開発者ガイド

Borland

C++Builder 6

for Windows

Borland Software Corporation, 100 Enterprise Way
P.O. Box 660001, Scotts Valley, CA 95067-0001

目次

第 30 章 データベースアプリケーションでの XML の使用 30-1

第 III 部 インターネットアプリケーションの作成

第 31 章 CORBA アプリケーションの作成 31-1

表目次

図目次

第 1 章

はじめに

この開発者ガイドでは，クライアント / サーバーデータベースアプリケーションの構築，カスタムコンポーネントの作成，インターネット Web サーバーアプリケーションの作成，SOAP，TCP/IP，COM+，Active X といった業界標準規格のサポートなど，中 / 上級者向けの開発について説明しています。Web 開発，先進の XML テクノロジー，データベース開発などをサポートする高度な機能の多くはコンポーネントやウィザードを必要としますが，C++Builder のバージョン（版）によっては，こうしたコンポーネントまたはウィザードの一部を使用できない場合があります。

この開発者ガイドは，C++Builder の操作に慣れ，C++Builder の基本的なプログラミング手法を理解している読者を対象としています。C++Builder を使ったプログラミングと統合開発環境（IDE）の概要については，『クイックスタート』およびオンラインヘルプを参照してください。

このマニュアルの内容

このマニュアルは，次の 5 部で構成されています。

- **第 I 部「C++Builder を使ったプログラミング」**では，汎用の C++Builder アプリケーションの作成方法について説明します。どのような C++Builder アプリケーションにも利用できるプログラミング手法について詳しく説明します。たとえば，VCL（ビジュアルコンポーネントライブラリ）と CLX（クロスプラットフォーム用クラスライブラリ）のよく使われるオブジェクトの使用方法について説明します。VCL/CLX オブジェクトを使うと，文字列の処理，テキストの操作，コモンダイアログの実装などのユーザーインターフェースのプログラミングが簡単に行えます。第 I 部には，グラフィックの操作，エラーおよび例外の処理，DLL および OLE オートメーションの使い方，国際化対応アプリケーションの作成に関する章も含まれます。

  通常，C++Builder の基盤となる VCL が Object Pascal で作成されていることが問題になることはほとんどありません。ただし，作成した C++Builder プログラムに影響が出てくることがたまにあります。C++ 言語のサポートおよび VCL に関する章では，VCL クラス使用時の C++ クラスのインスタンス化の特徴，また C++Builder の「コンポーネント - プロパティ - イベント」プログラミン

グモデルをサポートするために追加される C++ 言語の拡張機能など，言語の問題について詳しく説明します。

クロスプラットフォーム開発に関する章では，CLX のオブジェクトを使って Windows と Linux の両方でコンパイルし実行できるアプリケーションの開発方法について説明します。

配布に関する章では，作成したアプリケーションをアプリケーションユーザーに配布するときの作業について詳しく説明しています。たとえば，効果的なコンパイラオプション，InstallShield Express の使い方，ライセンスの問題などに関する情報があります。また，実稼働規模のアプリケーションを構築するときに，どのパッケージ，DLL，または他のライブラリを付属させるかを決定する方法についても説明しています。

- **第 II 部「データベースアプリケーションの開発」**では，データベースのツールとコンポーネントを使ったデータベースアプリケーションの構築方法について説明しています。C++Builder では，Paradox，dBASE などのローカルデータベースや，InterBase，Oracle，Sybase などのネットワーク SQL サーバーデータベースをはじめ，各種のデータベースにアクセスすることができます。dbExpress，BDE（ボーランドデータベースエンジン），InterBaseExpress，ActiveX Data Objects（ADO）など，さまざまなデータアクセスメカニズムを選択できます。高度なデータベースアプリケーションを実装するには Delphi の機能が必要です。ただし，C++Builder のバージョン（版）によっては使用できない機能があります。
- **第 III 部「インターネットアプリケーションの作成」**では，インターネットを介した分散アプリケーションの作成について説明します。C++Builder には，Web サーバーアプリケーションを作成するための各種ツールが付属しています。たとえば，Web ブローカ（クロスプラットフォーム サーバーアプリケーションを作成するアーキテクチャ），WebSnap（GUI 環境で Web ページ設計を行うアーキテクチャ），XML ドキュメント操作機能，BizSnap（SOAP ベースの Web サービスを使用するためのアーキテクチャ）などがあります。

  第 III 部では，C++Builder のソケットコンポーネントについても説明します。ソケットコンポーネントを使うと，TCP/IP および関連のプロトコルを使って他のシステムと通信できるアプリケーションを作成できます。ソケットは TCP/IP プロトコルに基づいた接続能力を提供しますが，汎用性が高く，Xerox Network System（XNS），DEC の DECnet プロトコル，Novell の IPX/SPX ファミリーなどの関連プロトコルにも使えます。
- **第 IV 部「COM ベースアプリケーションの開発」**では，COM ベースの API オブジェクトと相互運用できるアプリケーションの作成方法について説明します。C++Builder は，ATL（Active Template Library）に基づく COM アプリケーションをサポートします。ウィザードとタイプライブラリエディタによって COM サーバーの開発が簡略化され，インポートツールを使えばクライアントアプリケーションを短時間で作成できます。C++Builder のどのバージョン（版）も COM クライアントをサポートしています。COM サーバーは，C++Builder のバージョン（版）によってはサポートしていない場合があります。
- **第 V 部「カスタムコンポーネントの作成」**では，カスタムコンポーネントの設計と実装，およびそれらを IDE のコンポーネントパレットで利用できるようにする方法について説明します。コンポーネントとは，設計時に操作できるほとんどすべてのプログラム要素を意味します。カスタムコンポーネントの実装では，VCL または CLX のクラスライブラリにある既存のクラス型から新しいクラスを派生させます。

マニュアルの表記規則

このマニュアルでは，特別なテキストを表記するとき，表 1.1 に示す書体と記号を使用しています。

表 1.1　マニュアルで使用している書体と記号

書体または記号	説明
Monospace	この書体は画面に表示される内容や C++ コードの内容を表します。入力テキストを示す場合もあります。
[ ]	このカッコで囲まれたテキストや構文リスト項目は省略可能です。このようなテキストは，そのまま入力することはありません。
Boldface（太字）	コード内の太字で記された語は，C++ の予約語かコンパイラオプションです。
［メニュー名｜コマンド名］	この表記は，メニューバーのメニューとその中のコマンドを示します。 **例**：［ファイル｜印刷］を選択します。また，画面上の文字も［］で囲んで表記します。 **例**：［ファイルを開く］ダイアログボックスの［ファイル名］にファイル名を入力し，［OK］を選択します。
〔Keycaps〕	この表記はキーボードのキーを示します。 **例**：〔Esc〕を押してメニューを終了します。

開発者サポートサービス

ボーランドでは，インターネット上の無料サービス（大規模な情報ベースからの検索，他のボーランド製品ユーザーとの接続など），テクニカルサポート，有償のコンサルティングサービスなど，さまざまなサポートオプションを用意しています。

ボーランドの開発者サポートサービスについての詳細は，Web サイト http://www.borland.co.jp/support/ にアクセスしてください。

サポートにお問い合わせする際には，ご使用環境に関する詳しい情報，製品の版およびバージョン番号，問題点の詳細をお知らせください。

第 I 部

C++Builder を使ったプログラミング

第 I 部「C++Builder を使ったプログラミング」の各章では，製品のすべての版で共通の，C++Builder アプリケーションの作成に必要な概念およびテクニックを説明します。

第2章

C++Builder を使ったアプリケーション開発

Borland C++Builder は，32 ビットアプリケーションの開発を可能にする，オブジェクト指向のビジュアルプログラミング環境です。C++Builder を使うと，最小限のコーディングできわめて能率的なクロスプラットフォームアプリケーションを作成できます。

C++Builder は，プログラミングウィザード，アプリケーションテンプレート，フォームテンプレートなどの一連の RAD（Rapid Application Development）設計ツールを用意し，包括的なクラスライブラリを使用したオブジェクト指向プログラミングをサポートしています。

- VCL（ビジュアルコンポーネントライブラリ）には，Windows API をカプセル化する各種オブジェクトをはじめ有用なプログラミング手法が含まれています（Windows）。
- CLX（クロスプラットフォーム用クラスライブラリ）には，Qt ライブラリをカプセル化する各種オブジェクトが含まれています（Windows，Linux）。

この章では，C++Builder の開発環境と，開発の各段階について簡単に説明します。このマニュアルの残りの部分では，汎用，データベース，インターネット，イントラネットなど各目的別のアプリケーションの開発，および ActiveX コントロール，COM コントロール，独自のコンポーネントの作成について詳しく説明しています。

統合開発環境

C++Builder を起動すると，IDE と呼ばれる統合開発環境が開始されます。IDE にはアプリケーションの設計，開発，テスト，デバッグ，配布に必要なすべてのツールが用意されているので，プロトタイプを短期間に作成でき，開発期間を短縮することができます。

IDE には，アプリケーション設計を開始するのに必要なツールがすべて用意されています。主なツールは次のとおりです。

- フォームデザイナ（フォーム）：最初は空白のウィンドウであり，ここにアプリケーションのユーザーインターフェースを設計する
- コンポーネントパレット：ユーザーインターフェースの設計で使われるビジュアルコンポーネントと非ビジュアルコンポーネントを表示する
- オブジェクトインスペクタ：オブジェクトの特性およびイベントを調べたり変更したりする
- オブジェクトツリー：コンポーネント間の論理関係を表示，変更する
- コードエディタ：アプリケーションの基礎をなすプログラムロジックを記述，変更する
- プロジェクトマネージャ：1 つまたは複数のプロジェクトを構成するファイル群を管理する
- 統合デバッガ：作成したコードのエラーを発見，修正する
- プロパティエディタなどの他のツール：オブジェクトのプロパティ値を変更する
- コマンドラインツール：コンパイラやリンカなどのユーティリティ
- 大規模なクラスライブラリ。再利用可能なオブジェクトが数多く用意されているクラスライブラリに用意されたオブジェクトの多くは，IDE のコンポーネントパレットからアクセスできます。慣例的に，クラスライブラリのオブジェクト名は TStatusBar のように T で始まります。

製品のバージョン（版）によっては，一部のツールが付属していない場合があります。

開発環境に関する概要説明については，製品に付属する『クイックスタート』マニュアルを参照してください。また，オンラインヘルプは，メニュー，ダイアログボックス，およびウィンドウに関するヘルプを網羅しています。

アプリケーションの設計

C++Builder を使うと，汎用のユーティリティから洗練されたデータアクセスプログラム，分散アプリケーションに至るまで，あらゆる種類の 32 ビットアプリケーションを設計できます。

アプリケーションのユーザーインターフェースをビジュアルに設計する過程で，アプリケーションの基礎となる C++ コードが C++Builder によって自動的に生成されます。コンポーネントおよびフォームのプロパティを選択，変更するたびに，その変更の結果がソースコードに自動的に反映されます。逆にソースコードを変更すると，それがコンポーネントおよびフォームのプロパティに反映されます。C++Builder に付属するコードエディタなどのテキストエディタを使って，ソースファイルを直接変更できます。エディタで行った変更の結果は，ビジュアル環境に即時に反映されます。

C++Builder では，プログラマ独自のコンポーネントを作成することができます。C++Builder に用意されているコンポーネントは，ほとんどが Object Pascal で記述されています。プログラマは，自分で作成したコンポーネントをコンポーネントパレットに追加し，必要に応じて新しいタブを追加できます。

また，CLX を使えば，Linux と Windows の両方で動作するアプリケーションを設計することもできます。VCL のかわりに CLX のクラスを使うことにより，Windows と Linux の間でプログラムを移植することができます。クロスプラットフォームプログラミングに関する詳細および Windows 環境と Linux 環境の違いについては，第 14 章「クロスプラットフォームアプリケーションの開発」を参照してください。

C++Builder でサポートされているさまざまな種類のアプリケーションについては，第 7 章「アプリケーション，コンポーネント，ライブラリの構築」を参照してください。

プロジェクトの作成

プロジェクトは，C++Builder を使ったアプリケーション開発の中核を成します。C++Builder でアプリケーションを作成すると，プロジェクトファイルが生成されます。プロジェクトとは，アプリケーションを構成するファイルの集まりです。これらのファイルは，設計時に作成されるものもあれば，プロジェクトソースコードをコンパイルしたときに自動生成されるものもあります。

プロジェクトの中身を表示するには，「プロジェクトマネージャ」というプロジェクト管理ツールを使います。プロジェクトマネージャを使うと，ユニット名およびユニット内に格納されたフォーム（存在する場合）が階層状に一覧表示されます。このほか，プロジェクトを構成するファイルへのパスも表示されます。こうしたファイルの多くは直接編集することもできますが，C++Builder に付属のビジュアルツールを使った方が簡単かつ確実に編集できます。

プロジェクト階層の一番上にあるのがグループファイルです。複数のプロジェクトを 1 つのプロジェクトグループにまとめることができます。こうすると，プロジェクトマネージャを使って一度に複数のプロジェクトを開くことが可能です。プロジェクトグループを作成しておくと，互いに関連する複数のプロジェクト（たとえば，同時に機能するアプリケーション，多層アプリケーションのパーツなど）を整理して作業できます。単一のプロジェクトで作業する場合には，プロジェクトグループファイルを使わずにアプリケーションを作成できます。

プロジェクトファイルとは，個々のプロジェクト，ファイル，および関連オプションを記述したファイルです。プロジェクトファイルの拡張子は .bpr です。プロジェクトファイルには，アプリケーションや共有オブジェクトを構築するための指示が保存されています。プロジェクトマネージャを使ってファイルを追加，削除すると，プロジェクトファイルが更新されます。プロジェクトのオプションを設定するには，［プロジェクトオプション］ダイアログを使います。このダイアログには，フォーム，アプリケーション，コンパイラなど，プロジェクトの要素を示すタブがあります。設定したプロジェクトオプションは，プロジェクトと一緒にプロジェクトファイルに保存されます。

ユニットとフォームは，C++Builder アプリケーションの基本となる要素です。プロジェクトを作成するとき，既存のフォームやユニットファイルを共有することができます。既存のフォーム，既存のユニットファイルには，プロジェクトのディレクトリツリーの外にあるものも含まれます。具体的には，スタンドアロンルーチンとして記述されたカスタムの手続きや関数なども共有できます。

共有ファイルをプロジェクトに追加しても，現在のプロジェクトディレクトリにコピーされるのではありません。共有ファイル自身は元の場所に残っています。現在のプロジェクトに共有ファイルを追加すると，プロジェクトファイルにファイル名とパスが登録されます。プログラマがプロジェクトにユニットを追加すると，C++Builder 側で自動的に登録処理を行います。

プロジェクトをコンパイルするときも，プロジェクトを構成するファイルがどこにあるかは問題でありません。コンパイラは，プロジェクト自身が生成したファイルも共有ファイルも同じように処理します。

コードの編集

C++Builder のコードエディタは，完全な機能を備えたテキストエディタです。ビジュアルプログラミング環境では，新規プロジェクトを作成すると，1 つのフォームが自動的に表示されます。アプリケーションのインターフェースを設計するには，まずフォームにオブジェクトを配置して，オブジェクトインスペクタを使って機能を修正するという方法で作業を開始できます。しかし，オブジェクトのイベントハンドラを記述するなどのプログラミングタスクによっては，コードを入力する必要があります。

コードエディタを使うと，フォームの中身，プロパティ，フォーム上のコンポーネント，および各コンポーネントのプロパティをテキスト形式で表示し，編集することができます。コードエディタの中でコードを入力することにより，生成されたコードを調整したり，コンポーネントを追加したりできます。プログラマがコードエディタにコードを記述しているとき，コンパイラは絶えず変更内容をスキャンし，新規レイアウトを使ってフォームを更新しています。したがって，コードを記述する途中でフォームに戻り，変更内容を表示，テストしながら再びコードエディタでフォームを調整することができます。

C++Builder のコード生成およびプロパティストリーミングのシステムは，すべて表示され，容易に検査できます。VCL オブジェクト，CLX オブジェクト，RTL ソース，プロジェクトファイルなど，最終的に生成される実行形式ファイルを構成するすべてのソースコードを，コードエディタを使って表示，編集できます。

アプリケーションのコンパイル

フォーム上でアプリケーションインターフェースを設計し，必要に応じて追加コードを記述したら，IDE またはコマンドラインからプロジェクトをコンパイルできます。

どのプロジェクトも，配布可能な実行形式ファイル 1 つをターゲットとして持ちます。アプリケーションのコンパイル，ビルド（構築），または実行により，開発のさまざまな段階でアプリケーションの表示確認やテストを行えます。

- アプリケーションをコンパイルすると，前回のコンパイル以降に変更されたユニットだけが再コンパイルされます。
- アプリケーションをビルドする場合は，前回のコンパイル以降に変更されたかどうかを問わず，プロジェクトのすべてのユニットがコンパイルされます。どのファイルを変更したか（または変更していないか）不明のとき，あるいは，単にすべてのファイルを更新し同期させたいときは，ビルドという方法が便利です。また，グローバルなコンパイラ指令を変更した後もビルドを行うことが重要です。この場合，すべてのコードが適切な状態でコンパイルされます。さらに，プロジェクトをコンパイルせずにソースコードの妥当性をテストすることもできます。
- アプリケーションを実行する場合は，アプリケーションをコンパイルしてから実行することになります。前回のコンパイル以降にソースコードを修正した場合，コンパイラは修正モジュールを再コンパイルして，アプリケーションを再リンクします。

複数のプロジェクトをグループ化した場合は，同じグループ内のプロジェクトすべてを一度にコンパイルまたはビルドすることができます。プロジェクトマネージャで目的のプロジェクトグループを選択し，［プロジェクト｜すべてのプロジェクトをコンパイル］か［プロジェクト｜すべてのプロジェクトを再構築］を選択します。

CLX　CLX アプリケーションの Linux 上でのコンパイルについては，Linux C++ アプリケーションはまだ利用できませんが，現段階で C++Builder を使ってアプリケーションを開発することはできます。

アプリケーションのデバッグ

C++Builder には，アプリケーションエラーの発見および修復を支援するための統合デバッガが用意されています。統合デバッガを利用すると，プログラムの実行を制御したり，変数やデータ構造内の各要素を監視したり，デバッグ中にデータ値を変更できます。

統合デバッガは，実行時エラーと論理エラーの両方を追跡します。プログラムの特定位置を実行し，変数値，呼び出し履歴中の関数，およびプログラムの出力を表示します。これにより，プログラムが動作する様子をモニターし，設計どおりに動作していない領域を見つけることができます。デバッガについての詳細は，オンラインヘルプを参照してください。

例外処理を使ってエラーを識別し，その場所を特定して必要な処置を行うこともできます。C++Builder では，例外も一般のクラスと同様にクラスですが，慣例的に例外クラスの名前は T でなく E で始まります。例外処理についての詳細は，第 12 章「例外処理」を参照してください。

アプリケーションの配布

C++Builder には，アプリケーションの配布を支援するアドオンツールが含まれています。たとえば InstallShield Express を使って，配布するアプリケーションの実行に必要なすべてのファイルが含まれたインストールパッケージを作成できます（C++Builder のバージョン（版）によっては使用できない場合があります）。TeamSource を使うと，アプリケーションの更新履歴を追跡できます（C++Builder のバージョン（版）によっては使用できない場合があります）。

メモ　C++Builder のバージョン（版）によっては，配布機能がない場合があります。

CLX　CLX アプリケーションの Linux での配布については，Linux C++ アプリケーションはまだ利用できませんが，現段階で C++Builder を使ってアプリケーションを開発することはできます。

アプリケーションの配布についての詳細は，第 17 章「アプリケーションの配布」を参照してください。

第3章

クラスライブラリの使い方

この章では，クラスライブラリについて説明し，アプリケーションの開発時に使用できるコンポーネントをいくつか紹介します。C++Builder には，VCL（ビジュアルコンポーネントライブラリ）と CLX（クロスプラットフォーム用コンポーネントライブラリ）の両方が付属しています。VCL は，Windows アプリケーションの開発で使用するコンポーネントライブラリです。CLX は，Windows と Linux の両方で動作するクロスプラットフォームアプリケーションの開発で使用するコンポーネントライブラリです。VCL と CLX は互いに異なるクラスライブラリですが，多くの類似点があります。

クラスライブラリを理解する

VCL と CLX は，多数のオブジェクトから構成されるクラスライブラリです。アプリケーションを開発するとき，このクラスライブラリを使います。VCL と CLX は互いに類似したクラスライブラリであり，多くのオブジェクトを共通して持っています。VCL のオブジェクトには，Windows でしか使用できない機能を実装したオブジェクトが一部含まれています。具体的には，コンポーネントパレットの［ADO］，［BDE］，［QReport］，［COM+］，［Servers］の各ページに表示されるオブジェクトなどが Windows 専用のオブジェクトです。CLX オブジェクトは，ほとんどすべてが Windows，Linux の両方で使用できます。

VCL のオブジェクトはすべて TObject から派生しています。TObject とは，作成，破棄，メッセージ処理などの基本動作をカプセル化するメソッドだけを持つ抽象クラスです。独自のクラスを作成するには，使用するクラスライブラリの TObject からクラスを派生させます。

コンポーネントとは，VCL，CLX それぞれの下位セットです。抽象クラス TComponent から派生しています。フォームやデータモジュールにコンポーネントを配置して，設計時に操作することができます。大部分のコンポーネントは，ビジュアルコンポーネントか非ビジュアルコンポーネントのどちらかです（実行時に目に見えるものはビジュアル，目に見えないものは非ビジュアルです）。一部のコンポーネントはコンポーネントパレットに表示されます。

ビジュアルコンポーネント（TForm，TSpeedButton など）のことを「コントロール」と呼びます。コントロールは TControl から派生します。TControl から派生するオブジェクトは各種のプロパティを持っています。プロパティ，コントロールのビジュアル属性（高さ，幅など）を指定する働きをします。

非ビジュアルコンポーネントを使うと，さまざまなタスクができます。たとえばデータベースに接続するアプリケーションを作成する場合，フォームに TDataSource コンポーネントを配置すると，あるコントロールと，そのコントロールが使用するデータセットとを接続することができます。この接続はユーザーの目に見えないので，TDataSource は非ビジュアルコンポーネントです。設計時，非ビジュアルコンポーネントはアイコンの形で表示されます。このため，ビジュアルコンポーネントと同じようにプロパティやイベントを操作できます。

プログラミング中にオンラインヘルプを利用すると，VCL オブジェクトと CLX オブジェクトに関する詳しいリファレンス情報を参照できます。コードエディタを表示しているときは，オブジェクトのどこかにカーソルを置いて〔F1〕キーを押すと，ヘルプトピックが表示されます。VCL のオブジェクト，プロパティ，メソッドは「VCL リファレンス」，CLX のオブジェクト，プロパティ，メソッドは「CLX リファレンス」と表示されます。

プロパティ，メソッド，イベント

VCL，CLX はそれぞれオブジェクト階層を構成し，IDE と連携してアプリケーションの迅速な開発を可能にします。VCL オブジェクトと CLX オブジェクトは，どちらもプロパティ，メソッド，およびイベントを基本としています。各オブジェクトには，データメンバー（プロパティ），データに作用する関数（メソッド），およびクラスのユーザーとの対話方法（イベント）が含まれています。VCL も CLX も Object Pascal で記述されていますが，VCL は Windows API をベースとし，CLX は Qt ウィジェットライブラリをベースにしています。

プロパティ

プロパティとは，コンポーネントの特性のことです。プロパティはオブジェクトの可視動作または操作に影響を与えます。たとえば Visible プロパティは，オブジェクトをアプリケーションのインターフェースに表示するか非表示にするかを指定します。プロパティを適切に設計すれば，ユーザーにとっては使いやすく，開発者にとっては保守しやすいコンポーネントが作成できます。

プロパティの有用な機能をいくつか紹介します。

- メソッドは実行時でなければ使用できませんが，プロパティは設計時に表示，変更ができ，変更の結果はすぐに IDE を通してコンポーネントに反映されます。
- オブジェクトインスペクタからプロパティにアクセスでき，オブジェクトの値をビジュアルに変更できます。コードを記述するより，設計時にプロパティを設定する方が簡単です。コードの保守も簡単になります。
- データはカプセル化されるので，プロテクトされ，実際のオブジェクトに対してプライベートになります。
- 値を取得，設定するための実際の呼び出しを行うのはメソッドです。このため，オブジェクトのユーザーの目に見えない特殊な処理を実行できます。たとえば，テーブルにデータがあっても，プログラマに対しては通常のデータメンバーとして表示するなどが可能です。
- プロパティのアクセス中にイベントを発生させたり他のデータを変更するロジックを実装できます。たとえば，あるプロパティの値を変更するとき，別のプロパティも変更する必要が生じることがあります。その場合，目的のプロパティのメソッドを変更することができます。

- プロパティを仮想プロパティとすることもできます。
- 1 つのプロパティを複数のオブジェクトに適用することもできます。オブジェクトのプロパティを 1 つ変更するだけで，複数のオブジェクトを変更することもできます。たとえば，ラジオボタン 1 つの Checked プロパティを設定するだけで，同じグループ内のラジオボタン全体に影響を与えることが可能です。

メソッド

メソッドとは，同じクラスに属する関数のことです。メソッドはオブジェクトの動作を定義します。クラスメソッドは同じクラスの public プロパティ，protected プロパティ，private プロパティ，およびデータメンバーのすべてにアクセスできます。一般に，クラスメソッドのことを「メンバー関数」と呼びます。46-4 ページの「アクセスの制御」を参照してください。

イベント

イベントとは，プログラムにより検出されるアクションまたは出来事のことです。最近のアプリケーションは，ほとんどが「イベントドリブン」であると言われます。これは，イベントに応答するように設計されているからです。プログラムを作成するとき，プログラマは，ユーザーがどんな順序で操作するのか予想できません。メニュー項目を選択する，ボタンをクリックする，テキストの一部を強調表示するなど，ユーザーはさまざまな操作をします。プログラマは，常に一定の順序で実行されるコードを記述するのでなく，発生しそうなイベントに対処できるコードを記述することができます。

イベント名を問わず何かイベントが発生すると，C++Builder は，そのイベントを処理するコードがあるかどうかを確認します。何かコードが記述されていれば，そのコードを実行します。記述されていなければ，デフォルトのイベント処理動作を発生させます。

発生し得るイベントは，大きく分けて次の 2 種類があります。

- ユーザーイベント
- システムイベント

ユーザーイベント

ユーザーイベントとは，ユーザーが開始するアクションのことです。ユーザーイベントの例としては，OnClick イベント（マウスをクリックする），OnKeyPress イベント（キーボードのキーを押す），OnDblClick イベント（マウスをダブルクリックする）などがあります。

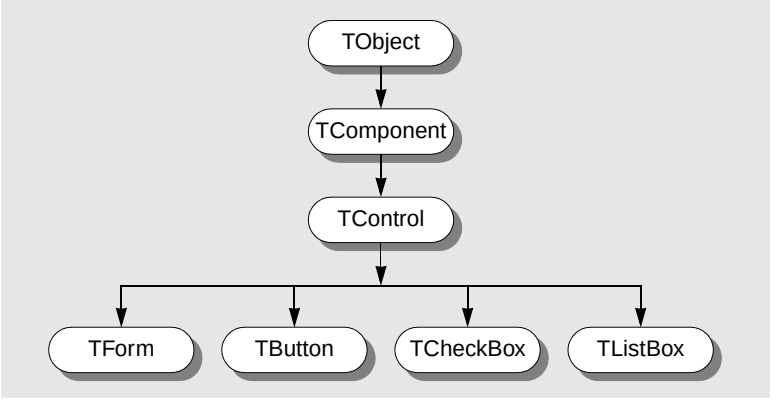
システムイベント

システムイベントとは，オペレーティングシステムがユーザーにかわって発生させるイベントのことです。システムイベントの例としては，OnTimer イベント（一定の時間が経過すると，Timer コンポーネントがシステムイベントを発行する），OnCreate イベント（コンポーネントが作成される），OnPaint イベント（コンポーネントやウィンドウを再描画する必要が生じる）などがあります。システムイベントは，通常，ユーザーのアクションと直接関係せずに開始します。

オブジェクト，コンポーネント，コントロール

図 3.2 は，継承の階層を単純化した図です。オブジェクト，コンポーネント，コントロール間の関係を示しています。

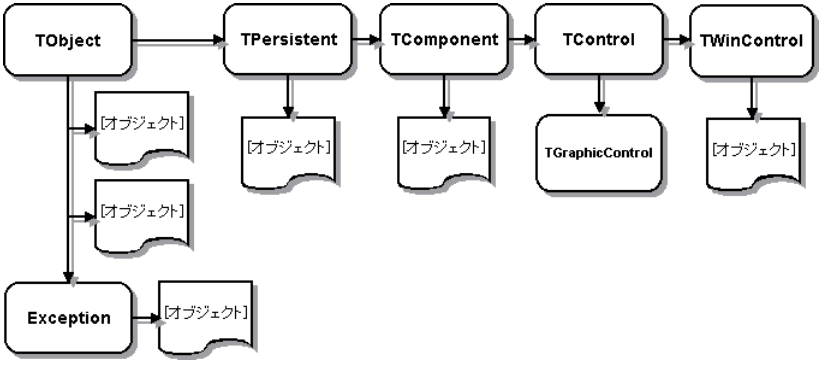
図 3.1　オブジェクト，コンポーネント，コントロール



どのオブジェクトも TObject から派生し，多くのオブジェクトが TComponent から派生します。コントロールは TControl から要素を継承するオブジェクトで，実行時にコントロール自身を表示する能力を持ちます。TCheckBox のようなコントロールの場合，TObject，TComponent，TControl の全機能を継承するだけでなく，独自の特殊機能も持っています。

図 3.2 は VCL（ビジュアルコンポーネントライブラリ）の構造を簡単に表したもので，継承ツリーの中の主要ブランチを示しています。この図のレベルでは，CLX（クロスプラットフォーム用コンポーネントライブラリ）と VCL はほとんど同じです。ただし，VCL の TWinControl は CLX では TWidgetControl になります。

図 3.2　単純化した階層図



前の図はいくつかの基本クラスを示しています。各基本クラスの説明を下の表に示します。

表 3.1　重要な基本クラス

クラス	説明
TObject	VCL または CLX のあらゆるクラスの最上位に位置する基本クラス。オブジェクトのインスタンスの作成，保守，破棄などの基本機能を実行するメソッドを導入する。これにより，すべての VCL/CLX オブジェクトに共通する基本動作をカプセル化する
Exception	例外に関連するクラスの基本クラス。エラー条件に対する一貫したインターフェースを提供し，アプリケーションが的確にエラーを処理できるようにする
TPersistent	プロパティを実装するオブジェクトの基本クラス。TPersistent に属するクラスは，ストリームへのデータ送信を処理し，クラスの割り当てを可能にする
TComponent	非ビジュアルコンポーネント（TApplication など）の基本クラス。TComponent は，すべてのコンポーネントに共通する上位クラスである。TComponent クラスでは，コンポーネントをコンポーネントパレットに表示でき，コンポーネント自身がほかのコンポーネントを所有できる。また，フォーム上で直接コンポーネントを操作できる
TControl	実行時に表示されるコントロールの基本クラス。TControl はすべてのビジュアルコンポーネントに共通する上位クラスであり，位置やカーソルのような標準のビジュアルコントロールを提供する。加えて，マウスアクションに応答するイベントも提供する
TWinControl	ユーザーインターフェースオブジェクトの基本クラス。TWinControl に属するコントロールは，キーボード入力を受け取ることのできるウィンドウコントロールである。CLX では，ウィンドウコントロールのことを「ウィジェット」と呼び，TWinControl の代わりに TWidgetControl が使われる

以下のセクションでは，各ブランチに属するクラスの種類について概要を説明します。VCL と CLX のオブジェクト階層の全体図については，製品に同梱されているポスター「VCL Object Hierarchy」および「CLX Object Hierarchy」を参照してください。

TObject ブランチ

TObject ブランチは TObject から派生した VCL オブジェクトおよび CLX オブジェクトで構成されます。ただし，TPersistent から派生したオブジェクトは含まれません。VCL/CLX オブジェクトの強力な機能の多くは，TObject が導入するメソッドによって実装されています。TObject は，以下の機能を提供するメソッドを導入して，VCL と CLX の全オブジェクトに共通する基本動作をカプセル化します。

- オブジェクトが作成 / 破棄されたときに応答できる
- オブジェクトのクラス型とインスタンス情報，およびパブリッシュプロパティに関する実行時型情報（RTTI：Runtime Type Information）を提供する
- メッセージ処理（VCL）またはシステムイベント（CLX）をサポートする

TObject から多くの単純なクラスが直接派生します。TObject ブランチに含まれるクラスには，重要な共通特性が 1 つあります。それは，どのクラスも一時的だということです。TObject ブランチに属するクラスは，破棄される前の状態を保存するメソッドを持っていません。すなわち持続性がありません。

その代表的な例が Exception クラスです。Exception クラスは，大規模な一連の組み込みの例外クラスを提供して，ゼロ除算，ファイル入出力エラー，無効な型キャスト，その他の例外条件を自動的に処理します。

TObject ブランチには，次のようにデータ構造をカプセル化するクラスもいくつかあります。

- TBits：論理値の「配列」を格納するクラス
- TList：リストとリンクするクラス
- TStack：ポインタの後入れ先出し配列を管理するクラス
- TQueue：ポインタの先入れ先出し配列を管理するクラス

VCL には，TPrinter や TRegistry といった，外部オブジェクトのラッパーもあります。TPrinter は，Windows のプリンタインターフェースをカプセル化します。TRegistry は，システムレジストリおよびその操作を行う関数向けの低レベルのラッパーです。これらは Windows 環境に固有のものです。

このほか，TObject ブランチには，TStream のようなクラスもあります。TStream は，ディスクファイルやダイナミックメモリなどの各種の記憶媒体に対して読み書きのできる，ストリームオブジェクトの基本となるクラス型です。

全体として，TObject にはアプリケーション開発に役立つ多種多様なクラスがあります。

TPersistent ブランチ

TPersistent ブランチは TPersistent から派生した VCL オブジェクトおよび CLX オブジェクトで構成されます。ただし，TComponent から派生したオブジェクトは含まれません。持続性によって，フォームファイルまたはデータモジュールとともに何が保存されるか，またメモリから取り出されるときにフォームファイルまたはデータモジュールに読み出すものを決定できます。

TPersistent ブランチに属するオブジェクトは，コンポーネントのプロパティを実装しています。フォームと一緒にプロパティの読み込みと保存をするには，プロパティのオーナーが存在していなければなりません。プロパティのオーナーとは，何らかのコンポーネントです。TPersistent ブランチから GetOwner 関数が導入されるので，この関数を使ってプロパティのオーナーを決定できます。

また，TPersistent ブランチ以降は published 部を持てるため，プロパティの読み込みと保存を自動的に行うことができます。また，DefineProperties メソッドを使うと，プロパティの読み込み方法と保存方法を指定できます。

このほか，TPersistent ブランチには以下のクラスもあります。

- TGraphicsObject：グラフィックオブジェクト（TBrush，TFont，TPen など）の抽象基本クラス
- TGraphic：ビジュアルイメージを保存，表示するオブジェクト（TBitmap，TIcon など）の抽象基本クラス
- TStrings：文字列リストを表すオブジェクトの基本クラス
- TClipboard：アプリケーションでカットまたはコピーしたテキストやグラフィックを格納するクラス
- TCollection，TOwnedCollection，TCollectionItem：特別に定義した項目のインデックス付きコレクションを管理するクラス

TComponent ブランチ

TComponent ブランチは TComponent から派生したオブジェクトで構成されます。ただし，TControl から派生したオブジェクトは含まれません。TComponent に属するオブジェクトは，設計時にフォー

ム上で操作できるコンポーネントです。TComponent オブジェクトは，以下の機能を持つ持続的なオブジェクトです。

- コンポーネントパレットに表示され，フォームデザイナ内で操作できる
- コンポーネント自身がほかのコンポーネントを所有 / 管理できる
- コンポーネント自身の読み込みと保存ができる

TComponent には，コンポーネントの動作を設計時に指示するメソッドや，コンポーネントと一緒に保存すべき情報を指示するメソッドがいくつかあります。ストリーミングが導入されるのは VCL および CLX の TComponent ブランチからです。C++Builder は，ほとんどのストリーミング操作を自動的に処理します。パブリッシュに設定されたプロパティは持続性を持ち，自動的にストリーミングされます。

また，VCL と CLX で広く使われている所有関係という考え方も TComponent から導入されます。所有関係をサポートするプロパティは，Owner プロパティと Components プロパティです。各コンポーネントに Owner プロパティがあります。Owner プロパティは別のコンポーネントをオーナーとして参照します。1 つのコンポーネントが複数のコンポーネントを所有することもあります。その場合，所有されている側のコンポーネントはすべて，所有する側のコンポーネントの Array プロパティで参照されます。

コンポーネントのコンストラクタはパラメータを 1 つ取り，そのパラメータを使って新規コンポーネントのオーナーを指定します。すでにオーナーが渡されていれば，そのオーナーのコンポーネントリストに新規コンポーネントが追加されます。Owner プロパティは，コンポーネントリストを使って被所有コンポーネントを参照するだけでなく，被所有コンポーネントを自動的に破棄する働きもします。オーナーを持つコンポーネントであれば，オーナーが破棄された時点でコンポーネントも破棄されます。たとえば，TForm は TComponent の下位オブジェクトなので，フォームが破棄されると，フォームが所有していたコンポーネントもすべて破棄され，そのメモリも解放されます。ここでは，デストラクタが呼び出されたとき，フォーム上のすべてのコンポーネントが自分自身を適切にクリーンアップすることを前提としています。

プロパティ型が TComponent またはその下位オブジェクトの場合，ストリームシステムは，読み出しをするときにその型のインスタンスを作成します。プロパティ型が TComponent でなく TPersistent の場合，ストリームシステムは，プロパティを通じて使用可能な既存のインスタンスを使ってインスタンスのプロパティ値を読み込みます。

フォームファイル（フォーム上のコンポーネントに関する情報を保存するファイル）を作成する段階になると，フォームデザイナはコンポーネント配列内をループして，フォーム上のすべてのコンポーネントを保存します。どのコンポーネントも，各自のプロパティに加えられた変更内容をストリーム（この場合はテキストファイル）に書き出す方法を知っています。逆に，フォームファイルからコンポーネントのプロパティを読み込むときは，コンポーネント配列内をループして各コンポーネントを読み込みます。

TComponent ブランチには以下のクラスがあります。

- TActionList：コンポーネントやコントロール（メニュー項目，ボタンなど）と一緒に使われるアクションリストを管理するクラス
- TMainMenu：フォームのメニューバーとそれに付随するドロップダウンメニューを提供するクラス

- TOpenDialog，TSaveDialog，TFontDialog，TFindDialog，TColorDialog など：共通して使われるダイアログボックスを提供する
- TScreen：アプリケーションによってインスタンス化されたフォームとデータモジュール，アクティブなフォームおよびフォーム内でアクティブなコントロール，画面のサイズと解像度，およびアプリケーションで使えるカーソルとフォントを追跡するクラス

ビジュアルインターフェースが不要なコンポーネントを作成するには，TComponent から直接派生させます。TTimer デバイスなどのツールを作成する場合は，TComponent から派生させます。この種のコンポーネントはコンポーネントパレットにありますが，実行時にユーザーインターフェースとして目に見える形で現れるのでなく，コードでアクセスされる内部関数を実行する働きをします。

CLX の TComponent ブランチには，THandleComponent というクラスもあります。THandleComponent は，アプリケーションの基礎を成す Qt オブジェクト（ダイアログ，メニューなど）へのハンドルを必要とする非ビジュアルコンポーネントの基本クラスです。

コンポーネントのプロパティの設定，メソッドの呼び出し，イベントの操作についての詳細は，第 5 章「コンポーネントの利用」を参照してください。

TControl ブランチ

TControl ブランチは，TControl から派生したコンポーネントで構成されます。ただし，TWinControl（CLX の TWidgetControl）から派生したコンポーネントは含まれません。TControl ブランチに属するオブジェクトはコントロールです。コントロールとは，実行時にユーザーが見たり操作したりできるビジュアルオブジェクトです。どのコントロールも，コントロールの表示に関係するプロパティ，メソッド，およびイベントを持っています。たとえば，コントロールの位置，コントロールのウィンドウ（CLX のウィジェット）に関連付けられているカーソル，コントロールをペイントしたり移動したりするメソッド，マウス動作に応答するイベントなどを持っています。コントロールはキーボード入力を受け付けません。

TComponent がコンポーネントの動作を定義するのに対し，TControl はビジュアルコントロールの動作を定義します。具体的には，描画ルーチン，標準イベント，包含関係などを定義します。

すべてのビジュアルコントロールが共通して持っているプロパティがいくつかあります。上に挙げたプロパティは TControl から継承され，オブジェクトインスペクタに表示されますが，該当するコンポーネント以外にはパブリッシュされません。たとえば，TImage の色は表示するグラフィックによって決まるので，TImage は Color プロパティをパブリッシュしません。

コントロールは大きく分けて次の 2 種類があります。

- 自分自身のウィンドウ（またはウィジェット）を持つコントロール
- 自分の親のウィンドウ（またはウィジェット）を使用するコントロール

自分自身のウィンドウを持つコントロールのことを，VCL では「ウィンドウコントロール」，CLX では「ウィジェットベースのコントロール」と呼びます。この種のコントロールは TWinControl（CLX では TWidgetControl）から派生します。ボタン，チェックボックスなどがこのクラスのコントロールです。

自分の親のウィンドウ（ウィジェット）を使用するコントロールを「グラフィックコントロール」と呼びます。この種のコントロールは TGraphicControl から派生します。イメージ，図形などがグラフィックコントロールです。グラフィックコントロールはハンドルを持たず，入力フォーカスを受け取ることができません。グラフィックコントロールはハンドルが不要なので，使用するシステムリソースが少なくなります。グラフィックコントロールは自身を描画するコントロールであり，他のコントロールの親にはなれません。

他のグラフィックコントロールに関する詳細は，9-18 ページの「グラフィックコントロール」を参照してください。各種のコントロールについての詳細は，第 9 章「コントロールの種類」を参照してください。実行時のコントロールとの相互作用については，第 6 章「コントロールの利用」を参照してください。

TWinControl/TWidgetControl ブランチ

VCL では，TWinControl ブランチは TWinControl から派生したコントロールで構成されます。TWinControl は，ウィンドウコントロール（すなわち，アプリケーションのユーザーインターフェースで使用するボタン，ラベル，スクロールバーなどの項目）の基本クラスです。ウィンドウコントロールは Windows のコントロールのラッパーです。

CLX では，TWidgetControl が TWinControl の代わりになります。TWidgetControl とは，ウィジェットコントロール（ウィジェットのラッパー）の基本クラスです。

ウィンドウコントロールとウィジェットコントロールの特徴は次のとおりです。

- アプリケーションの実行時にフォーカスを受け取ることができる。つまり，アプリケーションユーザーのキーボード入力を受け付ける。これに対して，他のコントロールはデータを表示することしかできない
- 1 つ以上の子コントロールの親になることができる
- ハンドル（一意の識別子）を持つ

TWinControl/TWidgetControl ブランチには，自動的に描画されるコントロール（TEdit，TListBox，TComboBox，TPageControl など）と，C++Builder が描画すべきカスタムコントロール（TDBNavigator，TMediaPlayer（VCL のみ），TGauge（VCL のみ））の両方があります。TWinControl/TWidgetControl から直接派生するオブジェクトは，一般に標準コントロール（編集フィールド，コンボボックス，リストボックス，ページコントロールなど）を実装することが多いため，ペイントの仕方はすでにわかっています。

ウィンドウハンドルを必要とするコンポーネントのうち，自身を再描画する機能を持つ標準コントロールをカプセル化しないコンポーネントに対しては，TCustomControl というクラスが用意されています。しかし，コントロールの描画やイベント応答に関する実装の細かいことを心配する必要はありません。プログラマにかわって C++Builder が完全にカプセル化してくれます。

第4章

BaseCLX の使い方

VCL と CLX の両方に存在するいくつかのユニットが，これら 2 つのコンポーネントライブラリに基本サポートを提供します。このようなユニットを総称して「BaseCLX」と呼びます。BaseCLX には，コンポーネントパレットに表示されるコンポーネントは 1 つもありません。BaseCLX は，コンポーネントパレットに表示されるコンポーネントではなく，コンポーネントが使用するクラスやグローバルルーチンで構成されます。プログラマがアプリケーションコードを記述したり独自のクラスを作成するときも，BaseCLX のクラスやグローバルルーチンを使用できます。

メモ BaseCLX を構成するグローバルルーチンは「ランタイムライブラリ」とも呼ばれます。このグローバルルーチンと C++ のランタイムライブラリを混同しないようにしてください。BaseCLX のグローバルルーチンの多くが C++ のランタイムライブラリに似た機能を実行しますが，関数名が大文字で始まる点と，ユニットのヘッダーファイルで宣言される点が異なります。

以降の節では，BaseCLX を構成するクラスとルーチンについて説明し，その使い方を示します。主なトピックは次のとおりです。

- ストリームの使用
- ファイルの処理
- .ini ファイルの処理
- リストの操作
- 文字列リストの操作
- 文字列の処理
- 計量単位の変換
- 描画スペースの作成

メモ 上に挙げたリスト以外にもタスクがあります。BaseCLX のランタイムライブラリには，ここで説明していないタスクを実行する多くのルーチンがあります。たとえば，多数の算術関数（Math ユニットに定義），日付 / 時刻値を処理するルーチン（SysUtils ユニットと DateUtils ユニットに定義），Object Pascal のバリアントを処理するルーチン（Variants ユニットに定義）などがあります。

ストリームの使用

ストリームとは，データの読み書きを可能にするクラスのことです。ストリームは，さまざまなメディア（メモリ，文字列，ソケット，データベースの BLOB フィールドなど）に対して読み書きの共通インターフェースを提供します。いくつかのストリームクラスがありますが，すべて TStream から派生しています。メディアの種類ごとに個別のストリームクラスがあります。たとえば，TMemoryStream はメモリイメージとの間で読み書きを行い，TFileStream はファイルとの間で読み書きをします。

ストリームを使ってデータの読み書きを行う

どのストリームクラスにも，データを読み書きするための共通のメソッドがいくつかあります。これらのメソッドは，以下の点で分類されます。

- 読み出された，または書き込まれたバイト数を返すかどうか
- バイト数を知る必要があるかどうか
- エラー時に例外を生成するかどうか

読み書きをするためのストリームメソッド

Read メソッドは，指定されたバイト数だけ現在位置からバッファへと読み出します。その後，実際に読み出したバイト数だけ読み書き位置を進めます。Read のプロトタイプは次のとおりです。

```
virtual int __fastcall Read(void *Buffer, int Count);
```

現在位置を越えるデータが Count バイト数に満たない場合には，実際に転送されるバイト数は Count より小さくなります。Read は実際に転送されたバイト数を返すため，ファイルのバイト数が事前にわからないときに便利な関数です。

Write メソッドは，バッファからストリームへと現在位置から Count バイト分書き込みます。Write のプロトタイプは，以下のとおりです。

```
virtual int __fastcall Write(const void *Buffer, int Count);
```

書き込み後，実際に書き込めたバイト数だけ読み書き位置を進め，関数の戻り値とします。バッファの終端に到達した場合，あるいはストリームがこれ以上のバイト数を受け付けられない場合には，書き込んだバイト数は Count より小さいことがあります。

上記関数と類似した手続きとして ReadBuffer と WriteBuffer があります。これらは，Read や Write とは異なり，読み出されたバイト数や書き込まれたバイト数を返しません。構造体から読み出しをする場合のように，必要なバイト数があらかじめわかっている場合に，これらの手続きは有効です。バイト数を正確に一致させられない場合，ReadBuffe と WriteBuffer は例外（EReadError と EWriteError）を生成します。この点が Read，Write メソッドと明確に違います。Read と Write は，要求値とのバイト数の差を返すことができます。ReadBuffer と WriteBuffer のプロトタイプは，次のようになります。

```
virtual int __fastcall ReadBuffer(void *Buffer, int Count);
virtual int __fastcall WriteBuffer(const void *Buffer, int Count);
```

これらのメソッドは，実際の読み出しや書き込みは Read メソッドと Write メソッドを呼び出すことで実行します。

コンポーネントの読み出しと書き込み

TStream には，コンポーネントの読み書きを行う専用メソッド（ReadComponent と WriteComponent）が定義されています。アプリケーションでこれらのメソッドを使うと，アプリケーションを作成したり実行時に変更したりしたとき，コンポーネントとそのプロパティを保存できます。

IDE は，この ReadComponent メソッドと WriteComponent メソッドを使ってフォームファイルとの間でコンポーネントの読み書きをします。フォームファイルとの間でコンポーネントを読み書きするとき，ストリームクラスは TFiler クラス，TReader，および TWriter と一緒に動作して，オブジェクトをフォームファイルから読み出したりディスクへ出力したりします。ストリームシステムについての詳細は，オンラインヘルプで TStream，TFiler，TReader，TWriter，および TComponent クラスを参照してください。

ストリーム間のデータのコピー

あるストリームから別のストリームにデータをコピーする場合は，明示的にデータの読み出しと書き込みを行う必要はありません。かわりに，次の例に示すように CopyFrom メソッドを使用できます。

アプリケーションには，2 つの編集コントロール（FromFile，ToFile）と［Copy File］ボタンが含まれています。

```
void __fastcall TForm1::CopyFileClick(TObject *Sender)
{
  TStream* stream1=TFileStream::Create(From.Text,fmOpenRead | fmShareDenyWrite);
  try
  {
    TStream* stream2 -> TFileStream::Create(To.Text fmOpenCreate | fmShareDenyRead);
    try
    {
      stream2 -> CopyFrom(stream1, stream1->Size);
    }
    __finally
    {
     delete stream2;
    }
  }
  __finally
  {
    delete stream1;
  }
}
```

ストリームの位置とサイズの指定

ストリームは，読み書き用のメソッドを提供することに加えて，アプリケーションがストリーム内の任意の位置を検索したり，ストリームのサイズを変更したりできるようにします。特定の位置を検索すると，次の読み書き操作ではその位置からストリームの読み書きを始めます。

特定の位置を検索する

ストリーム内の特定の位置に移動する方法としては，Seek メソッドがもっとも一般的なメカニズムです。Seek メソッドには次の 2 つのオーバーロードがあります。

```
virtual int __fastcall Seek(int Offset, Word Origin);
virtual __int64 __fastcall Seek(const __int64 Offset, TSeekOrigin Origin);
```

どちらのオーバーロードも同じように機能します。両者の違いは，一方が 32 ビット整数を使って位置とオフセットを表すのに対し，もう一方は 64 ビット整数を使う点です。

Origin パラメータは，Offset パラメータの解釈方法を示します。Origin の値は以下のいずれかになります。

値	説明
soFromBeginning	Offset の起点はリソースの先頭。Seek は Offset の位置まで移動する。Offset >= 0 でなければならない
soFromCurrent	Offset の起点はリソース内の現在の位置。Seek は Position + Offset に移動する
soFromEnd	Offset の起点はリソースの末尾。Offset <= 0 でなければならない。ファイル末尾からのバイト数をマイナスで示す

Seek は，現在のストリーム位置を指定されたオフセット分だけ移動して，再設定します。Seek は，ストリーム内の新しい現在位置を返します。

Position プロパティと Size プロパティの使い方

どのストリームも，ストリームの現在位置とサイズを保持するプロパティがあります。Seek メソッドのほか，ストリームとの間で読み書きを行うすべてのメソッドが位置とサイズのプロパティを使います。

Position プロパティは，ストリームデータの先頭からの現在のオフセットをバイト数で示します。

Size プロパティは，ストリームのサイズをバイト数で示します。Size プロパティを使うと，読み出しに使用できるバイト数を決定したり，ストリーム内のデータを切り捨てることができます。

Size プロパティはストリームへのデータの書き込み，またはストリームからのデータの読み出しをするルーチンによって内部的に使用されます。

Size プロパティを設定すると，ストリーム内のデータのサイズが変更されます。たとえばファイルストリームで Size を設定すると，ファイルを切り詰めるための EOF マーカーが挿入されます。ストリームの Size が変更できない場合は，例外が生成されます。たとえば，読み取り専用ファイルストリームの Size を変更しようとすると，例外が生成されます。

ファイルの処理

BaseCLX は，数通りの方法でファイルの処理をサポートします。ファイルストリームの使用のほかに，ファイル I/O を実行するランタイムライブラリルーチンがいくつかあります。ファイルストリーム，およびファイルの読み書き用のグローバルルーチンについては，4-5 ページの「ファイル入出力の方法」で説明します。

入出力操作だけでなく，ディスク上でファイルを操作することもできます。ファイルの中身でなくファイル自体に対する操作のサポートについては，4-7 ページの「ファイルの操作」で説明します。

メモ Object Pascal は大文字小文字を区別しませんが，Linux は区別します。クロスプラットフォームアプリケーションで CLX を使用するときは，このことを忘れないでください。ファイルを操作するオブジェクトやルーチンを使うときは，ファイル名の大文字小文字に注意してください。

ファイル入出力の方法

ファイルの読み書きを行うときは，次の 3 つの方法を使用できます。

- ファイルの処理で推奨される方法は，ファイルストリームを使うことです。ファイルストリームは TFileStream クラスのオブジェクトインスタンスであり，ディスクファイル内の情報へのアクセスに使用されます。ファイルストリームは移植性があり，ファイル I/O への高レベルのアプローチです。ファイルストリームはファイルハンドルを使用できるので，2 番目の方法と組み合わせて使うこともできます。次の節「ファイルストリームの使い方」で，TFileStream について詳しく説明します。
- ハンドルを使った方法でファイルを処理することもできます。ファイルの中身を操作するためにファイルを作成するか既存ファイルを開くと，オペレーティングシステム側でファイルハンドルを割り当てます。SysUtils ユニットに，ファイルハンドルを使ってファイルを操作するファイル処理ルーチンがいくつか定義されています。Windows では，これらのルーチンは通常，Windows API 関数のラッパーです。Delphi の関数は Object Pascal の構文を使用し，デフォルトのパラメータ値を備える場合もあるため，Windows API への便利なインターフェースとなります。その上 Linux 版のルーチンもあるので，クロスプラットフォームアプリケーションでも使用できます。ハンドルベースの方法を使用するには，最初に FileOpen 関数を使ってファイルを開くか，FileCreate 関数を使ってファイルを新規作成します。ファイルハンドルにアクセスしたら，ハンドルベースの各種ルーチンで中身を操作できます（線を書き込む，テキストを読み出すなど）。
- C のランタイムライブラリおよび標準 C++ ライブラリに，ファイルを処理する関数とクラスがいくつか含まれています。これらのライブラリは，VCL も CLX も使用しないアプリケーションで使えるという点でメリットがあります。これらの関数についての詳細は，C のランタイムライブラリまたは標準 C++ ライブラリに関するオンラインドキュメントを参照してください。

ファイルストリームの使い方

TFileStream は，アプリケーションがディスク上のファイルを読み書きできるようにするクラスで，ファイルストリームの高レベルなオブジェクトを表すために使用されます。TFileStream はストリームオブジェクトなので，共通のストリームメソッドをいくつか持っています。これらのメソッドを使うと，ファイルの読み書き，他のストリームクラスとの間のデータコピー，およびコンポーネント値の読み書きができます。ファイルストリームがストリームクラスから継承する機能についての詳細は，4-2 ページの「ストリームの使用」を参照してください。

加えて，ファイルストリームを使ってファイルハンドルにアクセスできるので，ファイルハンドルを必要とするグローバルなファイル処理ルーチンも使用できます。

ファイルストリームを使ってファイルを作成またはオープンする

ファイルを作成するかオープンしてそのファイルのハンドルにアクセスするには，TFileStream のインスタンスを作成します。インスタンスを作成すると，指定したファイルが開くか作成されて，そのファイルについての読み出しと書き込みができるようになります。ファイルが開けない場合には，TFileStream コンストラクタが例外を生成します。

```
__fastcall TFileStream(const AnsiString FileName, Word Mode);
```

Mode パラメータには，ファイルストリーム作成時のファイルのオープンモードを指定します。Mode パラメータは，オープンモードと共有モードを表す識別子を or で結んで指定します。オープンモードには次のいずれかの値を指定します。

表 4.1　オープンモード

値	説明
fmCreate	指定した名前でファイルを作成する。指定した名前のファイルがある場合は，fmOpenWrite と同じ
fmOpenRead	読み出し専用でファイルを開く
fmOpenWrite	書き込み専用でファイルを開く。現在の内容はすべて破棄される
fmOpenReadWrite	ファイルを開き，内容に変更を加える。現在の内容は維持される

共有モードには，下記の制約を条件として次のいずれかの値を指定できます。

表 4.2　共有モード

値	説明
fmShareCompat	共有モードは FCB を使って開く方法と互換性がある
fmShareExclusive	他のアプリケーションはファイルを開くことができない
fmShareDenyWrite	他のアプリケーションは読み出し用にファイルを開くことはできるが，書き込み用に開くことはできない
fmShareDenyRead	他のアプリケーションは書き込み用にファイルを開くことはできるが，読み出し用に開くことはできない
fmShareDenyNone	他のアプリケーションからのファイルの読み出しおよび書き込みを妨げる処理は行われない

どの共有モードを使用できるかは，使用するオープンモードによって決まります。次の表に，共有モードの使用可 / 不可をオープンモード別に示します。

表 4.3　共有モードの使用可 / 不可（オープンモード別）

オープンモード	fmShareCompat	fmShareExclusive	fmShareDenyWrite	fmShareDenyRead	fmShareDenyNone
fmOpenRead	使用できない	使用できない	使用できる	使用できない	使用できる
fmOpenWrite	使用できる	使用できる	使用できない	使用できる	使用できる
fmOpenReadWrite	使用できる	使用できる	使用できる	使用できる	使用できる

ファイルのオープンモードと共有モードの定数は SysUtils ユニットで定義されています。

ファイルハンドルの使い方

TFileStream のインスタンスを作成すると，ファイルハンドルへアクセスできるようになります。ファイルハンドルは，Handle プロパティに含まれます。Windows では，Handle プロパティは Windows のファイルハンドルです。Linux バージョンの CLX では，Handle プロパティは Linux のファイルハンドルです。Handle プロパティは読み出し専用で，ファイルが開いたときのモードを表します。ファイルハンドルの属性を変更したい場合は，新しいファイルストリームオブジェクトを作成する必要があります。

ファイル操作ルーチンの中には，パラメータとしてファイルハンドルをとるものもあります。一度ファイルストリームを得た後は，ファイルハンドルを使用するような状況ではいつでも Handle プロパティを使用できます。ファイルストリームは，ハンドルストリームとは違って，オブジェクトが破棄されたときにファイルハンドルを閉じることに注意してください。

ファイルの操作

BaseCLX のランタイムライブラリには，いくつか共通のファイル操作が組み込まれています。ファイルを操作する手続きや関数は上位レベルで動作します。大部分のルーチンでは，ファイル名を指定すると，ルーチンがオペレーティングシステムに対して必要な呼び出しを行います。そのかわりに，ファイルハンドルを使用する場合もあります。

注意 Object Pascal は大文字小文字を区別しませんが，Linux は区別します。クロスプラットフォームアプリケーションでファイルを操作するときは，大文字小文字に注意してください。

ファイルの削除

削除したファイルはディスクから消去され，またディスクのディレクトリからも除去されます。一般には削除したファイルは復元できないので，通常，アプリケーションではファイル削除の前にユーザーに確認します。ファイルを削除するには，ファイルの名前を DeleteFile 関数に渡します。

```
DeleteFile(FileName);
```

DeleteFile は，ファイルを削除すると **true** を返します。ファイルが存在しないか，または読み出し専用であるためにファイルを削除しなかった場合は **false** を返します。DeleteFile は，FileName で指定されたファイルをディスクから削除します。

ファイルの検索

ファイル検索用には，3 つのルーチン FindFirst，FindNext，および FindClose があります。FindFirst は，指定されたディレクトリ内で，与えられた属性を持つファイル名の最初のインスタンスを検索します。FindNext は，FindFirst を呼び出したときに指定した名前と属性に一致する次のエントリを返します。FindClose は FindFirst によって割り当てられたメモリを解放します。常に FindClose を使用して FindFirst/FindNext シーケンスを終了する必要があります。ファイルが存在するかどうか知りたい場合には，FileExists 関数を使用できます。この関数は，ファイルが存在する場合には **true** を返し，存在しない場合には **false** を返します。

3 つのファイル検索ルーチンは，パラメータの 1 つとして TSearchRec をとります。TSearchRec は，FindFirst や FindNext が検索するファイル情報を定義します。TSearchRec の宣言の例を次に示します。

```
struct TSearchRec
{
   int Time; // ファイルのタイムスタンプ
   int Size; // ファイルのサイズ（バイト数）
   int Attr; // ファイルの属性フラグ
   AnsiString Name; // ファイル名と拡張子
   int ExcludeAttr; // 無視するファイルの属性フラグ
   unsigned FindHandle;
   _WIN32_FIND_DATAA FindData; // 追加情報を持つ構造体
} ;
```

ファイルが見つかると，TSearchRec 型パラメータのフィールドが変更されて見つかったファイルを示します。以下の属性定数または値に対して Attr をチェックして，ファイルが特定な属性を持っているかどうかを判別できます。

表 4.4　属性定数と値

定数	値	説明
faReadOnly	0x00000001	読み出し専用ファイル
faHidden	0x00000002	非表示ファイル
faSysFile	0x00000004	システムファイル
faVolumeID	0x00000008	ボリューム ID ファイル
faDirectory	0x00000010	ディレクトリファイル
faArchive	0x00000020	アーカイブファイル
faAnyFile	0x0000003F	任意のファイル

属性をテストするには，Attr フィールドの値と属性定数を **&** 演算子で結合します。ファイルがその属性を持つ場合，結果 は 0 より大きくなります。たとえば，見つかったファイルが非表示ファイルの場合，式 (SearchRec.Attr & faHidden > 0) の評価は true になります。定数または値を OR で結ぶことにより，属性を複数指定できます。たとえば，通常ファイル以外に読み出し専用ファイルと非表示ファイルを検索するには，Attr パラメータに（faReadOnly|faHidden）を渡します。

例　この例では，フォーム上のラベル，［Search］ボタン，［Again］ボタンを使用します。ユーザーが［Search］ボタンをクリックすると，指定されたパス上の最初のファイルが検索され，ファイル名とバイト数がラベルの見出しに表示されます。ユーザーが［Again］ボタンを押すたびに，次に一致するファイル名とサイズがラベルに表示されます。

```
  TSearchRec SearchRec; // グローバル変数
void __fastcall TForm1::SearchClick(TObject *Sender)
{
  FindFirst("c:\\Program Files\\Cbuilder6\\bin\\*.*", faAnyFile, SearchRec);
  Label1->Caption = SearchRec->Name + " is " + IntToStr(SearchRec.Size) + " bytes in size";
}
void __fastcall TForm1::AgainClick(TObject *Sender)
{
  if (FindNext(SearchRec) == 0)
    Label1->Caption = SearchRec->Name + " is " + IntToStr(SearchRec.Size) + " bytes in size";
  else
    FindClose(SearchRec);
}
```

メモ クロスプラットフォームアプリケーションでは，ハードコードされたパス名は，システムに合った正しいパス名に置き換えるか，環境変数を使ってパス名を表す必要があります（[ツール｜環境オプション］を選択し，［環境変数］ページで設定する）。

ファイル名の変更

ファイル名の変更は，RenameFile 関数を使用するだけで行えます。

```
extern PACKAGE bool __fastcall RenameFile(const AnsiString OldName,
  const AnsiString NewName);
```

RenameFile 関数は，OldFileName で指定されたファイル名を NewFileName で指定された名前に変更します。ファイル名の変更が成功すると，RenameFile は true を返します。ファイル名の変更ができない場合（NewFileName というファイルがすでに存在していた場合など）には，RenameFile は false を返します。次に例を示します。

```
if (!RenameFile("OLDNAME.TXT","NEWNAME.TXT"))
  ErrorMsg("Error renaming file!");
```

RenameFile を使用して異なるディレクトリ間でのファイル名の変更（ファイルの移動）をすることはできません。ファイルをコピーしてから，古いファイルを削除する必要があります。

メモ 異なるドライブ間での RenameFile 関数が失敗するのは，BaseCLX ランタイムライブラリの RenameFile 関数が Windows API の MoveFile 関数のラッパーであるからです。

ファイルの日付ルーチン

FileAge，FileGetDate，および FileSetDate ルーチンはオペレーティングシステムの日付値を操作します。FileAge は，ファイルのタイムスタンプを返します。ファイルが存在しない場合は -1 を返します。FileSetDate は，指定されたファイルのタイムスタンプを設定します。成功した場合は 0 を返します。設定に失敗した場合は，エラーコードを返します。FileGetDate は指定されたファイルのタイムスタンプを返します。ハンドルが無効の場合は -1 を返します。

ほとんどのファイル操作ルーチンと同様に，FileAge は文字列のファイル名を使用します。しかし，FileGetDate と FileSetDate は，整数のパラメータを使ってファイルハンドルをとります。ファイルハンドルを取得するには，次の 2 通りの方法があります。

- FileCreate 関数を使ってファイルを新規作成するか，FileOpen 関数を使って既存ファイルを開く。FileCreate も FileOpen も，ファイルハンドルを返す
- TFileStream のインスタンスを作成し，ファイルを作成するか開く。その後，その Handle プロパティを使用する。詳細は，4-5 ページの「ファイルストリームの使い方」を参照

ファイルのコピー

BaseCLX ランタイムライブラリには，ファイルのコピー用のルーチンはありません。しかし，Windows 専用のアプリケーションを作成する場合は，Windows API の CopyFile 関数を直接呼び出してファイルをコピーすることができます。ランタイムライブラリのほとんどのルーチンと同様に，CopyFile はパラメータとしてファイルハンドルではなくファイル名をとります。ファイルをコピーする場合には，既存のファイルから新規ファイルにファイル属性はコピーされても，セキュリティ属性はコピーされないことに注意してください。CopyFile は，ドライブにまたがってファイルを移動する際にも有効です。RenameFile 関数も Windows API の MoveFile 関数も，ドライブをまたがってファイ

ル名を変更したりファイルを移動することはできません。詳細については，Microsoft Windows のオンラインヘルプを参照してください。

ini ファイルとシステムレジストリの操作

多くのアプリケーションが ini ファイルを使って環境設定情報を保存しています。BaseCLX には，ini ファイルを操作するクラスが 2 つあります（TIniFile と TMemIniFile）。ini ファイルを使う利点は，クロスプラットフォームアプリケーションで使用できることと，簡単に読み出しと編集ができることです。TIniFile と TMemIniFile の詳細は，4-10 ページの「TIniFile と TMemIniFile の使い方」を参照してください。

Windows アプリケーションの多くは，ini ファイルの代わりにシステムレジストリを使用します。Windows システムレジストリは階層状のデータベースで，環境設定情報を一元的に保存するスペースになっています。VCL には，システムレジストリを操作するクラスがいくつかあります。これらのクラスは，厳密には BaseCLX の一部ではありません（Windows でしか使用できないため）が，TRegistryIniFile クラスと TRegistry クラスについては，ini ファイルを操作するクラスと類似性があるため，ここで取り上げることにします。

TRegistryIniFile は，クロスプラットフォームアプリケーションで使うと便利です。その理由は，ini ファイルを操作するクラスと同じ上位クラス（TCustomIniFile）を持つからです。共通の上位クラス（TCustomIniFile）のメソッドだけに制限すれば，最小限の条件コードで ini ファイルとレジストリの両方を使用できます。TRegistryIniFile については，4-11 ページの「TRegistryIniFile の使い方」で説明しています。

クロスプラットフォームでないアプリケーションの場合は，TRegistry クラスを使用できます。TRegistry は ini ファイル用のクラスと互換性を持つ必要がないため，TRegistry のプロパティ名とメソッド名はシステムレジストリの構成に直接的に対応しています。TRegistry については，4-12 ページの「TRegistry の使い方」で説明しています。

TIniFile と TMemIniFile の使い方

ini ファイルフォーマットは今も広く使われており，多くの環境設定ファイル（DSK デスクトップ設定ファイルなど）も ini ファイルフォーマットです。クロスプラットフォームアプリケーションの場合，必ずしもシステムレジストリを使って環境設定情報を保存できるとは限りません。このため，ini ファイルフォーマットは特にクロスプラットフォームアプリケーションで使うと便利です。BaseCLX には，ini ファイルの読み書きをとても簡単にするクラスが 2 つあります。TIniFile と TMemIniFile です。

Linux では，TMemIniFile と TIniFile は同一です。Windows では，TIniFile がディスク上の ini ファイルを直接操作するのに対し，TMemIniFile は変更内容をバッファリングし，UpdateFile メソッドが呼び出されてからディスクに書き込みます。

TIniFile オブジェクトか TMemIniFile オブジェクトをインスタンス化するときに，パラメータとして ini ファイルの名前をコンストラクタに渡します。その名前のファイルが存在しなければ，自動的に作成されます。その後は，ReadString，ReadDate，ReadInteger，ReadBool などの読み出しメソッドを使って値を読み出せます。別の方法として ReadSection メソッドを使えば，ini ファイルのセクション

全体を読み出すことができます。同様に，WriteBool，WriteInteger，WriteString などのメソッドを使って値を書き込みます。

次に示す例は，フォームのコンストラクタで ini ファイルの環境設定情報を読み込み，OnClose イベントハンドラ内で値を書き込んでいます。

```
__fastcall TForm1::TForm1(TComponent *Owner) : TForm(Owner)
{
   TIniFile *ini;
   ini = new TIniFile( ChangeFileExt( Application->ExeName, ".INI" ) );

   Top     =  ini->ReadInteger( "Form", "Top", 100 );
   Left    =  ini->ReadInteger( "Form", "Left", 100 );
   Caption =  ini->ReadString( "Form", "Caption",
                               "Default Caption" );
   ini->ReadBool( "Form", "InitMax", false ) ?
         WindowState = wsMaximized :
         WindowState = wsNormal;

   delete ini;
}

void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
   TIniFile *ini;
   ini = new TIniFile(ChangeFileExt( Application->ExeName, ".INI" ) );
   ini->WriteInteger( "Form", "Top", Top );
   ini->WriteInteger( "Form", "Left", Left );
   ini->WriteString ( "Form", "Caption", Caption );
   ini->WriteBool   ( "Form", "InitMax",
                      WindowState == wsMaximized );

   delete ini;
}
```

どの Read ルーチンも 3 つのパラメータを取ります。1 番目のパラメータは，ini ファイルのセクションを識別します。2 番目のパラメータは，読み出す値を識別します。3 番目のパラメータは，指定したセクションまたは値が ini ファイルに存在しない場合のデフォルト値です。セクションや値が存在しない場合でも Read ルーチンが的確に処理するのと同様に，Write ルーチンもセクション / 値が存在しなければ作成します。上に示したコード例の場合，アプリケーションを最初に実行したときに次のような ini ファイルが作成されます。

```
[Form]
Top=185
Left=280
Caption=Default Caption
InitMax=0
```

2 回目以降の起動では，フォーム作成時に ini ファイルの値が読み出され，OnClose イベント中に再び書き込まれます。

TRegistryIniFile の使い方

多くの 32 ビット Windows アプリケーションは，各自の情報を（ini ファイルでなく）システムレジストリに保存しています。その理由は，レジストリは階層構造を成し，ini ファイルと違ってサイズ

に制限がないからです。プログラマが ini ファイルを熟知していて，ini ファイルの環境設定情報をレジストリに移したい場合には，TRegistryIniFile クラスを使うと便利です。クロスプラットフォームアプリケーションを作成する場合，Windows ではシステムレジストリを使用し，Linux では ini ファイルを使用したいときも，TRegistryIniFile を使います。たいていのアプリケーションは，TCustomIniFile 型を使うように作成できます。コードに条件を設定して TRegistryIniFile（Windows）または TMemIniFile（Linux）のインスタンスを作成し，それをアプリケーションの TCustomIniFile に代入するだけです。

TRegistryIniFile を使うと，レジストリのエントリが ini ファイルに似たエントリになります。TIniFile と TMemIniFile のメソッド（読み書き）はすべて，TRegistryIniFile にもあります。

TRegistryIniFile のオブジェクトを作成した場合，コンストラクタに渡したパラメータ（IniFile オブジェクトまたは TMemIniFile オブジェクトのファイル名に対応）は，レジストリのユーザーキー下のキー値になります。このルートを起点にしてセクションと値が分岐していきます。TRegistryIniFile はレジストリインターフェースを大幅に簡略化します。既存のコードを移植する必要がない場合や，クロスプラットフォームアプリケーションを作成しない場合でも，TRegistry のかわりに TRegistryIniFile を使用できます。

TRegistry の使い方

Windows 専用のアプリケーションを作成する場合，システムレジストリの構造に精通していれば，TRegistry を使用できます。TRegistryIniFile は ini ファイルコンポーネントと同じプロパティとメソッドを使いますが，TRegistry はこれと異なり，システムレジストリの構造と直接的に対応するプロパティとメソッドを使います。たとえば，TRegistry はルートキーとサブキーの両方を指定でき，また，デフォルトでは HKEY_CURRENT_USER というルートキーを想定しています。TRegistry のメソッドを使うと，キーのオープン，クローズ，保存，移動，コピー，削除のほか，アクセスレベルの設定も行えます。

メモ TRegistry は，クロスプラットフォームプログラミングでは使用できません。

レジストリエントリから値を取り出す例を以下に示します。

```
#include <Registry.hpp>

AnsiString GetRegistryValue(AnsiString KeyName)
{
  AnsiString S;
  TRegistry *Registry = new TRegistry(KEY_READ);
  try
  {
    Registry->RootKey = HKEY_LOCAL_MACHINE;
    // 存在しない場合でも新規作成しないので，false
    Registry->OpenKey(KeyName,false);
    S = Registry->ReadString("VALUE1");
  }
  __finally
  {
    delete Registry;
  }
  return S;
}
```

リストの操作

BaseCLX には，リスト（項目の集まり）を表すクラスが多数あります。格納する項目の種類，サポートする操作，および持続性の有無によって使用するクラスが異なります。

下の表に，各種のリストクラスと項目の種類を示します。

表 4.5　リストを管理するクラス

オブジェクト	管理対象
TList	ポインタのリスト
TThreadList	ポインタのスレッドセーフリスト
TBucketList	ポインタのハッシュリスト
TObjectBucketList	オブジェクトインスタンスのハッシュリスト
TObjectList	メモリ管理されるオブジェクトインスタンスのリスト
TComponentList	メモリ管理されるコンポーネント（TComponent から派生したクラスのインスタンス）のリスト
TClassList	クラス参照（メタクラス）のリスト
TInterfaceList	インターフェースポインタのリスト
TQueue	ポインタの先入れ先出しリスト
TStack	ポインタの後入れ先出しリスト
TObjectQueue	オブジェクトの先入れ先出しリスト
TObjectStack	オブジェクトの後入れ先出しリスト
TCollection	型付き項目を扱う多くの特殊クラスの基本クラス
TStringList	文字列のリスト
THashedStringList	Name=Value という形の文字列のリスト（ハッシュ化により効率を高める）

共通のリスト操作

リストクラスはさまざまな種類があり，格納される項目の型や派生元クラスも多種多様ですが，たいていのリストクラスは，共通のメソッドを使ってリスト内の項目の追加，削除，並べ替え，アクセスを行います。

リスト項目を追加する

ほとんどのリストクラスは Add メソッドを持っています。Add メソッドを使うと，リストの末尾（ソートされていない場合）または適切な位置（ソートされている場合）に項目を追加できます。Add メソッドは通常，リストに追加する項目をパラメータとしてとり，リスト内の項目の追加位置を返します。バケットリスト（TBucketList と TObjectBucketList）の場合は，追加する項目だけでなく，その項目と関連付けのできるデータもパラメータとしてとります。コレクションの場合は，パラメータをとらずに，追加する新しい項目を作成します。コレクションに対して Add メソッドを使うと，追加した項目が返されます。このため，新しい項目のプロパティに値を割り当てることができます。

一部のリストクラスは，Add メソッドに加えて Insert メソッドも持っています。Insert メソッドの働きは Add メソッドに似ていますが，Add メソッドにないパラメータが 1 つあります。このパラメータは，リスト内の新規項目の追加位置を指定します。Add メソッドを持つクラスであれば，項目位置

があらかじめ決定されている場合を除き，Insert メソッドも持っています。たとえばソートされているリストでは，項目はソート順に従わなければならないため，Insert メソッドは使えません。バケットリストの場合も，ハッシュアルゴリズムが項目位置を決定するため Insert メソッドは使えません。

Add メソッドを持たないクラスとは，番号付きのリストです。番号付きリストはキュー構造かスタック構造を成します。番号付きリストに項目を追加するには，Add メソッドのかわりに Push メソッドを使います。Add メソッドと同様に，Push メソッドも項目をパラメータとしてとり，適切な位置に項目を挿入します。

リスト項目を削除する

単一のリストクラスから項目を 1 つ削除するには，Delete メソッドか Remove メソッドを使います。Delete はパラメータを 1 つとります。このパラメータは，削除する項目のインデックスです。Remove もパラメータを 1 つとりますが，このパラメータはインデックスではなく，削除する項目への参照です。Delete メソッドのみサポートするリストクラスもあれば，Remove メソッドのみサポートするリストクラスもあります。両方をサポートするリストクラスもあります。

項目の追加と同様に，削除のときも番号付きリストだけは違った動作をします。番号付きリストから項目を削除するには，Delete メソッドあるいは Remove メソッドでなく，Pop メソッドを呼び出します。削除できる項目は 1 つだけなので，Pop メソッドは引数をとりません。

リスト内のすべての項目を削除するには，Clear メソッドを呼び出します。番号付きリストを除くどのリストも Clear を使用できます。

リスト項目にアクセスする

TThreadList および番号付きリスト以外のリストクラスはすべて，リスト内の項目にアクセスさせるプロパティを持っています。一般的には，このプロパティの名前は Items ですが，文字列リストでは，このプロパティ名は Strings になります。また，バケットリストの場合は Data プロパティと呼ばれます。Items プロパティ，Strings プロパティ，および Data プロパティはインデックス付きプロパティなので，どの項目にアクセスするかを指定できます。

TThreadList では，項目にアクセスする前にリストをロックしなければなりません。リストをロックすると，LockList メソッドが TList オブジェクトを返します。この TList オブジェクトを使って項目にアクセスできます。

番号付きリストの場合，リストの「トップ」の項目にしかアクセスできません。この項目への参照を取得するには，Peek メソッドを呼び出します。

リスト項目を並べ替える

一部のリストクラスには，リスト内の項目を並べ替えるメソッドがあります。Exchange メソッドを持つリストクラスがいくつかあります。Exchange は，2 つの項目の位置を入れ換えます。Move メソッドを持つリストクラスもあります。Move は，ある項目を指定位置に移動します。Sort メソッドを持つリストクラスもあります。Sort は，リスト内の項目をソートします。

どのメソッドが使えるかを確認するには，使用しているリストクラスに関するオンラインヘルプを参照してください。

持続性を持つリスト

リストに持続性があれば，フォームファイルに保存できます。このため，持続性を持つリストは，コンポーネントのパブリッシュプロパティの型としてよく使われます。設計時にリストに項目を追加して，その項目をオブジェクトと一緒に保存できます。このため，実行時にコンポーネントがメモリにロードされると，項目も読み込まれます。持続性のあるリストは主に 2 種類あります。文字列リストとコレクションです。

文字列リストは TStringList，THashedStringList などがあります。文字列リストは，その名が示すとおり文字列を格納します。Name=Value という形の文字列に対して特殊なサポートを提供するので，名前に関連付けられた値で検索することもできます。加えて，たいていの文字列リストでは，リスト内の各文字列とオブジェクトとを関連付けることができます。文字列リストについての詳細は，4-15 ページの「文字列リストの操作」を参照してください。

コレクションは，TCollection クラスから派生します。TCollection の各下位オブジェクトは，TCollectionItem から派生する個々の項目クラスを専門的に管理します。コレクションは，共通のリスト操作の多くをサポートします。どのコレクションもパブリッシュプロパティの型になるように設計されています。また，多くのコレクションは，コレクションを使ってプロパティのどれかを実装しているオブジェクトと無関係に機能することができません。値をコレクションとして持つプロパティなら，設計時にコレクションエディタを使って項目の追加，削除，再配置ができます。コレクションエディタを使うと，一般的なユーザーインターフェースを介してコレクションを操作できます。

文字列リストの操作

頻繁に使われるリストの 1 つに，文字列リストがあります。たとえば，コンボボックスの項目，メモのテキスト行，フォント名，文字列グリッドの行と列の名前などを管理する必要が生じます。BaseCLX は，TStrings オブジェクトとその下位オブジェクト（TStringList，THashedStringList など）を通して，あらゆる文字列リストに共通のインターフェースを提供します。TStringList は，TStrings により導入された抽象プロパティおよび抽象メソッドを実装し，以下の操作を行うプロパティ，イベント，メソッドを導入します。

- リスト内の文字列をソートする
- ソート後のリストに文字列の重複が生じるのを禁止する
- リスト内容の変更に応答する

文字列リストの管理機能に加え，TStrings と TStringList により簡単に文字列を交換できます。たとえば，メモのテキスト行（TStrings の下位オブジェクト）を編集した後，そのテキスト行をコンボボックスの項目（これも TStrings の下位オブジェクト）として使用することができます。

オブジェクトインスペクタの値列に (TStrings) と表示されているプロパティは，文字列リストです。TStrings をダブルクリックすると文字列リストエディタが開き，このエディタで行の編集，追加，削除が行えます。

文字列リストは実行時にも操作できます。一般に，以下のような処理で文字列リストを扱います。

- 文字列リストの読み込みと保存

- 新しい文字列リストの作成
- リスト内の文字列を操作する
- オブジェクトと文字列リストを関連付ける

文字列リストの読み込みと保存

テキストファイルに文字列を保存するには，文字列リストオブジェクトの提供する SaveToFile メソッドを使います。同様に，テキストファイルを文字列に読み込むには LoadFromFile メソッドを使います。テキストファイルの各行がリストの各文字列に対応します。SaveToFile メソッドと LoadFromFile メソッドを使うと，メモコンポーネントにファイルを読み込んで簡単なテキストエディタを作成したり，項目のリストをコンボボックスに保存したりできます。

以下の例は，WIN.INI ファイルを Memo コンポーネントに読み込み，WIN.BAK というバックアップコピーを作成しています。

```
void __fastcall EditWinIni()
{
  AnsiString FileName = "C:\\WINDOWS\\WIN.INI";   // ファイル名を設定する
  Form1->Memo1->Lines->LoadFromFile(FileName);      // ファイルから読み込む
  Form1->Memo1->Lines->SaveToFile(ChangeFileExt(FileName, ".BAK")); // バックアップファイル
                                                                    // に保存する
}
```

新しい文字列リストの作成

通常，文字列リストはコンポーネントの一部です。ただし，参照テーブルの文字列を保存する場合など，独立した文字列リストを作成すると便利な場合があります。文字列リストの作成と管理の方法は，文字列リストが短期リストか（単一のルーチンで作成，使用，破棄する）長期リストか（アプリケーションが終了するまで有効）によって異なります。どちらの文字列リストを作成するにしても，使わなくなる時点でリストを解放するように設計しなければなりません。

短期文字列リスト

単一ルーチンの存続期間の中でのみ文字列リストを使用する場合は，作成，使用，破棄のすべてを 1 つの場所で行えます。これは，文字列リストを操作するもっとも安全な方法です。メモリは文字列リスト自体と各文字列に割り当てられます。したがって，例外が生じた場合でもメモリが解放されるよう，**try...__finally** ブロックで保護する必要があります。

1. 文字列リストオブジェクトを作成します。
2. **try..__finally** ブロックの **try** 部で，文字列リストを使用します。
3. **__finally** 部で文字列リストオブジェクトを解放します。

次の例に示すイベントハンドラは，ボタンクリックに応答して文字列リストを作成して使用してから，文字列リストを破棄しています。

```
void __fastcall TForm1::ButtonClick1(TObject *Sender)
{
   TStringList *TempList = new TStringList; // リストを宣言する
   try{
      // 文字列リストを使用する
```

```
    }
    __finally{
      delete TempList; // リストオブジェクトを破棄する
    }
}
```

長期文字列リスト

アプリケーションの実行中に文字列リストをいつでも利用できるようにするには，アプリケーションの起動時に文字列リストを作成し，アプリケーションの終了前に破棄します。

1. アプリケーションのメインフォーム用のユニットファイルで，フォームの宣言部に TStrings 型の項目を追加します。
2. メインフォームのコンストラクタ（フォームが表示される前に実行されるコンストラクタ）を作成します。これにより文字列リストが作成され，手順 1 で宣言したフィールドに文字列リストが代入されます。
3. フォームの OnClose イベント用に，文字列リストを解放するイベントハンドラを作成します。

以下に示す例では，長期文字列リストを使ってユーザーのメインフォーム上のマウスクリックを記録し，アプリケーション終了前にリストをファイルに保存しています。

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
   ClickList = new TStringList;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
   ClickList->SaveToFile(ChangeFileExt(Application->ExeName, ".LOG")); // リストを保存する
   delete ClickList;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
      TShiftState Shift, int X, int Y)
{
   TVarRec v[] = {X,Y};
   ClickList->Add(Format("Click at (%d, %d)",v,ARRAYSIZE(v) - 1)); // リストへ文字列を追加
}
```

リスト内の文字列を操作する

文字列リストに対しては，以下のような操作が行われます。

- リスト内の文字列を数える
- 特定の文字列にアクセスする
- リスト内の文字列の位置を検索する
- リスト内の文字列を繰り返し処理する
- リストに文字列を追加する
- リスト内で文字列を移動する
- リストから文字列を削除する
- 文字列リスト全体をコピーする

リスト内の文字列を数える

Count プロパティは，リスト内の文字列の数を返す読み出し専用のプロパティです。文字列リストはゼロから始まるインデックスを使用するため，Count プロパティは最終文字列のインデックスより 1 つ多い値になります。

特定の文字列にアクセスする

Strings という配列プロパティは，リスト内の文字列を保持します。この文字列は 0 ベースのインデックスで参照されます。

```
StringList1->Strings[0] = "This is the first string.";
```

文字列リスト内の項目を検索する

文字列リスト内で文字列を探すには，IndexOf メソッドを使います。IndexOf メソッドは文字列をパラメータとして受け取り，リスト内で一致する最初の文字列のインデックスを返します。パラメータ文字列が見つからなければ -1 を返します。IndexOf メソッドは完全一致のみ探します。部分文字列の一致を検索するには，自分で文字列リストを繰り返し処理する必要があります。

以下の例のように，Index メソッドを使用して，リストボックスの項目に特定のファイル名があるかどうか調べることもできます。

```
if (FileListBox1->Items->IndexOf("WIN.INI") > -1) ...
```

リスト内の文字列を繰り返し処理する

リスト内の文字列を繰り返し処理するには，**for** ループを使ってゼロから Count - 1 まで実行します。

次の例は，リスト内の各文字列を大文字に変換しています。

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  for (int i = 0; i < ListBox1->Items->Count; i++)
    ListBox1->Items->Strings[i] = UpperCase(ListBox1->Items->Strings[i]);
}
```

リストに文字列を追加する

文字列リストの末尾に文字列を追加するには，Add メソッドを呼び出して新しい文字列をパラメータとして渡します。リストに文字列を挿入するには，Insert メソッドを呼び出し，2 つのパラメータ（文字列と，挿入先位置のインデックス）をパラメータとして渡します。たとえば文字列「Three」をリストの 3 番目の文字列にしたい場合には，次のコードを使用できます。

```
StringList1->Insert(2, "Three");
```

リストの末尾に別のリストの文字列を追加するには，AddStrings メソッドを呼び出します。

```
StringList1->AddStrings(StringList2);  // StringList2 から StringList1 の末尾に文字列を追加
```

リスト内で文字列を移動する

文字列リスト内で文字列を移動するには，Move メソッドを呼び出して，2 つのパラメータ（文字列の移動前のインデックスと，移動後のインデックス）を渡します。たとえば，リストの 3 番目の文字列を 5 番目に移動する場合には，次のコードを使用できます。

```
StringListObject->Move(2, 4);
```

リストから文字列を削除する

文字列リストから文字列を削除するには，文字列リストの Delete メソッドを呼び出し，削除する文字列のインデックスを渡します。削除したい文字列のインデックスがわからない場合には，IndexOf メソッドを使って調べます。文字列リスト内のすべての文字列を削除するには，Clear メソッドを使用します。

次の例では，IndexOf メソッドと Delete メソッドを使って文字列を検索，削除しています。

```
int BIndex = ListBox1->Items->IndexOf("bureaucracy");
if (BIndex > -1)
  ListBox1->Items->Delete(BIndex);
```

文字列リスト全体をコピーする

ある文字列リストから別の文字列リストへ文字列をコピーするには，Assign メソッドを使います。この場合，コピー先リストにあった文字列は上書きされます。上書きせずにコピー先リストの末尾に追加するには，AddStrings メソッドを使います。下に例を示します。

```
Memo1->Lines->Assign(ComboBox1->Item)s; // 元の文字列を上書きする
```

この例では，コンボボックスからメモへとテキスト行をコピー（メモに上書き）しています。

```
Memo1->Lines->AddStrings(ComboBox1->Items);// 文字列を末尾に追加する
```

この例では，コンボボックスのテキスト行をメモの末尾に追加しています。

文字列リストのローカルコピーを作成するには，Assign メソッドを使います。次のようにローカル変数を代入すると，元の文字列リストオブジェクトが失われることになります。

```
StringList1 = StringList2;
```

この例のようにすると，予期しない結果が発生することがあります。

オブジェクトと文字列リストを関連付ける

文字列リストは，Strings プロパティに格納されている文字列だけでなく，Objects プロパティに格納されるオブジェクトへの参照も管理することができます。Objects プロパティは，Strings プロパティと同様に，ゼロから始まるインデックスを持つ配列です。一般に，オーナー描画コントロール用文字列とビットマップを関連付ける場合に Objects プロパティを使います。

AddObject メソッドか InsertObject メソッドを使うと，文字列および関連付けられたオブジェクトを一度に文字列リストに追加できます。IndexOfObject メソッドは，文字列リスト内で指定したオブジェクトと関連付けられた最初の文字列のインデックスを返します。Delete，Clear，Move などのメ

ソッドは，文字列とオブジェクトの両方を対象に動作します。たとえば，文字列を削除すると，その文字列に対応するオブジェクト（もしあれば）も削除されます。

オブジェクトを既存の文字列と関連付けるには，同じインデックス位置の Objects プロパティにオブジェクトを代入します。対応する文字列を追加しないとオブジェクトは追加できません。

文字列の処理

BaseCLX ランタイムライブラリは，文字列型に対応した文字列処理ルーチンを数多く備えています。これらのルーチンは，ワイド文字列，AnsiString，およびヌルで終わる文字列（**char** *）のためのルーチンです。ヌルで終わる文字列を処理するルーチンは，ヌルでの終了を使用して文字列長を決定します。以下のトピックでは，ランタイムライブラリにある文字列処理ルーチンの多くを取り上げ，その概要を説明します。.

ワイド文字のルーチン

ワイド文字はさまざまな状況で使われます。XML などの技術では，ワイド文字をネイティブの型として使っています。複数のターゲットロケールを持つアプリケーションには文字列処理の問題がありますが，ワイド文字を使うと，そうした問題の一部を簡略化することもできます。1 バイト文字だけの文字列の処理では一般的であった仮定の多くは MBCS システムには適用できませんが，ワイド文字エンコード方式を採用することで，それらが同じように適用できるという利点があります。文字列のバイト数と文字列の文字数の関係も一定したものになります。文字を分断してしまう心配もなく，文字の第 2 バイトを別の文字と間違える心配もありません。

ワイド文字を扱う問題点の 1 つは，大部分の VCL コントロールが文字列値を 1 バイトまたは MBCS 文字列で表すことです（通常，CLX のコントロールはワイド文字を使用する）。文字列プロパティの設定や値の読み出しごとにワイド文字システムと MBCS システム間の翻訳を行うと，膨大なコードの記述が新たに必要になることがあり，その場合はアプリケーションの実行速度が低下します。しかし，文字と WideChar の 1 対 1 のマッピングの利用を必要とする一部の特殊な文字列処理アルゴリズムを使用する場合には，ワイド文字への翻訳が必要なこともあります。

以下の関数は，標準の 1 バイト文字列（または MBCS 文字列）と Unicode 文字列の間で変換を実行します。

- StringToWideChar
- WideCharLenToString
- WideCharLenToStrVar
- WideCharToString
- WideCharToStrVar

加えて，以下の関数は，WideString と他の表現方法の間で翻訳を実行します。

- UCS4StringToWideString
- WideStringToUCS4String
- VarToWideStr
- VarToWideStrDef

以下のルーチンは，WideString に直接作用します。

- WideCompareStr
- WideCompareText
- WideSameStr
- WideSameText
- WideSameCaption（CLX のみ）
- WideFmtStr
- WideFormat
- WideLowerCase
- WideUpperCase

以下に挙げるルーチンは，ワイド文字を操作するためのオーバーロードを持ちます。

- UniqueString
- Length
- Trim
- TrimLeft
- TrimRight

一般に使用される AnsiString 処理ルーチン

AnsiString を処理するルーチンは幅広い機能を提供します。これらの機能の中には，目的は同一でも判定基準が異なる複数のルーチンがあります。以下の分野でルーチンを分類します。

- 比較
- 大文字小文字の変換
- 修飾
- 部分文字列

これらの表は，ルーチンが以下の基準を満足するかどうかを表示する欄も適宜用意しています。

- 大文字小文字の区別の使用：ロケール設定を使用している場合，ロケール設定が大文字小文字を定義します。ルーチンがロケール設定を使用しない場合には，文字そのものの値に基づいて判断されます。ルーチンが大文字小文字を同じものとみなす場合，ASCII コードとみなされます。
- ロケール設定の使用：ロケール設定を使用すると，特定のロケールに合わせてアプリケーションをカスタマイズできます（特に日本語環境の場合）。大部分のロケール設定は，小文字を対応する大文字よりも小さい位置に置いています。これは，小文字が大文字よりも大きな位置にいる ASCII の順序とは反対になっています。Windows のロケールを使用するルーチンは，一般的に Ansi で始まります（つまり，AnsiXXX となる）。
- マルチバイト文字セット（MBCS）のサポート：マルチバイト文字を表すには 1 バイト文字と 2 バイト文字が混用されます。したがって，バイト数は必ずしも文字数に対応しません。MBCS をサポートするルーチンは 1 バイト文字と 2 バイト文字を解析します。

ByteType と StrByteType は，特定のバイトが 2 バイト文字の先頭バイトかどうかを判別します。マルチバイト文字を使用する場合は，文字列を 2 バイト文字の途中で分断しないように注意してください。また 1 バイト文字か 2 バイト文字かの判断ができなくなるため，文字列の途中の 1 バイトを関数

や手続きのパラメータとして渡してはいけません。全体を参照できるように文字列自身，または文字列へのポインタを渡してください。MBCS についての詳細は，第 16 章「国際化対応アプリケーションの作成」の 16-2 ページの「コードを多国語対応にする」を参照してください。

表 4.6　文字列比較ルーチン

ルーチン	大文字小文字の区別	ロケール設定の使用	MBCS のサポート
AnsiCompareStr	はい	はい	はい
AnsiCompareText	いいえ	はい	はい
AnsiCompareFileName	いいえ	はい	はい
AnsiMatchStr	はい	はい	はい
AnsiMatchText	いいえ	はい	はい
AnsiContainsStr	はい	はい	はい
AnsiContainsText	いいえ	はい	はい
AnsiStartsStr	はい	はい	はい
AnsiStartsText	いいえ	はい	はい
AnsiEndsStr	はい	はい	はい
AnsiEndsText	いいえ	はい	はい
AnsiIndexStr	はい	はい	はい
AnsiIndexText	いいえ	はい	はい
CompareStr	はい	いいえ	いいえ
CompareText	いいえ	いいえ	いいえ
AnsiResemblesText	いいえ	いいえ	いいえ

表 4.7　大文字小文字の変換ルーチン

ルーチン	ロケール設定の使用	MBCS のサポート
AnsiLowerCase	はい	はい
AnsiLowerCaseFileName	はい	はい
AnsiUpperCaseFileName	はい	はい
AnsiUpperCase	はい	はい
LowerCase	いいえ	いいえ
UpperCase	いいえ	いいえ

メモ　ファイル名の文字列に使われるルーチン AnsiCompareFileName，AnsiLowerCaseFileName，および AnsiUpperCaseFileName はすべて，Windows ロケールを使用します。ファイル名に使用されるロケール（文字セット）はデフォルトのユーザーインターフェースとは異なることがあるため，移植可能なファイル名を常に使用する必要があります。

表 4.8　文字列修飾ルーチン

ルーチン	大文字小文字の区別	MBCS のサポート
AdjustLineBreaks	（意味を持たない）	はい
AnsiQuotedStr	（意味を持たない）	はい
AnsiReplaceStr	はい	はい
AnsiReplaceText	いいえ	はい
StringReplace	フラグで選択（オプション）	はい

表 4.8　文字列修飾ルーチン（つづき）

ルーチン	大文字小文字の区別	MBCS のサポート
ReverseString	（意味を持たない）	いいえ
StuffString	（意味を持たない）	いいえ
Trim	（意味を持たない）	はい
TrimLeft	（意味を持たない）	はい
TrimRight	（意味を持たない）	はい
WrapText	（意味を持たない）	はい

表 4.9　部分文字列のルーチン

ルーチン	大文字小文字の区別	MBCS のサポート
AnsiExtractQuotedStr	（意味を持たない）	はい
AnsiPos	はい	はい
IsDelimiter	はい	はい
IsPathDelimiter	はい	はい
LastDelimiter	はい	はい
LeftStr	（意味を持たない）	いいえ
RightStr	（意味を持たない）	いいえ
MidStr	（意味を持たない）	いいえ
QuotedStr	いいえ	いいえ

一般に使用されるヌル終端文字列処理ルーチン

ヌルで終わる文字列を処理するルーチンは幅広い機能を提供します。これらの機能の中には，目的は同一でも判定基準が異なる複数のルーチンがあります。下の表は，以下の機能分野別にルーチンを列記しています。

- 比較
- 大文字小文字の変換
- 修飾
- 部分文字列
- コピー

下の表は，ルーチンが大文字小文字を区別するか，現在のロケールを使用するか，マルチバイト文字セット（MBCS）をサポートするかを表示する欄も適宜用意しています。

表 4.10　ヌルで終わる文字列の比較ルーチン

ルーチン	大文字小文字の区別	ロケール設定の使用	MBCS のサポート
AnsiStrComp	はい	はい	はい
AnsiStrIComp	いいえ	はい	はい
AnsiStrLComp	はい	はい	はい
AnsiStrLIComp	いいえ	はい	はい
StrComp	はい	いいえ	いいえ
StrIComp	いいえ	いいえ	いいえ

表 4.10 ヌルで終わる文字列の比較ルーチン（つづき）

ルーチン	大文字小文字の区別	ロケール設定の使用	MBCS のサポート
StrLComp	はい	いいえ	いいえ
StrLIComp	いいえ	いいえ	いいえ

表 4.11 ヌルで終わる文字列の大文字小文字変換ルーチン

ルーチン	ロケール設定の使用	MBCS のサポート
AnsiStrLower	はい	はい
AnsiStrUpper	はい	はい
StrLower	いいえ	いいえ
StrUpper	いいえ	いいえ

表 4.12 文字列修飾ルーチン

ルーチン
StrCat
StrLCat

表 4.13 部分文字列のルーチン

ルーチン	大文字小文字の区別	MBCS のサポート
AnsiStrPos	はい	はい
AnsiStrScan	はい	はい
AnsiStrRScan	はい	はい
StrPos	はい	いいえ
StrScan	はい	いいえ
StrRScan	はい	いいえ

表 4.14 文字列コピールーチン

ルーチン
StrCopy
StrLCopy
StrECopy
StrMove
StrPCopy
StrPLCopy

印刷

厳密に言えば，TPrinter クラスは BaseCLX に属しません。その理由は，VCL の TPrinter（Printers ユニット），CLX の TPrinter（QPrinters ユニット）という 2 つのバージョンに分かれているからです。VCL の TPrinter は，Windows プリンタの詳細をカプセル化するオブジェクトです。CLX の TPrinter

は，プリンタにペイントするペイントデバイスです。TPrinter は，ポストスクリプトを生成して lpr, lp，または別の印刷コマンドへ送信します。しかし，どちらの TPrinter もきわめてよく似ています。

インストール済みの使用可能なプリンタの一覧を取得するには，Printers プロパティを使います。どちらのプリンタオブジェクトも TCanvas を使います（フォームの TCanvas と同じ）。このため，フォーム上に描画できるものであればプリンタにも印刷できます。イメージを印刷するには，BeginDoc メソッドを呼び出した後，印刷したいキャンバスグラフィック（TextOut メソッドによるテキスト表示を含む）を描画し，EndDoc メソッドを呼び出して印刷ジョブをプリンタに送ります。

フォーム上のボタンとメモを使った例を以下に示します。ユーザーがボタンをクリックすると，ページの周囲に 200 ピクセルの境界が確保され，メモの中身が印刷されます。

このコード例を確実に実行するには，ユニットファイルに <Printers.hpp> を含めます。

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  TPrinter *Prntr = Printer();
  TRect r = Rect(200,200,Prntr->PageWidth - 200,Prntr->PageHeight- 200);
  Prntr->BeginDoc();
  for( int i = 0; i < Memo1->Lines->Count; i++)
    Prntr->Canvas->TextOut(200,200 + (i *
  Prntr->Canvas->TextHeight(Memo1->Lines->Strings[i])),
  Memo1->Lines->Strings[i]);
  Prntr->Canvas->Brush->Color = clBlack;
  Prntr->Canvas->FrameRect(r);
  Prntr->EndDoc();
}
```

TPrinter オブジェクトの使い方に関する詳細は，オンラインヘルプで「TPrinter」を参照してください。

計量単位の変換

ConvUtils ユニットに，汎用関数 Convert が宣言されています。Convert 関数を使うと，計量単位を別の単位に変換することができます。変換は同じ種類の単位同士で行えます（フィートとインチ，日と週など）。測定対象が同じ種類であれば，その計量単位は「同じ変換ファミリー」に属するとみなされます。単位を変換するには，変換前と変換後の単位が同じ変換ファミリーに属していなければなりません。変換の実行に関する詳細は，次の節の変換の実行，およびオンラインヘルプで「変換」を参照してください。

StdConvs ユニットでは，いくつかの変換ファミリーと，同一ファミリー内の計量単位を定義しています。また，カスタムの変換ファミリーとその単位を自分で作成することもできます。これを行うには，RegisterConversionType 関数と RegisterConversionFamily 関数を使います。変換および変換単位の拡張については，計量単位の種類を追加すると，オンラインヘルプの「変換」を参照してください。

変換の実行

変換が単純か複雑かを問わず，変換を実行するには Convert 関数を使います。Convert 関数は単純な構文と 2 次構文で構成されます。2 次構文は，複雑な測定対象を変換する働きをします。

単純な変換を実行する

Convert 関数を使うと，計量単位を別の計量単位に変換できます。Convert 関数は，同じ測定対象（距離，面積，時間，温度など）を測定する単位同士の変換を行います。

Convert 関数で変換するには，変換前の単位と変換後の単位を指定する必要があります。計量単位を識別するには TConvType 型を使います。

下の例は，温度の単位を華氏からケルビン（絶対温度単位）に変換しています。

```
TempInKelvin = Convert(StrToFloat(Edit1->Text), tuFahrenheit, tuKelvin);
```

複雑な変換を実行する

計量単位の比率を変換するという複雑な変換も，Convert 関数を使います。たとえば，1 時間あたりのマイル数を 1 分あたりのメートル数に変換して速度を計算する，1 分あたりのガロンを 1 時間あたりのリットルに変換して流量を計算する，といったケースです。

1 ガロンあたりのマイル数を 1 リットルあたりのキロメートルに変換する呼び出しの例を示します。

```
double nKPL = Convert(StrToFloat(Edit1.Text), duMiles, vuGallons, duKilometers, vuLiter);
```

変換前の単位と変換後の単位は同じ変換ファミリーに属していなければなりません（つまり，測定対象が同じでなければならない）。双方の単位に互換性がないと，Convert 関数は EConversionError 例外を生成します。2 つの TConvType 値が同じ変換ファミリーに属しているかを確認するには，CompatibleConversionTypes を呼び出します。

StdConvs ユニットには，TConvType 値のいくつかのファミリーが定義されています。定義済みの計量単位ファミリーと各ファミリーの計量単位については，オンラインヘルプの「変換ファミリー変数」にあるリストを参照してください。

計量単位の種類を追加する

変換したい計量単位が StdConvs ユニットに定義されていない場合は，目的の計量単位を表す変換ファミリーを新規に作成する必要があります（TConvType の値を指定する）。2 つの TConvType 値を同じ変換ファミリーに登録します。これにより Convert 関数は，これらの TConvType 値が表す単位を使って計量単位を変換できるようになります。

まず，TConvFamily の値を取得します。TConvFamily 値を取得するには，RegisterConversionFamily 関数を使って新しい変換ファミリーを登録します。TConvFamily 値を取得したら（新しい変換ファミリーを登録するか，StdConvs ユニットにあるグローバル変数のどれかを使用する），RegisterConversionType 関数を使って新しい単位を変換ファミリーに追加します。以下で取り上げる例は，この手順を示しています。

単純な変換ファミリーを作成し，単位を追加する

変換ファミリーを新規作成して新しい計量単位を追加するケースとは，たとえば，変換結果が不正確になる可能性のある長期間の単位の間（月と世紀など）で変換する場合などです。

以下，「日」を基本単位とする cbTime ファミリーを使って詳しく解説します。基本単位とは，同じファミリー内の変換で必ず使用する単位です。つまり，この例では常に日換算で変換しなければなり

ません。しかし，月以上の単位（月，年，10 年，世紀，千年期など）を使って変換すると，間違いが発生することがあります。これは，日数から月数，日数から年数などは正確に変換できないためです。月によって長さが違うだけでなく，年数も閏年，閏秒などの補正要因があります。

月以上の単位だけを使う場合には，年を基本単位とするより正確な変換ファミリーを作成できます。この例では，cbLongTime という新しい変換ファミリーを作成しています。

変数を宣言する

最初に，識別子の変数を宣言する必要があります。新規作成する LongTime 変換ファミリーと，そのメンバーである計量単位の中で識別子が使われます。

```
tConvFamily cbLongTime;
TConvType ltMonths;
TConvType ltYears;
TConvType ltDecades;
TConvType ltCenturies;
TConvType ltMillennia;
```

変換ファミリーを登録する

次に，変換ファミリーを登録します。

```
cbLongTime = RegisterConversionFamily ("Long Times");
```

UnregisterConversionFamily 手続きも用意されていますが，変換ファミリーを定義しているユニットを実行時に削除するのでない限り，変換ファミリーを登録解除する必要はありません。アプリケーションが終了すると，変換ファミリーは自動的にクリーンアップされます。

計量単位を登録する

次に，新規作成した変換ファミリーの中で使用する計量単位を登録します。特定の変換ファミリー内の計量単位を登録するには，RegisterConversionType 関数を使います。基本単位（この例では年）を定義し，この基本単位との関連を示す係数を使って他の単位を定義します。LongTime ファミリーの場合，基本単位は年なので，ltMonths の係数は 1/12 になります。変換後の単位についても記述します。

計量単位を登録するコードの例を下に示します。

```
ltMonths = RegisterConversionType(cbLongTime,"Months",1/12);
ltYears = RegisterConversionType(cbLongTime,"Years",1);
ltDecades = RegisterConversionType(cbLongTime,"Decades",10);
ltCenturies = RegisterConversionType(cbLongTime,"Centuries",100);
ltMillennia = RegisterConversionType(cbLongTime,"Millennia",1000);
```

新しい単位を使用する

この段階で，新規登録した単位を使って変換を実行できます。前に変換ファミリー cbLongTime に登録した変換型であれば，グローバル関数 Convert で変換ができます。

したがって，以下の Convert 呼び出しを使うかわりに，

```
Convert(StrToFloat(Edit1->Text),tuMonths,tuMillennia);
```

以下のコードを使った方が正確になります。

```
Convert(StrToFloat(Edit1->Text),ltMonths,ltMillennia);
```

変換関数の使い方

複雑な変換を行う場合には，変換係数を使う方法でなく，新しい関数を指定する別の構文を使って変換を実行することができます。たとえば，変換係数を使っても温度の値は変換できません。これは，温度の目盛りによって基点が異なるからです。

新しい変換型を登録する例を下に示します。この例は StdConvs ユニットから翻訳したもので，基本単位との間で変換を行う関数をいくつか示しています。

変数を宣言する

最初に，識別子の変数を宣言します。cbTemperature 変換ファミリーと，そのメンバーである計量単位の中で識別子が使われます。

```
TConvFamily cbTemperature;
TConvType tuCelsius;
TConvType tuKelvin;
TConvType tuFahrenheit;
```

メモ　ここに列記した計量単位は，実際に StdConvs ユニットに登録されている温度単位の下位セットです。

変換ファミリーを登録する

次に，変換ファミリーを登録します。

```
cbTemperature = RegisterConversionFamily ("Temperature");
```

基本単位を登録する

次に，変換ファミリーの基本単位（この例では摂氏）を定義し，登録します。なお，基本単位については，実行すべき変換が存在しないため単純な変換係数を使用できます。

```
tuCelsius = RegisterConversionType(cbTemperature,"Celsius",1);
```

基本単位との間で変換するメソッドを記述する

温度単位を摂氏に変換するか，摂氏から別の単位に変換する場合には，単純な変換係数が使用できないので，自分でコードを書く必要があります。下に示す関数は StdConvs ユニットから翻訳したものです。

```
double __fastcall FahrenheitToCelsius(const double AValue)
{
  return (((AValue - 32) * 5) / 9);
}
double __fastcall CelsiusToFahrenheit(const double AValue)
{
  return (((AValue * 9) / 5) + 32);
}
double __fastcall KelvinToCelsius(const double AValue)
{
  return (AValue - 273.15);
}
double __fastcall CelsiusToKelvin(const double AValue)
{
  return (AValue +273,15);
}
```

ほかの単位を登録する

以上で変換関数が作成できました。これで，同じ変換ファミリー内のほかの計量単位を登録できます。このとき，単位に関する説明も書いておきます。

ほかの単位を登録するコードの例を下に示します。

```
tuKelvin = RegisterConversionType(cbTemperature, "Kelvin", KelvinToCelsius,
CelsiusToKelvin);
  tuFahrenheit = RegisterConversionType(cbTemperature, "Fahrenheit", FahrenheitToCelsius,
CelsiusToFahrenheit);
```

新しい単位を使用する

この段階で，新規登録した単位を使ってアプリケーション内で変換を実行できます。前に変換ファミリー cbTemperature に登録した変換型であれば，グローバル関数 Convert で変換ができます。下の例は，温度の単位を華氏からケルビン（絶対温度単位）に変換しています。

```
Convert(StrToFloat(Edit1->Text), tuFahrenheit, tuKelvin);
```

クラスを使って変換を管理する

変換関数を使えば，どの変換単位も登録できます。しかし，場合によっては，ほぼ同じことを実行する関数を不要に大量に作成しなければならないことがあります。

パラメータか変数の値だけが異なる複数の変換関数をひとまとめにして記述できれば，クラスを 1 つ作成するだけで複数の変換を処理できます。たとえば欧州通貨の場合，ユーロの導入後，欧州各国通貨の変換を行うための基準設定手法が使われています。（ドルとユーロの間などで変換するのと異なり）変換係数は従来どおり一定ですが，単純な変換係数を使うだけでは，欧州各国の通貨間で正しく変換することはできません。その理由は次のとおりです。

- 変換後の丸めの桁数が通貨によって異なる
- 変換係数を使用する変換は，ユーロ変換基準が指定している係数に対して逆の係数を使用する

しかし，以下のように変換関数によりすべて処理できます。

```
double __fastcall FromEuro(const double AValue, const double Factor, TRoundToRange
FRound)
{
  return(RoundTo(AValue * Factor, FRound));
}
double __fastcall ToEuro(const double AValue, const double Factor)
{
  return (AValue / Factor);
}
```

この方法には問題が 1 つあります。それは，変換関数に対してパラメータが新たに必要になることです。つまり，各欧州通貨に対して単に同じ関数を登録するだけでは正しく変換できません。欧州通貨ごとに変換関数を 2 つずつ新規作成する労力をはぶくには，同じ 2 つの関数を同一クラスのメンバーとして利用します。

変換クラスを作成する

変換クラスは，TConvTypeFactor から派生させます。TConvTypeFactor は，2 つのメソッド（ToCommon と FromCommon）を定義しています。これにより，変換ファミリーの基本単位からの変換と，基本単位への変換（この例ではユーロとの変換）を行います。変換単位の登録時に使用する関数と同様に，これらのメソッドも追加のパラメータを持ちません。したがって，作成する変換クラスのプライベートメンバーとして，四捨五入の桁数や変換係数を自分で指定する必要があります。下に例を示します。この例は Examples \ ConvertIt ディレクトリにある EuroConv サンプルに収められています（euroconv.cpp を参照）。

```
class PASCALIMPLEMENTATION TConvTypeEuroFactor : public Convutils::TConvTypeFactor
{
  private:
    TRoundToRange FRound;
  public:
    __fastcall TConvTypeEuroFactor(const TConvFamily AConvFamily,
        const AnsiString ADescription, const double AFactor, const TRoundToRange ARound);
              TConvTypeFactor(AConvFamily, ADescription, AFactor);
    virtual double ToCommon(const double AValue);
    virtual double FromCommon(const double AValue);
}
```

次のように，コンストラクタがプライベートメンバーに値を割り当てます。

```
__fastcall TConvTypeEuroFactor::TConvTypeEuroFactor(const TConvFamily AConvFamily,
       const AnsiString ADescription, const double AFactor, const TRoundToRange ARound):
              TConvTypeFactor(AConvFamily, ADescription, AFactor);
{
  FRound = ARound;
}
```

下に示す通り，2 つの変換関数は単に上記のプライベートメンバーを使用するだけです。

```
virtual double TConvTypeEuroFactor::ToCommon(const double AValue)
{
  return(RoundTo(AValue * Factor, FRound));
}

virtual double TConvTypeEuroFactor::ToCommon(const double AValue)
{
  return (AValue / Factor);
}
```

変数を宣言する

以上で変換クラスが作成できました。今度は，識別子を宣言して新しい変換ファミリーを登録します。

```
TConvFamily cbEuro;
TConvType euEUR; // EU ユーロ
TConvType euBEF; // ベルギーフラン
TConvType euDEM; // ドイツマルク
TConvType euGRD; // ギリシャドラクマ
TConvType euESP; // スペインペセタ
TConvType euFFR; // フランスフラン
TConvType euIEP; // アイルランドポンド
TConvType euITL; // イタリアリラ
TConvType euLUF; // ルクセンブルグフラン
TConvType euNLG; // オランダギルダー
TConvType euATS; // オーストリアシリング
```

```
TConvType euPTE; // ポルトガルエスクード
TConvType euFIM; // フィンランドマルッカ
```

変換ファミリーおよび他の単位を登録する

新規作成した変換クラスを使って，変換ファミリーおよび欧州各国の通貨単位を登録します。他の変換ファミリーを登録したときとまったく同じように変換ファミリーを登録します。

```
cbEuro = RegisterConversionFamily ("European currency");
```

各変換型を登録するには，通貨の係数と丸めのプロパティを反映した変換クラスのインスタンスを作成し，RegisterConversionType メソッドを呼び出します。

```
TConvTypeInfo *pInfo = new TConvTypeEuroFactor(cbEuro, "EUEuro", 1.0, -2);
if (!RegisterConversionType(pInfo, euEUR))
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "BelgianFrancs", 40.3399, 0);
if (!RegisterConversionType(pInfo, euBEF))
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "GermanMarks", 1.95583, -2);
if (!RegisterConversionType(pInfo, euDEM))
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "GreekDrachmas", 340.75, 0);
if (!RegisterConversionType(pInfo, euGRD)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "SpanishPesetas", 166.386, 0);
if (!RegisterConversionType(pInfo, euESP)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "FrenchFrancs", 6.55957, -2);
if (!RegisterConversionType(pInfo, euFFR)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "IrishPounds", 0.787564, -2);
if (!RegisterConversionType(pInfo, euIEP)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "ItalianLire", 1936.27, 0);
if (!RegisterConversionType(pInfo, euITL)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "LuxembourgFrancs", 40.3399, -2);
if (!RegisterConversionType(pInfo, euLUF)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "DutchGuilders", 2.20371, -2);
if (!RegisterConversionType(pInfo, euNLG)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "AutstrianSchillings", 13.7603, -2);
if (!RegisterConversionType(pInfo, euATS)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "PortugueseEscudos", 200.482, -2);
if (!RegisterConversionType(pInfo, euPTE)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "FinnishMarks", 5.94573, 0);
if (!RegisterConversionType(pInfo, euFIM)
  delete pInfo;
```

新しい単位を使用する

この段階で，新規登録した単位を使ってアプリケーション内で変換を実行できます。新しい変換ファミリー cbEuro に登録した欧州通貨同士であれば，どれでもグローバル関数 Convert で変換ができます。イタリアリラからドイツマルクに変換するコード例を下に示します。

```
Edit2->Text = FloatToStr(Convert(StrToFloat(Edit1->Text), euITL, euDEM));
```

描画スペースの作成

TCanvas クラスは，VCL の Windows デバイスコンテキストおよび CLX のペイントデバイス（Qt ペインタ）をカプセル化したものです。TCanvas は，フォーム，ビジュアルコンテナ（パネルなど），およびプリンタオブジェクト（4-24 ページの「印刷」を参照）向けのあらゆる描画を処理します。キャンバスオブジェクトを利用すれば，ペン，ブラシ，パレットなどの割り当てを気にせずに簡単に描画ができます。割り当てと割り当て解除は自動的に行われます。

TCanvas には非常に多くの基本グラフィックルーチンがあるので，キャンバスを持つコントロールなら，どのコントロールにも線，図形，多角形，フォントなどを描画できます。下に示すボタンイベントハンドラは，フォームの左上隅から中央に線を引き，フォーム上に文字列を表示しています。

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   Canvas->Pen->Color = clBlue;
   Canvas->MoveTo( 10, 10 );
   Canvas->LineTo( 100, 100 );
   Canvas->Brush->Color = clBtnFace;
   Canvas->Font->Name = "Arial";
   Canvas->TextOut( Canvas->PenPos.x, Canvas->PenPos.y,"This is the end of the line" );
}
```

Windows アプリケーションでは，TCanvas オブジェクトを使うと一般的な Windows グラフィックの間違いを防ぐこともできます。たとえば，デバイスコンテキスト，ペン，ブラシなどの値を，描画操作前の値に復元できます。C++Builder では，描画が必要な場所や描画できる場所で TCanvas を使用して，グラフィックの描画を安全かつ簡単にしています。

TCanvas のプロパティおよびメソッドの一覧については，オンラインヘルプで「TCanvas」を参照してください。

第5章

コンポーネントの利用

C++Builder の統合開発環境（IDE）では，多くのビジュアルコンポーネントがコンポーネントパレットに用意されています。コンポーネントパレットでコンポーネントを選択し，フォームやデータモジュールにドロップします。ボタンやリストボックスなどのビジュアルコンポーネントをフォーム上に配置して，アプリケーションのユーザーインターフェースを設計します。データアクセスコンポーネントなどの非ビジュアルコンポーネントも，フォームかデータモジュールに配置できます。

C++Builder のコンポーネントは一見すると，他の C++ クラスとまったく区別が付きません。しかし，C++Builder のコンポーネントと，普通の C++ プログラマが扱う標準 C++ クラスの階層構造との間には，いくつかの違いがあります。主な相違点は次のとおりです。

- C++Builder のコンポーネントはすべて TComponent クラスから派生している
- コンポーネントは，機能を追加または変更してサブクラス化するための「基底クラス」としての役目を果たすよりもむしろ，たいていの場合そのままの形で利用され，プロパティを介して変更される。コンポーネントの継承は通常，既存のイベント処理メンバー関数に特定のコードを追加する目的で行われる
- コンポーネントは，スタックではなくヒープにのみ割り当てることができる。つまり，**new** 演算子を使って作成する必要がある
- コンポーネントの重要なプロパティとして，実行時の型情報が組み込まれている
- コンポーネントは，C++Builder のユーザーインターフェースであるコンポーネントパレットに追加でき，コンポーネントパレットからフォームに追加できる

C++Builder のコンポーネントは，標準 C++ クラスよりも高度なカプセル化を実現します。たとえば，プッシュボタンを持つダイアログを使うと仮定します。VCL コンポーネントを使って開発された C++ Windows プログラムの場合，ユーザーがボタンをクリックすると，システムが WM_LBUTTONDOWN というメッセージを生成します。プログラムは，このメッセージを受け取り（一般的には **switch** 文，メッセージマップ，および応答テーブルにより受け取り），メッセージに応答して実行されるルーチンにディスパッチします。

Windows メッセージ（VCL）とシステムイベント（CLX）の大部分は，C++Builder のコンポーネントによって処理されます。メッセージに応答する処理を行うには，イベントハンドラを記述するだけですみます。

第 8 章「アプリケーションユーザーインターフェースの作成」モード付きフォームの動的作成，フォームへパラメータを渡す方法，フォーム上のデータの検索といったフォームの使い方についての詳細は，第 8 章「アプリケーションユーザーインターフェースの作成」を参照してください。

コンポーネントのプロパティの設定

設計時は，オブジェクトインスペクタを使ってパブリッシュプロパティを設定できます。専用のプロパティエディタを使う場合もあります。実行時にプロパティを設定するには，アプリケーションのソースコードでプロパティの値を指定します。

各コンポーネントのプロパティについての詳細は，オンラインヘルプを参照してください。

設計時にプロパティを設定する

設計時にフォーム上でコンポーネントを選択すると，そのパブリッシュプロパティがオブジェクトインスペクタに表示されます。プログラマは，適宜オブジェクトインスペクタでプロパティを変更できます。【Tab】キーを押すたびに，左側の値列と右側のプロパティ列の間でカーソル位置が切り替わります。カーソルがプロパティ列にあるとき，プロパティ名の最初の文字を入力すると，目的のプロパティにすばやく移動できます。論理型か列挙型のプロパティの場合，値列にあるドロップダウンリストから目的の値を選択します。ダブルクリックして表示値を切り替えることもできます。

プロパティ名の隣に正符号（+）が表示されている場合，正符号をクリックするか，プロパティにフォーカスがあるときに「+」を入力すると，そのプロパティの下位項目が表示されます。同様に，プロパティ名の隣に負符号（-）が表示されている場合，負符号をクリックするか「-」を入力すると，そのプロパティの下位項目は非表示になります。

デフォルトでは，［旧式］に分類されるプロパティは表示されません。表示フィルタを変更するには，オブジェクトインスペクタで右クリックして［表示する項目］を選択します。詳細については，オンラインヘルプの「プロパティカテゴリ」を参照してください。

フォームで複数のコンポーネントを選択すると，オブジェクトインスペクタには，選択したコンポーネントのすべてで共有されるプロパティが表示されます（Name プロパティを除く）。選択したコンポーネントの共有プロパティの値が同一でない場合は，デフォルト値か，最初に選択したコンポーネントの値が表示されます。共有プロパティの値を変更すると，選択したコンポーネントの値がすべて変更されます。

オブジェクトインスペクタを使ってイベントハンドラの名前などコード関連のプロパティを変更すると，ソースコードの対応部分が自動的に変更されます。逆に，ソースコードを変更すると（たとえば，フォームクラスの宣言部でイベントハンドラのメソッドの名前を変更する）など，その変更はオブジェクトインスペクタに即時に反映されます。

プロパティエディタの使い方

Font プロパティなど，一部のプロパティには専用のプロパティエディタがあります。プロパティエディタを持つプロパティの場合，オブジェクトインスペクタでそのプロパティを選択すると，値列の隣に省略記号（...）が表示されます。プロパティエディタを開くには，値をダブルクリックするか，省略記号（...）をクリックするか，あるいはプロパティ名または値にフォーカスがあるときに〔Ctrl〕+〔Enter〕を押します。また，コンポーネントによっては，フォーム上でコンポーネントをダブルクリックしてもプロパティエディタが表示される場合があります。

プロパティエディタを使うと，ダイアログボックス 1 つで複雑なプロパティを簡単に設定できます。プロパティエディタは入力内容を検証し，多くの場合，プロパティの設定結果をプレビュー表示します。

実行時のプロパティの設定

設計時に設定できるプロパティは，ソースコードを使って実行時にも設定できます。たとえば，フォームのキャプションを次のように動的に割り当てることができます。

```
Form1->Caption = MyString;
```

メソッドの呼び出し

メソッドの呼び出しは，通常の手続きや関数と同じように行われます。たとえば，ビジュアルコントロールの Repaint メソッドで画面上のコントロールのイメージを再描画するとします。この場合，次のように描画グリッドオブジェクトの中で Repaint メソッドを呼び出します。

```
DrawGrid1->Repaint;
```

プロパティと同様に，限定子が必要かどうかはメソッド名のスコープによって決まります。たとえば，フォームの子コントロールのイベントハンドラの範囲内でフォームを再描画する場合には，メソッド呼び出しの前にフォーム名を付加する必要はありません。

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Repaint;
}
```

イベント / イベントハンドラでの作業

C++Builder では，記述するコードのほとんどはイベントに応じて直接または間接に実行されます。イベントとは，実行時の出来事（たいていはユーザーのアクション）を表す一種のプロパティです。イベントに直接応答するコードを「イベントハンドラ」といいます。イベントハンドラはオブジェクトのメソッドです。この節では，以下の内容を解説します。

- 新規イベントハンドラの生成
- コンポーネントのデフォルトイベントハンドラの生成
- イベントハンドラの検索

- イベントと既存のイベントハンドラの関連付け
- メニューイベントとイベントハンドラの関連付け
- イベントハンドラの削除

新規イベントハンドラの生成

C++Builder は，フォームなどのコンポーネント用にスケルトンのイベントハンドラを生成できます。イベントハンドラを作成する方法は次のとおりです。

1. コンポーネントを選択します。
2. オブジェクトインスペクタの［イベント］タブをクリックします。オブジェクトインスペクタの［イベント］ページに，選択したコンポーネントに定義されているイベントが一覧表示されます。
3. 目的のイベントを選択し，値列をダブルクリックするか，または〔Ctrl〕+〔Enter〕を押します。
4. イベントが起きたときに実行するコードを入力します。

コンポーネントのデフォルトイベントハンドラの生成

一部のコンポーネントはデフォルトのイベントを持っています。デフォルトのイベントとは，そのコンポーネントで処理する頻度がもっとも高いイベントです。たとえば，ボタンのデフォルトイベントは OnClick です。デフォルトイベントハンドラを作成するには，フォームデザイナで目的のコンポーネントをダブルクリックします。スケルトンのイベント処理手続きが生成され，コードエディタが開きます。コードエディタでは，すでに手続き本文の中にカーソルが置かれているので，簡単にコードを追加することができます。

デフォルトイベントを持たないコンポーネントもあります。TBevel など一部のコンポーネントは，どのイベントにも応答しません。フォーム上でコンポーネントをダブルクリックしても，コードエディタが表示されず，別の応答をする場合もあります。たとえば，設計時にコンポーネントをダブルクリックすると，デフォルトのプロパティエディタなどのダイアログが表示されることがあります。

イベントハンドラの検索

フォームでコンポーネントをダブルクリックして，デフォルトイベントハンドラを生成した場合，同じ方法で検索することができます。コンポーネントをダブルクリックするとコードエディタが開き，イベントハンドラ本体の先頭にカーソルが配置されます。

デフォルト以外のイベントハンドラを検索する手順は次のとおりです。

1. フォーム上で，検索するイベントハンドラに対応するコンポーネントを選択します。
2. オブジェクトインスペクタで［イベント］タブを選択します。
3. 目的のイベントを選択し，その値列をダブルクリックします。コードエディタが開き，イベントハンドラ本体の先頭にカーソルが配置されます。

イベントと既存のイベントハンドラの関連付け

複数のイベントに応答するイベントハンドラを記述しておくと，コードを再利用できます。たとえば，多くのアプリケーションはドロップダウンメニューのコマンドに相当するスピードボタンを提供します。ボタンとメニューコマンドが同じアクションを起動する場合，イベントハンドラを 1 つ書いておけば，ボタンとメニュー項目の両方の OnClick イベントに割り当てることができます。

既存のイベントハンドラとイベントを関連付ける方法は次のとおりです。

1. イベントハンドラと関連付けるコンポーネントをフォーム上で選択します。
2. オブジェクトインスペクタの［イベント］ページを表示し，イベントハンドラと結び付けるイベントを選択します。
3. 値列にあるイベント値の横の下矢印をクリックし，既存のイベントハンドラのリストを開きます（リストには，同じフォームの中で，同じ名前のイベントについて書かれたイベントハンドラだけが表示されます）。リストのイベントハンドラ名をクリックして選択します。

このように関連付けすると，イベントハンドラを簡単に再利用できます。なお，アクションリストおよび VCL のアクションバンドはこれ以上にパワフルなツールです。これらを使うと，ユーザーコマンドに応答するコードを一元管理できます。クロスプラットフォームアプリケーションの場合，アクションリストは使用できますが，アクションバンドは使用できません。アクションリストについての詳細は，8-16 ページの「ツールバーとメニューのアクションを構成する」を参照してください。

Sender パラメータの使い方

イベントハンドラの Sender パラメータは，どのコンポーネントがイベントを受け取ったか，つまり，どのコンポーネントがハンドラを呼び出したかを表します。複数のコンポーネントに同じイベントハンドラを共有させ，呼び出し元のコンポーネントに応じて異なる操作をさせると便利な場合があります。これを行うには，Sender パラメータを使います。

共通のイベントを表示してコードを書く

複数のコンポーネントの間に共通のイベントがある場合，その共通イベントもオブジェクトインスペクタに表示できます。まず，フォーム上で目的のコンポーネントをすべて選択（[Shift] を押しながらクリック）し，オブジェクトインスペクタの［イベント］ページを選択します。次に，オブジェクトインスペクタの値列を選択します。これで，共有イベントに新しいイベントハンドラを作成するか，または既存のイベントハンドラに共有イベントを割り当てることが可能になります。

メニューイベントとイベントハンドラの関連付け

メニューデザイナと MainMenu，PopupMenu の各コンポーネントを組み合わせて使うと，簡単にドロップダウンメニューやポップアップメニューを設計できます。ただし，メニューを使えるようにするには，ユーザーがメニュー項目を選択するか，アクセラレータキーまたはショートカットキーを押したとき，各メニュー項目が OnClick イベントに応答しなければなりません。この節では，イベントハンドラをメニュー項目と関連付ける方法について説明します。メニューデザイナおよび関連コンポーネントに関する詳細は，8-30 ページの「メニューの作成と管理」を参照してください。

メニュー項目のイベントハンドラを作成する手順は次のとおりです。

1. MainMenu コンポーネントか PopupMenu コンポーネントをダブルクリックしてメニューデザイナを開きます。
2. メニューデザイナでメニュー項目を選択します。オブジェクトインスペクタで，このメニュー項目の Name プロパティに値が設定されていることを確認します。
3. メニューデザイナに戻り，このメニュー項目をダブルクリックします。コードエディタにイベントハンドラが生成されます。
4. ユーザーがこのメニューコマンドを選択したときに実行するコードを入力します。

メニュー項目に既存の OnClick イベントハンドラを関連付ける手順は次のとおりです。

1. MainMenu コンポーネントか PopupMenu コンポーネントをダブルクリックしてメニューデザイナを開きます。
2. メニューデザイナでメニュー項目を選択します。オブジェクトインスペクタで，このメニュー項目の Name プロパティに値が設定されていることを確認します。
3. オブジェクトインスペクタの［イベント］ページで，値列の OnClick の横にある下矢印をクリックして，以前に作成したイベントハンドラのリストを表示します（リストには，このフォームの OnClick イベントに作成したイベントハンドラだけが表示されます）。リストのイベントハンドラ名をクリックして選択します。

イベントハンドラの削除

フォームデザイナでコンポーネントを削除すると，C++Builder はフォームの型宣言からそのコンポーネントを除去します。ただし，コンポーネントを削除しても，関連付けられたメソッドはユニットファイルから削除されずに残っています。フォーム上のほかのコンポーネントに呼び出される可能性があるからです。イベントハンドラなどのメソッドを手作業で削除することはできますが，その場合，メソッドの forward 宣言と実現部の両方を削除してください。両方を削除しないと，プロジェクト構築時にコンパイラエラーになります。

クロスプラットフォームコンポーネントと非クロスプラットフォームコンポーネント

コンポーネントパレットには，さまざまなプログラミングタスクを処理するコンポーネントが多数用意されています。コンポーネントは，目的と機能別に複数のページにグループ化されています。たとえば，メニュー，編集ボックス，またはボタンなど，さまざまな種類のアプリケーションに共通して使われるコンポーネントは［Standard］ページにあります。デフォルトのページ構成は製品のバージョン（版）によって異なります。

表 3.3 に，標準のデフォルトページと，アプリケーション作成で使用できるコンポーネント（非クロスプラットフォームコンポーネントを含む）の一覧を示します。CLX コンポーネントはすべて，Windows アプリケーションと Linux アプリケーションの両方で使用できます。VCL 固有のコンポーネントを使って Windows 専用の CLX アプリケーションを作成することも可能ですが，その場合，そ

のアプリケーションをクロスプラットフォームにするには，コードの Windows 専用部分を分離する必要があります。

表 5.1　コンポーネントパレットのページ

ページ名	説明	クロスプラットフォームか？
Standard	標準のコントロール，メニュー	Yes
Additional	特殊コントロール	Yes。ただし，ApplicationEvents, ActionManager, ActionMainMenuBar, ActionToolBar, CustomizeDlg を除く。LCDNumber は CLX のみ
Win32（VCL）/ Common Controls（CLX）	Windows のコモンコントロール	CLX アプリケーションの作成時に表示される［Common Controls］ページには，［Win32］ページと同じコンポーネントが多数表示される。
		RichEdit, UpDown, HotKey, Animate, DataTimePicker, MonthCalendar, Coolbar, PageScroller, ComboBoxEx は VCL のみ。
		TextBrowser, TextViewer, IconViewer, SpinEdit は CLX のみ
System	システムレベルのアクセス用コンポーネントおよびコントロール。タイマー，マルチメディア，および DDE を含む	No。ただし Timer と PaintBox を除く。Timer と PaintBox は，CLX アプリケーションを作成すると［Additional］ページに表示される
Data Access	特定のデータアクセスメカニズムと連携しないデータベースデータを操作するコンポーネント	Yes。ただし XMLTransform, XMLTransformProvider, XMLTransformClient を除く
Data Controls	ビジュアルなデータベース対応コントロール	Yes。ただし DBRichEdit，DBCtrlGrid, DBChart を除く
dbExpress	dbExpress（クロスプラットフォーム対応で，データベースに依存しないレイヤ。動的に SQL を処理する方法を提供する）を使用するデータベースコントロール。SQL サーバーにアクセスするための共通のインターフェースを定義する	Yes
DataSnap	多層データベースアプリケーションの作成に使用するコンポーネント	No
BDE	ボーランドデータベースエンジンを介してデータアクセスを提供するコンポーネント	No
ADO	ADO フレームワークを通じてデータアクセスを提供するコンポーネント	No
InterBase	InterBase データベースへのダイレクトアクセスを提供するコンポーネント	Yes
InterBaseAdmin	InterBase Services API 呼び出しにアクセスするコンポーネント	Yes
InternetExpress	Web サーバーアプリケーションと同時に多層データベースアプリケーションのクライアントとなるコンポーネント	No
Internet	インターネット通信プロトコルおよび Web アプリケーション用コンポーネント	No

表 5.1　コンポーネントパレットのページ（つづき）

ページ名	説明	クロスプラットフォームか？
WebSnap	Web サーバーアプリケーションを構築するためのコンポーネント	No
FastNet	NetMasters インターネットコントロール	No
QReport	組み込みのレポートを作成する QuickReport コンポーネント	No
Dialogs	一般に使われるダイアログボックス	Yes。ただし OpenPictureDialog, SavePictureDialog，PrintDialog, PageSetupDialog を除く
Win 3.1	Win 3.1 用の旧式コンポーネント	No
Samples	カスタムコンポーネントのサンプル	No
ActiveX	ActiveX サンプルコントロール。Microsoft のドキュメントを参照（msdn.microsoft.com）	No
COM+	COM+ イベントを処理するためのコンポーネント	No
WebServices	SOAP ベースの Web サービスを実装 / 使用するアプリケーションを記述するためのコンポーネント	No
Servers	Microsoft Excel や Word 用の COM サーバー例（Microsoft MSDN ドキュメントを参照）	No
Indy Clients	クライアント用のクロスプラットフォームインターネットコンポーネント（オープンソース Winshoes インターネットコンポーネント）	Yes
Indy Servers	サーバー用のクロスプラットフォームインターネットコンポーネント（オープンソース Winshoes インターネットコンポーネント）	Yes
Indy Misc	その他のクロスプラットフォームインターネットコンポーネント（オープンソース Winshoes インターネットコンポーネント）	Yes

パレットにコンポーネントを追加したり，パレット上のコンポーネントを削除，再配置したりできます。また，コンポーネントテンプレートとフレームを作成して，コンポーネントをグループ化することもできます。

コンポーネントパレットに表示される各コンポーネントに関する情報は，ヘルプを参照してください。ただし，［ActiveX］ページ，［Servers］ページ，［Samples］ページの一部のコンポーネントはサンプルとしてのみ提供されており，詳しいマニュアルは付属していません。

VCL と CLX の違いについての詳細は第 14 章「クロスプラットフォームアプリケーションの開発」を参照してください。

コンポーネントパレットへのカスタムコンポーネントの追加

カスタムコンポーネント（独自に作成するコンポーネントまたはサードパーティ製のコンポーネント）をコンポーネントパレットにインストールして，アプリケーションで使用することができます。カスタムコンポーネントの作成方法については，第 V 部「カスタムコンポーネントの作成」を参照

してください。既存コンポーネントのインストール方法については，15-5 ページの「コンポーネントパッケージのインストール」を参照してください。

第6章

コントロールの利用

コントロールとは，実行時にユーザーが対話できるビジュアルコンポートです。この章では，多くのコントロールに共通する各種機能について説明します。

ドラッグアンドドロップのコントロールへの実装

ドラッグアンドドロップは，ユーザーがオブジェクトを操作するときに頻繁に使われる便利な方法です。コンポーネント全体をドラッグすることも，リストボックスやツリービューなどのコンポーネントから別のコンポーネントへアイテムをドラッグすることもできます。

- ドラッグ操作の開始
- ドラッグした項目の受け入れ
- 項目のドロップ
- ドラッグ操作の終了
- ドラッグオブジェクトを使ったドラッグアンドドロップのカスタマイズ
- ドラッグマウスポインタの変更

ドラッグ操作の開始

どのコントロールにも DragMode プロパティがあります。DragMode は，ドラッグ操作の開始方法を決定するプロパティです。DragMode に dmAutomatic を設定すると，ユーザーがコントロールにポインタを置いてマウスボタンを押すと自動的にドラッグが始まります。しかしながら，dmAutomatic を設定すると通常のマウス動作を制御できなくなる可能性があるため，一般的には DragMode に dmManual（デフォルト）を設定し，マウスボタンを押したことを示すイベントを処理することでドラッグを開始します。

コントロールのドラッグを（自動的にでなく）明示的に開始するには，そのコントロールの BeginDrag メソッドを呼び出します。BeginDrag は Immediate という論理パラメータおよび Threshold という整数パラメータをとります。Immediate を **true** で渡すと，すぐにドラッグが始まります。**false**

を渡すと，Threshold で指定したピクセル数だけユーザーがマウスを動かすまで，ドラッグは開始しません。Threshold が -1 の場合はデフォルト値が使われます。下のコードを呼び出せば，

```
BeginDrag (false, -1);
```

コントロールをクリックしただけではドラッグ操作は開始されません。

BeginDrag を呼び出す前に，マウスボタンが押されたことを示すイベントのパラメータをチェックし，ユーザーが押したボタンを調べてから，条件が合う場合にのみドラッグを開始することもできます。たとえば，マウスボタンが押されたことを示すファイルリストボックスのイベントに応答して有効な項目の上で左ボタンが押された場合のみ，ドラッグ操作を開始するハンドラを次に示します。

```
void __fastcall TFMForm::FileListBox1MouseDown(TObject *Sender,
  TMouseButton Button, TShiftState Shift, int X, int Y)
{
  if (Button == mbLeft)   // 左ボタンが押されたときのみドラッグ
  {
    TFileListBox *pLB = (TFileListBox *)Sender; // TFileListBox へキャストする
    if (pLB->ItemAtPos(Point(X,Y), true) >= 0)  // 項目があるかどうかチェック
      pLB->BeginDrag(false, -1);                // 項目がある場合には，ドラッグする
  }
}
```

ドラッグした項目の受け入れ

ユーザーがコントロール上で何かをドラッグすると，コントロールは OnDragOver イベントを受け取り，その時点でユーザーがその項目をドロップした場合に受け入れることができるかどうかを示さなければなりません。ドラッグカーソルが変化し，ドラッグされた項目をコントロールが受け入れられるかどうかをユーザーに示します。コントロールがドラッグした項目を受け入れるには，コントロールの OnDragOver イベントに対応するイベントハンドラを定義します。

このドラッグオーバーイベントには Accept というパラメータがあります。項目を受け入れる場合，イベントハンドラはこのパラメータに **true** を設定します。Accept を設定することにより，カーソルの種類を変更します（受け入れカーソル，非受け入れカーソルのどちらかに変更）。

ドラッグオーバーイベントには，ドラッグのソースやマウスカーソルの現在位置を示すパラメータもあります。イベントハンドラは各パラメータを参照して，ドラッグを受け入れるかどうかを判断します。次の例では，ファイルリストボックスからドラッグされた項目だけをディレクトリツリービューで受け入れます。

```
void __fastcall TForm1::TreeView1DragOver(TObject *Sender, TObject *Source,
  int X, int Y, TDragState State, bool &Accept)
{
  if (Source->InheritsFrom(__classid(TFileListBox)))
    Accept = true;
}
```

項目のドロップ

ドラッグした項目を受け入れることを意思表示したコントロールは，ドロップした項目を処理する必要があります。ドロップした項目を処理するには，ドロップを受け入れるコントロールの OnDragDrop イベントに対応するイベントハンドラを定義します。ドラッグオーバーイベントと同じ

く，ドラッグアンドドロップイベントは，受け入れるコントロール上のマウスポインタの座標と，ドラッグした項目のソースを通知します。マウスポインタの座標を示すパラメータにより，ドラッグ中に項目がとる経路をモニタできます。このような仕様になっているので，たとえば，項目がドロップされた場合にコンポーネントの色を変更できます。

次に示す例では，ファイルリストボックスからドラッグした項目をディレクトリツリービューが受け入れ，ドロップ先のディレクトリにファイルを移動するという応答をしています。

```
void __fastcall TForm1::TreeView1DragDrop(TObject *Sender, TObject *Source,
  int X, int Y){
  if (Source->InheritsFrom(__classid(TFileListBox)))
  {
    TTreeNode *pNode = TreeView1->GetNodeAt(X,Y);  // pNode は，ドロップ先のターゲット
    if (pNode)
    {
      AnsiString NewFile = pNode->Text + AnsiString("\\") +
        ExtractFileName(FileListBox1->FileName);  // ドロップ先ターゲットのファイル名を作成
      MoveFileEx(FileListBox1->FileName.c_str(), NewFile.c_str(),
        MOVEFILE_REPLACE_EXISTING | MOVEFILE_COPY_ALLOWED);  // ファイルを移動する
    }
  }
}
```

ドラッグ操作の終了

ドラッグした項目をドロップするか，または項目を受け入れないコントロール上でマウスボタンを放した時点で，ドラッグ操作が終了します。ドラッグ操作が終了すると，ドラッグ元のコントロールに対してドラッグ終了イベントが送られます。項目のドラッグ元のコントロールでドラッグの終了に応答するには，コントロールの OnEndDrag イベントに対応するイベントハンドラを定義します。

OnEndDrag イベントのもっとも重要なパラメータは Target です。このパラメータはドロップを受け入れたコントロールを示します。Target がヌルであれば（何も入っていなければ），ドラッグした項目を受け入れたコントロールがないことを示します。OnEndDrag イベントには，受け入れ側コントロールの座標を示すパラメータもあります。

次の例では，ファイルリストボックスがドラッグ終了イベントに応答してファイルリストを更新しています。

```
void __fastcall TFMForm::FileListBox1EndDrag(TObject *Sender, TObject *Target,
    int X, int Y)
  if (Target)
    FileListBox1->Update();
};
```

ドラッグオブジェクトを使ったドラッグアンドドロップのカスタマイズ

TDragObject の下位オブジェクトを使って，オブジェクトのドラッグアンドドロップ動作をカスタマイズできます。標準のドラッグオーバーイベントとドラッグアンドドロップイベントは，ドラッグされる項目のソースと，受け入れ側のコントロールにおけるマウスカーソルの座標をパラメータとして

受け取ることができます。付加的な状態情報を取得するには，TDragObject または TdragObjectEx（VCL のみ）からカスタムドラッグオブジェクトを派生させ，その仮想メソッドをオーバーライドします。カスタムドラッグオブジェクトを OnStartDrag イベントで作成します。

通常，ドラッグオーバーイベントおよびドラッグアンドドロップイベントのソースパラメータは，ドラッグ操作を開始したコントロールです。種類の異なる複数のコントロールが同一種類のデータの操作を開始できる場合，ソースは各種類のコントロールをサポートする必要があります。ただし，TDragObject の子オブジェクトを使う場合は，ドラッグオブジェクト自身がソースです。つまり，各コントロールが自分自身の OnStartDrag イベントに同一種類のドラッグオブジェクトを生成すれば，受け入れ先では 1 種類のオブジェクトだけを処理すればよいことになります。ドラッグを開始したコントロールとは対照的に，ドラッグオーバーイベントおよびドラッグアンドドロップイベントは IsDragObject 関数を呼び出すことにより，ソースがドラッグオブジェクトであるかどうかを識別できます。

TDragObjectEx の下位オブジェクト（VCL のみ）が自動的に解放されるのに対し，TDragObject の下位オブジェクトは自動的に解放されません。TDragObject の下位オブジェクトを明示的に解放させていない場合は，TDragObjectEx から派生するように変更するとメモリの損失を防止できます。

アプリケーションの実行形式ファイルに実装されたフォームと，.DLL ファイルを使って実装されたフォームの間でもドラッグ項目をドラッグできます。複数の .DLL ファイルを使って実装されたフォーム間でもドラッグできます。

ドラッグマウスポインタの変更

ドラッグ操作中のマウスポインタの外観をカスタマイズするには，ソースコンポーネントの DragCursor プロパティを設定します（VCL のみ）。

ドラッグアンドドックのコントロールへの実装

TWinControl からの派生クラスはドッキングサイトとして動作し，TControl からの派生クラスはドッキングサイトにドックされる子ウィンドウとして動作します。たとえば，フォームウィンドウの左端にドッキングサイトを備えるには，パネルをフォームの左端に配置してパネルをドッキングサイトにします。ドッキング可能コントロールがパネルにドラッグされてリリースされると，パネルの子コントロールになります。

- ウィンドウコントロールをドッキングサイトにする
- コントロールをドッキング可能な子コントロールにする
- 子コントロールがドッキングサイトにどのようにドッキングされるかを制御する
- 子コントロールがどのようにアンドックされるか制御する
- 子コントロールがどのようにドラッグアンドドック操作に応答するかを制御する

メモ ドラッグアンドドックプロパティは，VCL では使用できますが CLX では使用できません。

ウィンドウコントロールをドッキングサイトにする

ウィンドウコントロールをドッキングサイトにする手順は以下のとおりです。

1. DockSite プロパティを **true** に設定します。
2. ドッククライアントを含むとき以外に，ドックサイトオブジェクトが表れないようにするには，AutoSize プロパティを **true** にします。AutoSize が **true** の場合，ドックサイトはドッククライアントの大きさに合わせてサイズ変更されます。ドッククライアントが存在しない場合には，サイズは 0 になります。次に，子コントロールに合うようにサイズ変更します。

コントロールをドッキング可能な子コントロールにする

コントロールをドッキング可能な子コントロールにする手順は以下のとおりです。

1. DragKind プロパティを dkDock に設定します。DragKind が dkDock の場合，コントロールをドラッグすると，新しいドッキングサイトにコントロールを移動したり，コントロールをアンドックしてフローティングウィンドウにしたりすることができます。DragKind が dkDrag の場合（デフォルト），コントロールをドラッグすると，ドラッグアンドドロップ操作を開始します。ドラッグアンドドロップ操作は，OnDragOver，OnEndDrag，および OnDragDrop イベントを定義して実装されなければなりません。
2. DragMode を dmAutomatic に設定します。DragMode が dmAutomatic である場合，ユーザーがマウスでコントロールをドラッグすると，ドラッグ（ドラッグアンドドロップ操作かドッキング操作かは DragKind に依存）が自動的に開始されます。DragMode が dmManual の場合は，BeginDrag メソッドを呼び出して，ドラッグアンドドック（またはドラッグアンドドロップ）操作を開始させることができます。
3. コントロールがアンドックされてフローティングウィンドウとして残されたときに TWinControl からの派生クラスがコントロールをホストすることを示すように，FloatingDockSiteClass プロパティを設定します。コントロールがドッキングサイト上でない場所でリリースされると，このクラスのウィンドウコントロールが動的に作成され，ドッキング可能な子コントロールの親になります。ドッキング可能な子コントロールが TWinControl の下位クラスである場合，コントロールをホストするための別の浮動ドックサイトを作成する必要はありません。ダイナミックコンテナーウィンドウを省略するときは，FloatingDockSiteClass をコントロールと同じクラスに設定すると，親のないフローティングウィンドウになります。

子コントロールがドッキングサイトにどのようにドッキングされるかを制御する

ドッキングサイトは，子コントロールがドッキングサイト上でリリースされると自動的に受け入れます。ほとんどのコントロールでは，最初の子がクライアント領域のサイズに合わせてドッキングされ，2 番目の子がクライアント領域を分離した領域に分割します。この分割が順番に続いていきます。ページコントロールでは，子を新しいタブシートにドッキングします。

ドッキングサイトで子コントロールがどのようにドッキングされるかを制限するには，以下に説明する 3 つのイベントを使います。

```
__property TGetSiteInfoEvent OnGetSiteInfo = {read=FOnGetSiteInfo, write=FOnGetSiteInfo};
typedef void __fastcall (__closure *TGetSiteInfoEvent)(System::TObject* Sender, TControl*
DockClient, Windows::TRect &InfluenceRect, const Windows::TPoint &MousePos, bool &CanDock);
```

OnGetSiteInfo イベントは，ユーザーがドッキング可能な子をコントロール上にドラッグすると，ドッキングサイト側で発生します。このイベントにより，サイトが子として DockClient パラメータで指定されたコントロール（その場合，子がドッキング用であるとみなされなければならない）を受け入れるかどうかを示すことができます。OnGetSiteInfo が発生すると，InfluenceRect がドッキングサイトの画面座標に，CanDock が **true** に初期化されます。InfluenceRect を変更するとドッキング領域を制限することができます。CanDock を **false** に設定すると，子のドッキングを拒否できます。

```
__property TDockOverEvent OnDockOver = {read=FOnDockOver, write=FOnDockOver};
typedef void __fastcall (__closure *TDockOverEvent)(System::TObject* Sender,
TDragDockObject* Source, int X, int Y, TDragState State, bool &Accept);
```

OnDockOver イベントは，ユーザーがドッキング可能な子をコントロール上にドラッグすると，ドッキングサイト側で発生します。これはドラッグアンドドロップ操作の OnDragOver イベントに類似しています。このイベントの Accept パラメータを設定して，ドッキングのために子をリリースできることを知らせます。ドッキング可能コントロールが OnGetSiteInfo イベントハンドラで（たとえば，コントロールの種類が違うため）拒否された場合，OnDockOver は起こりません。

```
__property TDockDropEvent OnDockDrop = {read=FOnDockDrop, write=FOnDockDrop};
typedef void __fastcall (__closure *TDockDropEvent)(System::TObject* Sender,
TDragDockObject* Source, int X, int Y);
```

OnDockDrop イベントは，ユーザーがドッキング可能な子をコントロール上でリリースすると，ドッキングサイト側で発生します。これはドラッグアンドドロップ操作の OnDragDrop イベントに類似しています。このイベントを使用して，コントロールを子コントロールとして受け入れるために必要な処理を実行します。子コントロールへのアクセスは，Source パラメータで指定された TDockObject の Control プロパティを使用して取得できます。

子コントロールがどのようにアンドックされるか制御する

DragMode プロパティが dmAutomatic に設定されている子コントロールが，他のドッキングサイトに移動するときのようにドラッグされた場合に，ドッキングサイトはそのコントロールを自動的にアンドックすることができます。ドッキングサイトは OnUnDock イベントハンドラ内を定義することで，アンドック時に特別な処理を行ったり，アンドックすることを禁止したりできます。

```
__property TUnDockEvent OnUnDock = {read=FOnUnDock, write=FOnUnDock};
typedef void __fastcall (__closure *TUnDockEvent)(System::TObject* Sender, TControl*
Client, TWinControl* NewTarget, bool &Allow);
```

Client パラメータはアンドックしようとする子コントロールを示します。子コントロールのアンドックを許可するには，Allow パラメータに true を，拒否するには false を設定します。OnUnDock イベントハンドラを実装するときには，ほかの子コントロール（存在する場合）が現在ドッグされているか知っておくと便利です。現在ドッキングされているほかの子コントロール（存在する場合）の情報を取得するには，TControl 型の配列である DockClient プロパティが使えます。ドッキングされている子コントロールの数は，DockClientCount プロパティに格納されています。

子コントロールがどのようにドラッグアンドドック操作に応答するかを制御する

ドッキング可能な子コントロールには，ドラッグアンドドック操作中に発生するイベントが 2 つあります。OnStartDock イベントは，ドラッグアンドドロップ操作の OnStartDrag イベントと同様に，ドッキング可能な子コントロールがカスタムドラッグオブジェクトを作成するのを可能にします。OnEndDock イベントは，ドラッグアンドドロップ操作の OnEndDrag イベントと同様に，ドラッグ操作が終了したときに発生します。

コントロール内のテキストを操作する

以降の節では，書式付きエディタとメモコントロールの各種機能の使い方について説明します。これらの各種機能の一部は，編集コントロールでも使用できます。

- テキストの位置揃えの設定
- 実行時のスクロールバーの追加
- クリップボードオブジェクトの追加
- テキストの選択
- すべてのテキストの選択
- テキストの切り取り，コピー，貼り付け
- 選択したテキストの削除
- メニュー項目を使用不可にする
- ポップアップメニューの提供
- OnPopup イベントを処理する

テキストの位置揃えの設定

書式付きエディタやメモコンポーネントでは，テキストの右揃え，左揃え，中央揃えができます。テキストの位置揃えをするには，エディタコンポーネントの Alignment プロパティを設定します。位置揃えは WordWrap プロパティが **true** の場合にだけ有効になります。ワードラップをオフにすると，位置揃えするマージンがなくなります。

たとえば，以下の RichEdit の例で取り上げたコードは，選択されたボタンに応じて位置揃えの設定を行います。

```
switch((int)RichEdit1->Paragraph->Alignment)
{
  case 0: LeftAlign->Down   = true; break;
  case 1: RightAlign->Down  = true; break;
  case 2: CenterAlign->Down = true; break;
}
```

実行時のスクロールバーの追加

書式付エディタとメモコンポーネントには，水平スクロールバーと垂直スクロールバーの両方または一方を必要に応じて表示することができます。ワードラップが有効な場合，コンポーネントが必要とするのは垂直スクロールバーだけです。ワードラップを無効にした場合，テキストが右マージンで制限されなくなるので，コンポーネントは水平スクロールバーも必要とすることがあります。

実行時にスクロールバーを追加する手順は次のとおりです。

1. テキストが右マージンを超えることがあるかどうかを調べます。通常は，ワードラップが有効になっているかどうかを確認します。実際にテキスト行がコンポーネント幅を超えているかどうかを調べる場合もあります。
2. 編集コンポーネントやメモコンポーネントの ScrollBars プロパティを設定して，スクロールバーを表示または非表示にします。

以下の例は，［Character ｜ WordWrap］メニューの OnClick イベントハンドラです。

```
void __fastcall TEditForm::WordWrap1Click(TObject *Sender)
{
  Editor->WordWrap = !(Editor->WordWrap);   // ワードラップを切り替える
  if (Editor->WordWrap)
    Editor->ScrollBars = ssVertical;        // ラップする場合，垂直方向のみ
  else
    Editor->ScrollBars = ssBoth;            // アンラップする場合，両方向
  WordWrap1->Checked = Editor->WordWrap;    // プロパティを一致させるために
                                            // メニュー項目にチェックマークを付ける
}
```

書式付きエディタとメモコンポーネントは，少し異なる方法でスクロールバーの処理をします。書式付きテキストコンポーネントでは，テキストがコンポーネントの範囲内に収まる場合はスクロールバーを非表示にできます。メモコンポーネントでは，スクロールバーは使用可能な場合は必ず表示されます。

クリップボードオブジェクトの追加

ほとんどのテキスト処理アプリケーションでは，異なるアプリケーションの文書も含めて，文書どうしの間で選択したテキストを移動できます。C++Builder の Clipboard オブジェクトはクリップボード（Windows のクリップボードなど）をカプセル化するもので，テキスト（およびグラフィックなどの他の形式）の切り取り，コピー，貼り付けのためのメソッドが入っています。Clipboard オブジェクトは Clipbrd ユニットで宣言されています。

クリップボードオブジェクトをアプリケーションに追加する手順は次のとおりです。

1. クリップボードを使うユニットを選択します。
2. フォームの .h ファイルに次の文を追加します。

```
#include <Clipbrd.hpp>
```

テキストの選択

編集コントロールの中にあるテキストの場合，最初にテキストを選択しておかないとクリップボードに送ることができません。ここでは，編集コンポーネントの場合について説明します。選択したテキストを強調表示する機能は編集コンポーネントに組み込まれています。ユーザーがテキストを選択すると，そのテキストは強調表示されます。

表 6.1 に，選択したテキストを処理するために通常使用されるプロパティを示します。

表 6.1　選択したテキストのプロパティ

プロパティ	説明
SelText	コンポーネントで選択されているテキストに対応する文字列が入る
SelLength	選択した文字列の長さが入る
SelStart	文字列の開始位置（編集コントロールのテキストバッファの先頭に対する相対位置）が入る

すべてのテキストの選択

SelectAll メソッドは，編集コントロール（書式付きエディタ，メモコンポーネントなど）の内容全体を選択します。この機能は，コンポーネントの内容が表示領域に収まらない場合に特に役立ちます。その他のほとんどの場合は，ユーザーはキー操作またはマウスのドラッグでテキストを選択します。

書式付き編集コントロールやメモコントロールの内容全体を選択するには，RichEdit1 コントロールの SelectAll メソッドを呼び出します。

次に例を示します。

```
void __fastcall TMainForm::SelectAll(TObject *Sender)
{
  RichEdit1->SelectAll();     // RichEdit 内のすべてのテキストを選択する
}
```

テキストの切り取り，コピー，貼り付け

Clipbrd ユニットを使うアプリケーションでは，クリップボードを介してテキスト，グラフィック，オブジェクトの切り取り，コピー，貼り付けができます。標準テキスト処理コントロールがカプセル化されているすべてのコンポーネントに，クリップボードとのやり取りのためのメソッドが組み込まれています（クリップボードでグラフィックを使用する場合の詳細については 10-21 ページの「グラフィックに対するクリップボードの使い方」を参照してください）。

クリップボードを使ってテキストの切り取り，コピー，貼り付けを行うには，編集コンポーネントの CutToClipboard メソッド，CopyToClipboard メソッド，PasteFromClipboard メソッドをそれぞれ呼び出します。

たとえば，［編集｜切り取り］コマンド，［編集｜コピー］コマンド，［編集｜貼り付け］コマンドの OnClick イベントにそれぞれ次のイベントハンドラを定義します。

```
void __fastcall TMainForm::EditCutClick(TObject* Sender)
{    RichEdit1->CutToClipboard();
}
```

```
void __fastcall TMainForm::EditCopyClick(TObject* Sender)
{    RichEdit1->CopyToClipboard();
}
void __fastcall TMainForm::EditPasteClick(TObject* Sender)
{    RichEdit1->PasteFromClipboard();
}
```

選択したテキストの削除

テキストを切り取ってクリップボードに入れるのでなく，選択したテキストをエディタから削除することもできます。選択したテキストをエディタから削除するには，ClearSelection メソッドを呼び出します。たとえば［編集］メニューに［削除］を追加する場合，次のようなコードを記述できます。

```
void __fastcall TMainForm::EditDeleteClick(TObject *Sender)
{
  RichEdit1->ClearSelection();
}
```

メニュー項目を使用不可にする

メニューコマンドをメニューに残したまま，使用不可にすることがよくあります。たとえば，テキストエディタでテキストを選択していないとき，［切り取り］，［コピー］，［削除］の各コマンドは使用できません。メニュー項目を使用可または使用不可にするタイミングとして適しているのは，ユーザーがメニューを選択したときです。メニュー項目を使用不可にするには，そのメニュー項目の Enabled プロパティに **false** を設定します。

例では，子フォームのメニューバーにある［編集］項目の OnClick イベントに次のイベントハンドラを定義しています。［編集］メニューの［切り取り］，［コピー］，［削除］の各メニュー項目については，RichEdit1 がテキストを選択しているかどうかに基づいて Enabled を設定します。［貼り付け］コマンドについては，クリップボードにテキストがあるかどうかに基づいて使用可能または使用不可にします。

```
void __fastcall TMainForm::EditEditClick(TObject *Sender)
{
  // ［貼り付け］メニュー項目を使用可能または使用不可にする
  Paste1->Enabled = Clipboard()->HasFormat(CF_TEXT);
  bool HasSelection = (RichEdit1->SelLength > 0);  // テキストが選択されている場合は true
  Cut1->Enabled = HasSelection;    // HasSelection が true であれば
                                   // メニュー項目を使用可能にする
  Copy1->Enabled = HasSelection;
  Delete1->Enabled = HasSelection;
}
```

クリップボードの HasFormat メソッドは，クリップボードの内容が特定形式のオブジェクト，テキスト，イメージかどうかに基づいて論理値を返します。CF_TEXT パラメータを使って HasFormat を呼び出すと，クリップボードの内容がテキストかどうかを調べることができ，適宜に［貼り付け］項目を使用可能または使用不可にできます。

クリップボードでグラフィックを使用する場合の詳細については，第 10 章「グラフィックとマルチメディアの処理」を参照してください。

ポップアップメニューの提供

ポップアップまたはローカルメニューは，アプリケーションの操作性を高めるためによく使われる機能です。ポップアップメニューがあると，ユーザーはアプリケーション作業領域内でマウスの右ボタンをクリックするだけで，よく利用するコマンドのリストにアクセスできます。そのためマウスの動きを最小限にできます。

このサンプルでは，たとえば［切り取り］，［コピー］，［貼り付け］の各編集コマンドを繰り返し実行するためのポップアップメニューを追加できます。これらのポップアップメニューの項目では，［編集］メニュー中の対応する項目と同じイベントハンドラが使えます。一般に，対応する標準メニュー項目にはすでにショートカットキーがあるので，ポップアップメニュー用にアクセラレータキーやショートカットキーを作成する必要はありません。

フォームの PopupMenu プロパティは，ユーザーがフォームの項目を右クリックしたときにどのポップアップメニューが表示されるかを指定します。個々のコントロールにも PopupMenu プロパティがあり，これはフォームの PopupMenu プロパティに優先するため，特定のコントロールに合わせてカスタマイズしたメニューが使えます。

ポップアップメニューをフォームに追加する手順は次のとおりです。

1. フォームにポップアップメニューコンポーネントを追加します。
2. メニューデザイナを使ってポップアップメニューの項目を定義します。
3. ポップアップメニューを表示するフォームまたはコントロールの PopupMenu プロパティにポップアップメニューコンポーネントの名前を設定します。
4. ポップアップメニューの項目の OnClick イベントにハンドラを結び付けます。

OnPopup イベントを処理する

ポップアップメニューを表示する前に，標準メニューの項目を使用可能または使用不可にすることがあるのと同じように，ポップアップメニューの項目を調整しなければならないことがあります。標準メニューでは，6-10 ページの「メニュー項目を使用不可にする」で前述したとおり，メニューの最上位にある項目の OnClick イベントを処理します。

しかし，ポップアップメニューでは最上位レベルのメニューバーがないので，ポップアップメニューのコマンドを作成するには，メニューコンポーネント自体のイベントを処理します。ポップアップメニューコンポーネントには，この処理のために OnPopup というイベントがあります。

表示する前にポップアップメニューの項目を調整する手順は次のとおりです。

1. ポップアップメニューコンポーネントを選択します。
2. ポップアップメニューコンポーネントの OnPopup イベントハンドラを作成します。
3. イベントハンドラでコードを記述して，メニュー項目を使用可能，使用不可，あるいは非表示にしたり，表示したりします。

次のコードでは，6-10 ページの「メニュー項目を使用不可にする」で説明した EditClick イベントハンドラをポップアップメニューコンポーネントの OnPopup イベントに結び付けます。次に，ポップアップメニューのそれぞれの項目に対応した 1 行の新しいコードを EditClick に追加します。

```
void __fastcall TMainForm::EditEditClick(TObject *Sender)
{
  // ［貼り付け］メニュー項目を使用可能または使用不可にする
  Paste1->Enabled = Clipboard()->HasFormat(CF_TEXT);
  Paste2->Enabled = Paste1->Enabled;            // この行を追加する
  bool HasSelection = (RichEdit1->SelLength > 0);  // テキストが選択されている場合は true
    Cut1->Enabled = HasSelection;  // HasSelection が true であれば
                                   // メニュー項目を使用可能にする
  Cut2->Enabled = HasSelection;    // この行を追加する
  Copy1->Enabled = HasSelection;
  Copy2->Enabled = HasSelection;   // この行を追加する
  Delete1->Enabled = HasSelection;
}
```

コントロールへのグラフィックの追加

コントロールの中には，コントロールの描画方法をカスタマイズできるものがいくつかあります。具体的には，リストボックス，コンボボックス，メニュー，ヘッダー，タブコントロール，リストビュー，ステータスバー，ツリービュー，ツールバーなどです。標準の描画方法を使ってコントロールや項目を描画するかわりに，コントロールのオーナー（通常はフォーム）が実行時に各項目を描画します。一般に，オーナー描画コントロールを使用するのは，テキストではなくグラフィックを項目に描画する場合や，テキストに加えてグラフィックも描画する場合などです。オーナー描画を使用してメニューにイメージを追加する方法については，8-36 ページの「イメージをメニュー項目に追加する」. を参照してください。

すべてのオーナー描画コントロールは，項目のリストを使用します。項目のリストとは，一般に，テキストとして表示される文字列リストか，テキストとして表示される文字列を含んだオブジェクトリストです。オブジェクトにリスト内の各項目を関連付けて，このオブジェクトを利用して項目を描画できます。

通常，C++Builder でオーナー描画コントロールを作成する手順は次のとおりです。

1. コントロールがオーナー描画であることを示す .
2. 文字列リストへのグラフィックオブジェクトの追加 .
3. オーナー描画項目の描画

コントロールがオーナー描画であることを示す

コントロールの描画をカスタマイズするには，ペイントすべきときにコントロールのイメージを描画するイベントハンドラを追加しなければなりません。こうしたイベントを自動的に受け取るコントロールがいくつかあります。たとえば，リストビュー，ツリービュー，ツールバーの場合，プログラマがプロパティをまったく設定しなくても，描画プロセスのさまざまな段階でイベントを受け取ります。受け取るイベントの名前は，「OnCustomDraw」，「OnAdvancedCustomDraw」などです。

このほかのコントロールについては，あらかじめプロパティを設定しておかないと，オーナー描画イベントを受け取れません。リストボックス，コンボボックス，ヘッダーコントロール，およびステータスバーには，Style プロパティがあります。Style プロパティは，コントロールをデフォルト描画（Standard スタイル）とするかオーナー描画（OwnerDraw スタイル）とするかを指定します。グリッ

ドは DefaultDrawing というプロパティを使って，デフォルト描画を有効または無効にします。リストビューとタブコントロールには，OwnerDraw プロパティがあります。OwnerDraw プロパティもデフォルト描画を有効または無効にします。

リストボックスやコンボボックスのオーナー描画スタイルには，表 6.2 に示すように固定（Fixed）スタイルと可変（Variable）スタイルがあります。また，ほかのコントロールは常に固定スタイルです。テキストを含む項目のサイズは可変にできますが，コントロールを描画する前の各項目のサイズは固定です。

表 6.2　固定と可変のオーナー描画スタイル

オーナー描画スタイル	説明	例
固定	各項目は ItemHeight プロパティの指定する共通の高さになる	lbOwnerDrawFixed, csOwnerDrawFixed
可変	各項目は実行時のデータに応じて高さが変わる	lbOwnerDrawVariable, csOwnerDrawVariable

文字列リストへのグラフィックオブジェクトの追加

各文字列リストは，文字列のリストとは別にオブジェクトのリストも保持できます。

たとえばファイルマネージャアプリケーションであれば，ドライブ文字の横にドライブの種類を示すビットマップを追加します。このためには，アプリケーションにビットマップイメージを追加した後で，次の節で説明しているように，このイメージを文字列リストの適切な場所にコピーします。

アプリケーションへのイメージの追加

イメージコントロールは，ビットマップなどのグラフィックイメージを入れる非ビジュアルコントロールです。イメージコントロールはフォームにグラフィックイメージを表示するのに使います。アプリケーションで使用するイメージを非表示状態で保持するためにも使用できます。たとえば，次の手順でオーナー描画コントロールのビットマップを非表示のイメージコントロールに格納できます。

1. メインフォームにイメージコントロールを追加します。
2. イメージコントロールの Name プロパティに名前を設定します。
3. 各イメージコントロールの Visible プロパティに **false** を設定します。
4. オブジェクトインスペクタから画像エディタを使って，各イメージの Picture プロパティにビットマップを設定します。

イメージコントロールは，アプリケーションを実行しても表示されません。

文字列リストへのイメージの追加

アプリケーションに追加したグラフィックイメージは，文字列リストの文字列と関連付けることができます。グラフィックイメージオブジェクトは，文字列と同時に追加することも，既存の文字列に後から関連付けることもできます。必要なデータがすべてある場合，AddObject メソッドはオブジェクトと文字列を同時に追加します。

次の例では，文字列リストにイメージを追加する方法を示します。これは，ファイルマネージャアプリケーションのサンプルで，各ドライブを示す文字と各ドライブの種類を示すビットマップを同時に追加します。このために，フォームの OnCreate イベントハンドラを次のとおりに更新します。

```
void __fastcall TFMForm::FormCreate(TObject *Sender)
{
  int AddedIndex;
  char DriveName[4] = "A:\\";
  for (char Drive = 'A'; Drive <= 'Z'; Drive++) // すべてのドライブを調べる
  {
    DriveName[0] = Drive;
    switch (GetDriveType(DriveName))
    {
      case DRIVE_REMOVABLE:  // リスト項目を加える
        DriveName[1] = '\0'; // ドライブを示す文字を一時的に文字列にする
        AddedIndex = DriveList->Items->AddObject(DriveName,
          Floppy->Picture->Graphic);
        DriveName[1] = ':'; // コロンに置換する
        break;
      case DRIVE_FIXED:  // リスト項目を加える
        DriveName[1] = '\0'; // ドライブを示す文字を一時的に文字列にする
        AddedIndex = DriveList->Items->AddObject(DriveName,
          Fixed->Picture->Graphic);
        DriveName[1] = ':'; // コロンに置換する
        break;
      case DRIVE_REMOTE:  // リスト項目を加える
        DriveName[1] = '\0'; // ドライブを示す文字を一時的に文字列にする
        AddedIndex = DriveList->Items->AddObject(DriveName,
          Network->Picture->Graphic);
        DriveName[1] = ':' // コロンに置換する
        break;
    }
    if ((int)(Drive - 'A') == getdisk()) // 現在のドライブかどうかチェック
      DriveList->ItemIndex = AddedIndex; // 現在のドライブのリスト項目を選択状態にする
  }
}
```

オーナー描画項目の描画

あるコントロールがオーナー描画であることを示す（プロパティを設定するか，カスタム描画イベントハンドラを作成する）と，それ以降，コントロールは画面に描画されません。かわりに，オペレーティングシステムがイベントをコントロールの表示項目ごとに生成します。アプリケーションでこのイベントを処理し，各項目を描画します。

オーナー描画コントロールの項目を描画するには，コントロールに表示されている各項目に対して次のことを行います。すべての項目は共通な 1 つのイベントハンドラを使用します。

1. 必要な場合，オーナー描画項目をサイズ変更します。

   同じサイズの項目であればサイズ変更の必要はありません。たとえば，リストボックスでスタイルが lsOwnerDrawFixed である場合などです。

2. オーナー描画項目を描画します。

オーナー描画項目のサイズ変更

アプリケーションで可変オーナー描画コントロールに各項目を描画する前に，オペレーティングシステムは項目サイズを取得するイベントを生成します。このイベントによって，アプリケーションはコントロール上に項目を表示する位置を知ることができます。

C++Builder は，項目のテキストを現在のフォントで表示できるサイズに決定します。アプリケーションはイベントを処理して，選択された長方形のサイズを変更できます。たとえば，テキストのかわりにビットマップを項目に表示する場合，ビットマップのサイズに合わせてサイズを変更します。ビットマップとテキストの両方を表示する場合は，両方を合わせた長方形のサイズにします。

オーナー描画項目のサイズを変更するには，オーナー描画コントロールの項目のサイズを取得するイベントに対して，イベントハンドラを定義します。イベントの名前はオーナー描画コントロールによって異なります。リストボックスとコンボボックスには，OnMeasureItem を使います。グリッドにはサイズ取得イベントはありません。

サイズ取得イベントには，2 つの重要なパラメータがあります。項目のインデックス番号と項目のサイズです。サイズは可変です。アプリケーションでサイズを小さくすることも，大きくすることもできます。先行する項目のサイズによって，後続の項目の位置が決まります。

たとえば，可変オーナー描画リストボックスで最初の項目の高さを 5 ピクセルに設定すると，2 番目の項目は上端から 6 ピクセル目に配置されます。以下同様に続いていきます。リストボックスとコンボボックスの場合，アプリケーションで変更できるのは項目の高さだけです。項目の幅は常にコントロールの幅になります。

オーナー描画グリッドでは，描画時にセルのサイズを変更することはできません。各行と各列のサイズは，ColWidths と RowHeights の両プロパティで描画前に設定します。

次の例では，オーナー描画タブの OnMeasureItem イベントに対してハンドラを割り当て，タブ項目に関連付けたビットマップを表示するための必要な幅を設定しています。

```
void __fastcall TForm1::ListBox1MeasureItem(TWinControl *Control, int Index,
  int &Height)      // Height が参照を使って渡されることに注意
{
  int BitmapHeight = ((TBitmap *)ListBox1->Items->Objects[Index])->Height + 2;
  // リスト項目にビットマップ用の十分な余裕（2 ピクセル分）があるか確認
  if (BitmapHeight > Height)
    Height = BitmapHeight;
}
```

メモ 文字列リストの Objects プロパティの項目は目的の型にキャストしなければ使うことができません。このオブジェクトは，TObject 型のプロパティであり，このプロパティはすべての種類のオブジェクトを保持することができます。配列からオブジェクトを検索するときは，実際に使う元の型にキャストし直す必要があります。

オーナー描画項目の描画

オーナー描画コントロールをアプリケーションで描画または再描画しなければならないとき，オペレーティングシステムは，コントロールに表示されている項目ごとに項目を描画するイベントを生成

します。対象となるコントロールによって，項目全体の描画イベントを受け取る場合と，下位項目の描画イベントを受け取る場合があります。

オーナー描画コントロールの各項目を描画するには，このコントロールの項目を描画するイベントに対して，イベントハンドラを定義します。

一般に，オーナー描画イベントの名前は次のどちらかから始まります。

- OnDraw（OnDrawItem，OnDrawCell など）
- OnCustomDraw（OnCustomDrawItem など）
- OnAdvancedCustomDraw（OnAdvancedCustomDrawItem など）

項目描画イベントは，描画する項目と描画領域の長方形をパラメータとして持っています。さらに，項目にフォーカスがあるかどうかなど，項目の状態を示すパラメータも持ちます。アプリケーションはイベントを処理して，指定の長方形に適切な項目を描画します。

たとえば次のコードは，タブにビットマップと文字列をどのように描画するかを示しています。このハンドラをタブセットの OnDrawItem イベントとして定義します。

```
void __fastcall TForm1::ListBox1DrawItem(TWinControl *Control, int Index,
  TRect &Rect, TOwnerDrawState State)
{
  TBitmap *Bitmap = (TBitmap *)ListBox1->Items->Objects[Index];
  ListBox1->Canvas->Draw(R.Left, R.Top + 2, Bitmap); // ビットマップを描画する
  ListBox1->Canvas->TextOut(R.Left + Bitmap->Width + 2, R.Top + 2,
    ListBox1->Items->Strings[Index]);        // テキストを右側に書き込む
}
```

第 7 章

アプリケーション，コンポーネント，ライブラリの構築

この章ではアプリケーション，ライブラリおよびコンポーネントを作成するための C++Builder の使い方について説明します。

アプリケーションの作成

C++Builder は，主に以下のアプリケーションの設計と構築を目的としています。

- GUI アプリケーション
- コンソールアプリケーション
- サービスアプリケーション（Windows アプリケーションのみ）
- パッケージと DLL

一般に，GUI アプリケーションは使いやすいインターフェースを持っています。コンソールアプリケーションはコンソールウィンドウから実行します。サービスアプリケーションは，Windows サービスとして実行されます。これらのアプリケーションは，起動コードと一緒に実行形式ファイルとしてコンパイルされます。

パッケージ，DLL といった別のタイプのプロジェクトを作成することもできます。この場合，パッケージファイルまたはダイナミックリンクライブラリが作成されます。DLL やパッケージの場合，起動コードのない実行形式コードが生成されます。7-10 ページの「パッケージと DLL の作成」を参照してください。

GUI アプリケーション

GUI（グラフィカルユーザーインターフェース）アプリケーションとは，ウィンドウ，メニュー，ダイアログボックスなどのグラフィカルな機能を使って設計されたアプリケーションです。アプリケーションを使いやすくしている点が特徴です。GUI アプリケーションをコンパイルすると，起動コー

ドを持つ実行形式ファイルが作成されます。実行形式ファイルは通常，プログラムの基本的な機能を備えており，簡単なプログラムの場合には 1 つの実行形式ファイルだけで構成されていることもあります。実行形式ファイルから DLL，パッケージ，およびその他のサポートファイルを呼び出すことによってアプリケーションを拡張できます。

C++Builder は次の 2 つの UI モデルを提供します。

- シングルドキュメントインターフェース（SDI：Single document interface）
- マルチドキュメントインターフェース（MDI：Multiple document interface）

アプリケーションの実装モデルに加えて，プロジェクトの設計時の動作とアプリケーションの実行時の動作も IDE でプロジェクトオプションを設定することによって操作できます。

ユーザーインターフェースモデル

どのフォームも，マルチドキュメントインターフェース（MDI）フォームまたはシングルドキュメントインターフェース（SDI）フォームとして実装できます。MDI アプリケーションでは，1 つの親ウィンドウ内で複数のドキュメントや子ウィンドウを開くことができます。これは表計算またはワードプロセッサなどのアプリケーションで一般的です。それとは対照的に，SDI アプリケーションは通常 1 つのドキュメントしか開けません。フォームを SDI アプリケーションにするには，Form オブジェクトの FormStyle プロパティを fsNormal に設定します。

アプリケーションの UI の作成については，第 8 章「アプリケーションユーザーインターフェースの作成」を参照してください。

SDI アプリケーション

SDI アプリケーションの作成手順は以下のとおりです。

1. ［ファイル｜新規作成｜その他］を選択して，［新規作成］ダイアログを開きます。
2. ［プロジェクト］ページをクリックして［SDI アプリケーション］をダブルクリックします。
3. ［OK］を選択します。

デフォルトでは Form オブジェクトの FormStyle プロパティは fsNormal に設定されるので，C++Builder はすべての新しいアプリケーションは SDI アプリケーションであると想定します。

MDI アプリケーション

MDI アプリケーションの作成手順は以下のとおりです。

1. ［ファイル｜新規作成｜その他］を選択して，［新規作成］ダイアログを開きます。
2. ［プロジェクト］ページをクリックして［MDI アプリケーション］をダブルクリックします。
3. ［OK］を選択します。

MDI アプリケーションは SDI アプリケーションよりも綿密な計画が必要とされ，設計が複雑になります。MDI アプリケーションからクライアントウィンドウ内に常駐する子ウィンドウが作成されます。メインフォームには子フォームが含まれます。フォームが子フォーム（fsMDIChild）またはメインフォーム（fsMDIForm）なのかを指定するには，TForm オブジェクトの FormStyle プロパティを設定します。子フォームの基本クラスを定義し，各子フォームをこの基本クラスから派生させると，子フォームのプロパティを再設定する必要がなくなります。

MDI アプリケーションには，メインメニューのウィンドウポップアップとして，複数のウィンドウの表示方法を指定する［重ねて表示］や［並べて表示］などの項目が含まれていることがよくあります。子ウィンドウが最小化されると，そのアイコンは MDI 親フォーム内に配置されます。

MDI アプリケーションの作成に必要な作業をまとめると次のようになります。

1. メインウィンドウフォームまたは MDI 親ウィンドウを作成します。その FormStyle プロパティを fsMDIForm に設定します。
2. ［ファイル｜開く］，［ファイル｜上書き保存］，および［ウィンドウ］を含むメニューを作成します（［ウィンドウ］には［重ねて表示］，［並べて表示］，［アイコンの整列］を入れます）。
3. MDI 子フォームを作成し，その FormStyle プロパティを fsMDIChild に設定します。

IDE，プロジェクト，およびコンパイルオプションの設定

プロジェクトに対するさまざまなオプションを指定するには，［プロジェクト｜オプション］を選択します。詳細については，オンラインヘルプを参照してください。

デフォルトプロジェクトオプションの設定

今後作成されるプロジェクトすべてに適用されるデフォルトオプションを変更するには，［プロジェクトオプション］ダイアログボックスのオプションを設定し，ウィンドウの左下にある［デフォルト］ボックスをチェックします。これで，新しいプロジェクトはすべて，現在選択しているオプションをデフォルトで使用します。

プログラミングテンプレート

プログラミングテンプレートとは，コーディングに共通して使われるスケルトン構造のことです。ソースコードにテンプレートを挿入してから，必要なコードを記述します。C++Builder には，配列宣言，クラス宣言，関数宣言などの標準テンプレートに加え，多くの文が用意されています。

よく使うコーディング構造のテンプレートを自分で作成することもできます。たとえば **for** ループを使うには，以下のテンプレートをコードに挿入します。

```
for (; ;)
{

}
```

コードエディタにテンプレートを挿入するには，〔Ctrl〕+〔J〕を押し，使用したいテンプレートを選択します。独自のテンプレートを追加することもできます。テンプレートを追加する手順は，次のとおりです。

1. ［ツール｜エディタオプション］を選択します。
2. ［支援機能］タブをクリックします。
3. ［コードテンプレート］セクションで［追加］をクリックします。
4. ［名前］の後ろにテンプレート名を入力し，新しいテンプレートの簡単な説明を入力して，［OK］をクリックします。
5. ［コード］テキストボックスにテンプレートのコードを追加します。

6. ［OK］を選択します。

コンソールアプリケーション

コンソールアプリケーションは，32 ビットの Windows プログラムで，通常はグラフィカルインターフェースなしでコンソールウィンドウ内で実行されます。これらのアプリケーションは一般的にはユーザーからの入力をあまり必要とせず，限られた機能で実行されます。

コンソールアプリケーションの作成手順は以下のとおりです。

1. ［ファイル｜新規作成｜その他］を選択し，［新規作成］ダイアログの［コンソールウィザード］をダブルクリックします。
2. ［コンソールウィザード］ダイアログボックスが表示されます。［コンソールアプリケーション］をチェックし，プロジェクトのメインモジュール用の［ソースの種類］（C または C++）を選びます。必要に応じて，main 関数または WinMain 関数を含む既存ファイルを指定します。［OK］をクリックします。

すると，このタイプのソースファイル用のプロジェクトファイルが作成され，コードエディタが表示されます。

コンソールアプリケーションでの VCL と CLX の使用

メモ 新しいコンソールアプリケーションを作成するとき，IDE は新しいフォームを作成しません。コードエディタだけが表示されます。

しかし，コンソールアプリケーションでは VCL オブジェクトと CLX オブジェクトを使用できます。このためには，VCL または CLX を使うことをコンソールウィザードで指定する必要があります（［VCL を使う］か［CLX を使う］をチェックする）。ウィザードで VCL か CLX を使用するように指定しないと，後から VCL/CLX を使用することはできません。指定せずに VCL/CLX を使おうとするとリンカエラーが発生します。

コンソールアプリケーションでは，実行中にウィンドウがダイアログを表示しないようにするために，すべての例外を処理する必要があります。

サービスアプリケーション

サービスアプリケーションは，クライアントから要求を受け，処理し，結果を返します。一般的に，サービスアプリケーションはバックグラウンドで実行され，ユーザーの入力を直接受けることはほとんどありません。Web/FTP サーバーや電子メールサーバーなどがサービスアプリケーションの良い例です。

Win32 サービスを実装するアプリケーションを作成するには，次のようにします。

1. ［ファイル｜新規作成｜その他］を選択し，［新規作成］ダイアログの［サービスアプリケーション］をダブルクリックします。すると，プロジェクトに TServiceApplication 型の Application というグローバル変数が追加されます。

2. サービス（TService）に対応するサービスウィンドウが表示されます。オブジェクトインスペクタにプロパティとイベントハンドラを設定して，サービスを実装します。
3. ［ファイル｜新規作成｜その他］を選択すると，サービスアプリケーションにサービスを追加できます。［新規作成］ダイアログボックスで［サービス］をダブルクリックします。サービスアプリケーションでないアプリケーションにサービスを追加しないでください。サービスアプリケーションでないアプリケーションにも TService オブジェクトを追加することはできますが，アプリケーションはサービスが必要とするイベントを生成しません。また，サービスを呼び出すこともありません。

サービスアプリケーションを作成したら，/INSTALL オプションを付けて実行し，サービスをインストールします。アプリケーションはサービスのインストールを開始し，インストールが完了したかどうかの確認メッセージを表示した後，終了します。この確認メッセージを抑止するには，/SILENT オプションを使ってサービスアプリケーションを実行します。

サービスの登録を解除するには，コントロールパネルから［停止］ボタンでサービスを停止し，サービスアプリケーションを /UNINSTALL オプションを付けて実行します（アンインストール時も，/SILENT オプションを使うと確認メッセージを抑止できます）。

例 このサービスではポート番号 80 を使う TServerSocket を作成します。このポートは，Web ブラウザと Web サーバーが対話する際の一般的なポート番号です。この例では，WebLogxxx.log（xxx は ThreadID）というファイルが C: \ Temp ディレクトリに作られます。任意のポート番号に対して監視するサーバーは 1 つだけなので，すでに Web サーバーを実行している場合には既存の Web サーバーを停止してください。

コントロールパネルの［サービス］ダイアログからサービス名を探し，［開始］ボタンを押します。［状態］欄に［開始］と表示されたら，サービスとして起動していることになります。

準備が整ったら，ローカルマシン上でブラウザを開き，アドレスに「localhost」と入力します。しばらくするとブラウザはタイムアウトしますが，C: \ Temp ディレクトリに weblogxxx.log が作成されています。

1. サンプルを作成するには，［ファイル｜新規作成｜その他］を選択し，［新規作成］ダイアログの［サービスアプリケーション］を選択します。Service1 という名前のウィンドウが表示されます。
2. コンポーネントパレットの［Internet］ページから，ServerSocket コンポーネントをサービスウィンドウ（Service1）に追加します。
3. TMemoryStream 型のプライベートデータメンバーを TService1 クラスに追加します。ユニットのヘッダーは次のようになります。

```
//---------------------------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Classes.hpp>
#include <SvcMgr.hpp>
#include <ScktComp.hpp>
//---------------------------------------------------------------------------
class TService1 : public TService
{
```

```
__published:  // IDE 管理コンポーネント
        TServerSocket *ServerSocket1;
private:      // ユーザー宣言
        TMemoryStream *Stream; // ここに，この行を追加する
public:       // ユーザー宣言
   __fastcall TService1(TComponent* Owner);
   PServiceController __fastcall GetServiceController(void);

   friend void __stdcall ServiceController(unsigned CtrlCode);
};
//---------------------------------------------------------------------------
extern PACKAGE TService1 *Service1;
//---------------------------------------------------------------------------
#endif
```

4. 手順 1 で追加した ServerSocket1 コンポーネントを選択します。オブジェクトインスペクタで OnClientRead イベントをダブルクリックし，次のイベントハンドラを追加します。

```
void __fastcall TService1::ServerSocket1ClientRead(TObject *Sender,
      TCustomWinSocket *Socket)
{
  char *Buffer = NULL;
  int len = Socket->ReceiveLength();
  while (len > 0)
  {
    try
    {
      Buffer = (char *)malloc(len);
      Socket->ReceiveBuf((void *)Buffer, len);
      Stream->Write(Buffer, len);
    }
    __finally
    {
      free(Buffer);
    }
    Stream->Seek(0, soFromBeginning);
    AnsiString LogFile = "C:\\Temp\\WebLog";
    LogFile = LogFile + IntToStr(ServiceThread->ThreadID) + ".log";
    Stream->SaveToFile(LogFile);
  }
}
```

5. 最後に，ウィンドウのクライアント領域（[ServiceSocket] 以外の領域）をクリックして Service1 を選択します。オブジェクトインスペクタで OnExecute イベントをダブルクリックし，次のイベントハンドラを追加します。

```
void __fastcall TService1::Service1Execute(TService *Sender)
{
  Stream = new TMemoryStream();
  try
  {
    ServerSocket1->Port = 80; // WWW ポート
    ServerSocket1->Active = true;
    while (!Terminated)
       ServiceThread->ProcessRequests(true);
    ServerSocket1->Active = false;
  }
  __finally
```

```
  {
    delete Stream;
  }
}
```

サービスアプリケーションを作成するときは，以下の点に注意してください。

- サービススレッド
- サービス名プロパティ
- サービスアプリケーションのデバッグ

メモ サービスアプリケーションは Windows 専用です。

サービススレッド

各サービスはそれぞれ個別のサービススレッド（TServiceThread）を 1 つ持ちます。そのため，サービスアプリケーションが複数のサービスを実装する場合には，各サービスはスレッドセーフとして実装されていなければなりません。サービスで行いたい処理を TService の OnExecute イベントハンドラに記述する場合に，サービススレッドは使われます。各サービススレッドは個別の Execute メソッドを持ちます。サービススレッドの Execute メソッドからサービスの OnStart や OnExecute ハンドラが呼び出されます。

サービス要求に時間がかかる場合やサービスアプリケーションが複数のクライアントから同時に多数の要求を受ける可能性のある場合などは，サービススレッドの代わりに，TThread から派生させた独自のスレッドを要求ごとに作成し，そのスレッドの Execute メソッドに実行コードを記述するほうが効果的です。こうすることで OnExecute ハンドラがすぐに終了するため，サービススレッドの Execute ループが新しい要求を速やかに受け付けることができるようになります。以下に例を示します。

例 このサービスでは 標準のスレッドを直接使って 500 ミリ秒ごとに警告音を鳴らします。サービスは一時停止，再開，終了を要求されたとき，スレッドの一時停止，再開，終了を処理します。

1. ［ファイル｜新規作成｜その他］を選択し，［新規作成］ダイアログボックスの［サービスアプリケーション］をダブルクリックします。「Service1」という名前のウィンドウが表示されます。
2. ユニットのヘッダーファイルで，TSparkyThread という名前の TThread の派生クラスを宣言します。これがサービスのための作業を実行するスレッドです。宣言は次のようになります。

```
class TSparkyThread : public TThread
{
private:
protected:
        void __fastcall Execute();
public:
        __fastcall TSparkyThread(bool CreateSuspended);
};
```

3. ユニットの .cpp ファイルで TSparkyThread のインスタンスで使用するグローバル変数を作成します。

```
TSparkyThread *SparkyThread;
```

4. TSparkyThread コンストラクタの .cpp ファイルに次のコードを追加します。

```
__fastcall TSparkyThread::TSparkyThread(bool CreateSuspended)
        : TThread(CreateSuspended)
{
}
```

5. TSparkyThread の Execute メソッド（スレッド関数）の .cpp ファイルに次のコードを追加します。

```
void __fastcall TSparkyThread::Execute()
{
  while (!Terminated)
  {
    Beep();
    Sleep(500);
  }
}
```

6. Service ウィンドウ（Service1）を選択し，オブジェクトインスペクタで OnStart イベントをダブルクリックします。次の OnStart イベントハンドラを追加します。

```
void __fastcall TService1::Service1Start(TService *Sender, bool &Started)
{
  SparkyThread = new TSparkyThread(false);
  Started = true;
}
```

7. オブジェクトインスペクタで OnContinue イベントをダブルクリックします。次の OnContinue イベントハンドラを追加します。

```
void __fastcall TService1::Service1Continue(TService *Sender, bool &Continued)
{
  SparkyThread->Resume();
  Continued = true;
}
```

8. オブジェクトインスペクタで OnPause イベントをダブルクリックします。次の OnPause イベントハンドラを追加します。

```
void __fastcall TService1::Service1Pause(TService *Sender, bool &Paused)
{
  SparkyThread->Suspend();
  Paused = true;
}
```

9. 最後にオブジェクトインスペクタで OnStop イベントをダブルクリックし，次の OnStop イベントハンドラを追加します。

```
void __fastcall TService1::Service1Stop(TService *Sender, bool &Stopped)
{
  SparkyThread->Terminate();
  Stopped = true;
}
```

サーバー用途のサービスアプリケーション開発する場合に，独立したスレッドを作成すべきかどうかは，提供するサービスの性質，予想される接続数，サービスを実行するコンピュータ上のプロセッサの数によって決めなければなりません。

サービス名プロパティ

VCL には，Windows で動作するサービスアプリケーションを作成するためのクラスがいくつかあります（クロスプラットフォームアプリケーションでは使用できません）。サービスアプリケーションを作成する主なクラスは TService と TDependency です。これらのクラスを使う場合，種類の異なる Name プロパティが複数あるため紛らわしくなることがあります。このセクションでは，それぞれの Name プロパティの違いについて示します。

サービスに関連する名前には，サービススタート名と呼ばれるユーザーアカウント名とパスワード名，マネージャやエディタウィンドウ内に表示される表示名，そしてサービスの実際の名前があります。依存関係はサービス間であることもあり，またロードされるグループの順番に依存している場合もあります。サービス間の依存関係を定義する場合にも，実名と表示名がある場合があります。また，サービスオブジェクトは TComponent を祖先に持つため，Name プロパティを継承します。さまざまな Name プロパティについて以下に概略を示します。

TDependency のプロパティ

TDependency の DisplayName はサービスの表示名と実名の両方を表します。ほとんどの場合，DisplayName は TDependency の Name プロパティと同じになります。

TService のプロパティ

TService の Name プロパティは TComponent から継承されたものです。この Name プロパティはコンポーネント名であり，またサービス名でもあります。サービス間の依存関係を定義するために，このプロパティは TDependency の Name と DisplayName で使われます。

TService の DisplayName はコントロールパネルの［サービス］ダイアログで表示されるサービス一覧に表示される名前です。ユーザーの目に直接触れるためにサービスの内容を説明する文章が好まれ，実際のサービス名（TService::Name，TDependency::DisplayName，TDependency::Name）とは異なる場合が一般的です。特に TDependency および TService の DisplayName とは異なる場合が多くなっています。

ServiceStartName プロパティはサービススタート名です。コントロールパネルの［サービス］ダイアログの［スタートアップ］ボタンで表示されるダイアログで指定するユーザーアカウント名です。

サービスアプリケーションのデバッグ

サービスアプリケーションのデバッグは，実行中にそのプロセスにアタッチする（つまり，まずサービスを起動してから，デバッガにアタッチする）ことによって行えます。サービスアプリケーションのプロセスにアタッチするには，［実行｜プロセスにアタッチ］を選択してダイアログを表示し，目的のサービスアプリケーションを選択します。

この方法を使った場合，権利が不十分であるために失敗することがあります。その場合は，次のようにサービスコントロールマネージャ（SCM）を使ってデバッガを操作することができます。

1. まず，次のレジストリ位置に「Image File Execution Options」という名前のキーを作成します。

   ```
   HKEY_LOCAL_MACHINE \ SOFTWARE \ Microsoft \ Windows NT \ CurrentVersion
   ```

2. 作成したサービスと同じ名前（MYSERV.EXE など）のサブキーを作成します。このサブキーに，Debugger という名前の REG_SZ 型の値を追加します。文字列値として BCB.exe までのフルパスを使用してください。
3. ［サービス］コントロールパネルアプレットで目的のサービスを選択し，［スタートアップ］をクリックして，［デスクトップとの対話をサービスに許可］をチェックします。

Windows NT システムでは，サービスアプリケーションをデバッグする方法がもう 1 つあります。ただし，この方法には多少注意が必要です。それは，短時間のインターバルで操作する必要があるからです。

1. まず，デバッガで目的のサービスアプリケーションを開始します。読み込みが完了するまで数秒待ちます。
2. この後，コントロールパネルまたは次のコマンドラインですぐにサービスを開始させます。
   ```
   start MyServ
   ```

サービスが開始されないとアプリケーションが終了してしまうため，すぐに（アプリケーションの起動後 15 ～ 30 秒以内）にサービスを起動する必要があります。

パッケージと DLL の作成

ダイナミックリンクライブラリ（DLL）は，実行形式ファイルと連携して動作するコンパイル済みのモジュールで，アプリケーションに機能を提供します。クロスプラットフォームプログラムでも DLL を作成できます。ただし Linux 上では，DLL を共有オブジェクトとして再コンパイルします（パッケージも同様です）。

パッケージとは C++Builder アプリケーションと IDE の両方で使用される特別な DLL です。パッケージには，実行時パッケージと設計時パッケージの 2 種類があります。実行時パッケージは，プログラムの実行中にプログラムに機能を提供します。設計時パッケージは IDE の機能を拡張します。

DLL とライブラリは，Windows ダイアログでエラーや警告が表示されるのを防ぐために，すべての例外を処理する必要があります。

ライブラリプロジェクトファイルに挿入できるコンパイラ指令を下の表に示します。

表 7.1　ライブラリ用のコンパイラ指令

コンパイラ指令	説明
{$LIBPREFIX 'string'}	出力ファイル名に先頭文字を追加する。たとえば，設計時パッケージに対して {$LIBPREFIX 'dcl'} を指定できる。{$LIBPREFIX ' '} を使うと，先頭文字全体が削除される
{$LIBSUFFIX 'string'}	出力ファイル名の拡張子の前に末尾文字を追加する。たとえば，something.cpp で {$LIBSUFFIX '-2.1.3'} を使うと，something-2.1.3.bpl となる
{$LIBVERSION 'string'}	出力ファイル名の拡張子 .bpl の後ろに第 2 拡張子を追加する。たとえば，something.cpp で {$LIBVERSION '2.1.3'} を使うと，something.bpl.2.1.3 となる

パッケージの詳細については，第 15 章「パッケージとコンポーネントの操作」を参照してください。

パッケージと DLL をいつ使用するか

C++Builder で記述されているほとんどのアプリケーションでは，パッケージは DLL よりも柔軟性が高く，作成が簡単です。しかし，以下の場合には DLL の方がパッケージよりもプロジェクトに適しています。

- コードモジュールが C++Builder 以外のアプリケーションから呼び出される場合
- Web サーバーの機能を拡張している場合
- サードパーティの開発者が使用するコードモジュールを作成している場合
- プロジェクトが OLE コンテナである場合

DLL ファイル間で RTTI（実行時型情報）を渡したり，DLL から実行形式ファイルに RTTI を渡すことはできません。理由は，どの DLL も独自のシンボル情報を保持しているからです。**is** 演算子または **as** 演算子を使って DLL から TStrings オブジェクトを渡す必要がある場合には，DLL でなくパッケージを作成します。パッケージの場合，パッケージ間でシンボル情報を共有します。

C++Builder での DLL の使用

通常の C++ アプリケーションと同じように，C++Builder アプリケーションでも Windows の DLL を使用できます。

C++Builder アプリケーションをロードするときに DLL を静的に読み込むには，その DLL 用のインポートライブラリファイルを C++Builder アプリケーションにリンクします。インポートライブラリを C++Builder アプリケーションに追加するには，［プロジェクト｜プロジェクトに追加］を選択し，追加したい .LIB ファイルを指定します。

これにより，その DLL のエクスポートされた関数がアプリケーションで使用可能となります。**__declspec** (dllimport) 修飾子を使用して，アプリケーションで使用する DLL 関数のプロトタイプを作ります。

```
__declspec(dllimport) return_type imported_function_name(parameters);
```

C++Builder アプリケーションの実行中に DLL を動的にロードするには，静的に読み込むときと同様にインポートライブラリを追加し，［プロジェクト｜オプション］の［リンカ（詳細）］タブで遅延ロードオプションを設定します。Windows API 関数の LoadLibrary() を使って DLL を読み込み，次に API 関数 GetProcAddress() を使用して必要な各関数のポインタを取得することもできます。

DLL の使い方の詳細は，Microsoft Win32 SDK Reference を参照してください。

C++Builder での DLL の作成

C++Builder での DLL の作成方法は，標準的な C++ での作成方法と同じです。

1. ［ファイル｜新規作成｜その他］を選択して［新規作成］ダイアログボックスを表示します。
2. ［DLL ウィザード］をダブルクリックします。
3. メインモジュールの［ソースの種類］（C または C++）を選びます。
4. DLL のエントリポイントを MSVC++ スタイルの DllMain にしたい場合は，［VC++ スタイルの DLL］をチェックします。これをチェックしなければ，エントリポイントに DllEntryPoint が使われます。
5. VCL コンポーネントか CLX コンポーネントを含む DLL を作成するには，［VCL を使う］と［CLX を使う］のどちらかをチェックします。ただし，［C++］を選択しておかないと，このオプションは使用できません。
   7-12 ページの「VCL/CLX コンポーネントを含む DLL の作成」を参照してください。
6. DLL をマルチスレッドにしたい場合は，［マルチスレッド］をチェックします。

7. ［OK］を選択します。

コード内のエクスポートされる関数には，Borland C++ または Microsoft Visual C++ の場合と同様に **__declspec** (dllexport) 修飾子が必要です。たとえば，次のコードは C++Builder および他の Windows 用 C++ コンパイラで有効です。

```
// MyDLL.cpp
double dblValue(double);
double halfValue(double);
extern "C" __declspec(dllexport) double changeValue(double, bool);

double dblValue(double value)
{
   return value * value;
};

double halfValue(double value)
{
   return value / 2.0;
}

double changeValue(double value, bool whichOp)
{
   return whichOp ? dblValue(value) : halfValue(value);
}
```

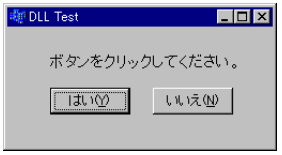
上記のコードでは，関数 changeValue がエクスポートされ，呼び出し側アプリケーションで利用できます。dblValue と halfValue 関数は，内部関数なので，DLL の外部から呼び出すことはできません。

DLL の作成に関する詳細は，Microsoft Win32 SDK Reference を参照してください。

VCL/CLX コンポーネントを含む DLL の作成

DLL の利点の 1 つは，ある開発ツールで作成した DLL を，別の開発ツールで書いたアプリケーションで使用できることです。DLL が，呼び出し元のアプリケーションが使用する VCL/CLX コンポーネント（たとえばフォーム）を含む場合には，標準の呼び出し規則を使用するエクスポートインターフェースルーチンを提供する必要があります。ただし，C++ の命名規則に違反しないようにしてください。また，呼び出し元アプリケーションからは VCL/CLX ライブラリのみでサポートされている機能を直接要求しないでください。エクスポート可能な VCL/CLX コンポーネントを作成するには，実行時パッケージを使用します。詳細については，第 15 章「パッケージとコンポーネントの操作」を参照してください。

たとえば，次のような簡単なダイアログボックスを表示させる DLL を作成する場合



ダイアログボックス DLL のコードは次のようになります。

```
// DLLMAIN.H
//---------------------------------------------------------------------------
#ifndef dllMainH
#define dllMainH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
class TYesNoDialog : public TForm
{
__published:    // IDE 管理コンポーネント
   TLabel *LabelText;
   TButton *YesButton;
   TButton *NoButton;
   void __fastcall YesButtonClick(TObject *Sender);
   void __fastcall NoButtonClick(TObject *Sender);
private:        // ユーザー宣言
   bool returnValue;
public:         // ユーザー宣言
   virtual __fastcall TYesNoDialog(TComponent *Owner);
   bool __fastcall GetReturnValue();
};

// エクスポートされたインターフェース関数
extern "C" __declspec(dllexport) bool InvokeYesNoDialog();

//---------------------------------------------------------------------------
 extern TYesNoDialog *YesNoDialog;
//---------------------------------------------------------------------------
#endif


// DLLMAIN.CPP
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "dllMain.h"
//---------------------------------------------------------------------------
#pragma resource "*.dfm"
TYesNoDialog *YesNoDialog;
//---------------------------------------------------------------------------
__fastcall TYesNoDialog::TYesNoDialog(TComponent *Owner)
 : TForm(Owner)
{
  returnValue = false;
}
//---------------------------------------------------------------------------
void __fastcall TYesNoDialog::YesButtonClick(TObject *Sender)
{
   returnValue = true;
   Close();
}
//---------------------------------------------------------------------------
void __fastcall TYesNoDialog::NoButtonClick(TObject *Sender)
```

```
{
   returnValue = false;
   Close();
}
//---------------------------------------------------------------------
bool __fastcall TYesNoDialog::GetReturnValue()
{
   return returnValue;
}
//---------------------------------------------------------------------
// エクスポートされた VCL を呼び出す標準 C++ インターフェース関数
bool InvokeYesNoDialog()
{
   bool returnValue;
   TYesNoDialog *YesNoDialog = new TYesNoDialog(NULL);
   YesNoDialog->ShowModal();
   returnValue = YesNoDialog->GetReturnValue();
   delete YesNoDialog;
   return returnValue;
}
//---------------------------------------------------------------------
```

この例にあるコードは，ダイアログを表示し，「Yes」ボタンが選択されるとプライベートデータメンバー returnValue に値 **true** を格納します。他のボタンが選択されると戻り値は **false** になります。パブリックの GetReturnValue() 関数が，returnValue の現在値を取り出します。

ダイアログを呼び出し，どのボタンが押されたか調べるときには，呼び出し元のアプリケーションがエクスポート関数 InvokeYesNoDialog() を呼び出します。この関数は，C++ の命名規則を守るための C リンクと，標準的な C の呼び出し規則を使用するエクスポート関数として DLLMAIN.H で宣言されます。この関数は DLLMAIN.CPP で定義されます。

C の標準関数を DLL とのインターフェースとして使用すると，C++Builder で作られたかどうかに関係なく，どの呼び出し元のアプリケーションでも DLL を使用できます。ダイアログを作成するのに必要な VCL/CLX 機能は，DLL そのものにリンクされますが，呼び出し元のアプリケーションは，VCL/CLX の機能を知る必要はありません。

VCL か CLX を使用する DLL を作成すると，必要な VCL コンポーネントまたは CLX コンポーネントが DLL にリンクされ，ある程度のオーバーヘッドが生じます。このオーバーヘッドがアプリケーションの全体のサイズに与える影響を最小限に抑えるには，数個のコンポーネントを 1 つの DLL に組み込みます。これにより，DLL が使用する VCL/CLX サポートコンポーネントのコピーは 1 つだけになります。

DLL のリンク

［プロジェクトオプション］ダイアログの［リンカ］ページで DLL 用のリンカオプションを設定できます。このページのデフォルトのチェックボックスでも，DLL 用のインポートライブラリを作成できます。コマンドラインからコンパイルするときは，-Tpd スイッチを使用してリンカ（ILINK32.EXE）を起動します。次に例を示します。

```
ilink32 /c /aa /Tpd c0d32.obj mydll.obj, mydll.dll, mydll.map, import32.lib cw32mt.lib
```

インポートライブラリが必要であれば，-Gi スイッチも使用するとインポートライブラリが生成されます。

コマンドラインユーティリティ IMPLIB.EXE を使用してもインポートライブラリを作成できます。以下に例を示します。

```
implib mydll.lib mydll.dll
```

DLL をリンクするためのさまざまなオプションについての詳細，および，それを静的または動的に実行時ライブラリにリンクされる他のモジュールと一緒に使う方法については，オンラインヘルプを参照してください。

データベースアプリケーションの作成

C++Builder の利点の 1 つは，拡張データベースアプリケーションの作成をサポートしていることです。C++Builder がサポートしているツールを使うと，SQL サーバーおよび各種データベース（Oracle，Sybase，InterBase，MySQL，MS-SQL，Informix，DB2 など）と接続できます。同時に，アプリケーション間で意識せずにデータの共有ができます。

C++Builder には，データベースにアクセスするコンポーネントや，データベース内の情報を表示するコンポーネントが多数用意されています。コンポーネントパレットは，データアクセスのメカニズムと機能によってデータベースコンポーネントを次のようにグループ分けしています。

表 7.2　コンポーネントパレットのデータベース関連ページ

ページ名	内容
BDE	BDE（ボーランドデータベースエンジン）を使用するコンポーネント。BDE はデータベースと相互作用する大規模 API であり，もっとも広範な機能をサポートする。BDE には非常に有用なユーティリティ（Database Desktop，Database Explorer，SQL Monitor，BDE Administrator など）が付属している。詳細については，第 24 章「ボーランドデータベースエンジンの使い方」を参照
ADO	Microsoft ADO（ActiveX Data Objects）を使ってデータベース情報にアクセスするコンポーネント。多くの ADO ドライバがあり，各種データベースサーバーに接続できる。ADO ベースのコンポーネントを使うと，自分のアプリケーションを ADO ベース環境に統合できる。詳細については，第 25 章「ADO コンポーネントの操作」を参照
dbExpress	dbExpress を使ってデータベース情報にアクセスするクロスプラットフォームコンポーネント。dbExpress 用ドライバによりデータベースに高速にアクセスできるが，更新を実行するには TClientDataSet と TDataSetProvider を使う必要がある。詳細については，第 26 章「単方向データセットの使い方」を参照
InterBase	別個のエンジン層を通らずに直接 InterBase データベースにアクセスするコンポーネント。InterBase コンポーネントの使い方は，オンラインヘルプを参照
Data Access	TClientDataSet，TDataSetProvider などのデータアクセスメカニズムを使用できるコンポーネント。クライアントデータセットの詳細は，第 27 章「クライアントデータセットの使い方」を参照。プロバイダの詳細は，第 28 章「プロバイダコンポーネントの使い方」を参照
Data Controls	データソースから情報にアクセスできるデータベース対応コントロール。詳細については，第 19 章「データコントロールの使い方」を参照

データベースアプリケーションを設計するには，どのデータアクセスメカニズムを使うかを決定する必要があります。データアクセスメカニズムによって，機能サポートの範囲，配布しやすさ，ドライバ（各種データベースサーバーをサポートするドライバ）の可用性が異なります。

C++Builder を使ってデータベースクライアントアプリケーションとアプリケーションサーバーを作成する方法については，このマニュアルの第 II 部「データベースアプリケーションの開発」を参照してください。配布についての詳細は，17-6 ページの「データベースアプリケーションの配布」を参照してください。

メモ C++Builder のバージョン（版）によっては，データベース作成機能が含まれていない場合があります。

データベースアプリケーションの配布

C++Builder は，コンポーネントの活用による分散データベースアプリケーションの作成をサポートしています。DCOM，TCP/IP，SOAP など多様な通信プロトコル上に分散データベースアプリケーションを構築することができます。

分散データベースアプリケーションの構築の詳細については，第 29 章「多層アプリケーションの作成」を参照してください。

データベースアプリケーションの配布では，アプリケーションファイルに加えて BDE（Borland Database Engine）を配布する必要が生じる場合があります。BDE の詳細については，17-6 ページの「データベースアプリケーションの配布」を参照してください。

Web サーバーアプリケーションの作成

Web サーバーアプリケーションとは，インターネットを介して Web コンテンツ（HTML Web ページ，XML ドキュメントなど）を配信するサーバーで動作するアプリケーションです。たとえば，Web サイトへのアクセスを制御するアプリケーション，注文書を作成するアプリケーション，問い合わせに応答するアプリケーションなどが Web サーバーアプリケーションです。

C++Builder の持つ以下のテクノロジーを使って，さまざまな Web サーバーアプリケーションを作成できます。

- WebBroker
- WebSnap
- InternetExpress
- Web サービス

WebBroker の使い方

WebBroker（NetCLX アーキテクチャとも呼ばれる）を使うと，CGI アプリケーションや DLL（ダイナミックリンクライブラリ）などの Web サーバーアプリケーションを作成できます。この種の Web サーバーアプリケーションには，非ビジュアルコンポーネントを入れることができます。コンポーネントパレットの［Internet］ページにあるコンポーネントを使うと，イベントハンドラを作成し，プ

ログラミングによって HTML/XML ドキュメントを作成し，クライアントに転送することができます。

WebBroker アーキテクチャを使って Web サーバーアプリケーションを新規作成するには，［ファイル｜新規作成｜その他］を選択し，［新規作成］ダイアログボックスの［Web サーバーアプリケーション］をダブルクリックします。次に，以下に示す Web サーバーアプリケーションの種類のいずれかを選択します。

表 7.3　Web サーバーアプリケーション

Web サーバーアプリケーションの種類	説明
ISAPI/NSAPI ダイナミックリンクライブラリ	Web サーバーによってロードされる DLL。クライアントのリクエスト情報は構造体として DLL に渡され，TISAPIApplication に評価される。各リクエストメッセージは別個のスレッドで処理される。 この種類を選択すると，プロジェクトファイルの uses 節と exports 節に，プロジェクトファイルのライブラリヘッダーおよび必要なエントリが追加される
CGI スタンドアロン実行形式	CGI Web サーバーアプリケーションは，標準入力でクライアントから要求を受け取り，それらの要求を処理し，標準出力で結果をサーバーに戻し，クライアントに送るコンソールアプリケーションである
Win-CGI スタンドアロン実行形式	サーバーが書き込んだ環境設定（INI）ファイルからクライアントの要求を受け取り，サーバーがクライアントに送るファイルに結果を書き込む Windows アプリケーション。この INI ファイルは TCGIApplication に評価される。アプリケーションの個別のインスタンスが各リクエストメッセージを処理する
Apache 共有モジュール（DLL）	この種類を選択すると，プロジェクトが DLL として設定される。Apache Web サーバーアプリケーションは，Web サーバーによってロードされる DLL である。情報は Web サーバーによって DLL に渡され，処理され，クライアントに返される
Web アプリケーションデバッガスタンドアロン実行形式	この種類を選択すると，Web サーバーアプリケーションを開発，テストする環境がセットアップされる。Web アプリケーションデバッガは，Web サーバーでロードされる実行形式ファイルである。この種類のアプリケーションは配布されない

CGI と Win-CGI アプリケーションは，サーバー上でより多くのシステムリソースを利用するので，複雑なアプリケーションは ISAPI，NSAPI，Apache のいずれかの DLL アプリケーションとして作成した方がよいでしょう。クロスプラットフォームアプリケーションを作成する場合は，Web サーバー開発には CGI スタンドアロンか Apache 共有モジュールを選択してください。WebSnap アプリケーションと Web サービスアプリケーションを作成するときも，上記と同じオプションがあります。

Web サーバーアプリケーションの構築についての詳細は，第 32 章「インターネットサーバーアプリケーションの作成」を参照してください。

WebSnap アプリケーションの作成

WebSnap のコンポーネント群とウィザードを使うと，Web ブラウザとやり取りする高度な Web サーバーを構築することができます。WebSnap のコンポーネントは，Web ページ用の HTML コンテンツや MIME コンテンツを生成します。WebSnap はサーバー側の開発で使います。現時点では，WebSnap はクロスプラットフォームアプリケーションには使用できません。

新しい WebSnap アプリケーションを作成するには，［ファイル｜新規作成｜その他］を選択し，［新規作成］ダイアログボックスの［WebSnap］を選択します。［WebSnap アプリケーション］を選択し

ます。次に，Web サーバーアプリケーションの種類（ISAPI/NSAPI，CGI，Win-CGI，Apache）を選択します。詳細については，表 7.3「Web サーバーアプリケーション」を参照してください。

WebSnap の詳細については，第 34 章「WebSnap を使用しての Web サーバーアプリケーションの作成」を参照してください。

InternetExpress の使い方

InternetExpress のコンポーネント群を使うと，Web サーバーアプリケーションの基本アーキテクチャを拡張して，アプリケーションサーバーのクライアントとして機能させることができます。InternetExpress を使って作成したアプリケーションは，クライアント側で実行され，そこではブラウザベースのクライアントがプロバイダからデータを取得し，プロバイダに対して更新を解決することができます。

InternetExpress アプリケーションは，HTML，XML，および Java スクリプトを混在させた HTML ページを生成します。HTML は，ユーザーがブラウザで表示したときのページのレイアウトと外観を決定します。XML は，データベース情報を表すデータパケットとデルタパケットをコード化します。Java スクリプトは，HTML コントロールが XML データパケット内のデータをクライアントマシン上で解釈，操作するのを可能にします。

InternetExpress の詳細については，29-32 ページの「InternetExpress 使用による Web アプリケーションの構築」を参照してください。

Web サービスアプリケーションの作成

Web サービスとは，World Wide Web などのネットワークを介して公開し，起動することのできる自立型のモジュラーアプリケーションです。Web サービスには，提供するサービスを表す定義済みインターフェースが用意されています。Web サービスを使うと，XML，XML Schema，SOAP（Simple Object Access Protocol），WSDL（Web Service Definition Language）といった新しい規格を使ってインターネット経由でプログラマブルサービスを作成したり利用したりできます。

Web サービスでは，SOAP という標準の軽量プロトコルを使って分散環境で情報交換を行います。また，通信プロトコルは HTTP を使用し，リモート手続き呼び出しのコード化には XML を使います。

C++Builder を使うと，Web サービスを実装するサーバーと，Web サービスを呼び出すクライアントを構築できます。任意のサーバー用のクライアントを作成して，SOAP メッセージに応答する Web サービスを実装することができます。また，C++Builder サーバーを作成して，任意のクライアントが使用する Web サービスを公開することができます。

Web サービスに関する詳細は，第 36 章「Web サービスの使い方」を参照してください。

COM を使ってアプリケーションを作成する

COM とは Component Object Model の頭字語で，インターフェースという定義済みのルーチンを使用してオブジェクトの相互利用ができるように設計された，Windows ベースの分散オブジェクトアー

キテクチャを意味します。COM アプリケーションは，DCOM を使う場合は異なるマシンで，それ以外の場合は異なるプロセスによって実装されるオブジェクトを使用します。COM+，ActiveX，ASP（Active Server Pages）も使用できます。

COM は，言語に依存しないソフトウェアコンポーネントモデルです。Windows プラットフォームで動作するアプリケーションやソフトウェアコンポーネント間の相互作用を可能にします。COM のもっとも重要な機能は，明確に定義されたインターフェースを通じて，コンポーネント間，アプリケーション間，およびクライアント / サーバー間での通信を可能にすることです。インターフェースを通じて，クライアントは，どの機能をサポートしているかを実行時に COM コンポーネントにたずねます。コンポーネントに機能を追加するには，追加機能用のインターフェースを追加するだけで済みます。

COM と DCOM の使い方

C++Builder には，COM アプリケーション，OLE アプリケーション，および ActiveX アプリケーションの作成を簡易化するクラスとウィザードが用意されています。COM クライアントまたは COM サーバーを作成して，そこに COM オブジェクト，オートメーションサーバー（Active Server Objects を含む），ActiveX コントロール，または ActiveForm を実装することができます。COM は，他のテクノロジー（オートメーション，ActiveX コントロール，Active ドキュメント，Active ディレクトリなど）の土台の働きもします。

C++Builder を使用して COM アプリケーションを作成すると，アプリケーションの内部のインターフェースを使ったソフトウェア設計の改良から，Win9x のシェルエクステンションや DirectX マルチメディアサポートのような，システム上の他の COM API オブジェクトと対話できるオブジェクトの作成まで，さまざまな作業が可能となります。アプリケーションは，同じコンピュータ上にある COM コンポーネントのインターフェースにアクセスでき，また，分散 COM（DCOM）と呼ばれるメカニズムを使ってネットワーク上の別のコンピュータ上にある COM コンポーネントのインターフェースにアクセスすることもできます。

COM と ActiveX コントロールの詳細については第 38 章「COM テクノロジーの概要」第 43 章「ActiveX コントロールの作成」と 29-31 ページの「ActiveX コントロールとしてクライアントアプリケーションを配布する」を参照してください。

DCOM の詳細については，29-9 ページの「DCOM 接続を使う」を参照してください。

MTS と COM+ の使い方

特殊なサービスを使って COM アプリケーションを拡張し，大規模な分散環境でオブジェクトを管理することができます。特殊なサービスとは，MTS（Microsoft Transaction Server）（Windows 2000 より前のバージョン）および COM+（Windows 2000 以降のバージョン用）の提供するトランザクションサービス，セキュリティ，リソース管理です。

MTS と COM+ の詳細については，第 44 章「MTS オブジェクトまたは COM+ オブジェクトの作成」と 29-6 ページの「トランザクションデータモジュールを使う」を参照してください。

データモジュールの使い方

データモジュールとは，非ビジュアルコンポーネントの入ったいわば特殊フォームです。データモジュール内のコンポーネントは，ビジュアルコントロールと一緒に通常のフォームに置くこともできます。しかし，データベースオブジェクトとシステムオブジェクトのグループを再利用したり，アプリケーションのうちデータベース接続とビジネスルールを処理する部分を切り離そうとする場合には，データモジュールをツールとして使うと簡単に整理できます。

データモジュールには，標準，リモート，Web モジュール，アプレットモジュール，サービスなどいくつかの種類があります。C++Builder のバージョン（版）によって，使用できるデータモジュールの種類が異なります。どの種類も，それぞれ特別な目的で使用します。

- 標準データモジュールは特に 1 層および 2 層のデータベースアプリケーションで使うと便利ですが，どのアプリケーションでも非ビジュアルコンポーネントを整理するのに使用できます。詳細については，7-20 ページの「標準データモジュールの作成と編集」を参照してください。
- リモートデータモジュールは，多層データベースアプリケーションでアプリケーションサーバーの基本になります。C++Builder のバージョンによっては，リモートデータモジュールを使用できない場合があります。リモートデータモジュールは，アプリケーションサーバーに非ビジュアルコンポーネントを保持します。これに加えて，クライアントがアプリケーションサーバーと通信するときに使用するインターフェースを公開します。リモートデータモジュールの使い方については，7-24 ページの「リモートデータモジュールをアプリケーションサーバープロジェクトに追加する」を参照してください。
- Web モジュールは，Web サーバーアプリケーションの基本になります。Web モジュールは，HTTP レスポンスメッセージのコンテンツを作成するコンポーネントを保持します。これに加えて，クライアントアプリケーションからの HTTP メッセージのディスパッチを処理します。Web モジュールの使い方については，第 32 章「インターネットサーバーアプリケーションの作成」を参照してください。
- アプレットモジュールは，コントロールパネルアプレットの基本になります。アプレットモジュールは，コントロールパネルアプレットを実装する非ビジュアルコントロールを保持します。これに加えて，コントロールパネル上でのアプレットアイコンの表示方法を決めるプロパティを定義します。ユーザーがアプレットを実行したときに呼び出されるイベントも含まれています。アプレットモジュールについての詳細は，オンラインヘルプを参照してください。
- サービスモジュールは，NT サービスアプリケーション内で個々のサービスをカプセル化します。サービスモジュールは，各サービスの実装に使われる非ビジュアルコントロールを保持します。サービスモジュールには，サービスを開始または停止したときに呼び出されるイベントが含まれています。サービスの詳細については，7-4 ページの「サービスアプリケーション」を参照してください。

標準データモジュールの作成と編集

プロジェクト用に標準のデータモジュールを作成するには，［ファイル｜新規作成｜データモジュール］を選択します。デスクトップにデータモジュールコンテナが開きます。同時に，コードエディタ

に新規データモジュールのユニットファイルが表示され，プロジェクトファイルにデータモジュールが追加されます。

設計時，データモジュールの外観は C++Builder の標準フォームに似ています。背景は白色で，位置揃えグリッドは表示されていません。フォームを操作するときと同様に，コンポーネントパレットからモジュールへと非ビジュアルコンポーネントを配置し，オブジェクトインスペクタでプロパティを変更できます。追加したコンポーネントに合わせてモジュールのサイズを変えることもできます。

モジュールを右クリックすれば，コンテキストメニューが表示されます。データモジュールのコンテキストメニューの概要は次のとおりです。

表 7.4　データモジュールのコンテキストメニュー項目

メニュー項目	目的
編集	データモジュール内のコンポーネントの切り取り，コピー，貼り付け，削除，選択に関するコンテキストメニューを表示する
位置	モジュールの非表示グリッドに合わせて非ビジュアルコンポーネントを位置揃えする（グリッドに合わせる）。または，［位置合わせ］ダイアログボックスで指定した基準に従って非ビジュアルコンポーネントを位置揃えする（位置合わせ）
タブ順序	〔Tab〕キーを押したときにコンポーネントからコンポーネントへとフォーカスが移る順序を変更する
作成順序	起動時にデータアクセスコンポーネントが作成される順序を変更する
継承元の値に戻す	オブジェクトリポジトリで別のモジュールから継承したモジュールの変更内容を破棄し，最初の継承モジュールに戻す
リポジトリに追加	オブジェクトリポジトリにあるデータモジュールとのリンクを保存する
エディタで表示	データモジュールのプロパティをテキストで表示する
テキスト形式 DFM	特定のフォームファイルの保存フォーマットをバイナリとテキストの間で切り替える

データモジュールについての詳細は，オンラインヘルプを参照してください。

データモジュールとそのユニットファイルに名前を付ける

データモジュールのタイトルバーにデータモジュール名が表示されます。デフォルトのデータモジュール名は「DataModule*N*」です。*N* は，プロジェクトの中で一番小さい未使用ユニット番号を表します。たとえば，新規プロジェクトを開始して，他のアプリケーション構築を行う前にモジュールを 1 つ追加した場合，モジュール名はデフォルトで「DataModule2」となります。DataModule2 に対応するユニットファイル名は，デフォルトで「Unit2」です。

わかりやすい名前にするには，設計時にデータモジュールおよびそのユニットファイルの名前を変更します。特に，オブジェクトリポジトリにデータモジュールを追加した場合には，オブジェクトリポジトリやアプリケーションの中にある他のデータモジュールと名前の衝突が起こるおそれがあるため，データモジュール名を変更する必要があります。

データモジュールの名前を変更するには，次のようにします。

1. フォーム上でモジュールを選択します。
2. オブジェクトインスペクタで，モジュールの Name プロパティを変更します。

オブジェクトインスペクタ内のフォーカスが Name プロパティ以外のところに移ると，タイトルバーに新しいモジュール名が表示されます。

設計時にデータモジュールの名前を変更すると，コードの interface 部にある変数名が変更されます。手続き宣言で型名を使用している場合も，変更されます。プログラマが自分で記述したコードにデータモジュールへの参照がある場合には，手作業で変更しなければなりません。

データモジュールのユニットファイル名を変更するには，次のようにします。

1. ユニットファイルを選択します。
2. ［ファイル｜名前を付けて保存］を選択し，ユニットのファイル名を入力して保存します。

コンポーネントを配置して名前を付ける

データモジュールに非ビジュアルコンポーネントを配置する方法は，フォームにビジュアルコンポーネントを配置するのとまったく同じです。コンポーネントパレットの目的のページでコンポーネントをクリックしてから，データモジュール内をクリックします。グリッドなどのビジュアルコントロールはデータモジュールに置けません。ビジュアルコントロールを置こうとすると，エラーメッセージが表示されます。

コンポーネントを使いやすくするため，データモジュール内ではコンポーネントの名前も表示されます。コンポーネントは，最初にデータモジュールに配置された時点で一般名称（コンポーネントを識別する名前）が割り当てられ，その後ろに 1 が付きます。たとえば，TDataSource コンポーネントの名前は DataSource1 になります。このように名前が付いているので，プロパティやメソッドを操作すべきコンポーネントを簡単に探すことができます。

また，コンポーネントの種類や目的を表す別の名前に変更することもできます。

データモジュール内のコンポーネントの名前を変更するには，次のようにします。

1. フォーム上でコンポーネントを選択します。
2. オブジェクトインスペクタでコンポーネントの Name プロパティを変更します。

オブジェクトインスペクタ内のフォーカスが Name プロパティ以外のところに移ると，データモジュール内のコンポーネントアイコンの下に新しい名前が表示されます。

たとえば次のようなケースは，コンポーネント名を変更すると便利です。作成したデータベースアプリケーションで CUSTOMER というテーブルを使うとします。このテーブルにアクセスするには，データアクセスコンポーネントが少なくとも 2 つ必要です。1 つはデータソースコンポーネント（TDataSource），もう 1 つはテーブルコンポーネント（TClientDataSet）です。この 2 つのコンポーネントを最初にデータモジュールに配置した時点では，コンポーネント名は「DataSource1」と「ClientDataSet1」になっています。この場合，コンポーネント名をそれぞれ CustomerSource と CustomerTable に変更すれば，コンポーネントの種類，およびアクセスするデータベースの名前（CUSTOMER）を表すことができます。

データモジュールでのコンポーネントのプロパティとイベントの使い方

複数のコンポーネントを同じデータモジュールに配置すると，これらのコンポーネントの動作をアプリケーション全体に渡って一元化することができます。たとえば，TClientDataSet などのデータセットコンポーネント群のプロパティを設定することにより，これらのデータセットを使用するデータソースコンポーネントでどのデータを使うかを制御できます。あるデータセットの ReadOnly プロパティ

を **true** に設定しておけば，フォーム上のデータベース対応ビジュアルコントロールにデータが表示されていても，ユーザーはそのデータを編集できません。また，データセットの項目エディタを使用すれば，同じテーブル内または問い合わせ内で参照できる項目をデータソースに制限できます（つまり，フォーム上のデータベース対応コントロールに制限できる）。項目エディタを表示するには，ClientDataSet1 の上をダブルクリックします。データモジュールでコンポーネントのプロパティを設定すると，アプリケーション内でそのモジュールを使用するすべてのフォームに一貫して適用されます。

プロパティを設定するだけでなく，コンポーネントのイベントハンドラを作成することもできます。たとえば TDataSource コンポーネントの場合，発生しうるイベントは，OnDataChange，OnStateChange，OnUpdateData の 3 つです。TClientDataSet コンポーネントの場合は，20 個以上のイベントがあります。これらのイベントを使うことで，アプリケーション全体のデータ操作を管理する一貫したビジネスルールを作成できます。

データモジュールにメソッドを記述する

データモジュールに入れたコンポーネントのイベントハンドラを作成するだけでなく，データモジュールのユニットファイルに直接メソッドを記述することもできます。記述したメソッドは，データモジュールをビジネスルールとして使用するフォームに適用できます。たとえば，月次，四半期，あるいは年間の帳簿を作成する手続きなどを記述できます。データモジュールに置いたコンポーネントのイベントハンドラから手続きを呼び出したりもできます。

フォームからデータモジュールへのアクセス

フォームのビジュアルコントロールをデータモジュールに関連付けるには，まずデータモジュールのヘッダーファイルをフォームの .CPP ファイルに追加する必要があります。次のように何通りかの方法があります。

- コードエディタにフォームのユニットファイルを開き，#include 指令を使ってデータモジュールのヘッダーファイルをインクルードする
- フォームのユニットファイルをクリックし，［ファイル｜ユニットヘッダーファイルの追加］を選択して，モジュール名を入力するか，または［使用するユニットの選択］ダイアログのリストボックスから選択する
- データベースコンポーネントの場合は，データモジュール内でデータセットコンポーネントか問い合わせコンポーネントをダブルクリックして項目エディタを開き，項目エディタからフォームへと既存の項目をドラッグする。この場合，モジュールをフォームに追加するかどうかを確認するプロンプトが表示された後で，ドラッグした項目のコントロール（編集ボックスなど）が作成される

たとえば，データモジュールに TClientDataSet コンポーネントを追加したとします。項目エディタを開くには，このコンポーネントをダブルクリックします。項目のどれかを選択し，フォームへドラッグします。すると，編集ボックスコンポーネントが表示されます。

まだデータソースが定義されていないので，C++Builder は新しいデータソースコンポーネント（DataSource1）をフォームに追加し，編集ボックスの DataSource プロパティを DataSource1 に設定します。追加されたデータソースは，データモジュール内で自分の DataSet プロパティをデータセットコンポーネント（ClientDataSet1）に自動的に設定します。

項目エディタの項目をフォームにドラッグする前なら，プログラマ自身でデータソースを定義できます。まず，データモジュールに TDataSource コンポーネントを追加します。次に，データソースの DataSet プロパティを ClientDataSet1 に設定します。項目エディタの項目をフォームにドラッグしたあとでは，すでに TDataSource プロパティが DataSource1 に設定された状態で編集ボックスが表示されます。後者の方法の方が，データアクセスモデルを簡潔にできます。

リモートデータモジュールをアプリケーションサーバープロジェクトに追加する

C++Builder のバージョンによっては，リモートデータモジュールをアプリケーションサーバプロジェクトに追加することができます。リモートデータモジュールとは，多層アプリケーションのクライアントをネットワーク経由でアクセスさせるインターフェースを持つデータモジュールです。

リモートデータモジュールをプロジェクトに追加するには，次のようにします。

1. ［ファイル｜新規作成｜その他］を選択します。
2. ［新規作成］ダイアログボックスで［多層サポート］ページを選択します。
3. ［リモートデータモジュール］アイコンをダブルクリックして，リモートデータモジュールウィザードを開きます。

プロジェクトにリモートデータモジュールを追加した後は，標準データモジュールのように使用します。

多層データベースアプリケーションについての詳細は，第 29 章「多層アプリケーションの作成」を参照してください。

オブジェクトリポジトリの使い方

オブジェクトリポジトリ（［ツール｜リポジトリ］）を使うと，簡単にフォーム，ダイアログボックス，フレーム，データモジュールを共有できます。また，オブジェクトリポジトリには，新規プロジェクト用テンプレートやウィザード（一連の操作を通じてユーザーにフォームとプロジェクトを完成させるアプリケーション）があります。リポジトリを管理するファイルは BCB.DRO というテキストファイルです（デフォルトで BIN ディレクトリに作成される）。このファイルの中に，［オブジェクトリポジトリ］ダイアログボックスと［新規作成］ダイアログボックスに表示される項目への参照が収められています。

プロジェクト内での項目の共有

同じプロジェクトの中であれば，オブジェクトリポジトリに項目を追加しなくても項目を共有できます。［ファイル｜新規作成｜その他］を選択して［新規作成］ダイアログボックスを開くと，現在のプロジェクト名が付いたページタブが表示されます。このページを選択すると，現在のプロジェクトにあるフォーム，ダイアログボックス，データモジュールがすべて表示されます。必要に応じて，既存の項目から新しい項目を派生させてカスタマイズできます。

オブジェクトリポジトリへの項目の追加

独自に作成したプロジェクト，フォーム，フレーム，データモジュールをオブジェクトリポジトリに追加することもできます。オブジェクトリポジトリへ項目を追加するには，次のようにします。

1. 項目がプロジェクト自身であるか，項目がプロジェクト内にある場合は，そのプロジェクトを開きます。
2. プロジェクトの場合は，［プロジェクト｜リポジトリに追加］を選択します。フォームまたはデータモジュールの場合は，項目を右クリックして［リポジトリに追加］を選択します。
3. コメント，タイトル，および作者を入力します。
4. ［新規作成］ダイアログボックスのどのページに表示させるかを指定します。［ページ］コンボボックスから選択するか，ページ名を直接入力します。存在しないページ名を入力した場合は，自動的に新規ページが作成されます。
5. ［参照］をクリックすると，オブジェクトリポジトリで表示されるアイコンイメージを選択できます。
6. ［OK］をクリックします。

チーム環境でオブジェクトを共有する

ネットワークを介してリポジトリを使用できるようにすれば，ワークグループや開発チームなどのチーム環境でオブジェクトを共有できます。チームメンバーが共有リポジトリを使用するには，［環境オプション］ダイアログボックスに指定された共有リポジトリディレクトリをメンバー全員が選択しなければなりません。共有リポジトリディレクトリを指定する手順は次のとおりです。

1. ［ツール｜環境オプション］を選択します。
2. ［設定］ページで［共有リポジトリ］パネルを選択します。［ディレクトリ］編集ボックスに，共有リポジトリを置くディレクトリを入力します。チームメンバー全員がアクセスできるディレクトリを指定してください。

初めて項目をリポジトリに追加した場合，BCB.DRO ファイルが存在しなければ，共有リポジトリ用ディレクトリに BCB.DRO ファイルが作成されます。

プロジェクトにおけるオブジェクトリポジトリ項目の使い方

オブジェクトリポジトリ内の項目にアクセスするには，［ファイル｜新規作成｜その他］を選択します。［新規作成］ダイアログが開き，使用可能な項目が一覧表示されます。項目をプロジェクトに追加するオプションは以下の 3 種類があります。どれが使えるかは項目の種類によります。

- コピー
- 継承
- 直接使用

項目のコピー

コピーを選択すると，選択した項目とまったく同じものが複製され，プロジェクトにそのコピーが追加されます。オブジェクトリポジトリの項目を後で変更してもコピーには反映されません。また，コピーを変更しても元のオブジェクトリポジトリの項目には反映されません。

プロジェクトテンプレートの場合，使用できるオプションはコピーだけです。

項目の継承

継承を選択すると，オブジェクトリポジトリで選択した項目から新しいクラスが派生し，派生後の新しいクラスがプロジェクトに追加されます。プロジェクトを再コンパイルした場合，プロジェクト内の項目に加えた変更だけでなく，オブジェクトリポジトリの項目に加えた変更も派生クラスに反映されます。派生クラスに加えた変更は，オブジェクトリポジトリの共有項目には反映されません。

フォーム，ダイアログボックス，データモジュールでは継承が使えますが，プロジェクトテンプレートでは使えません。また，同一プロジェクト内の項目を再利用する場合には，継承オプションだけが使えます。

項目の直接使用

選択した項目自体をプロジェクトの一部にする場合には，直接使用を選択します。項目に加えた変更は，継承や直接使用オプションを使って項目を追加した他の全プロジェクトに反映されます。このオプションを使用するときは注意が必要です。

直接使用オプションは，フォーム，ダイアログボックス，データモジュールで使用できます。

プロジェクトテンプレートの使い方

テンプレートとは，プロジェクトの土台として使える設計済みのプロジェクトです。プロジェクトテンプレートを使って新規プロジェクトを作成する手順は次のとおりです。

1. ［ファイル｜新規作成｜その他］を選択して［新規作成］ダイアログボックスを表示します。
2. ［プロジェクト］タブを選択します。
3. 目的のプロジェクトテンプレートを選択して［OK］をクリックします。
4. ［ディレクトリの選択］ダイアログで，新規プロジェクトファイル用のディレクトリを指定します。

C++Builder 側で，テンプレートファイルを指定ディレクトリにコピーします。コピー先のテンプレートファイルは後で変更できます。コピー先のテンプレートファイルを変更しても，元のプロジェクトテンプレートは影響を受けません。

共有項目の変更

直接使用オプションか継承オプションを使って項目をプロジェクトに追加した場合，オプションリポジトリ内の項目に変更を加えると，既存のプロジェクトに影響するだけでなく，同じ項目を使用する将来のプロジェクトにも影響が及びます。変更内容がほかのプロジェクトに広がるのを防ぐには，以下のいずれかの方法を使います。

- 現在のプロジェクトの中だけで項目をコピーして修正する
- 現在のプロジェクトに項目をコピーし，修正した後，その項目を別名でリポジトリに追加する
- その項目からコンポーネント，DLL，コンポーネントテンプレート，またはフレームを作成する。コンポーネントか DLL を作成すると，ほかの開発者との間で共有することができる

デフォルトプロジェクト，新しいフォーム，およびメインフォームを指定する

デフォルトでは，［ファイル｜新規作成｜アプリケーション］または［ファイル｜新規作成｜フォーム］を選択すると，空白のフォームが表示されます。次のようにリポジトリの設定を変えると，このデフォルトの動作を変更することができます。

1. ［ツール｜リポジトリ］を選択します。
2. デフォルトのプロジェクトを指定するには，［ページ］リストから［プロジェクト］を選択し，［オブジェクト］リストから目的の項目を選択します。［プロジェクトの新規作成時に使用］をチェックします。
3. デフォルトのフォームを指定するには，［オブジェクトリポジトリ］ダイアログボックスで［フォーム］などのページを選択し，［オブジェクト］リストからフォームを選択します。デフォルトの新規フォーム（［ファイル｜新規作成｜フォーム］）を指定するには，［フォームの新規作成時に使用］をチェックします。新規プロジェクトに使用するデフォルトのメインフォームを指定するには，［メインフォームとして使用］をチェックします。
4. ［OK］を選択します。

アプリケーションでヘルプを使用可能にする

VCL と CLX のどちらも，アプリケーションからのヘルプ表示をサポートしています。オブジェクトベースのメカニズムを使用して，ヘルプリクエストを外部ヘルプビューアのどれかに渡します。これを実行するには，あるクラスがアプリケーションに含まれていなければなりません。あるクラスとは，ICustomHelpViewer インターフェースを実装し，そのクラス自身をグローバルの Help Manager に登録したクラスです。

VCL の場合，すべてのアプリケーションに TWinHelpViewer のインスタンスを供給します。これにより，ヘルプシステムのインターフェースすべてが実装され，アプリケーション と WinHelp の間のリンクが確立します。CLX の場合，アプリケーション開発者が自分で実装する必要があります。Windows 上では，VCL の一部として供給されている WinHelpViewer ユニットを CLX アプリケーションが使用できます。そのためには，WinHelpViewer ユニットを CLX アプリケーションと静的にバインドします（つまり，WinHelpViewer ユニットを VCL パッケージとリンクさせるのでなく，プロジェクトの一部として含める）。

Help Manager は，登録済みヘルプビューアのリストを管理し，リクエストをヘルプビューアに渡します。リクエストを渡す動作は次の 2 段階で行われます。最初に，特定のキーワードまたはコンテキストをサポートするかどうかを各ヘルプビューアにたずねます。次に，キーワードかコンテキストをサポートできると応答したビューアに，ヘルプリクエストを渡します。

同じアプリケーションが WinHelp と HyperHelp の両方のビューア（Windows），または Man と Info の両方のビューア（Linux）を登録したケースなど，複数のビューアが同じキーワードをサポートする場合，Help Manager は次のように処理します。可能な場合は選択ボックスを表示して，アプリケーションのユーザーがヘルプビューアを選択できるようにします。そうでなければ，最初に応答したヘルプシステムを表示します。

ヘルプシステムのインターフェース

ヘルプシステムでは，一連のインターフェースを介してアプリケーションとヘルプビューアの間で通信することを可能にします。これらのインターフェースはすべて HelpIntfs に定義されています。このファイルには，Help Manager の実装もあります。

ICustomHelpViewer：キーワードに基づくヘルプ表示をサポートするほか，特定のヘルプビューアに表示できるヘルプ項目を一覧にした目次の表示もサポートします。

IExtendedHelpViewer：数値で表されるヘルプコンテキストに基づくヘルプ表示と，トピックの表示をサポートします。ほとんどのヘルプシステムでは，トピックは高レベルのキーワードとして機能します（たとえば，ヘルプシステムでは「IntToStr」はキーワードですが，「文字列操作ルーチン」はトピック名になります）。

ISpecialWinHelpViewer：WinHelp 用の特殊メッセージ（Windows で動作するアプリケーションが受信でき，簡単には汎用化できないメッセージ）への応答をサポートします。基本的に，このインターフェースを実装する必要があるのは，Windows 環境で動作するアプリケーションだけです。また，Windows 環境で動作するアプリケーションであっても，非標準の WinHelp メッセージを広範囲に使用するのでなければ，このインターフェースを実装する必要はありません。

IHelpManager：アプリケーションの Help Manager への応答として，ヘルプビューアが追加情報を要求するためのメカニズムを提供します。ヘルプビューアが自分自身を登録した時点で，IHelpManager が取得されます。

IHelpSystem：TApplication がヘルプリクエストをヘルプシステムに渡すためのメカニズムを提供します。TApplication は，IHelpSystem と IHelpManager の両方を実装したオブジェクトのインスタンスをアプリケーションのロード時に取得し，そのインスタンスをプロパティとしてエクスポートします。これにより，アプリケーション内の他のコードから，適宜ヘルプリクエストを直接送信できます。

IHelpSelector：複数のヘルプビューアがヘルプリクエストを処理できるケースに対処するメカニズムを提供します。ヘルプシステムは，ユーザーインターフェースを起動して，どのヘルプビューアを使用するかをたずねます。目次を表示することも可能です。この目次表示機能は Help Manager に直接的には組み込まれていません。このため，使用するウィジェットセットやクラスライブラリの種類によらず，Help Manager のコードは同一です。

ICustomHelpViewer の実装

ICustomHelpViewer インターフェースには 3 種類のメソッドが含まれています。Help Manager との間でシステムレベルの情報（特定のヘルプリクエストに関係しない情報など）を通信するメソッド，

Help Manager が提示したキーワードに基づいてヘルプを表示するメソッド，および目次を表示するメソッドです。

Help Manager との通信

ICustomHelpViewer には，Help Manager との間でシステム情報について通信する関数が 4 つあります。

- GetViewerName
- NotifyID
- ShutDown
- SoftShutDown

Help Manager は，以下の状況のときにこれらの関数を使って呼び出しをします。

- AnsiString ICustomHelpViewer::GetViewerName()：Help Manager がビューア名を知りたいとき，この関数を呼び出します（たとえば，登録済みビューアのリストを表示するようアプリケーションに要求するときなど）。ビューア名情報は文字列を介して返されます。また，この情報は論理的に静的でなければなりません（つまり，アプリケーションの動作中は変更できません）。マルチバイト文字セットはサポートしていません。
- **void** ICustomHelpViewer::NotifyID(**const int** ViewerID)：登録直後にこの関数を呼び出し，ビューアを識別する一意の cookie をビューアに与えます。ビューアの識別情報は，後で使えるように保存しておく必要があります。つまり，ビューアが（Help Manager からの通知に対する応答としてでなく）自分からシャットダウンする場合，ビューアから Help Manager に cookie 情報を提供しなければなりません。そうすれば，Help Manager は，そのビューアに対するすべての参照を解放できます（cookie を提供しないか，または間違った cookie 情報を提供すると，Help Manager は間違ったビューアに対する参照を解放する可能性があります）。
- **void** ICustomHelpViewer::ShutDown()：この関数を呼び出すことにより，Help Manager からヘルプビューアに 2 つのことを通知します。1 つは Help Manager がシャットダウン中であること，もう 1 つは，ヘルプビューアが割り当てたリソースがあれば，それを解放する必要があるということです。リソースの解放は，すべてこのメソッドに委任するようにしてください。
- **void** ICustomHelpViewer::SoftShutDown()：この関数を呼び出すことにより，Help Manager は，ヘルプシステムを外部に表示するもの（ヘルプ情報を表示しているウィンドウなど）があれば，ヘルプビューアをアンロードせずにそれを閉じるようヘルプビューアに要求します。

Help Manager に情報を要求する

ヘルプビューアと Help Manager との通信は，IHelpManager インターフェースを介して行います。ヘルプビューアが Help Manager に登録されると，IHelpManager インターフェースのインスタンスがヘルプビューアに返されます。IHelpManager インターフェースにより，ヘルプビューアは次の 4 つの情報を伝達できます。

- 現在アクティブになっているコントロールのウィンドウハンドルを求めるリクエスト
- 現在アクティブになっているコントロールのヘルプが保存されていると Help Manager が判断するヘルプファイルの名前を求めるリクエスト
- そのヘルプファイルまでのパスを求めるリクエスト

- Help Manager からの要求（シャットダウン要求を除く）に応答してヘルプビューアが自身をシャットダウン中であるという通知

int IHelpManager::GetHandle()：現在アクティブになっているコントロールのハンドルを知りたいとき，ヘルプビューアはこの関数を呼び出します。結果はウィンドウハンドルです。

AnsiString IHelpManager::GetHandle()：現在アクティブになっているコントロールのヘルプが保存されていると判断されるヘルプファイルの名前を知りたいとき，ヘルプビューアはこの関数を呼び出します。

void IHelpManager::Release()：ヘルプビューアが切断するとき，この関数を呼び出して Help Manager に切断を通知します。IHelpManager::ShutDown() を介したリクエストに応答するときは，絶対にこの関数を呼び出してはいけません。IHelpManager::Release を使うのは，予期されない切断を Help Manager に伝えるときだけです。

キーワードベースのヘルプを表示する

一般に，ヘルプリクエストはキーワードベースのヘルプかコンテキストベースのヘルプとしてヘルプビューアに送られます。キーワードベースの場合，ヘルプビューアに対し，特定の文字列に基づいてヘルプを表示するよう要求します。コンテキストベースの場合は，特定の数値の識別子に基づいてヘルプを表示するよう要求します。

CLX Windows で動作するアプリケーションの場合，デフォルトのヘルプリクエストは数値のヘルプコンテキストです。Windows アプリケーションは WinHelp システムを使います。CLX は WinHelp システムをサポートしますが，Linux ヘルプシステムの大部分は WinHelp システムを理解しないので，CLX アプリケーションでは使わないようにしてください。

ICustomHelpViewer を実装するには，キーワードベースのヘルプリクエストをサポートする必要があります。これに対し，IExtendedHelpViewer を実装するには，コンテキストベースのヘルプリクエストをサポートする必要があります。

ICustomHelpViewer には，キーワードベースのヘルプを処理するメソッドが 3 つあります。

- UnderstandsKeyword
- GetHelpStrings
- ShowHelp

```
int__fastcall ICustomHelpViewer::UnderstandsKeyword(const AnsiString HelpString)
```

Help Manager は，3 つのメソッドのうち前記のメソッドを最初に呼び出します。Help Manager は，登録したヘルプビューアを 1 つずつ同じ文字列で呼び出し，その文字列に対するヘルプを表示できるかたずねます。このヘルプリクエストへの応答として，ヘルプビューアは，表示できるヘルプページ数を整数で応答します。ヘルプビューア側は，IDE の内部で任意のメソッドを使って応答内容を決定できます。たとえば HyperHelp ビューアは独自のインデックスを管理しており，そのインデックスを検索します。ヘルプビューアが指定のキーワードのヘルプをサポートしていないければ，ゼロを返します。現時点では負の数はゼロと解釈されますが，今後のリリースでは，この動作は保証されません。

```
Classes::TStringList*__fastcall ICustomHelpViewer::GetHelpStrings(const AnsiString
HelpString)
```

同じトピックに対して複数のヘルプビューアがヘルプを表示できる場合，Help Manager は前記のメソッドを呼び出します。ヘルプビューアは TStringList を返し，Help Manager がそれを解放します。このリストにある文字列を，当該キーワードで使用できるページにマッピングする必要があります。ただし，ヘルプビューアによってマッピングの特性が異なります。Windows の WinHelp ビューアおよび Linux の HyperHelp ビューアの場合，文字列リストには必ずエントリが 1 つ入っています。HyperHelp は独自にインデックスを持っているので，そのエントリの複製を別に作成するのは，むだな重複になるでしょう。Man ページビューア（Linux）の場合，文字列リストは複数の文字列で構成され，当該キーワードに対するページを格納するマニュアルの各セクションにつき文字列 1 つが対応します。

```
void__fastcall ICustomHelpViewer::ShowHelp(const AnsiString HelpString)
```

特定のキーワードに関するヘルプをヘルプビューアに表示させたいとき，Help Manager は前記のメソッドを呼び出します。このメソッドは，一連の操作の最後に呼び出されるメソッドです。すなわち，最初に UnderstandsKeyword が呼び出されない限り，このメソッドが呼び出されることはありません。

目次を表示する

ICustomHelpViewer には，目次の表示に関連するメソッドが 2 つあります。

- CanShowTableOfContents
- ShowTableOfContents

この操作の原理は，キーワードベースのヘルプを要求する関数と同じです。つまり，Help Manager は最初に ICustomHelpViewer::CanShowTableOfContents() を呼び出して各ヘルプビューアに問い合わせを行い，次に，ICustomHelpViewer::ShowTableOfContents() を呼び出して特定のヘルプビューアを起動します。

目次表示のリクエストに対してヘルプビューアが要求を拒否することも当然あり得ます。たとえば，Man ページビューアは実際に目次表示リクエストを拒否します。その理由は，目次のコンセプトと Man ページの機能方法とがうまくマッピングできないからです。一方，HyperHelp ビューアは目次をサポートします。HyperHelp ビューアは，目次表示リクエストを直接 Linux の HyperHelp および Windows の WinHelp に渡すことにより，目次要求に対応します。ただし，ICustomHelpViewer の実装が CanShowTableOfContents による問い合わせに対して **true** と応答し，かつ，ShowTableOfContents によるリクエストを無視することは非現実的です。

IExtendedHelpViewer の実装

ICustomHelpViewer は，キーワードベースのヘルプを直接サポートするにすぎません。ヘルプシステムによっては（特に WinHelp），ヘルプシステムの内部でキーワードと数字（コンテキスト ID）を関連付けて，アプリケーションに対しては非表示の形で機能するものがあります。このようなヘルプシステムの場合，アプリケーションがコンテキストベースのヘルプをサポートしている必要があります。コンテキストベースのヘルプでは，アプリケーションは（文字列でなく）コンテキストをもとにヘルプシステムを起動し，ヘルプシステムは数値自体を翻訳します。

VCL か CLX で作成したアプリケーションであれば，コンテキストベースのヘルプを必要とするシステムと通信できます。そのためには，ICustomHelpViewer を実装するオブジェクトを拡張して，

IExtendedHelpViewer を実装します。IExtendedHelpViewer は，トピックのジャンプを可能にするヘルプシステムとの通信もサポートします。このため，ユーザーはキーワード検索を使うかわりに，高レベルのトピックへと直接ジャンプすることができます。組み込みの WinHelp ビューアは，自動的にこの実装を行います。

IExtendedHelpViewer は 4 つの関数をエクスポーズ（公開）します。そのうちの 2 つ（UnderstandsContext と DisplayHelpByContext）は，コンテキストベースのヘルプをサポートするのに使います。残りの 2 つ（UnderstandsTopic と DisplayTopic）は，トピックのサポートで使用します。

アプリケーションのユーザーが〔F1〕キーを押すと，Help Manager は次の関数を呼び出します。

```
int__fastcall IExtendedHelpViewer::UnderstandsContext(const int ContextID, AnsiString
HelpFileName)
```

現在アクティブになっているコントロールは，キーワードベースでなくコンテキストベースのヘルプをサポートしています。ICustomHelpViewer::UnderstandsKeyword() と同様に，Help Manager は，登録済みの各ヘルプビューアに問い合わせをします。ただし，ICustomHelpViewer::UnderstandsKeyword() と異なり，指定のコンテキストをサポートするビューアが複数存在する場合は，最初に登録したビューアが呼び出されます。

次に，Help Manager は以下の関数を呼び出します。

```
void__fastcall IExtendedHelpViewer::DisplayHelpByContext(const int ContextID, AnsiString
HelpFileName)
```

この関数を呼び出すのは，Help Manager が各ヘルプビューアに問い合わせをした後です。

トピックをサポートする関数も同じように機能します。

```
bool__fastcall IExtendedHelpViewer::UnderstandsTopic(const AnsiString Topic)
```

この関数は，指定されたトピックをサポートしているかをヘルプビューアに問い合わせます。

```
void__fastcall IExtendedHelpViewer::DisplayTopic(const AnsiString Topic)
```

この関数は，指定されたトピックのヘルプを表示できると答えたヘルプビューアのうち，最初に登録されたビューアを呼び出します。

IHelpSelector の実装

IHelpSelector は，ICustomHelpViewer とペアで使います。指定のキーワード，コンテキスト，トピックに対するヘルプ，または目次をサポートすると答えた登録済みヘルプビューアが複数ある場合，Help Manager はどれか 1 つを選択しなければなりません。コンテキストかトピックの場合，Help Manager は，サポートすると応答した最初のヘルプビューアを必ず選択します。キーワードか目次の場合，デフォルトでは最初のヘルプビューアを選択します。ただし，アプリケーション側でこの動作をオーバーライドすることができます。

キーワードか目次に対する Help Manager の選択をオーバーライドするには，アプリケーションは，IHelpSelector インターフェースを実装するクラスを登録する必要があります。IHelpSelector は 2 つの関数（SelectKeyword と TableOfContents）をエクスポーズします。両方とも引数として TStrings をとります。TStrings は，キーワードマッチ候補か，目次を表示できると応答したヘルプビューアの名前を 1 つずつ格納します。実装クラスでは，選択された文字列を表す（TStringList 内の）インデックスを返す必要があります。その後，Help Manager が TStringList を解放します。

メモ 文字列を並べ替えると Help Manager が混乱することがあるので，IHelpSelector の実装クラスではこれを行わないようにしてください。ヘルプシステムがサポートする HelpSelector は 1 つだけです。つまり，別のセレクタが新規登録されると，前に存在していたセレクタは切断されます。

ヘルプシステムオブジェクトの登録

Help Manager がヘルプシステムオブジェクトと通信するには，4 つのインターフェース（ICustomHelpViewer，IExtendedHelpViewer，ISpecialWinHelpViewer，IHelpSelector）を実装したオブジェクトを Help Manager に登録する必要があります。

ヘルプシステムオブジェクトを Help Manager に登録する手順は次のとおりです。

- ヘルプビューアを登録する
- ヘルプセレクタを登録する

ヘルプビューアを登録する

オブジェクトの実装を記述するユニットには HelpIntfs を使います。実装ユニットのヘッダーファイルで，オブジェクトのインスタンスを宣言します。

実装ユニットに pragma startup 指令（インスタンス変数を割り当て，それを RegisterViewer 関数に渡すメソッドを呼び出す指令）をインクルードします。RegisterViewer は，HelpIntfs.pas からエクスポートされるフラットな関数で，ICustomHelpViewer を引数にとり，IHelpManager を返します。その後も使えるように IHelpManager を保存しておきます。

対応する .cpp ファイルに，インターフェースを登録するコードが格納されます。上で説明したインターフェースの場合，登録コードは次のようになります。

```
void InitServices()
{
  THelpImplementor GlobalClass;
  Global = dynamic_cast<ICustomHelpViewer*>(GlobalClass);
  Global->AddRef;
  HelpIntfs::RegisterViewer(Global, GlobalClass->Manager);
}
#pragma startup InitServices
```

メモ Help Manager のオブジェクトがまだ解放されていない場合は，GlobalClass オブジェクトのデストラクタで解放しなければなりません。

ヘルプセレクタを登録する

オブジェクトの実装を記述するユニットには，VCL の Forms か CLX の QForms を使います。実装ユニットの .cpp ファイルで，オブジェクトのインスタンスを宣言します。

実装ユニットでヘルプセレクタを登録します。これを行うには，グローバルオブジェクト Application の HelpSystem プロパティを使います。

```
Application->HelpSystem->AssignHelpSelector(myHelpSelectorInstance)
```

このメソッドは値を返しません。

VCL アプリケーションでのヘルプの使い方

次の節では，VCL アプリケーション内でのヘルプの使い方について説明します。

- TApplication はどのように VCL ヘルプを処理するか
- VCL コントロールはどのようにヘルプを処理するか
- ヘルプシステムを直接呼び出す
- IHelpSystem の使い方

TApplication はどのように VCL ヘルプを処理するか

VCL の TApplication は，ヘルプ処理に関連するメソッドを 4 つ提供しています。いずれもアプリケーションコードからアクセスできます。

表 7.5　TApplication のヘルプメソッド

メソッド	動作
HelpCommand	Windows のヘルプスタイル HELP_COMMAND を受け取り，そのまま WinHelp に渡す。このメカニズムを通じて送られたヘルプリクエストは，ISpecialWinHelpViewer の実装部分にのみ渡される
HelpContext	コンテキストベースヘルプのリクエストを使ってヘルプシステムを起動する
HelpKeyword	キーワードベースのヘルプを使ってヘルプシステムを起動する
HelpJump	特定のトピックの表示を要求する

これら 4 つの関数は，自分に渡されたデータを受け取り，TApplication のデータメンバー（ヘルプシステムを表すデータメンバー）を通じてそのデータを転送します。このデータメンバーは，HelpSystem プロパティから直接アクセスできます。

VCL コントロールはどのようにヘルプを処理するか

TControl から派生するどの VCL コントロールも，ヘルプシステムが使用するいくつかのプロパティ（HelpType，HelpContext，HelpKeyword）をエクスポーズします。

HelpType プロパティは列挙型のインスタンスを保持し，コントロールデザイナがキーワードベース，コンテキストベースのどちらのヘルプが表示されると予期するかを決定します。HelpType が htKeyword に設定されていれば，ヘルプシステム側では，コントロールがキーワードベースのヘルプを使うものと想定します。よって，ヘルプシステムは HelpKeyword プロパティの中身だけに注目します。逆に，HelpType が htContext に設定されていれば，ヘルプシステム側では，コントロールがコンテキストベースのヘルプを使うものと想定します。よって，ヘルプシステムは HelpContext プロパティの中身だけに注目します。

これらのプロパティに加えて，コントロールは InvokeHelp という単一のメソッドをエクスポーズします。リクエストをヘルプシステムに渡すには，InvokeHelp を呼び出します。InvokeHelp はパラメータをとらず，グローバルオブジェクト Application にあるメソッドのうち，コントロールがサポートするヘルプの型に対応するメソッドを呼び出します。

〔F1〕キーが押されると，ヘルプメッセージが自動的に呼び出されます。これは，TWinControl の KeyDown メソッドが InvokeHelp を呼び出すからです。

CLX アプリケーションでのヘルプの使い方

次の節では，CLX アプリケーション内でのヘルプの使い方について説明します。

- TApplication はどのように CLX ヘルプを処理するか
- CLX コントロールはどのようにヘルプを処理するか
- ヘルプシステムを直接呼び出す
- IHelpSystem の使い方

TApplication はどのように CLX ヘルプを処理するか

CLX の TApplication は，ヘルプ処理に関連するメソッドを 2 つ提供しています。どちらもアプリケーションコードからアクセスできます。

- ContextHelp：コンテキストベースヘルプのリクエストを使ってヘルプシステムを起動します。
- KeywordHelp：キーワードベースヘルプのリクエストを使ってヘルプシステムを起動します。

両方とも，渡されたコンテキストまたはキーワードを引数としてとり，TApplication のデータメンバー（ヘルプシステムを表すデータメンバー）を介してリクエストを先へと送ります。このデータメンバーは，読み出し専用プロパティ HelpSystem から直接アクセスできます。

CLX コントロールはどのようにヘルプを処理するか

TControl から派生するどのコントロールも，ヘルプシステムが使用する 4 つのプロパティ（HelpType，HelpFile，HelpContext，HelpKeyword）をエクスポーズします。HelpFile プロパティは，コントロールのヘルプを保存しているファイルの名前を保持します。コントロールのヘルプが外部のヘルプシステムに保存され，そのヘルプシステムがファイル名を考慮しない場合（Man ページシステムなど）には，HelpFile プロパティは空白のまま残されます。

HelpType プロパティは列挙型のインスタンスを保持し，コントロールデザイナがキーワードベース，コンテキストベースのどちらのヘルプが表示されると予期するかを決定します。残る 2 つのプロパティは，HelpFile，HelpType と連携して使われます。HelpType が htKeyword に設定されていれば，ヘルプシステム側では，コントロールがキーワードベースのヘルプを使うものと想定します。よって，ヘルプシステムは HelpKeyword プロパティの中身だけに注目します。逆に，HelpType が htContext に設定されていれば，ヘルプシステム側では，コントロールがコンテキストベースのヘルプを使うものと想定します。よって，ヘルプシステムは HelpContext プロパティの中身だけに注目します。

これらのプロパティに加えて，コントロールは InvokeHelp という単一のメソッドをエクスポーズします。リクエストをヘルプシステムに渡すには，InvokeHelp を呼び出します。InvokeHelp はパラメータをとらず，グローバルオブジェクト Application にあるメソッドのうち，コントロールがサポートするヘルプの型に対応するメソッドを呼び出します。

〔F1〕キーが押されると，ヘルプメッセージが自動的に呼び出されます。これは，TWidgetControl の KeyDown メソッドが InvokeHelp を呼び出すからです。

ヘルプシステムを直接呼び出す

VCL と CLX が提供していない追加のヘルプシステム機能として，TApplication は，ヘルプシステムに直接アクセスさせる読み出し専用のプロパティを提供します。この読み出し専用プロパティは，IHelpSystem インターフェースのインスタンスです。IHelpSystem も IHelpManager も同じオブジェクトで実装されますが，一方のインターフェースはアプリケーションから Help Manager への通信を可能にし，他方のインターフェースはヘルプビューアから Help Manager への通信を可能にします。

IHelpSystem の使い方

IHelpSystem を使うと，VCL/CLX アプリケーションは次の 3 つのことができます。

- Help Manager にパス情報を提供する
- 新しいヘルプセレクタを提供する
- Help Manager にヘルプの表示を要求する

同一キーワードに対して外部の複数のヘルプシステムがヘルプを表示できる場合に備えてヘルプセレクタを割り当てておくと，Help Manager は判断を委任することができます。詳細については，7-32 ページの「IHelpSelector の実装」を参照してください。

IHelpSystem は以下の 4 つの手続きと関数 1 つをエクスポートして，Help Manager にヘルプ表示を要求します。

- ShowHelp
- ShowContextHelp
- ShowTopicHelp
- ShowTableOfContents
- Hook

Hook は WinHelp 互換だけを目的とした関数なので，CLX アプリケーションでは使いません。Hook を使うと，キーワードベース，コンテキストベース，およびトピックベースのヘルプの要求に直接マッピングできない WM_HELP メッセージを処理できます。残る 3 つの関数は，引数を 2 つとります。1 つは，要求するヘルプのキーワード，コンテキスト ID，またはトピック，もう 1 つは，当該ヘルプを保存していると予想されるヘルプファイルです。

基本的に，トピックベースのヘルプを要求するのでない限り，コントロールの InvokeHelp を通じて Help Manager にヘルプ要求を渡しても効果は同じです。また，InvokeHelp の方が明確な方法です。

IDE ヘルプシステムのカスタマイズ

C++Builder IDE も，VCL/CLX アプリケーションとまったく同じ方法で複数のヘルプビューアをサポートします。すなわち，Help Manager にヘルプ要求を委任し，Help Manager から登録済みヘルプビューアへとヘルプ要求を送ります。IDE も，VCL が使用しているのと同じ WinHelpViewer を利用します。

IDE に新しいヘルプビューアをインストールする手順は，VCL/CLX アプリケーションでインストールするときと同様ですが，1 点だけ違いがあります。ICustomHelpViewer（および必要に応じて IExtendedHelpViewer）を実装したオブジェクトを記述して，目的の外部ヘルプビューアにヘルプリクエストを送ります。次に，IDE に ICustomHelpViewer を登録します。

カスタムのヘルプビューアを IDE に登録する手順は次のとおりです。

1. 目的のヘルプビューアを実装するユニットに HelpIntfs.cpp が含まれているか確認します。
2. IDE に登録された設計時パッケージへとユニットをビルドし，実行時パッケージをオンにしてその設計時パッケージをビルドします（ユニットが使用する Help Manager インスタンスと，IDE が使用する Help Manager インスタンスを同じにするため，この作業を行う必要があります）。
3. ユニット内に目的のヘルプビューアがグローバルインスタンスとして存在しているか確認します。
4. #pragma startup ブロックの初期化関数で，このインスタンスが RegisterHelpViewer 関数に渡されているか確認します。

第8章

アプリケーションユーザーインターフェースの作成

C++Builder を起動するか，新規プロジェクトの作成を開始すると，空白のフォームが画面に表示されます。アプリケーションのユーザーインターフェース（UI）を設計するには，まず，ウィンドウ，メニュー，ダイアログボックスなどのビジュアルコンポーネントをコンポーネントパレットからフォームに配置します。

次に，配置したコンポーネントの位置，大きさなどの設計時プロパティを設定し，そのイベントハンドラを記述します。このときアプリケーションの基礎となるプログラムコードが，C++Builder によって自動的に記述されます。

以下の節では，ユーザーインターフェースに関する主な作業（フォームの操作，コンポーネントパレットの作成，ダイアログボックスの追加，メニューとツールバー用のアクションの構成など）について説明します。

アプリケーションの動作を制御する

TApplication，TScreen，および TForm は，ユーザーのプロジェクトの動作を制御して C++Builder アプリケーションのバックボーンを形成するクラスです。TApplication クラスは，標準プログラムの動作をカプセル化するプロパティおよびメソッドを提供して アプリケーションの基礎を形成します。TScreen は実行時に使用され，ロードされたフォームおよびデータモジュールを，画面解像度および表示に利用可能なフォントなどのシステム固有の情報と同様にトラッキングします。TForm クラスのインスタンスは，ユーザーアプリケーションのユーザーインターフェースの構築ブロックです。アプリケーションのウィンドウとダイアログボックスは TForm に基づいて作成されます。

アプリケーションレベルでの作業

TApplication 型のグローバル変数 Application は，すべての VCL アプリケーションおよび CLX アプリケーションに含まれています。Application は，プログラムのバックグラウンドで行われる多くの機能を提供するとともに，アプリケーションをカプセル化します。たとえば Application は，プログラムのメニューからヘルプファイルをどのように呼び出すかを処理します。TApplication がどのように働くかを理解することは，アプリケーションの開発者よりもコンポーネントの記述者にとって重要ですが，プロジェクトを作成する際には，Application が処理するオプションをプロジェクトオプションの［アプリケーション］ページで設定する必要があります。

加えて，Application は，アプリケーション全体で発生する多くのイベントを受け取ります。たとえば OnActivate イベントを使うと，アプリケーションの起動時に処理を実行させることができます。同様に，OnIdle イベントを使ってアプリケーションのアイドル時にバックグラウンド処理を実行させる，OnMessage イベントを使って Windows メッセージをトラップする，OnEvent イベントを使ってイベントをトラップする，などの操作ができます。IDE を使ってグローバル変数 Application のプロパティとイベントを調べることはできませんが，TApplicationEvents というコンポーネントを使うと，Application のイベントに割り込みを入れ，IDE を介してイベントハンドラを挿入することができます。

スクリーンの扱い方

プロジェクトを作成すると，TScreen 型のグローバル変数 Screen が作成されます。Screen は，ユーザーのアプリケーションが実行されているスクリーンの状態をカプセル化します。Screen によって実行される一般的なタスクは次のとおりです。

- カーソルの外観を指定する
- アプリケーションが実行しているウィンドウのサイズを指定する
- スクリーンデバイスで利用可能なフォントのリストを指定する
- 複数のスクリーン動作を指定する（Windows のみ）

Windows アプリケーションが複数のモニタ上で実行されている場合，Screen はモニタとその寸法のリストを保持してユーザーインターフェースのレイアウトを効果的に管理できるようにします。

CLX を使用するクロスプラットフォームプログラミングの場合，デフォルトの動作では，アプリケーションが現在のスクリーンデバイス情報をもとにスクリーンコンポーネントを作成し，それを Screen に代入します。

フォームの設定

TForm は，GUI アプリケーションを作成するためのキーとなるクラスです。C++Builder を起動してデフォルトのプロジェクトを表示するか，新規プロジェクトの作成を開始すると，フォームが 1 つ表示されます。このフォームから UI 設計を開始します。

メインフォームの使い方

デフォルトでは，最初に作成してプロジェクトに保存したフォームがプロジェクトのメインフォームになります。プロジェクトにフォームを追加するときには，アプリケーションのメインフォームとして別のフォームを指定できます。また，フォーム作成順序を変更しない限りアプリケーション実行時に最初に表示されるフォームがメインフォームになるので，フォームをメインフォームに指定することで，実行時に簡単にテストできます。

プロジェクトのメインフォームを変更する手順は次のとおりです。

1. ［プロジェクト｜オプション］を選択して，［フォーム］ページを選びます。
2. ［メインフォーム］コンボボックスでプロジェクトのメインフォームにするフォームを選択して［OK］を選びます。

アプリケーションを実行すると，メインフォームに指定したフォームが表示されます。

メインフォームを非表示にする

アプリケーションが最初に起動したときに，メインフォームを非表示にすることができます。そのためには，グローバル変数 Application を使います（第 8 章-2 の「アプリケーションレベルでの作業」を参照）。

起動時にメインフォームを非表示にするには，次のようにします。

1. ［プロジェクト｜ソース表示］を選択し，メインプロジェクトファイルを表示します。
2. `Application->CreateForm()` と `Application->Run()` の呼び出しの間に，以下の行を追加します。

   ```
   Application->ShowMainForm = false;
   ```

3. オブジェクトインスペクタで，メインフォームの Visible プロパティを **false** に設定します。

フォームを追加する

プロジェクトにフォームを追加するには，［ファイル｜新規作成｜フォーム］を選択します。プロジェクトのフォームおよび関連ユニットを一覧表示するには，プロジェクトマネージャ（［表示｜プロジェクトマネージャ］）を使います。フォームだけを一覧表示するには，［表示｜フォーム］を選択します。

フォームをリンクする

プロジェクトへのフォームの追加はプロジェクトファイルにそのフォームへの参照を追加しますが，プロジェクト内のほかのユニットには追加しません。フォームへの参照を参照フォームのユニットファイルに追加しなければ，新しいフォームを参照するコードは記述できません。これをフォームリンクといいます。

フォームをリンクする一般的な理由は，そのフォームのコンポーネントへのアクセスが提供されるためです。たとえば，フォームリンクを使ってデータベース対応コンポーネントを持つフォームをデータモジュールのデータアクセスコンポーネントに接続して利用できるようにします。

フォームをリンクする手順は次のとおりです。

1. 別のフォームへの参照が必要なフォームを選択します。
2. ［ファイル｜ユニットヘッダーファイルの追加］を選択します。
3. 参照されるフォームのフォームユニット名を選択します。
4. ［OK］をクリックします。

フォームを別のフォームにリンクすることは，フォームユニットが別のフォームユニットのヘッダーを持つことを意味します。つまり，リンクされたフォームとフォーム上のコンポーネントはリンクしたフォームのスコープ内になります。

レイアウトを管理する

もっとも簡単な方法として，フォーム内のどこにコントロールを配置するかによってユーザーインターフェースのレイアウトを管理できます。選択された配置は，コントロールの Top，Left，Width，および Height プロパティに反映されます。実行時にこれらの値を変更して，フォーム内のコントロールの位置とサイズを変更できます。

この他にも，コントロールには，自動的にコンテンツまたはコンテナを調整できるプロパティがあります。これによって，フォームを全体的に統一された形式に収まるようにレイアウトできます。

コントロールの位置合わせおよびサイズが親との関連でどのように決定されるかに影響するプロパティが 2 つあります。Align プロパティを使用すると，ほかのコントロールが配置された後で，コントロールを親の特定な端に沿ってまたはクライアント領域全体を埋めるように親のサイズに完全に合わせるように強制できます。親がサイズ変更されると，それに合わせて位置合わせされたコントロールが自動的にサイズ変更され，特定の端に合わせて配置されます。

コントロールを親の特定の端に対して位置合わせしたいが，必ずしもその端に接触させたり，サイズ変更したくない場合，つまり端全体に対して位置合わせしたい場合には，Anchors プロパティを使用できます。

コントロールが大きくなりすぎたり，小さくなりすぎないようにしたい場合は，Constraints プロパティを使用します。Constraints を使用すると，コントロールの高さの最大値と最小値，および幅の最大値と最小値を指定できます。これらの値を設定してコントロールの高さと幅のサイズ（ピクセル単位）を制限します。たとえば，コンテナオブジェクト上の Constraints の MinWidth と MinHeight を設定すると，子オブジェクトが常に見えるようにできます。

配置される子がサイズ制限を持っているため，Constraits の値は親 / 子階層を通じて継承されるので，オブジェクトのサイズは制限できます。Constraints は，ChangeScale メソッドが呼び出されたときにコントロールが特定の寸法にスケールされるのを防ぐこともできます。

TControl は，TConstrainedResizeEvent 型のプロテクトイベント OnConstrainedResize を導入します。

```
void __fastcall (__closure *TConstrainedResizeEvent)(System::TObject* Sender,
  int &MinWidth, int &MinHeight, int &MaxWidth, int &MaxHeight);
```

このイベントを利用すると，コントロールをリサイズしようとしたときのサイズ制限をオーバーライドできます。制限の値は参照パラメータとして渡されるので，イベントハンドラ内で変更できます。OnConstrainedResize はコンテナオブジェクト（TForm，TScrollBox，TControlBar，および TPanel）でパブリッシュになります。また，コンポーネントを作成するときに，このイベントを TControl の下位オブジェクトで使用したりパブリッシュにできます。

内容によってサイズを変更できるコントロールは，フォントや含まれているオブジェクトに合わせてサイズを調整する AutoSize プロパティを持っています。

フォームの使い方

C++Builder の IDE でフォームを作成すると，C++Builder は自動的にアプリケーションのメインエントリポイントにコードを生成して，メモリ上にフォームを作成します。通常，これは望ましい動作であり，これを変更する必要はありません。つまり，メインウィンドウがプログラムの持続期間を通じて固定されており，メインウィンドウ用のフォームを作成するときにデフォルトの C++Builder の動作を変更する必要はないでしょう。

しかし，プログラム実行中にアプリケーションのすべてのフォームがメモリ中にあっては困る場合があります。つまり，アプリケーションのすべてのダイアログが同時にメモリ中に存在しないようにするには，ダイアログを表示させたいときだけ動的にダイアログを作成します。

フォームは，モード付きまたはモードなしにできます。モード付きフォームは，ユーザーが別のフォームに切り替える前に何らかの対話をしなければならないフォームです（たとえば，ダイアログボックスがユーザーの入力を要求する）。モードなしフォームは，別のウィンドウの下になって隠されるか，またはユーザーが閉じるか最小化するまで表示されているウィンドウです。

フォームをメモリに格納するタイミングを制御する

デフォルトでは，C++Builder は自動的に次のコードをアプリケーションのメインエントリポイントに挿入し，アプリケーションのメインフォームをメモリ上に作成します。

```
Application->CreateForm(__classid(TForm1), &Form1);
```

この関数はフォームと同じ名前のグローバル変数を作成します。したがって，アプリケーションのすべてのフォームは自分自身に対応するグローバル変数を持ちます。この変数は，フォームのクラスのインスタンスへのポインタで，アプリケーションの実行中にフォームを参照するために使用されます。フォームのヘッダー（.h）ファイルを含むソースコード（.cpp）ファイルは，この変数を使ってフォームにアクセスできます。

このフォームはアプリケーションのメインエントリポイントに追加されるため，プログラムが起動されるとフォームが表示され，アプリケーションの持続期間中メモリ内に存在しています。

自動生成されるフォームを表示する

プログラム開始時にひとまずフォームを作成しておき，実行中の特定の時点でそのフォームを表示するように指定するには，フォームのイベントハンドラは ShowModal メソッドを使って，すでにメモリに読み出されているフォームを表示します。

```
void __fastcall TMainMForm::FirstButtonClick(TObject *Sender)
{
   ResultsForm->ShowModal();
}
```

この場合，フォームはすでにメモリ上に存在するため，新しくインスタンスを作成したり破棄したりする必要はありません。

フォームを動的に生成する

アプリケーションのすべてのフォームを，常にメモリに格納しておく必要はありません。読み出し時に必要なメモリ容量を減らすには，特定のフォームを必要時にだけ作成するようにします。たとえばダイアログボックスがメモリ上に存在しなければならないのは，ユーザーがそれを使って対話している間だけです。

IDE を使って，実行中の任意の時点で生成されるフォームを作成するには，以下のようにします。

1. メインメニューから［ファイル｜新規作成｜フォーム］を選択して，新しいフォームを表示します。
2. ［プロジェクトオプション］ダイアログの［フォーム］ページにある［自動作成の対象］リストからフォームを削除します。

   これにより，フォームの呼び出しが削除されます。別の方法として，プログラムのメインエントリポイントから次の行を手動で削除することもできます。

   ```
   Application->CreateForm(__classid(TResultsForm), &ResultsForm);
   ```
3. フォームがモードなしの場合は，フォームの Show メソッドを使って，必要時にフォームを呼び出します。モード付きフォームの場合は，ShowModal メソッドを使用します。

フォームを動的に生成する場合，メインフォームのイベントハンドラでフォームのインスタンスを作成し，破棄しなければなりません。結果のフォームを呼び出す方法の 1 つは，以下のようにグローバル変数を使うことです。ResultsForm がモード付きフォームのため，ハンドラが ShowModal メソッドを使用することに注意してください。

```
void __fastcall TMainMForm::FirstButtonClick(TObject *Sender)
{
  ResultsForm = new TResultsForm(this);
  ResultsForm->ShowModal();
  delete ResultsForm;
}
```

前の例のイベントハンドラではフォームが閉じたときにフォームを削除しているため，アプリケーションの別の場所で ResultsForm が必要になったときには **new** を使用してフォームをもう一度作成する必要があります。Show を使用してフォームが表示された場合には，フォームが開いている間に Show から戻るので，イベントハンドラ内ではフォームを削除できません。

メモ **new** 演算子を使ってフォームを作成する場合，［プロジェクトオプション］ダイアログの［フォーム］ページの［自動作成の対象］リストにそのフォームが含まれていないことを確認してください。特に，リストにある同じ名前のフォームを削除せずに新しいフォームを作成した場合，プログラム開始時にリストにある方のフォームが生成されてしまいます。生成されたフォームのイベントハンドラはフォームの新しいインスタンスを生成し，自動生成されるインスタンスへの参照を上書きします。自動生成されたインスタンスは引き続き存在しますが，アプリケーションからそのインスタンスにアクセスすることはできません。イベントハンドラの実行が終了したとき，グローバル変数が指している

フォームは正しいものではありません。ここでグローバル変数を逆参照しようとすると，アプリケーションがクラッシュする可能性があります。

ウィンドウのようなモードなしフォームを作成する

モードなしフォームが使われている間は，そのフォームへの参照変数が確実に存在しているようにしなければなりません。つまり，これらの変数のスコープはグローバルである必要があります。たいていの場合，フォームを作成したときに定義されたグローバルな参照変数（フォームの Name プロパティに適合する名前を持つ変数）を使用します。アプリケーションがフォームの追加のインスタンスを必要とする場合は，各インスタンスごとにフォームクラスを指すポインタ型の別のグローバル変数を宣言します。

ローカル変数を使ってフォームのインスタンスを作成する

モード付きフォームの一意なインスタンスを安全に作成する方法は，イベントハンドラの中でローカル変数を使って新しいインスタンスを参照することです。ローカル変数を使うと，ResultsForm が自動作成されるかどうかは問題ではなくなります。イベントハンドラの中でグローバル変数への参照は行われません。次に例を示します。

```
void __fastcall TMainMForm::FirstButtonClick(TObject *Sender)
{
  TResultsForm *rf = new TResultsForm(this);  // rf はローカルフォームのインスタンス
  rf->ShowModal();
  delete rf;                                  // フォームが安全に破棄された
}
```

このイベントハンドラで，どうやってフォームのグローバルインスタンスを使わないようにしているかに注意してください。

一般的に，アプリケーションはフォームのグローバルインスタンスを使います。しかし，モード付きフォームの新しいインスタンスが必要で，そのフォームの使用がアプリケーションの独立したセクション（1 つの関数内など）に限定される場合は，フォームを取り扱うもっとも安全で効果的な方法がローカルインスタンスです。

モードなしフォームのイベントハンドラで，ローカル変数を使うことはできません。これは，フォームが使用されている間，確実にフォームが存在することを保証するために，フォームのスコープがグローバルでなければならないからです。Show 関数はフォームが開くと同時に終了するので，使われていたローカル変数は，その時点でスコープの外に出ることになります。

フォームへの追加の引数の受け渡し

一般的に，アプリケーションで使用されるフォームは IDE で作成します。IDE で作成されたフォームには，作成されたフォームのオーナーへのポインタを示す 1 つの引数 Owner をとるコンストラクタを持ちます。このオーナーは，呼び出し側のアプリケーションオブジェクトまたはフォームオブジェクトです。Owner は，NULL にすることもできます。

フォームに別の引数を渡すには別のコンストラクタを作成し，そのフォームを **new** 演算子を使ってインスタンス化します。たとえば，次のフォームクラスは，追加の引数 whichButton を持つ新しいコンストラクタを示しています。この新しいコンストラクタは，ユーザーが手作業でフォームクラスに追加します。

```
class TResultsForm : public TForm
{
__published:    // IDE 管理コンポーネント
   TLabel *ResultsLabel;
   TButton *OKButton;
   void __fastcall OKButtonClick(TObject *Sender);
private:        // ユーザー宣言
public:         // ユーザー宣言
   virtual __fastcall TResultsForm(TComponent* Owner);
   virtual __fastcall TResultsForm(int whichButton, TComponent* Owner);
};
```

次に示すのは，追加の引数である whichButton が渡されるコンストラクタのコードです。このコンストラクタは whichButton 引数を使って，フォーム上の Label コントロールの Caption プロパティを設定します。

```
void__fastcall TResultsForm::TResultsForm(int whichButton, TComponent* Owner)
 : TForm(Owner)
{
   switch (whichButton) {
      case 1:
         ResultsLabel->Caption = "You picked the first button!";
         break;
      case 2:
         ResultsLabel->Caption = "You picked the second button!";
         break;
      case 3:
         ResultsLabel->Caption = "You picked the third button!";
   }
}
```

複数のコンストラクタを持つフォームのインスタンスを作成する場合は，目的に最適なコンストラクタを選択することができます。たとえば次に示すフォーム上のボタン用の OnClick ハンドラでは，2 つ目のパラメータを使って TResultsForm のインスタンスを作成しています。

```
void __fastcall TMainMForm::SecondButtonClick(TObject *Sender)
{
   TResultsForm *rf = new TResultsForm(2, this);
   rf->ShowModal();
   delete rf;
}
```

フォームからのデータの取得

ほとんどの実際のアプリケーションは，複数のフォームから構成されています。これらのフォーム間で情報をやりとりする必要が発生します。受け取りフォームのコンストラクタへのパラメータまたはフォームのプロパティへの値の割り当てにより，情報をフォームに送ることができます。フォームから情報を取得する方法は，フォームがモード付きかモードなしかに依存します。

モードなしフォームからデータを取得する

モードなしフォームからの情報は，フォームのパブリックメンバー関数を呼び出すか，プロパティを読み出すことで簡単に取得できます。たとえばフォームに，色のリスト（"Red"，"Green"，"Blue"，など）の入った ColorListBox というリストボックスを持つ ColorForm というモードなしフォームがあ

るとします。ColorListBox で選択された色の名前文字列は，ユーザーが色を選択するたびに自動的に CurrentColor というプロパティに格納されます。このフォームのクラス宣言は次のようになります。

```
class TColorForm : public TForm
{
   __published:    // IDE 管理コンポーネント
      TListBox *ColorListBox;
      void __fastcall ColorListBoxClick(TObject *Sender);
   private:        // ユーザー宣言
      AnsiString getColor();
      void   setColor(AnsiString);
      AnsiString curColor;
   public:         // ユーザー宣言
      virtual __fastcall TColorForm(TComponent* Owner);
      __property AnsiString CurrentColor = {read=getColor, write=setColor};
};
```

リストボックス ColorListBox のイベントハンドラ OnClick は，リストボックスで項目が選択されるごとに CurrentColor プロパティにその値を設定します。OnClick は，色の名前の入ったリストボックスから文字列を取り出して CurrentColor に代入します。CurrentColor プロパティは，色設定関数 setColor を使ってプロパティの実際の値をプライベートデータメンバー curColor に格納します。

```
void __fastcall TColorForm::ColorListBoxClick(TObject *Sender)
{
   int index = ColorListBox->ItemIndex;
   if (index >= 0) {  // 色を確実に選択する
      CurrentColor = ColorListBox->Items->Strings[index];
   }
   else           // 色が選択されていない場合
      CurrentColor = "";
}
//---------------------------------------------------------------------------
void TColorForm::setColor(AnsiString s)
{
   curColor = s;
}
```

ここで，アプリケーションにもう 1 つ ResultsForm というフォームが必要になったとします。これは，ResultsForm 上のボタン（UpdateButton）がクリックされたときに ColorForm で現在選択されている色を調べるために必要なものです。UpdateButton の OnClick イベントは，たとえば次のようになります。

```
void __fastcall TResultsForm::UpdateButtonClick(TObject *Sender)
{
   if (ColorForm) {  // ColorForm があることを確認する
      AnsiString s = ColorForm->CurrentColor;
      // 色の名前文字列を処理する
   }
}
```

イベントハンドラは，最初にポインタがヌルかどうかをチェックして ColorForm が存在していることを確認します。次に，ColorForm の CurrentColor プロパティの値を取り出します。CurrentColor の値を調べるとき，次に示される取得関数 getColor が呼び出されます。

```
String TColorForm::getColor()
{
   return curColor;
}
```

また，ColorForm の getColor 関数がパブリックの場合，別のフォームが現在選択されている色を CurrentColor プロパティ を使用せずに取り出せます（例：AnsiString=ColorForm->getColor();）。実際，もう一方のフォームから次のようにリストボックスでの選択内容を調べて，ColorForm で現在選択されている色を取り出す障害になるものは何もありません。

```
String s = ColorListBox->Items->Strings[ColorListBox->ItemIndex];
```

ただし，プロパティを使うと ColorForm へのインターフェースがずっと単純なものになります。あるフォームで ColorForm についての情報を知るために必要なのは，CurrentColor の値を調べることだけになります。

モード付きフォームからデータを取得する

モードなしフォームと同様に，たいていの場合モード付きフォームにもその他のフォームで必要とされる情報があります。一般的な例として，フォーム A からモード付きフォーム B を起動するような場合があります。フォーム B が閉じるときにフォーム A では，フォーム A の処理をどのように進めるかを決めるために，ユーザーがフォーム B で何をしたかを知る必要があります。フォーム B がまだメモリ上にあれば，前記のモードなしフォームのときと同様にプロパティやメンバー関数を使って情報を得ることができます。しかし，フォーム B が閉じたときにメモリ上からも削除されている場合のことを考えなければなりません。フォームには明示的な戻り値はないので，フォームが破棄される前にフォームの重要な情報を保持しなければなりません。

ここで，ColorForm がモード付きフォームとなるように変更したバージョンについて考えてみましょう。クラス宣言は次のようになります。

```
class TColorForm : public TForm
{
   __published:    // IDE 管理コンポーネント
      TListBox *ColorListBox;
      TButton *SelectButton;
      TButton *CancelButton;
      void __fastcall CancelButtonClick(TObject *Sender);
      void __fastcall SelectButtonClick(TObject *Sender);
   private:        // ユーザー宣言
      AnsiString* curColor;
   public:         // ユーザー宣言
      virtual __fastcall TColorForm(TComponent* Owner);
      virtual __fastcall TColorForm(AnsiString* s, TComponent* Owner);
};
```

このフォームには，色の名前のリストが入った ColorListBox というリストボックスがあります。SelectButton というボタンは，押されたときに ColorListBox で選択されている色の名前の記録を作ってからフォームを閉じます。CancelButton は，フォームを閉じるだけのボタンです。

引数 AnsiString* をとるユーザー定義のコンストラクタがクラスに追加されている点に注意してください。この String* は，ColorForm を起動するフォームが認識している文字列を指します。このコンストラクタの実装コードは次のようになります。

```
void __fastcall TColorForm::TColorForm(AnsiString* s, TComponent* Owner)
 : TForm(Owner)
{
   curColor = s;
   *curColor = "";
}
```

コンストラクタは，プライベートデータメンバー curColor へのポインタの保存と，文字列を空文字列に初期化することだけを行っています。

メモ 上記のユーザー定義コンストラクタを使用するには，フォームを明示的に作成しなければなりません。アプリケーション起動時に自動作成はされません。詳細は，8-5 ページの「フォームをメモリに格納するタイミングを制御する」を参照してください。

アプリケーションでは，ユーザーがリストボックスから色を選択し，SelectButton を押して選択内容を保存し，フォームを閉じることになります。SelectButton の OnClick イベントは，たとえば次のようになります。

```
void __fastcall TColorForm::SelectButtonClick(TObject *Sender)
{
   int index = ColorListBox->ItemIndex;
   if (index >= 0)
      *curColor = ColorListBox->Items->Strings[index];
   Close();
}
```

イベントハンドラが，コンストラクタに渡された文字列のアドレスに，選択された色の名前を格納していることに注意してください。

ColorForm を効率よく使うために，呼び出し側のフォームからは既存の文字列へのポインタをコンストラクタに渡さなければなりません。たとえば，ColorForm がフォーム ResultsForm 上の UpdateButton ボタンがクリックされたことに応答して，ResultsForm によってインスタンス化されたとします。イベントハンドラは，以下のようになります。

```
void __fastcall TResultsForm::UpdateButtonClick(TObject *Sender)
{
   AnsiString s;
   GetColor(&s);
   if (s != "") {
      // 色の名前文字列を処理する
   }
   else {
      // 色が選択されていない場合の処理
   }
}
//---------------------------------------------------------------------
void TResultsForm::GetColor(AnsiString *s)
{
   ColorForm = new TColorForm(s, this);
   ColorForm->ShowModal();
   delete ColorForm;
   ColorForm = 0; // ポインタをヌルにする
}
```

UpdateButtonClick は s という文字列を作成します。s のアドレスは，ColorForm を作成する GetColor 関数に渡され，s へのポインタが引数としてコンストラクタに渡されます。ColorForm は閉じるとすぐに削除されますが，選択されていた色の名前は MainColor に保存されています。色が何も選択されていなかった場合は MainColor に空文字列が入り，ユーザーが色を選択せずに ColorForm を閉じたことを示します。

この例では，モード付きフォームの情報を保持する文字列変数を 1 つ使います。もちろん必要に応じてもっと複雑なオブジェクトを使うこともできます。ここで覚えておくべきことは，何も変更や選択

が行われずにモード付きフォームが閉じたかどうかが，呼び出し側のフォームからわかる方法を必ず提供するということです（MainColor を空文字列に初期化する，などです）。

コンポーネントとコンポーネントグループの再利用

C++Builder には，コンポーネントを使った操作を保存，再利用する方法が何通りかあります。

- コンポーネントテンプレートを使うと，コンポーネントグループを簡単に構成，保存できます。8-12 ページの「コンポーネントテンプレートの作成と使い方」を参照してください。
- フォーム，データモジュール，プロジェクトをリポジトリに保存すると，再利用可能な要素のデータベースとして一元管理できます。フォームの継承を使えば，変更内容をほかのプロジェクトなどに適用できます。
- コンポーネントパレットやリポジトリにフレームを保存するという方法もあります。フレームを使うと，フォームの継承を使用して，ほかのフォームやフレームに埋め込むことができます。8-13 ページの「フレームの操作」を参照してください。
- カスタムコンポーネントはコードの再利用方法としてもっとも複雑な方法ですが，柔軟性は最高です。第 45 章「コンポーネント作成の概要」を参照してください。

コンポーネントテンプレートの作成と使い方

1 つ以上のコンポーネントをまとめてテンプレートにすることができます。コンポーネントをフォームに配置し，プロパティを設定して，コードを記述したら，単一のコンポーネントテンプレートとして保存します。あとは，コンポーネントパレットからテンプレートを選択するだけで，前に構成したコンポーネントがフォームに配置されます。同時に，関連のプロパティとイベント処理コードもプロジェクトに追加されます。

テンプレートをフォームに配置した後は，個々のコンポーネントの位置を変えたり，プロパティを再設定したりできます。また，単独のコンポーネントを貼り付ける場合と同様に，イベントハンドラの作成，修正ができます。

コンポーネントテンプレートを作成する手順は次のとおりです。

1. コンポーネントをフォームに置き，位置を揃えます。オブジェクトインスペクタでプロパティとイベントを設定します。
2. フォーム上でコンポーネントを選択します。複数のコンポーネントを一度に選択するには，すべてが含まれるようにドラッグして囲みます。各コンポーネントの隅に淡色のハンドルが表示されます。
3. ［コンポーネント｜コンポーネントテンプレートの作成］を選択します。
4. ［コンポーネントテンプレートの情報］編集ボックスにコンポーネント名を入力します。デフォルトのコンポーネント名として，手順 2 で最初に選択したコンポーネントの型に「Template」を付加した名前が最初に表示されます。たとえば，ラベル，編集ボックスの順に選択した場合，デフォルト名は「TLabelTemplate」となります。デフォルト名を変更してかまいませんが，既存のコンポーネントと重複しないように注意してください。

5. ［パレットページ名］編集ボックスに，テンプレートを置くコンポーネントパレットページを指定します。存在しないページ名を入力すると，テンプレートの保存時に新しいページが作成されます。
6. ［パレットアイコン］で，パレットに表示するアイコンのビットマップを選択します。デフォルトのアイコンとして，手順 2 で最初に選択したコンポーネントのビットマップが表示されます。ほかのビットマップを参照するには，［変更］をクリックします。選択するビットマップは，大きさが 24 × 24 ピクセル以下でなければなりません。
7. ［OK］を選択します。

コンポーネントパレットからテンプレートを削除するには，［コンポーネント｜パレットの設定］を選択します。

フレームの操作

フレーム（TFrame）は，フォームと同じように，ほかのコンポーネントを入れるコンポーネントです。フォームに入れるコンポーネントの自動インスタンス化，自動破棄については，フォームと同じオーナーシップメカニズムを使います。同様に，コンポーネントプロパティの同期についても，フォームと同じ親子関係を持ちます。

また，フォームよりむしろ，カスタマイズしたコンポーネントに似た部分もあります。フレームはコンポーネントパレットに保存できるので，簡単に再利用できます。また，フォーム内，ほかのフォーム内，ほかのコンテナオブジェクト内でフレームをネストできます。フレームを作成，保存した後も，引き続きユニットとして機能し，フレーム内のコンポーネント（別のフレームを含む）から変更内容を継承し続けます。フレームを別のフレームやフォームの中に埋め込んだ場合も，派生元フレームに加えられた変更内容を継承し続けます。

アプリケーション内の複数の場所でコントロールグループを使用する場合，フレームを使ってコントロールを整理すると便利です。たとえば，同じビットマップを複数のフォームで使用する場合は，そのビットマップをフレームに入れておきます。そうすれば，そのビットマップのコピー 1 つをアプリケーションのリソースに含めるだけですみます。また，テーブル編集を目的とした編集フィールド群を表す場合，テーブルにデータ入力する必要があるときに常にフレームを使用することもできます。

フレームの作成

空のフレームを作成するには，［ファイル｜新規作成｜フレーム］を選択するか，または［ファイル｜新規作成｜その他］を選択して［フレーム］をダブルクリックします。あとは，新規フレームにコンポーネント（別のフレームも含む）をドロップします。

プロジェクトの一部としてフレームを保存するのが通常はベストですが，必ずしもそうする必要はありません。フォームのない，フレームのみのプロジェクトを作成するには，まず［ファイル｜新規作成｜アプリケーション］を選択し，新規フォームとユニットを保存せずに閉じます。次に，［ファイル｜新規作成｜フレーム］を選択し，プロジェクトを保存します。

メモ フレームを保存するときは，Unit1，Project1 などのデフォルト名は使わないようにしてください。デフォルト名を使うと，保存したフレームを後で使う際に混乱が生じるおそれがあります。

現在のプロジェクトに含まれているフレームを設計時に表示するには，［表示｜フォーム］を選択し，目的のフレームを選択します。フォームやデータモジュールと同様に，フレームのフォームファイルとフォームデザイナの表示を交互に切り替えることができます。表示を切り替えるには，右クリックして［フォームとして表示］か［エディタで表示］を選択します。

コンポーネントファイルにフレームを追加する

フレームをコンポーネントパレットに追加する場合は，コンポーネントテンプレートとして追加されます。コンポーネントパレットにフレームを追加するには，フォームデザイナにフレームを開き（ここでは，別のコンポーネントに埋め込まれているフレームは使用不可），フレーム上で右クリックして［パレットに追加］を選択します。［コンポーネントテンプレートの情報］ダイアログボックスが表示されたら，コンポーネント名，パレットページ名，パレットアイコンを指定します。

フレームの使い方と変更方法

アプリケーションの中でフレームを使うには，直接または間接にフォーム上にフレームを配置しなければなりません。フォーム上だけでなく，別のフレームや別のコンテナオブジェクト（パネル，スクロールボックスなど）にも直接フレームを配置できます。

フォームデザイナでは，以下の 2 通りの方法でフレームをアプリケーションに追加できます。

- コンポーネントパレットでフレームを選択し，フォーム，別のフレーム，または別のコンテナオブジェクトにドロップする。必要に応じて，プロジェクトにフレームのユニットファイルを含めるようフォームデザイナが許可を求める
- コンポーネントパレットの［Standard］ページで［Frames］を選択し，フォーム上または別のフレーム上をクリックする。プロジェクトに追加済みのフレーム一覧を示すダイアログが表示されるので，目的のフレームを選択し，［OK］をクリックする

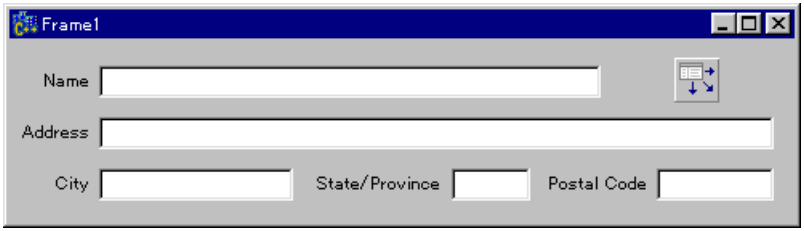
フレームをフォームまたは他のコンテナにドロップすると，選択したフレームから派生する新しいクラスが自動的に宣言されます（同様に，新規フォームをプロジェクトに追加すると，TForm から派生する新しいクラスが宣言されます）。つまり，元の（上位の）フレームに加えた変更は，埋め込みフレームに継承されます。これに対し，埋め込みフレームに加えた変更は，元の上位フレームには伝わりません。

たとえば，データアクセスコンポーネントとデータベース対応コントロールのグループをひとまとめにして，複数のアプリケーションで利用するとします。該当するデータベースコンポーネントを単一のコンポーネントテンプレートにまとめるという方法も可能ですが，その場合，テンプレートを使い始めた後でコントロールの並べ方を変えようとしたとき，テンプレートを置いたプロジェクトに戻って 1 つずつ手作業で修正しなければなりません。

これに対し，該当するコンポーネントを単一のフレームに置いておけば，後で修正の必要が生じても 1 か所を直すだけで済みます。元のフレームに加えた変更は，プロジェクトの再コンパイル時に自動的に埋め込み下位オブジェクトに継承されるからです。また，埋め込みフレームに加えた変更は元の

フレームにも他の埋め込み下位オブジェクトにも影響しないので，自由に埋め込みフレームを修正できます。埋め込みフレームの修正に関して 1 つだけ制限があります。それは，コンポーネントを追加できないことです。

図 8.1　データベース対応コントロールとデータソースコンポーネントを置いたフレーム



フレームを使うと，保守を簡易化できる上，リソースを有効に使用することができます。たとえばアプリケーションの中でビットマップなどのグラフィックを使う場合，TImage コントロールの Picture プロパティにグラフィックを読み込むという方法があります。しかしながら，同じグラフィックをアプリケーションの中で何度も使う場合，フォーム上に Image オブジェクトを 1 つずつ置いていくと，同じグラフィックのコピーがフォームのリソースファイルに次々と追加されることになります（TImage::Picture を一度設定して，イメージコントロールをコンポーネントテンプレートとして保存した場合でも同じです）。この方法より有効にリソースを使う方法があります。それは，イメージオブジェクトをフレームにドロップし，このフレームにグラフィックを読み込み，グラフィックを表示させたい場所でこのフレームを使うことです。この方がフォームファイルが小さくなります。その上，元のフレームでイメージを修正するだけで，グラフィックの使用箇所すべてを変更できるという利点もあります。

フレームの共有

ほかの開発者との間でフレームを共有できます。次の 2 通りの共有方法があります。

- オブジェクトリポジトリにフレームを追加する
- フレームのユニットファイル（.cpp と .h）およびフォームファイル（.dfm または .xfm）を配布する

リポジトリにフレームを追加するには，フレームの入ったプロジェクトのどれかを開き，フォームデザイナで右クリックして［リポジトリに追加］を選択します。詳細については，7-24 ページの「オブジェクトリポジトリの使い方」を参照してください。

フレームのユニットファイルとフォームファイルを配布した場合，受け取った開発者はファイルを開いた後，コンポーネントパレットに追加できます。フレームに埋め込みフレームが含まれている場合は，プロジェクトの一部として開く必要があります。

ダイアログボックスの作成

コンポーネントパレットの［Dialogs］ページにあるダイアログボックスコンポーネントを使うと，さまざまなダイアログボックスを自分のアプリケーションで使用できます。コモンダイアログボックスは，使いやすく一貫性のあるインターフェースをアプリケーションに提供します。このインター

フェースを使ってユーザーは，ファイルを開く，保存する，印刷するなどの共通のファイル操作を実行できます。ダイアログボックスはデータの表示 / 取得を行います。

Execute メソッドが呼び出されると，各ダイアログボックスが表示されます。つまり，ユーザーが［OK］を選択してダイアログボックス内の変更を受け入れた場合には，Execute メソッドは **true** を返します。ユーザーが［キャンセル］を選択して変更の反映も保存もしないでダイアログボックスを終了した場合には，Execute メソッドは **false** を返します。

CLX　クロスプラットフォームアプリケーションを開発する場合，CLX の QDialogs ユニットにあるダイアログボックスを使用できます。オペレーティングシステムに一般的タスク（ファイルを開く，保存する，フォントや色を変更するなど）用のネイティブのダイアログボックスがある場合は，UseNativeDialog プロパティを使用できます。このような環境でアプリケーションが動作し，かつ，Qt ダイアログのかわりにネイティブダイアログを使用したい場合には，UseNativeDialog プロパティを **true** に設定します。

［開く］ダイアログボックスの使い方

よく使われるダイアログボックスコンポーネントの代表が TOpenDialog です。通常は，フォームのメインメニューバーにある［ファイル］から［新規作成］か［開く］を選択すると，TOpenDialog が呼び出されます。このダイアログボックスにあるコントロールを使って，ユーザーは，ワイルドカード文字を使ってファイルのグループを選択したり，ディレクトリ間を移動したりできます。

TOpenDialog コンポーネントは，［開く］ダイアログボックスをアプリケーションから利用できるようにします。［開く］ダイアログボックスの目的は，オープンするファイルをユーザーに指定させることです。［開く］ダイアログボックスを表示するには，Execute メソッドを使います。

ユーザーが［OK］をクリックすると，選択されたファイルが TOpenDialog の FileName プロパティに格納されます。必要に応じて，この FileName プロパティを操作できます。

たとえば，以下の短いコードをアクションに追加して TMainMenu の下位項目の Action プロパティとリンクさせることができます。あるいは，下位項目の OnClick イベントに追加することもできます。

```
if(OpenDialog1->Execute()){
   filename = OpenDialog1->FileName;
};
```

このコードはダイアログボックスを表示し，ユーザーが［OK］ボタンを押すと，filename という名前で宣言されている AnsiString 変数にファイル名がコピーされます。

ツールバーとメニューのアクションを構成する

C++Builder には，メニューとツールの作成，カスタマイズ，および管理を簡易化する機能がいくつかあります。これらの機能を使うと，アクションのリストを整理することができます。アクションとは，ツールバーボタンを押す，メニューコマンドを選択する，アイコンをクリックするなど，アプリケーションのユーザーが開始する動作をいいます。

複数の UI（ユーザーインターフェース）要素で同じアクション群を使うことがよくあります。たとえば，切り取り，コピー，貼り付けといったコマンドは，［編集］メニューとツールバーの両方にあ

るのが一般的です。同じアクションなら 1 回追加するだけで，アプリケーション内の複数の UI 要素でそのアクションを使用できます。

Windows プラットフォームでは，設計時か実行時かを問わず，各種ツールを使って簡単にアクションを定義，グループ化し，さまざまなレイアウトを作成し，メニューをカスタマイズすることができます。この種のツールを総称して「アクションバンドツール」と呼び，アクションバンドツールを使って作成するメニューとツールを「アクションバンド」と呼びます。ActionBand ユーザーインターフェースを作成する基本的な手順は次のとおりです。

- アクションリストを作成し，アプリケーションで使用するアクション群を設定する（アクションマネージャを使う）
- アプリケーションに UI 要素を追加する（TActionMainMenuBar，TActionToolBar などの ActionBand コンポーネントを使う）
- アクションマネージャから UI 要素へアクションをドラッグアンドドロップする

下の表は，メニューとツールバーの設定に関連する用語集です。

表 8.1　アクション設定に関連する用語

用語	定義
アクション	ユーザーが何かに対して行う応答（例：メニュー項目をクリックするなど）。よく使われる標準のアクションが多数用意されているので，アプリケーションの中でそのまま使えます。たとえば，ファイルのオープン，名前を付けて保存，ファイル名を指定して実行，終了といったファイル操作をはじめ，編集，書式設定，検索，ヘルプ，ダイアログ，ウィンドウアクションなどのさまざまなアクションがあります。自分でカスタムアクションをプログラミングして，アクションリストとアクションマネージャからアクセスすることもできます。
アクションバンド	カスタマイズ可能なメニューやツールバーと関連付けられているアクション群を収容するコンテナ。たとえば，メインメニュー用の ActionBand コンポーネント（TActionMainMenuBar），ツールバー用の ActionBand コンポーネント（TActionToolBar）などがアクションバンドです。
アクションのカテゴリ	アクションをグループ分けする際の基準になるもの。1 つのグループ全体をメニュー / ツールバーにドロップできます。たとえば Find（検索）などが標準のカテゴリです。Find カテゴリの中に Find，FindFirst，FindNext，Replace といったアクションがすべて含まれます。
アクションクラス	アプリケーションで使われるアクションを実行するクラス。標準のアクションはすべて，TEditCopy，TEditCut，TEditUndo などのアクションクラスに定義されています。これらのアクションクラスを使うには，［カスタマイズ］ダイアログでアクションクラスを選び，アクションバンドにドラッグアンドドロップします。
アクションのクライアント	たいていはメニュー項目かボタンのことを指す（アクション開始通知を受け取るもの）。クライアントがユーザーコマンド（マウスクリックなど）を受け取ると，関連のアクションを開始します。
アクションリスト	ユーザーの動作に応答してアプリケーションがとるアクションのリストを管理する働きをします。
アクションマネージャ	ActionBand コンポーネントで再使用できる論理アクション群をグループ化し，整理する働きをします（TActionManager を参照）。
メニュー	アプリケーションのユーザーがクリックで実行できるコマンドを一覧にしたもの。ActionBand メニュークラスの TActionMainMenuBar や，クロスプラットフォームコンポーネントの TMainMenu，TPopupMenu などを使ってメニューを作成できます。

表 8.1　アクション設定に関連する用語（つづき）

用語	定義
ターゲット	アクションが作用する相手先の項目。普通は，メモコントロール，データコントロールといったコントロールが「ターゲット」です。ターゲットを必要としないアクションもあります。たとえば標準のヘルプアクションの場合，ターゲットを無視し，単にヘルプシステムを起動するだけです。
ツールバー	目に見えるボタンアイコンを列状に表示したもの。ボタンアイコンをクリックすると，プログラムは何らかのアクション（現在の文書を印刷するなど）を実行します。ツールバーを作成するには，ActionBand ツールバーコンポーネントの TActionToolBar や，クロスプラットフォームコンポーネント TToolBar を使います。

クロスプラットフォーム開発を行っている場合は，8-23 ページの「アクションリストの使い方」を参照してください。

アクションとは

アプリケーションの開発において，アクション群を作成し，各種の UI（ユーザーインターフェース）要素でそれを利用することができます。アクションをいくつかのカテゴリに分類しておけば，アクションをひとまとめにして（切り取り，コピー，貼り付けなど）メニューにドロップできます。メニュー項目にドロップすることもできます（［ツール｜カスタマイズ］など）。

アクション 1 つに対し，1 つ以上の UI 要素（メニューコマンド，ツールバーボタンなど）が対応します。アクションは 2 つの働きをします。(1) ユーザーインターフェース要素に共通するプロパティを表す（コントロールにチェックが付いて使用可になっているかなど）。(2) コントロールが起動したとき（アプリケーションのユーザーがボタンをクリックしたり，メニュー項目を選択したときなど）に応答する。アプリケーションのメニュー，ボタン，ツールバー，コンテキストメニューといった UI 要素別にアクション群を作成できます。

アクションは，以下のような他のコンポーネントと関連しています。

- **クライアント**：1 つ以上のクライアントがアクションを使用します。クライアントとは，たいていはメニュー項目かボタンです（TToolButton, TSpeedButton, TMenuItem, TButton, TCheckBox, TRadioButton など）。アクションは ActionBand コンポーネント（TActionMainMenuBar, TActionToolBar など）にも置かれます。クライアントがユーザーコマンド（マウスクリックなど）を受け取ると，関連のアクションを開始します。通常は，クライアントの OnClick イベントとアクションの OnExecute イベントが関連付けられます
- **ターゲット**：アクションはターゲットに対して作用します。ターゲットとは，普通はコントロールのことです（メモコントロール，データコントロールなど）。コンポーネント記述者は，設計，使用するコントロールのニーズに合わせた独自のアクションを作成でき，それらのユニットをパッケージにしてモジュラーアプリケーションを作成できます。ターゲットを使用しないアクションもあります。たとえば標準のヘルプアクションの場合，ターゲットを無視し，単にヘルプシステムを起動するだけです。

  コンポーネントがターゲットになることもあります。たとえばデータコントロールは，ターゲットを関連のデータセットに変えます。

クライアントからアクションに影響を与えます。つまり，クライアントがアクションを起動すると，アクションが応答します。アクションからクライアントにも影響を与えます。つまり，アクションのプロパティが動的にクライアントのプロパティを更新します。たとえば，実行時に何らかのアクションを使用不可にする（Enabled プロパティを **false** に設定する）と，そのアクションのクライアントもすべて使用不可になり，淡色表示になります。

アクションを追加，削除，再配置するには，アクションマネージャかアクションリストエディタを使います。アクションリストエディタを表示するには，アクションリストオブジェクト TActionList をダブルクリックします。アクションリストエディタで追加などを行ったアクションは，後でクライアントコントロールに接続されます。

アクションバンドの設定

アクション自体は，いわゆるレイアウト情報（外観，位置）を保持しません。このため C++Builder は，レイアウトデータを保存できるアクションバンドを用意しています。アクションバンドが提供するメカニズムにより，レイアウト情報とコントロール群を指定することができます。アクションを UI 要素（ツールバー，メニューなど）として描画することが可能です。

アクション群を構成するには，アクションマネージャ（TActionManager）を使います。あらかじめ用意された標準のアクションはもちろん使用できますが，自分で新しいアクションを作成することもできます。

以下のようにしてアクションバンドを作成します。

- TActionMainMenuBar を使ってメインメニューを作成する
- TActionToolBar を使ってツールバーを作成する

アクションバンドは，アクション群を収容し描画するコンテナの働きをします。設計時は，アクションマネージャエディタからアクションバンドにドラッグアンドドロップできます。実行時は，アクションマネージャエディタに似たダイアログボックスを使ってアプリケーションユーザーがメニューとツールバーをカスタマイズできます。

メニューとツールバーの作成

メモ この節では，Windows アプリケーションのメニューとツールバーを作成するための推奨方法を説明します。クロスプラットフォームアプリケーションのメニューとツールバーを作成する場合は，TToolBar とメニューコンポーネント（TMainMenu など）を使って作成し，アクションリスト（TActionList）を使ってアクションを構成します。8-24 ページの「アクションリストの設定」を参照してください。

アクションマネージャを使うと，アプリケーションに含まれているアクションをもとに，ツールバーとメインメニューを自動的に生成することができます。アクションマネージャの働きは，標準アクションとカスタムアクションを管理することです。プログラマは，これらのアクションをもとに UI 要素を作成し，アクションバンドを使ってアクション項目をメニュー項目またはツールバーボタンとして描画します。

メニュー，ツールバー，その他のアクションバンドを作成する基本的な手順は次のとおりです。

- アクションマネージャをフォームにドロップする
- アクションをアクションマネージャに追加する。アクションマネージャにより，アクション群を適切なアクションリストにまとめる
- 目的の UI に対応するアクションバンド（メニューかツールバー）を作成する
- アクションを UI にドラッグアンドドロップする

以下の手順は，上の手順をより詳しく説明したものです。

アクションバンドを使ってメニューとツールバーを作成するには，次のようにします。

1. コンポーネントパレットの［Additional］ページから，アクションマネージャコンポーネント（TActionManager）をフォームにドロップします。このフォームでメニューかツールバーを作成することになります。
2. メニューかツールバーにイメージを表示したい場合は，コンポーネントパレットの［Win32］ページから ImageList コンポーネントをフォームにドロップします（あらかじめ用意されたイメージを使うか，そうでなければイメージを ImageList に追加する必要があります）。
3. コンポーネントパレットの［Additional］ページから，以下のアクションバンドをフォームにドロップします（1 つまたは複数個をドロップする）。
   - TActionMainMenuBar（メインメニューを設計する場合）
   - TActionToolBar（ツールバーを設計する場合）
4. ImageList と アクションマネージャを結び付けます（アクションマネージャにフォーカスがある状態でオブジェクトインスペクタを表示し，Images プロパティから ImageList の名前を選択する）。
5. 次のようにしてアクションマネージャエディタのアクションペインにアクションを追加します。
   - アクションマネージャをダブルクリックして，アクションマネージャエディタを表示します。
   - ［新規アクション］ボタン（［アクション］タブの右上の左端にあるボタン）の隣にあるドロップダウン矢印をクリックして（図 8.2 参照），［アクションの新規追加］か［標準アクションの新規追加］を選択します。ツリービューが表示されます。アクションマネージャのアクションペインにアクションまたはカテゴリを追加します。アクションがアクションリストに追加されます。

図 8.2　アクションマネージャエディタ

6. アクションマネージャエディタでアクションを個別またはカテゴリごとに選択し，現在設計しているメニューまたはツールバーへとドラッグアンドドロップします。

ユーザー定義アクションを追加するには，［新規アクション］ボタンを押して新しい TAction を作成し，イベントハンドラを作成します。イベントハンドラでは，アクションが起動したときの応答方法を定義します。詳細については，8-25 ページの「アクションの起動時に何が発生するか」を参照してください。ユーザー定義アクションが作成できたら，標準アクションと同じようにメニューかツールバーにドラッグアンドドロップできます。

メニュー，ボタン，ツールバーに色，模様，画像を追加する

メニュー項目またはボタンに使う色，模様，またはビットマップを指定するには，Background プロパティと BackgroundLayout プロパティを使います。また，この 2 つのプロパティを使って，メニューの左側か右側にバナーを表示することもできます。

アクションクライアントオブジェクトの下位項目に背景とレイアウトを割り当てます。メニュー項目の背景を設定するには，フォームデザイナを表示し，目的の項目が入ったメニュームをクリックします。たとえば［ファイル］を選択すると，［ファイル］メニューに表示される項目の背景を変更することになります。色，模様，ビットマップを割り当てるには，オブジェクトインスペクタで Background プロパティを設定します。

BackgroundLayout プロパティは，背景をエレメント上にどう配置するかを設定します。キャプションの影に配置する（ノーマル），項目領域に合わせて拡大する，項目領域全体に小さい四角を並べる，の 3 つの選択肢があります。

ノーマル（blNormal），拡大（blStretch），並べて表示（blTile）のどの背景も，塗りつぶしのない透明の背景として項目が描画されます。バナーを作成する場合，項目の左（blLeftBanner）か右（blRightBanner）にフル画像を配置できます。バナーは領域に合わせた拡大 / 縮小は行わないので，正しいサイズにしておく必要があります。

アクションバンドの背景（つまり，メインメニューまたはツールバー全体の背景）を変えるには，目的のアクションバンドを選択してから，アクションバンドコレクションエディタを使って TActionClientBar を選択します。Background プロパティと BackgroundLayout プロパティを設定すると，メニュー全体またはツールバー全体の色，模様，ビットマップを指定できます。

メニューとツールバーにアイコンを追加する

メニューの隣にアイコンを表示したり，ツールバーのキャプションのかわりにアイコンを表示したりできます。ImageList コンポーネントを使ってビットマップやアイコンを構成します。

1. コンポーネントパレットの［Win32］ページから，ImageList コンポーネントをフォームにドロップします。
2. 次のようにしてイメージリストにイメージを追加します。まず［ImageList］アイコンをダブルクリックします。［追加］をクリックし，使用したいイメージを選択して［OK］をクリックします。Program Files \ Common Files \ Borland Shared \ Images にサンプルイメージがあります（ボタンイメージはアクティブと非アクティブの 2 通りの表示があります）。
3. コンポーネントパレットの［Additional］ページから，以下のアクションバンドをフォームにドロップします（1 つまたは複数個をドロップする）。

- TActionMainMenuBar（メインメニューを設計する場合）
- TActionToolBar（ツールバーを設計する場合）

4. 次のようにしてイメージリストとアクションマネージャを結び付けます。まず，アクションマネージャにフォーカスを当てます。次に，オブジェクトインスペクタで Images プロパティからイメージリストの名前（ImageList1 など）を選択します。
5. アクションマネージャエディタを表示し，アクションをアクションマネージャに追加します。イメージとアイコンを関連付けるには，ImageIndex プロパティをイメージリスト内の対応する番号に設定します。
6. アクションマネージャエディタでアクションを個別またはカテゴリごとに選択し，メニューまたはツールバーにドラッグアンドドロップします。
7. ツールバーにキャプションを表示せず，アイコンだけを表示するには，次のようにします。Toolbar アクションバンドを選択し，その Items プロパティをダブルクリックします。コレクションエディタで項目を 1 つまたは複数個選択し，Caption プロパティを設定します。
8. 以上でメニューまたはツールバーにイメージが表示されます。

ユーザーがカスタマイズできるメニューとツールバーを作成する

アクションバンドとアクションマネージャを使って，カスタマイズ可能なメニューとツールバーを作成することができます。実行時は，アクションマネージャエディタに似たカスタマイズダイアログを使ってアプリケーションユーザーがメニューとツールバー（アクションバンド）をカスタマイズできます。

アプリケーションユーザーがアクションバンドをカスタマイズできるようにするには，以下の手順を行います。

1. アクションマネージャコンポーネントをフォームにドロップします。
2. アクションバンドコンポーネント（TActionMainMenuBar，TActionToolBar）をフォームにドロップします。
3. アクションマネージャをダブルクリックして，アクションマネージャエディタを表示します。
   - アプリケーションで使用するアクションを追加します。標準アクションリストの一番下にあるカスタマイズアクションも追加します。
   - ［Additional］ページにある TCustomizeDlg コンポーネントをフォームにドロップし，その ActionManager プロパティを使ってアクションマネージャと結び付けます。ユーザーが行ったカスタマイズ内容の出力先となるファイル名を指定します。
   - アクションをアクションバンドコンポーネントにドラッグアンドドロップします（カスタマイズアクションを必ずツールバーかメニューに追加してください）。
4. アプリケーションの作成を完了させます。

アプリケーションをコンパイルして実行すると，ユーザーが［カスタマイズ］コマンドを選択できるようになります。このコマンドを選択すると，アクションマネージャエディタに似たカスタマイズ用ダイアログボックスが表示されます。アプリケーションのユーザーは，プログラマがアクションマネージャで追加したのと同じアクションを使ってメニュー項目をドラッグアンドドロップしたり，ツールバーを作成したりできます。

アクションバンドの未使用の項目 / カテゴリを非表示にする

アクションバンドを使うメリットの 1 つは，使われていない項目とカテゴリをユーザーに対して非表示にできることです。アクションバンドは，時間がたつにつれてアプリケーションユーザーに応じてカスタマイズされていきます。その結果，ユーザーが使用する項目のみ表示し，残りの項目は非表示になります。非表示の項目がある場合でも，ユーザーがドロップダウンボタンを押すだけですぐに表示できます。また，ユーザーがカスタマイズダイアログで使用解析をリセットすれば，すべてのアクションバンド項目を元通りに表示できます。項目の非表示がアクションバンドのデフォルトの動作ですが，この動作を変更して，個々の項目の非表示を無効にする，特定の集合（［ファイル］メニューなど）にある全項目の非表示を無効にする，あるいは，特定のアクションバンドにある全項目の非表示を無効にすることができます。

アクションマネージャは，ユーザーがアクションを呼び出した回数を追跡し，TActionClientItem の UsageCount フィールドに保存します。アクションマネージャは，アプリケーションが実行された回数（「セッション回数」と呼びます）も記録します。さらに，前回のアクションが使われたときのセッション回数も記録しています。項目が使われずに非表示になるまでの最大セッション回数を UsageCount の値を使って調べ，次に，現在のセッション回数と前回の項目使用時のセッション回数との差と比較します。その差が PrioritySchedule に定義されている数より大きい場合には，その項目は非表示になります。PrioritySchedule のデフォルト値を下の表に示します。

表 8.2　アクションマネージャの PrioritySchedule プロパティのデフォルト値

アクションバンド項目が使われたときのセッション回数	前回使用時のあとにアクションバンド項目が表示を続けているセッション回数
0, 1	3
2	6
3	9
4, 5	12
6-8	17
9-13	23
14-24	29
25 以上	31

設計時に項目の非表示を無効にすることができます。個々の項目（および，その項目が入っているすべての集合）の非表示を無効にするには，その項目の TActionClientItem オブジェクトを探し，UsageCount を -1 に設定します。項目の集合全体（［ファイル］メニューなど），またはメニューバー全体の非表示を無効にするには，関連の TActionClients オブジェクトを探し，その HideUnused プロパティを **false** に設定します。

アクションリストの使い方

メモ　この節の内容は，クロスプラットフォーム開発で設計するメニューとツールバーに適用できます。Windows アプリケーションを開発する場合には，この節で説明する方法も使用できますが，アクションバンドの方が簡単に操作でき，オプションも豊富です。アクションリストの処理は，アクショ

ンマネージャにより自動的に行われます。アクションバンドとアクションマネージャの使い方については，8-16 ページの「ツールバーとメニューのアクションを構成する」を参照してください。

アクションリストは，ユーザーの行為に応答してアプリケーションがとるアクションのリストを管理する働きをします。アクションオブジェクトを使うと，ユーザーインターフェースを介してアプリケーションが実行する機能を一元管理することができます。これにより，共通のコードを共有してアクションを実行でき（ツールバーボタンとメニュー項目が同じ動作をする場合など），さらに，アプリケーションの状態に応じてアクションを使用可 / 使用不可にする操作を一元化できます。

アクションリストの設定

アクションリストの設定は，以下の基本ステップを覚えてしまえば簡単にできます。

- アクションリストを作成する
- アクションリストにアクションを追加する
- アクションのプロパティを設定する
- アクションとクライアントを結び付ける

詳しい設定手順は次のとおりです。

1. フォームまたはデータモジュールに TActionList オブジェクトをドロップします（ActionList はコンポーネントパレットの［Standard］ページにあります）。
2. TActionList オブジェクトをダブルクリックして，アクションリストエディタを表示します。次に，以下のいずれかの操作を行います。
   a エディタに表示されている定義済みアクションのどれかを使用します（右クリックして［標準アクションの新規追加］を選択します）。
   b ［標準アクションクラス］ダイアログボックスには，定義済みアクションがカテゴリ別に表示されています（データセット，編集，ヘルプ，ウィンドウなど）。目的の標準アクションをすべて選択してアクションリストに追加し，［OK］をクリックします。

      または

   c 独自のアクションを新規作成します（右クリックして［アクションの新規追加］を選択します）。
3. オブジェクトインスペクタで，各アクションのプロパティを設定します（設定したプロパティは，アクションの各クライアントに影響します）。

   Name プロパティはアクションを識別し，その他のプロパティとイベント（Caption，Checked，Enabled，HelpContext，Hint，ImageIndex，ShortCut，Visible，Execute）は，クライアントコントロールのプロパティとイベントに対応します。これらは一般にクライアントのプロパティと同じ名前になっていますが，必ずしも同じである必要はありません。たとえば，アクションの Enabled プロパティは，TToolButton の Enabled プロパティに対応しています。ただし，アクションの Checked プロパティは，TToolButton の Down プロパティに対応しています。
4. 定義済みのアクションには，自動的に発生する標準のレスポンスが含まれています。独自のアクションを新規作成する場合には，アクション起動時の応答方法を定義するイベントハンドラを記述する必要があります。詳細については，8-25 ページの「アクションの起動時に何が発生するか」を参照してください。

5. アクションリスト内のアクションと，そのアクションを必要とするクライアントとを以下の手順で結び付けます。
   - フォームまたはデータモジュール上のコントロール（ボタン，メニュー項目など）をクリックします。オブジェクトインスペクタで確認すると，Action プロパティにアクションのリストが表示されています。
   - 目的のアクションを選択します。

TEditDelete，TDataSetPost などの標準アクションであれば，プログラマの予想どおりのアクションが実行されます。標準アクションの動作に関する詳細は，オンラインヘルプを参照してください。独自のアクションを記述する場合は，アクションの起動時に何が発生するかをよく理解しておく必要があります。

アクションの起動時に何が発生するか

あるイベントが発生すると，主に汎用アクションを目的とした一連のイベントが発生します。ここでイベントがアクションを処理しなければ，別の一連のイベントが発生します。

イベントによる応答

クライアントコンポーネントまたはクライアントコントロールに対してクリック動作などの操作をすると，ユーザーが応答できる一連のイベントが発生します。たとえば次のコードは，アクションが実行されたときにツールバーの表示を切り替えるアクションの OnExecute イベントハンドラを示しています。

```
void __fastcall TForm1::Action1Execute(TObject *Sender)
{
   // Toolbar1 の表示 / 非表示を切り替える
   ToolBar1->Visible = !ToolBar1->Visible;
}
```

イベントハンドラを作成する場合は，3 レベル（アクション，アクションリスト，アプリケーション）のどれかで応答するイベントハンドラを記述します。これが問題になるのは，新しい汎用アクションを使うときだけです。定義済みの標準アクションを使うときは問題ありません。標準アクションには，イベント発生時に実行する動作が組み込まれています。このため，標準アクションを使う場合は上記の問題を考慮する必要がありません。

イベントハンドラは，以下の順でイベントに応答します。

- アクションリスト
- アプリケーション
- アクション

ユーザーがクライアントコントロールをクリックすると，C++Builder はこのアクションの Execute メソッドを呼び出します。Execute メソッドは，最初にアクションリストに従い，次に Application オブジェクトに従います。アクションリストも Application もアクションを処理しなければ，アクション自身に従います。これについて，もう少し詳しく解説します。C++Builder は，ユーザーアクションへの応答方法を探すとき，以下のディスパッチシーケンスに従います。

1. アクションリスト用に作成した OnExecute イベントハンドラがアクションを処理すれば，アプリケーションは続行します。

   アクションリストのイベントハンドラは，デフォルトで **false** を返すパラメータ Handled を持っています。このハンドラが割り当てられていて，このイベントを処理する場合は，**true** を返し，処理シーケンスはここで終了します。次に例を示します。

   ```
   void __fastcall TForm1::ActionList1ExecuteAction(TBasicAction *Action, bool &Handled)
   {
      Handled = true;
   }
   ```

   アクションリストのイベントハンドラで Handled を **true** に設定しなければ，処理が続行します。

2. プログラマがアクションリスト用の OnExecute イベントハンドラを作成しなかった場合，あるいは，OnExecute イベントハンドラがアクションを処理しない場合，アプリケーションの OnActionExecute イベントハンドラが起動します。OnActionExecute イベントハンドラがアクションを処理すれば，アプリケーションは続行します。

   イベントを処理しないアクションリストがアプリケーションに 1 つでもあれば，グローバルオブジェクト Application が OnActionExecute イベントを受け取ります。アクションリストの OnExecute イベントハンドラと同様に，OnActionExecute ハンドラも，デフォルトで **false** を返すパラメータ Handled を持っています。イベントハンドラが割り当てられていて，このイベントを処理する場合は，**true** を返し，処理シーケンスはここで終了します。次に例を示します。

   ```
   void __fastcall TForm1::ApplicationExecuteAction(TBasicAction *Action, bool &Handled)
   {
     // Application 内のすべてのアクションを実行しない
     Handled = true;
   }
   ```

3. アプリケーションの OnExecute イベントハンドラがアクションを処理しない場合，そのアクションの OnExecute イベントハンドラが起動します。

組み込みのアクションを使用するか，独自のアクションクラスを作成できます。独自のアクションクラスを作成する場合は，特定のターゲットクラス（編集コントロールなど）に対してどう作用するかを定義してください。どのレベルにもイベントハンドラが見つからないと，アプリケーションは次に，アクションの実行対象のターゲットを見つけようとします。アプリケーションが何かターゲットを見つけると，そのターゲットにどう対処するかわかっているアクションがあれば，そのアクションを呼び出します。定義済みアクションクラスに応答可能なターゲットをアプリケーションがどう見つけるかについては，次の節を参照してください。

アクションがどのようにターゲットを見つけるか

8-25 ページの「アクションの起動時に何が発生するか」では，ユーザーがアクションを呼び出したときに発生する実行サイクルを説明しました。3 レベル（アクションリスト，アプリケーション，アクション）のどれかでアクションに応答するイベントハンドラが 1 つも割り当てられていない場合，アプリケーションは，アクションが自分自身を適用できるターゲットオブジェクトを見つけようとします。

アプリケーションは，以下のシーケンスでターゲットを探します。

1. アクティブコントロール：最初に，潜在的ターゲットとしてアクティブコントロールを検索します。

2. アクティブフォーム：アクティブコントロールが見つからないか，アクティブコントロールがターゲットの働きをしない場合は，Screen の ActiveForm を検索します。
3. フォーム上のコントロール：アクティブフォームが適切なターゲットでなければ，アクティブフォーム上の他のコントロールにターゲットがないか探します。

ターゲットが見つからなければ，イベントが発生しても何も起こりません。

コントロールの種類によっては，検索範囲を広げて関連コンポーネントもターゲットとすることができます。たとえばデータベース対応コントロールは，関連のデータセットコンポーネントをターゲットにします。また，［ファイルを開く］ダイアログなどの一部の定義済みアクションは，ターゲットを使用しません。

アクションを更新する

アプリケーションがアイドル状態の場合，コントロールまたは表示中のメニュー項目にリンクされているすべてのアクションに対して OnUpdate イベントが発生します。これは，使用可能 / 不可にする，チェックマークを付ける / はずすなどのコードを 1 個所にまとめ，アプリケーションにその集中化されたコードを実行させる方法を提供します。たとえば，次のコードは，ツールバーが表示されている場合，「チェックマークが付いている」アクションの OnUpdate イベントハンドラを示します。

```
void __fastcall TForm1::Action1Update(TObject *Sender)
{
   // ToolBar1 が現在表示されているかどうかを示す
   ((TAction *)Sender)->Checked = ToolBar1->Visible;
}
```

警告 処理時間のかかるコードを OnUpdate イベントハンドラに追加しないでください。このイベントハンドラは，アプリケーションがアイドルであると必ず実行されます。イベントハンドラが時間を取りすぎると，アプリケーション全体の処理効率に影響します。

定義済みのアクションクラス

定義済みのアクションをアプリケーションに追加するには，アクションマネージャ上をダブルクリックして［標準アクションの新規追加］を選択します。［新規標準アクションクラス］ダイアログボックスが開き，定義済みアクションクラスおよび関連の標準アクションが一覧表示されます。ここに表示されるのは C++Builder に組み込まれているアクションであり，アクションを自動的に実行するオブジェクトです。定義済みアクションは以下のクラスに分類されています。

表 8.3　アクションクラス

クラス	説明
Edit	標準の編集アクション：編集コントロールをターゲットとして使うアクション。TEditAction は基本クラスで，その下位クラスがそれぞれ ExecuteTarget メソッドをオーバーライドし，クリップボードを使用してコピー，切り取り，貼り付けタスクを実装する
Format	標準の書式設定アクション：リッチテキストをターゲットとして，太字，イタリック体，下線，取消線などのテキストの書式設定に適用される。TRichEditAction は基本クラスで，その下位クラスがそれぞれ ExecuteTarget と UpdateTarget メソッドをオーバーライドしてターゲットの書式設定を実装する

表 8.3　アクションクラス（つづき）

クラス	説明
Help	標準のヘルプアクション：どのターゲットにも使用できる。THelpAction は基本クラスで，その下位クラスがそれぞれ ExecuteTarget メソッドをオーバーライドして，コマンドをヘルプシステムに渡す
Window	標準のウィンドウアクション：MDI アプリケーションにおいて，フォームをターゲットとして使う。TWindowAction は基本クラスで，その下位クラスがそれぞれ ExecuteTarget メソッドをオーバーライドして，MDI 子フォームの整列，重ねて表示，閉じる，並べて表示，および最小化を実装する
File	ファイルアクション：ファイルオープン，ファイル実行，ファイル終了などのファイル操作に使用する
Search	検索アクション：いくつかの検索オプションと一緒に使う。TSearchAction は，ユーザーが検索文字列を入力して編集コントロールを検索するときの，モードなしダイアログを表示するアクションの共通動作を実装する
Tab	タブコントールアクション：ウィザードの［戻る］ボタンや［次へ］ボタンなど，タブコントロール上のタブ間の移動に使用する
List	リストコントロールアクション：リストビュー内の項目を管理するのに使用する
Dialog	ダイアログアクション：ダイアログコンポーネントで使用する。TDialogAction は，実行時にダイアログを表示するアクションの共通動作を実装する。各下位クラスが個々のダイアログを表す
Internet	インターネットアクション：ブラウズ，ダウンロード，メール送信などのインターネット機能を行う
DataSet	データセットアクション：データセットコンポーネントをターゲットとして使う。TDataSetAction は基本クラスで，その下位クラスがそれぞれ ExecuteTarget と UpdateTarget メソッドをオーバーライドしてターゲットのナビゲーションと編集を実装する。 TDataSetAction は，アクションがデータセット上で実行されることを保証する DataSource プロパティを導入する。DataSource がヌルの場合には，現在フォーカスがあるデータベース対応コントロールが使われる
Tools	ツール：追加のツール類（アクションバンドのカスタマイズダイアログを自動的に表示する TCustomizeActionBars など）

各アクションオブジェクトの説明を参照するには，オンラインヘルプでアクションオブジェクト名を検索してください。アクションオブジェクトの機能に関する詳細は，ヘルプを参照してください。

アクションコンポーネントを記述する

定義済みアクションクラスを独自に作成することもできます。独自のアクションクラスを作成すると，オブジェクトの一定のターゲットクラスに対して実行する能力を組み込めます。すると，定義済みアクションクラスを使うときと同じように，独自のカスタムアクションを使用できます。すなわち，アクションが自らを認識して，アクション自身をターゲットクラスに適用させることができれば，プログラマは単にクライアントコントロールにアクションを割り当てるだけで，ターゲットに対して作用させることができます。イベントハンドラを記述する必要はありません。

コンポーネント開発者は，StdActns と DBActns ユニット内のクラスを独自のアクションクラスを派生させるサンプルとして使用して，特定のコントロールやコンポーネントに対する動作を実装することもできます。これらの特化されたアクション（TEditAction，TWindowAction）の基本クラスは，一般的に HandlesTarget，UpdateTarget およびほかのメソッドをオーバーライドして，ターゲットをオブ

ジェクトの特定のクラスに対するアクションに制限します。下位クラスは，一般的に ExecuteTarget をオーバーライドして，特化されたタスクを実行します。これらのメソッドに関する説明を次の表に示します。

メソッド	説明
HandlesTarget	アクションとリンクしたオブジェクト（ツールボタン，メニュー項目など）をユーザーが起動すると，自動的に呼び出される。Target パラメータが「ターゲット」として指定するオブジェクトを使ってその時点で実行するのが適切かどうか，アクションオブジェクトに表示させる。詳細は，8-26 ページの「アクションがどのようにターゲットを見つけるか」を参照
UpdateTarget	アプリケーションがアイドルのときに自動的に呼び出されるため，最新の条件に従ってアクションが自分自身を更新できる。OnUpdateAction のかわりに使用する。詳細は，8-27 ページの「アクションを更新する」を参照
ExecuteTarget	OnExecute のかわりにユーザーアクションに応答してアクションが起動したとき（アクションとリンクしたメニュー項目やツールボタンをユーザーが選択したときなど），自動的に呼び出される。詳細は，8-25 ページの「アクションの起動時に何が発生するか」を参照

アクションを登録する

独自のアクションを作成した場合，そのアクションを登録してアクションリストエディタに表示することができます。アクションの登録 / 登録解除を行うには，Actnlist ユニットのグローバルルーチンを使います。

```
extern PACKAGE void __fastcall RegisterActions(const AnsiString CategoryName,
TMetaClass* const * AClasses, const int AClasses_Size, TMetaClass* Resource);
extern PACKAGE void __fastcall UnRegisterActions(TMetaClass* const * AClasses, const int
AClasses_Size);
```

RegisterActions を呼び出すと，アクションリストエディタに登録アクションが表示され，アプリケーションでそのアクションを使用できるようになります。カテゴリ名を指定してアクションを整理することができます。また，Resource パラメータを使うと，アクションのプロパティにデフォルト値を代入できます。

例として，IDE でアクションを MyAction ユニットに登録するコードを示します。

```
namespace MyAction
{
  void __fastcall PACKAGE Register()
  {
     // ここでコンポーネントとエディタを登録
      TMetaClass classes[2] = {__classid(TMyAction1), __classid(TMyAction2)};
      RegisterActions("MySpecialActions", classes, 1, NULL);
  }
}
```

UnRegisterActions を呼び出すと，アクションはその後，アクションリストエディタに表示されなくなります。

メニューの作成と管理

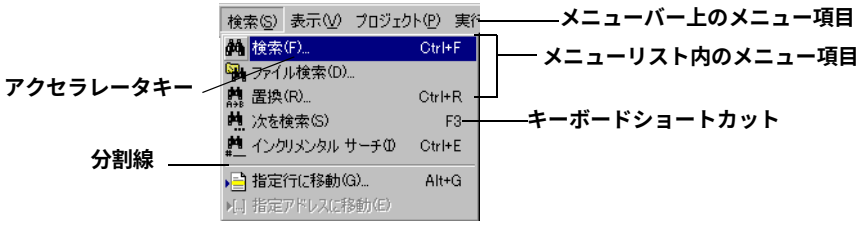
メニューを使用すると，ユーザーは論理的にまとめられたコマンド群を簡単に実行できます。メニューデザイナを使用すると，設計済みのメニューあるいはカスタマイズしたメニューを簡単にフォームに追加できます。フォームにメニューコンポーネントを追加し，メニューデザイナを開き，メニューデザイナウィンドウにメニュー項目を直接入力するだけでメニューを構築できます。設計時にメニュー項目を追加または削除したり，メニュー項目をドラッグアンドドロップして順序を入れ替えたりできます。

結果を確認するためにプログラムを実行する必要はありません。設計時のメニューは実行時とまったく同じようにフォームに表示されます。また，実行時にコードでメニューを変更し，追加情報やオプションをユーザーに提供することもできます。

このセクションでは，メニューデザイナを使用してメニューバーやポップアップ（ローカル）メニューを設計する方法を説明します。ここで説明する設計時と実行時のメニューに関する操作は以下のとおりです。

- メニューデザイナを開く
- メニューの構築
- オブジェクトインスペクタでのメニュー項目の編集
- メニューデザイナのコンテキストメニューの使い方
- メニューテンプレートの使い方
- メニューをテンプレートとして保存する
- イメージをメニュー項目に追加する

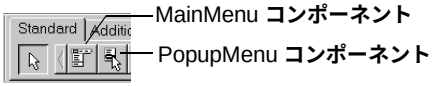
図 8.3　メニュー用語



メニューデザイナを開く

メニューを設計するには，メニューデザイナを使います。メニューデザイナの使用を始めるには，MainMenu コンポーネントまたは PopupMenu コンポーネントをフォームに追加します。どちらのメニューコンポーネントもコンポーネントパレットの［Standard］ページにあります。

図 8.4　MainMenu および PopupMenu コンポーネント



MainMenu コンポーネントでは，フォームのタイトルバーに結び付けるメインメニューを作成します。PopupMenu コンポーネントでは，ユーザーがフォーム内で右クリックをしたときに表示されるメニューを作成します。なお，ポップアップメニューにメニューバーはありません。

メニューデザイナを開くには，フォーム上でメニューコンポーネントを選択し，以下のいずれかを行います。

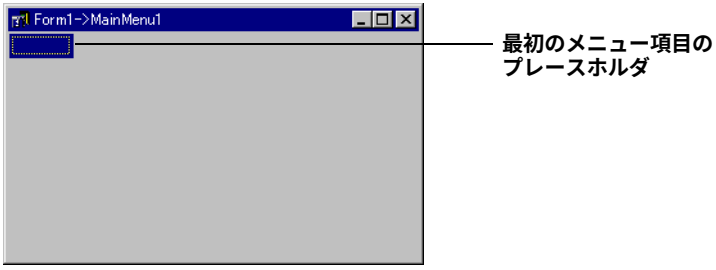
- メニューコンポーネントをダブルクリックする

  または

- オブジェクトインスペクタの［プロパティ］ページで Items プロパティを選択し，値列にある［(メニュー)］をダブルクリックするか，［...］（省略記号）ボタンをクリックする

  メニューデザイナが表示されます。メニューデザイナでは，最初に空白のメニュー項目が強調表示され，入力を始めるとオブジェクトインスペクタの Caption プロパティが選択されます。

図 8.5　メインメニューのためのメニューデザイナ



メニューの構築

アプリケーションに含めるすべてのメニューのために，フォームに対してメニューコンポーネントを追加します。各メニューを作成するには，最初から独自に構築するか，または設計済みのメニューテンプレートを使用します。

このセクションでは，設計時にメニューを作成するための基本を説明します。メニューテンプレートについての詳細は，8-39 ページの「メニューテンプレートの使い方」を参照してください。

メニューの名前

すべてのコンポーネントに共通することですが，フォームにメニューコンポーネントを追加するとデフォルトの名前（たとえば MainMenu1）が付けられます。別の具体的なメニュー名を付けることもできます。

C++Builder ではフォームの型宣言にメニュー名を追加します。このメニュー名はオブジェクトインスペクタのコンポーネントリストに表示されます。

メニュー項目の名前

メニューコンポーネント自体とは異なり，メニュー項目には明示的に名前を付けてフォームに追加します。メニュー項目に名前を付けるには以下の 2 通りの方法があります。

- Name プロパティに値を直接入力する
- まず，Caption プロパティに値を入力し，C++Builder にそのキャプションから Name プロパティを派生させる

  たとえば，メニュー項目に Caption プロパティの値として File を入力した場合，C++Builder はメニュー項目に Name プロパティ File1 を割り当てます。Caption プロパティに値を入力する前に Name プロパティに値を入力しても，Caption プロパティは値が入力されるまで空白になります。

メモ C++ の識別子として無効な文字を Caption プロパティに入力した場合は，C++Builder が Name プロパティを適切に変更します。たとえば，Caption の先頭に数字を使用した場合，C++Builder は数字の前に文字を付けて Name プロパティ値を作成します。

次の表に例を示します。表中のすべてのメニュー項目は同一のメニューバーに表示されているものとします。

表 8.4 キャプションの例と派生名

コンポーネントの Caption プロパティ値	作成される Name プロパティ値	説明
&File	File1	アンド記号（&）を除去する
&File（2 つ目）	File2	重複項目を数字で区別する
1234	N12341	先頭に文字，末尾に数字を追加する
1234（2 つ目）	N12342 同上（重複項目を区別）	派生した名前を明確にするために数を追加する
$@@@#	N1	命名規則に合わない文字をすべて除去し，先頭に文字，末尾に数字を追加する
-（ハイフン）	N2	同様に，命名規則に合わない文字を除去し，先頭に文字，末尾に重複しないように数字を追加する

メニューコンポーネントと同じように，フォームの型宣言にメニュー項目名が追加されます。このメニュー項目名はオブジェクトインスペクタのコンポーネントリストに表示されます。

メニュー項目の追加，挿入，削除

次の手順は，メニュー構造を構築するための基本的な作業を示しています。なお，各ステップでは，メニューデザイナウィンドウを開いていることを前提にしています。

設計時にメニュー項目を追加する手順は次のとおりです。

1. メニュー項目を作成する位置を選択します。

   初めて開いたメニューデザイナでは，メニューバーの最初の位置がすでに選択されています。

2. Caption プロパティの値を入力します。または，はじめにオブジェクトインスペクタの Name プロパティに値を入力します。この場合は，Caption プロパティを再び選択して値を入力する必要があります。
3. 〔Enter〕を押します。

   メニュー項目を配置するための次のプレースホルダが選択されます。

最初に Caption プロパティを入力した場合には，矢印キーを使用して作成したメニュー項目に戻ります。キャプションに入力した値に基づいて Name プロパティが設定されます（8-31 ページの「メニュー項目の名前」を参照してください）。

4. さらに新しく作成する各項目の Name プロパティと Caption プロパティに引き続き値を入力するか，【Esc】を押してメニューバーに戻ります。

   矢印キーでメニューバーからメニューへ移動し，さらにリスト内の項目間を移動します。操作を完了するには【Enter】を押します。メニューバーに戻るには【Esc】を押します。

新しい空白のメニュー項目を挿入する手順は次のとおりです。

1. メニュー項目にカーソルを配置します。
2. 【Ins】を押します。

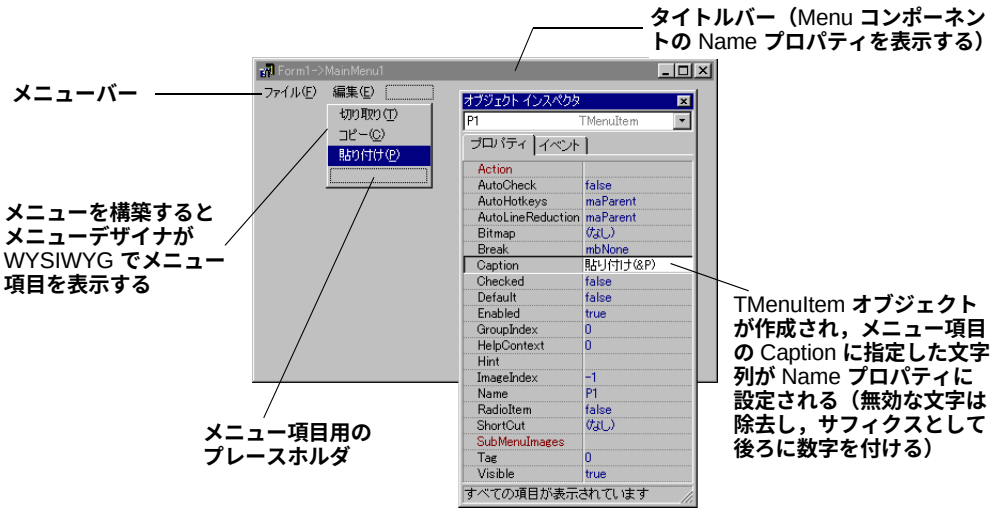
   メニュー項目は，メニューバーでは選択されている項目の左に挿入されます。メニューリストでは選択されている項目の上に挿入されます。

メニュー項目，つまりメニューコマンドを削除する手順は次のとおりです。

1. 削除するメニュー項目にカーソルを配置します。
2. 【Del】を押します。

メモ デフォルトのプレースホルダは削除できません。デフォルトのプレースホルダはメニューリストに最後に入力した項目の下か，またはメニューバーの最後の項目の横に表示されます。このプレースホルダは実行時のメニューには表示されません。

図 8.6　メインメニューにメニュー項目を追加する



分割線の追加

分割線はメニュー項目間に引かれる線です。分割線を使って，メニューリスト内のグループ分けを示したり，リストを単に見やすいように分けたりできます。

メニュー項目を分割線にするには，キャプションにハイフン（-）を入力します。

アクセラレータキーとキーボードショートカットの指定

アクセラレータキーを使用すると，ユーザーはキーボードで【Alt】キーと適切な文字キーを同時に押すことによってメニューコマンドにアクセスできます。適切な文字とは，コードでアンド記号（&）の次にある文字です。アンド記号に続く文字はメニューに下線付きで表示されます。

C++Builder 側でアクセラレータの重複の有無をチェックし，重複があれば実行時に自動的に調整します。これにより実行時にメニューが動的に構築され，アクセラレータの重複なしに全メニュー項目にアクセラレータを持たせることができます。この自動チェックをオフにするには，メニュー項目の AutoHotkeys プロパティを maManual に設定します。

アクセラレータを指定する手順は次のとおりです。

- 適切な文字の前に & 記号を追加する

  たとえば，メニューコマンド Save にアクセラレータキーとして S を付けるには「保存 (&S)」と入力します。

キーボードショートカットを使用すると，ユーザーはメニューを使用せずに，ショートカットキーの組み合わせを入力して直接操作することができます。

キーボードショートカットを指定する手順は次のとおりです。

- オブジェクトインスペクタを使用して，ShortCut プロパティに値を入力するか，またはドロップダウンリストからキーの組み合わせを選択する

  ただしこのリストは，入力できる有効な組み合わせのほんの一部です。

追加したショートカットは，メニュー項目のキャプションの横に表示されます。

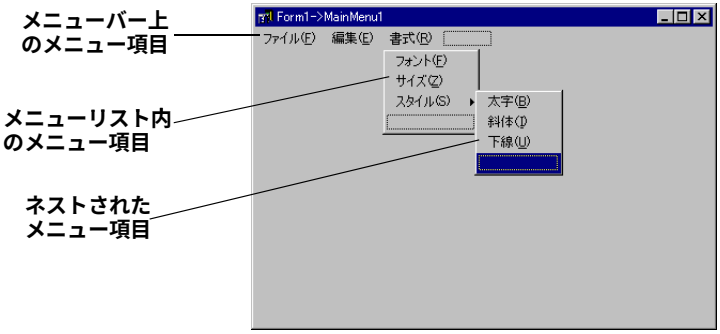
注意 アクセラレータキーと異なり，キーボードショートカットの重複は自動チェックされません。そのため，重複がないか自分で確認する必要があります。

サブメニューの作成

アプリケーションメニューには，さらにドロップダウンリストを開き，関連した複数のコマンドを提供するものがあります。このような階層構造をネストといい，下のレベルのコマンド群をネストしたメニューまたはサブメニューと呼びます。ネストしたメニューを持つメニュー項目では，その右に右向きの三角形が示されます。C++Builder では，いくらでも必要なだけネストしたメニューを構築できます。

メニューをネスト構造にすると，縦に長すぎるメニューを短くすることができます。ただし，インターフェースの使いやすさを考えて，ネストの階層は 2 ～ 3 レベル程度（ポップアップメニューの場合は 1 レベル）に抑えるべきです（ポップアップメニューでは，必要な場合にサブメニューを 1 つだけ使用します）。

図 8.7　ネストされたメニュー構造



サブメニューを作成する手順は次のとおりです。

1. サブメニューを作成するメニュー項目を選択します。
2. 〔Ctrl〕+〔→〕を押して最初のプレースホルダを作成するか，右クリックして［サブメニューの作成］を選択します。
3. このプレースホルダ内にサブメニュー項目の名前を入力するか，既存のメニュー項目をドラッグします。
4. 〔Enter〕または〔↓〕を押して，次のプレースホルダを作成します。
5. サブメニュー内に作成する項目ごとに手順の 3 と 4 を繰り返します。
6. 〔Esc〕を押して前のメニューレベルに戻ります。

既存のメニューのレベルを下げてサブメニューを作成する

メニューバーやメニューテンプレートのメニュー項目を，リストのメニュー項目間に挿入して，サブメニューを作成することもできます。メニューを既存のメニュー構造内に移動すると，このメニューに関連付けられたすべての項目も含めた全体をサブメニューにすることができます。これもサブメニューに属します。メニュー項目を既存のサブメニュー内に移動すると，ネストのレベルがもう 1 つ作成されます。

メニュー項目の移動

設計時にはドラッグアンドドロップだけでメニュー項目を移動できます。たとえば，メニューバー上を移動したり，メニューリストのほかの場所に移動したり，ほかのメニュー内に移動できます。

ただし，直系のネスト階層間での移動はできません。つまり，メニューバーのメニュー項目をそのメニュー自身の中へ移動したり，あるいは，あるメニュー項目をそのメニュー項目自身の下にあるサブメニュー内へ移動することはできません。一方，ほかの系列のメニューに移動するのであれば，メニュー項目の移動先と移動元のレベルに制限はありません。

メニュー項目をドラッグするとカーソルの形が変わり，その位置にメニュー項目を挿入できるかどうかが示されます。メニュー項目を移動すると，そのメニュー項目に属するすべての項目も移動します。

メニューバー上にメニュー項目を移動する手順は次のとおりです。

1. メニューバー上でメニュー項目をドラッグし，ドラッグカーソルの矢印で新しい位置を指します。

2. マウスボタンを放してメニュー項目を新しい位置にドロップします。

メニュー項目をメニューリスト内に移動する手順は次のとおりです。

1. メニューバー上で移動元のメニュー項目をドラッグし，ドラッグカーソルの矢印で移動先のメニュー項目を指します。

   移動先のメニューが開き，移動元のメニュー項目を新しい位置にドラッグできる状態になります。

2. リスト内に移動元のメニュー項目をドラッグし，マウスボタンを放して新しい位置にドロップします。

イメージをメニュー項目に追加する

イメージを使用すると，ツールバーでのイメージと同様に，メニュー項目動作と一致するグリフとイメージによってメニューを操作する助けとなります。メニュー項目にビットマップを 1 つずつ追加することもできますが，同じアプリケーションで使う複数のイメージをイメージリストにまとめ，イメージリストからメニューへと追加することも可能です。同一サイズの複数のビットマップを使う場合は，イメージリストにまとめておくと便利です。

メニューまたはメニュー項目に単一イメージを追加するには，Bitmap プロパティを使います。メニューまたはメニュー項目で使用するビットマップ名を参照するように Bitmap プロパティを設定します。

イメージリストを使ってイメージをメニュー項目に追加するには，以下のようにします。

1. TMainMenu または TPopupMenu オブジェクトをフォーム上にドロップします。
2. TImageList オブジェクトをフォーム上にドロップします。
3. TImageList オブジェクトをダブルクリックしてイメージリストエディタを開きます。
4. ［追加］をクリックしてメニュー内で使用したいビットマップまたはビットマップグループを選択します。［OK］を選択します。
5. TMainMenu または TPopupMenu オブジェクトの Image プロパティを作成した ImageList に設定します。
6. 前に説明した手順でメニュー項目とサブメニューを作成します。
7. イメージを持たせたいメニュー項目をオブジェクトインスペクタ内で選択して，ImageIndex プロパティを ImageList 内のイメージの対応する番号に設定します（ImageIndex のデフォルト値は -1 で，イメージを表示しません）。

メモ メニュー内で適切に表示するには，16 × 16 ピクセルのイメージを使用してください。メニューイメージには，これ以外のサイズを使用できますが，16 × 16 ピクセルよりも大きいまたは小さいイメージを使用すると，位置合わせや一貫性で問題が起きる場合があります。

メニューの表示

メニューは設計時のフォームに表示できます。あらかじめプログラムコードを実行する必要もありません。なお，設計時のフォームにはポップアップメニューコンポーネントは表示されますが，ポップアップメニュー自体は表示されません。設計時にポップアップメニューを表示するには，メニューデザイナを使用します。

メニューを表示する手順は次のとおりです。

1. メニューを表示するフォームが表示されていればメニューを表示するフォームをクリックします。フォームが表示されていなければ，［表示］メニューでフォームを選択します。
2. フォームに複数のメニューがあれば，表示するメニューをフォームの Menu プロパティのドロップダウンリストで選択します。

   プログラムの実行時に表示されるとおりのメニューがフォームに表示されます。

オブジェクトインスペクタでのメニュー項目の編集

これまでは，メニューデザイナを使用した Name および Caption プロパティなど，メニュー項目への複数のプロパティの設定方法を説明しました。

また，フォームでコンポーネントを選択した場合と同じように，ShortCut プロパティなどのメニュー項目のプロパティをオブジェクトインスペクタで直接設定する方法も説明しました。

メニューデザイナでメニュー項目を編集しているとき，オブジェクトインスペクタにはそのメニュー項目のプロパティが表示されています。そのため，オブジェクトインスペクタにフォーカスを切り替えるだけで，そのメニュー項目のプロパティを編集できます。一方，メニューデザイナを使用せずに，オブジェクトインスペクタのオブジェクトセレクタでメニュー項目を選択してプロパティを編集することもできます。

メニューデザイナウィンドウを閉じてメニュー項目の編集を続ける手順は，次のとおりです。

1. オブジェクトインスペクタの［プロパティ］ページをクリックし，メニューデザイナウィンドウからオブジェクトインスペクタへフォーカスを切り替えます。
2. メニューデザイナのシステムメニューで［閉じる］を選択するか，クローズボタンをクリックします。

   フォーカスはオブジェクトインスペクタ内に残ります。したがって，選択したメニュー項目のプロパティを引き続き編集できます。別のメニュー項目を編集するには，オブジェクトセレクタで項目を選択します。

メニューデザイナのコンテキストメニューの使い方

メニューデザイナのコンテキストメニューを使うと，一般的なメニューデザイナコマンドやメニューテンプレートオプションを簡単に利用できます（メニューテンプレートについての詳細は，8-39 ページの「メニューテンプレートの使い方」を参照してください）。

スピードメニューを表示するに，メニューデザイナウィンドウを右クリックするか，メニューデザイナウィンドウ内にフォーカスがあるときに〔Alt〕+〔F10〕を押します。

コンテキストメニューのコマンド

次の表にメニューデザイナのコンテキストメニューコマンドをまとめます。

表 8.5　メニューデザイナのコンテキストメニューコマンド

メニューコマンド	動作
挿入	カーソルの上または左にプレースホルダを挿入する
削除	選択したメニュー項目を削除する。サブ項目がある場合は，それも一括して削除する
サブメニューの作成	ネストレベルのプレースホルダを作成し，選択したメニュー項目の右に右向き三角を表示する
メニューの選択	現在のフォーム内のメニューリストを開く。メニュー名をダブルクリックすると，メニュー用のデザイナウィンドウが開く
テンプレートとして保存	［テンプレートを保存］ダイアログボックスを開く。将来の再利用のためにメニューを保存できる
テンプレートを挿入	［テンプレートを挿入］ダイアログボックスを開く。再利用するテンプレートを選択できる
テンプレートを削除	［テンプレートを削除］ダイアログボックスを開く。既存のテンプレートを削除できる
リソースから追加	［リソースから追加］ダイアログボックスを開く。現在のフォームで開く .rc ファイルまたは .mnu ファイルを選択できる

設計時のメニュー間の切り替え

1 つのフォーム上で複数のメニューを設計している場合，メニューデザイナのコンテキストメニューやオブジェクトインスペクタを使って，メニュー間を簡単に移動，選択できます。

コンテキストメニューでフォームのメニュー間を切り替える手順は次のとおりです。

1. メニューデザイナ内で右クリックして［メニューの選択］を選択します。

   ［メニューの選択］ダイアログボックスが表示されます。

   図 8.8　［メニューの選択］ダイアログボックス

   このダイアログボックスには，メニューデザイナで現在開いているメニューが属するフォームに関連付けられた，すべてのメニューが一覧表示されます。

2. ［メニューの選択］ダイアログボックスのリストで，表示または編集するメニューを選択します。

オブジェクトインスペクタを使ってフォームのメニュー間を切り替える手順は次のとおりです。

1. 選択するメニューを含むフォームにフォーカスを与えます。

2. オブジェクトセレクタで，編集するメニューを選択します。
3. オブジェクトインスペクタの［プロパティ］ページでこのメニューの Items プロパティを選択し，［...］ボタンをクリックするか，［メニュー］をダブルクリックします。

メニューテンプレートの使い方

C++Builder では，頻繁に使用するコマンドを入れた設計済みのメニュー，つまりメニューテンプレートがいくつか提供されています。これらのメニューテンプレートは，そのまま変更せずにアプリケーションで使用できます。また，これを基にしてカスタマイズし，最初から独自に設計したメニューと同じように扱うこともできます。なお，メニューテンプレートにはイベントハンドラコードは含まれていません。

C++Builder に添付されたメニューテンプレートは，デフォルトインストールの状態で \ Bin ディレクトリに格納されます。このファイルには拡張子 .dmt（C++Builder メニューテンプレートを意味します）が付いています。

独自に設計したメニューを，メニューデザイナでテンプレートとして保存することもできます。テンプレートとして保存したメニューは，設計済みのメニューと同じように使用できます。また，必要でないメニューテンプレートはリストから削除することもできます。

アプリケーションにメニューテンプレートを追加する手順は次のとおりです。

1. メニューデザイナで右クリックして［テンプレートを挿入］を選択します。

   なお，テンプレートが存在しない場合，コンテキストメニューの［テンプレートを挿入］コマンドは淡色表示になります。

   ［テンプレートを挿入］ダイアログボックスが開き，使用可能なメニューテンプレートのリストを表示します。

   図 8.9　［テンプレートを挿入］ダイアログボックス

2. 挿入するメニューテンプレートを選択し，【Enter】を押すか［OK］を選択します。

   メニューはフォームのカーソル位置に追加されます。たとえば，カーソルがリストのメニュー項目上にあると，メニューテンプレートはこの項目の上に挿入されます。カーソルがメニューバー上にあると，メニューテンプレートはカーソルの左に挿入されます。

メニューテンプレートを削除する手順は次のとおりです。

1. メニューデザイナを右クリックして［テンプレートを削除］を選択します。

   なお，テンプレートが存在しない場合，コンテキストメニューの［テンプレートを削除］コマンドは淡色表示になります。

   ［テンプレートの削除］ダイアログボックスが開き，使用可能なテンプレートのリストが表示されます。

2. 削除するメニューテンプレートを選択し，［OK］を選択します。

   テンプレートリストとハードディスクから選択したテンプレートが削除されます。

メニューをテンプレートとして保存する

独自に設計したメニューも，テンプレートとして保存すれば後で再利用できます。メニューテンプレートを使うと，アプリケーションに一貫した外観を確保できるだけでなく，これを基にしてカスタマイズを進めることもできます。

保存したメニューテンプレートは，BIN サブディレクトリ中に .dmt ファイルとして格納されます。

メニューをテンプレートとして保存する手順は次のとおりです。

1. 再利用できるメニューを設計します。

   このメニューにはいくつでも項目，コマンド，サブメニューを入れることができます。アクティブなメニューデザイナウィンドウ内のすべてが，1 つの再利用可能なメニューとして保存されます。

2. メニューデザイナで右クリックして［テンプレートとして保存］を選択します。

   ［テンプレートとして保存］ダイアログボックスが表示されます。

   図 8.10 ［テンプレートとして保存］ダイアログボックス

3. ［テンプレートのコメント］編集ボックスに，このメニューの簡単な説明文を入力して［OK］を選択します。

   ［テンプレートとして保存］ダイアログボックスが閉じるとメニュー設計が保存され，メニューデザイナウィンドウに戻ります。

メモ 入力した説明文は［テンプレートとして保存］,［テンプレートを挿入］,［テンプレートを削除］の各ダイアログボックスに表示されます。この説明文は，メニューの Name プロパティや Caption プロパティには影響しません。

テンプレートメニュー項目とイベントハンドラの命名規則

メニューをテンプレートとして保存するとき，このメニューの Name プロパティは保存されません。オーナーとなるフォームのスコープ内で，各メニューはユニークな名前を持つ必要があるからです。そして，メニューデザイナを使って新しいフォームにテンプレートのメニューを挿入すると，このメニューとそのすべての項目には新しい名前が付けられます。

たとえば，あるファイルメニュー（メニューバーに「ファイル」と表示される）をテンプレートとして保存するとします。元のメニュー名は MyFile とします。このメニューをテンプレートとして新しいメニューに挿入すると，このメニューの名前は File1 になります。ただし，挿入先にすでに File1 というメニュー項目がある場合は，挿入するメニューの名前は File2 になります。

テンプレートでは，保存するメニューに関連付けられた OnClick イベントハンドラは保存されません。イベントハンドラコードが新しいフォームにも適用できるかどうかわからないからです。メニューテンプレート項目に新しいイベントハンドラを作成すると，そのイベントハンドラの名前が生成されます。

メニューテンプレート内の項目は，フォームの既存の OnClick イベントハンドラと簡単に関連付けることができます。

実行時のメニュー項目の追加

追加の情報やオプションをユーザーに提供するために，アプリケーションの実行中にメニュー項目を既存のメニュー構造に追加する場合もあります。このようにメニュー項目を実行時に追加するには，Add メソッドまたは Insert メソッドを使用します。また，Visible プロパティを変更することで，メニュー項目の表示 / 非表示を切り替える方法もあります。Visible プロパティは，メニューにメニュー項目を表示するかどうかを指定します。メニュー項目を（非表示ではなく）淡色表示にするには，Enabled プロパティを使用します。

メニュー項目の Visible および Enabled プロパティを使用する例については，6-10 ページの「メニュー項目を使用不可にする」を参照してください。

MDI（マルチドキュメントインターフェース）と OLE（Object Linking and Embedding）アプリケーションでは，メニュー項目を既存のメニューバーにマージすることもできます。この点については，次の節で詳しく説明します。

メニューのマージ

テキストエディタのサンプルのような MDI アプリケーション，あるいは OLE のクライアントアプリケーションでは，アプリケーションのメインメニューは別のフォームや OLE サーバーオブジェクトのメニュー項目を受け取れなければなりません。ほかのメニュー項目を取り込むことを，メニューのマージと呼びます。なお，OLE 技術は Windows アプリケーションに限定されており，クロスプラットフォームプログラミングには使用できません。

メニューの 2 つのプロパティに値を指定して，マージの準備をします。

- フォームの Menu プロパティ
- メニュー内のメニュー項目の GroupIndex プロパティ

Menu プロパティでアクティブメニューを指定する

Menu プロパティで，フォームのアクティブメニューを指定します。メニューのマージ操作はアクティブメニューだけに適用されます。フォームに複数のメニューコンポーネントがある場合は，コードで Menu プロパティを設定して実行時にアクティブメニューを変更できます。下に例を示します。

```
Form1->Menu = SecondMenu;
```

GroupIndex プロパティでメニュー項目のマージ順を指定する

GroupIndex プロパティは，マージするメニュー項目を共有メニューバーに表示する順番を指定します。メニュー項目のマージでは，メインメニューバーのメニュー項目を置き換えることも追加することもできます。

GroupIndex のデフォルト値は 0 です。GroupIndex の値の指定には，以下の規則が適用されます。

- 小さい数字のメニュー項目ほどメニューの最初（左端）に表示される

  たとえば［ファイル］メニューなど，常に左端に表示するメニューの GroupIndex プロパティには 0（ゼロ）を設定します。同じように，ヘルプメニューなど常に右端に表示するメニューには大きな数字を指定します。なお，GroupIndex の値は連続した数でなくてもかまいません。

- メインメニューの項目を置き換えるには，子メニューの項目にメインメニューと同じ GroupIndex 値を指定する

  これはグループ化された項目や単一の項目に対して使えます。たとえば，メインフォームに GroupIndex の値が 1 に設定されている Edit メニュー項目がある場合，このメニュー項目を置き換えるには子フォームの 1 つ以上のメニュー項目に，GroupIndex の値として 1 を指定します。

  子フォームの複数のメニュー項目に同じ GroupIndex 値を指定しメインメニューにマージした場合は，各メニュー項目は元の順番を保持します。

- メインメニューに項目を置き換えずに追加するには，メインメニューの項目の数値の範囲を広げて，子フォームの数値を間に「はめ込む」ようにする

  たとえば，メインメニューの項目には 0 と 5 の数値を指定し，子フォームのメニュー項目には 1，2，3，4 の数値を指定して挿入します。

リソースファイルのインポート

C++Builder では，ほかのアプリケーションで構築されたメニューも利用できます。ただし，標準の Windows リソース（.RC）ファイル形式のものに限ります。この種のメニューは C++Builder プロジェクトに直接インポートできるので，同じメニューを再構築する時間と手間を省くことができます。

既存の .RC ファイル形式のメニューを読み込む手順は次のとおりです。

1. メニューデザイナで，メニューを表示したい位置にカーソルを配置します。

インポートしたメニューは，そのまま変更せずに使用することも，設計中のメニューの一部として組み込むこともできます。

2. 右クリックして［リソースから追加］を選択します。

   ［リソースから追加］ダイアログボックスが表示されます。

3. このダイアログボックスで，読み込むリソースファイルを指定して［OK］を選択します。

   メニューデザイナウィンドウにメニューが表示されます。

メモ リソースファイルに複数のメニューがある場合は，最初に各メニューを個別のリソースファイルとして保存してからインポートします。

ツールバーとクールバーの設計

ツールバーはメニューバーの下に位置するパネルで，通常はフォームの上端に水平に表示されます。ツールバーには主にボタンなどのコントロールが入ります。クールバー（リバーとも呼ばれる）はツールバーの一種で，移動やサイズ変更が可能なバンドにコントロールを表示します。複数のパネルをフォームの上端に配置した場合は，縦方向に追加順に並べられます。

メモ クールバーは CLX に用意されていないので，クロスプラットフォームアプリケーションでは使用できません。

ツールバーにはさまざまなコントロールを配置できます。ボタンのほかにカラーグリッド，スクロールバー，ラベルなどを置くこともできます。

複数の方法でツールバーをフォームに追加できます。

- フォームにパネルを置き（TPanel），そのパネルにコントロール（一般にスピードボタン）を追加します。
- TPanel のかわりにツールバーコンポーネント（TToolBar）を置き，それにコントロールを追加する。TToolBar は，ボタンその他のコントロールを管理し，配置を決め，サイズと位置を自動的に調整します。ツールバーのツールボタン（TToolButton）を使用すると，TToolBar によりボタンが機能によって簡単にグループ化され，その他の表示オプションが提供されます。
- クールバーコンポーネント（TCoolBar）を置き，それにコントロールを追加する。クールバーは，個々に移動やサイズ変更が可能なバンドの上にコントロールを表示します。

ツールバーをどのように実装するかは，作成するアプリケーションの種類によります。パネルコンポーネントを利用することの利点は，ツールバーのルックアンドフィールを一元的に制御できる点です。

ツールバーおよびクールバーコンポーネントを使うことは，ネイティブの Windows コントロールを使うことを意味するため，作成するアプリケーションが Windows アプリケーションのルックアンドフィールを持つことを保証します。これらのオペレーションシステムコントロールが将来変更された場合，作成したアプリケーションも同様に変更できます。また，ツールバーとクールバーは同じ Windows コンポーネントに依存するため，作成するアプリケーションには COMCTL32.DLL が必要となります。ツールバーとクールバーは，Windows NT 3.51 のアプリケーションではサポートされていません。

次の節では，以下の方法について説明します。

- パネルコンポーネントを使ったツールバーと対応するスピードボタンコントロールの追加
- ToolBar コンポーネントを使ったツールバーと対応するツールボタンコントロールの追加
- クールバーコンポーネントを使ったクールバーの追加
- クリックへの応答
- 非表示のツールバーとクールバーの追加
- ツールバーとクールバーの表示および非表示

パネルコンポーネントを使ってツールバーを追加する

パネルコンポーネントを使ってフォームにツールバーを追加する手順は，次のとおりです。

1. ［コンポーネント］パレットの［Standard］ページから，パネルコンポーネントをフォームに追加します。
2. パネルの Align プロパティに alTop を設定します。フォームの上端に配置されると，パネルはウィンドウのサイズが変更されても常に一定の高さを持ち，幅はフォームのクライアント領域の幅になります。
3. このパネルにスピードボタンなどを配置します。

スピードボタンは，ツールバーパネルでの作業用に設計されています。通常，スピードボタンにはキャプションがありません。かわりに，ボタンの機能を示すグリフという小さなグラフィックイメージがあります。

スピードボタンには，以下の 3 つの動作モードがあります。

- 通常のプッシュボタンとして機能する
- クリックに応じてオンとオフに切り替わる
- ラジオボタンとして機能する

ツールバーのスピードボタンを実装するには，次のようにします。

- パネルにスピードボタンを追加する
- スピードボタンへのグリフの割り当て
- スピードボタンの初期設定
- スピードボタンのグループ化
- 切り替えボタンとしての設定

パネルにスピードボタンを追加する

ツールバーのパネルにスピードボタンを追加するには，コンポーネントパレットの［Additional］ページから，パネルにスピードボタンを挿入します。

スピードボタンのオーナーはフォームではなくパネルです。したがって，パネルの移動や非表示化に応じてスピードボタンも移動したり非表示になります。

ツールバーのデフォルトの高さは 41 で，スピードボタンのデフォルトの高さは 25 です。各ボタンの Top プロパティに 8 を設定すれば，各ボタンの縦位置はツールバーの中央になります。デフォルトでは，この位置にスピードボタンが配置されます。

スピードボタンのグリフを割り当てる

各スピードボタンには，ボタンの機能をユーザーに示すグリフというグラフィックイメージを付けます。スピードボタンに 1 つのイメージを指定すると，ボタンを押した状態，押していない状態，選択した状態，使用不可の状態のそれぞれの表示が，そのイメージを基に作成されます。また，各状態に個別のイメージを指定することもできます。

通常は，設計時にスピードボタンにグリフを割り当てます。ただし，実行時に別のグリフを割り当てることもできます。

設計時にスピードボタンにグリフを割り当てる手順は，次のとおりです。

1. スピードボタンを選択します。
2. オブジェクトインスペクタで Glyph プロパティを選択します。
3. Glyph の横の値列をダブルクリックし，画像エディタを開き，適切なビットマップを選択します。

スピードボタンを初期設定する

ユーザーは，スピードボタンの機能や状態をスピードボタンの外観から判断します。スピードボタンではキャプションを使用しないので，表示するグリフは機能状態を示す的確なイメージでなければなりません。

表 8.6 に，スピードボタンの外観を変えるアクションを示します。

表 8.6　スピードボタンの外観の設定

スピードボタンの外観	プロパティの設定
押された状態	GroupIndex プロパティを 0 以外の値に，Down プロパティを **true** に設定する
使用不可の状態	Enabled プロパティを **false** に設定する
左マージンを設定	ツールバーの Indent プロパティを 0 よりも大きな値に設定する

たとえば，アプリケーションにデフォルトの描画ツールがある場合，そのツールに対応するボタンが，アプリケーションの開始時に押下されているように設定します。これを行うには，GroupIndex プロパティの値を 0 以外にし，Down プロパティを **true** にします。

スピードボタンをグループ化する

いくつかのスピードボタンをまとめて，相互に排他的なオプションにすることができます。この場合，ボタンをグループとして関連付けして，ユーザーがグループ内のいずれかのボタンをクリックすると，グループ内のほかのボタンが押されていない状態になります。

複数のスピードボタンをグループとして関連付けるには，各スピードボタンの GroupIndex プロパティに同じ番号を割り当てます。

この作業をもっとも簡単に行うには，グループにまとめるすべてのボタンを選択した状態で，GroupIndex に重複しない値を設定します。

切り替えボタンとして設定する

グループ内ですでに押されているボタンを再度クリックして元に戻し，グループ内のボタンが 1 つも押されていない状態にしたい場合もあります。このようなボタンを切り替えボタンと呼びます。グ

ループ内のボタンを切り替えボタンにするには，AllowAllUp を使用します。切り替えボタンは 1 回クリックすると押し下げられ，もう一度クリックすると押し上げられます。

グループ内のスピードボタンを切り替えボタンにするには，そのボタンの AllowAllUp プロパティに **true** を設定します。

グループ内でいずれかのスピードボタンの AllowAllUp プロパティに **true** を設定すると，グループ内のすべてのボタンに同じプロパティ値が自動的に設定されます。このグループでは，通常のグループと同じように 1 つのボタンを押された状態に設定できますが，さらに，すべてのボタンを押されていない状態にすることもできます。

ツールバーコンポーネントを使ってツールバーを追加する

ツールバーコンポーネント（TToolBar）は，パネルコンポーネントが行わない機能の管理と表示を行います。ツールバーコンポーネントを使ってフォームにツールバーを追加する手順は次のとおりです。

1. コンポーネントパレットの［Win32］ページから，フォームにツールバーコンポーネントを追加します。ツールバーは自動的にフォームの上端に配置されます。
2. バーにツールボタンを追加するか，またはほかのコントロールを追加します。

ツールボタンは，ツールバーコンポーネントで機能するよう設計されています。スピードボタンと同じように，ツールボタンも以下のことができます。

- 通常のプッシュボタンとして機能する
- クリックに応じてオンとオフに切り替わる
- ラジオボタンとして機能する

ツールバーにボタンを実装するには，以下のことを行います。

- ツールボタンを追加する
- ツールボタンにイメージを割り当てる
- ツールボタンの外観の設定
- ツールボタンをグループ化する
- 切り替えツールボタンとして設定する

ツールボタンを追加する

ツールバーにツールボタンを追加するには，ツールバーを右クリックして，コンテキストメニューから［ボタン新規作成］を選択します。

ツールボタンのオーナーはツールバーです。したがって，ツールバーの移動や非表示化に応じてツールボタンも移動したり非表示になります。またツールバー上のすべてのツールボタンは，自動的に常に一定の高さと幅を維持します。コンポーネントパレットからツールバーにコントロールをドロップすると，自動的に一定の高さになります。幅がツールバーに入りきらないときは，折り返して新しい列が作成されます。

ツールボタンにイメージを割り当てる

各ツールボタンには，実行時に表示するイメージを決める ImageIndex プロパティが含まれています。ツールボタンに 1 つのイメージを指定すると，ボタンが使用不可であるかどうかの表示が，そのイ

メージを基に作成されます。また，各状態に（使用不可，使用可能，押されている）個別のイメージを指定することもできます。設計時にツールボタンにイメージを割り当てる方法は次のとおりです。

1. ボタンを表示するツールバーを選択します。
2. オブジェクトインスペクタで，TImageList オブジェクトをツールバーの Images プロパティに割り当てます。イメージリストは同じサイズのアイコンやビットマップの集まりです。
3. ツールボタンを選択します。
4. オブジェクトインスペクタで，ツールボタンの ImageIndex プロパティに整数を割り当てます。これはボタンに割り当てるイメージリストのイメージに対応します。

ツールボタンが使用不可のときやマウスポインタが置かれたときに，ツールボタンに個別のイメージを表示するよう指定することもできます。それには，個別イメージリストをツールバーの DisabledImages プロパティと HotImages プロパティにそれぞれ割り当てます。

ツールボタンの外観と初期状態を設定する

表 8.7 に，ツールボタンの外観を変えるアクションを示します。

表 8.7　ツールボタンの外観の設定

ツールボタンの外観	プロパティの設定
押された状態	（ツールボタンで）Style プロパティを tbsCheck に設定し，Down プロパティを **true** に設定する
使用不可の状態	Enabled プロパティを **false** に設定する
左マージンを設定	ツールバーの Indent プロパティを 0 よりも大きな値に設定する
ポップアップボーダーを設定し，ツールバーを透明に見せる	Flat プロパティに **true** を設定する

メモ　TToolBar の Flat プロパティを使用するには，COMCTL32.DLL のバージョン 4.70 以降が必要です。

特定のツールボタンの後に強制的に新しいコントロールの列を開始するには，列の最後に表示するツールボタンを選択し，その Wrap プロパティに **true** を設定します。

ツールバーの自動折り返し機能を解除するには，ツールバーの Wrapable プロパティに **false** を設定します。

ツールボタンをグループ化する

ツールボタンをグループ化するには，関連付けるボタンを選択して，その Style プロパティに tbsCheck を設定してから，Grouped プロパティに **true** を設定します。グループ化されたツールボタンを選択すると，同じグループ内のボタンが押し上げられます。これは，相互に排他的な選択を表現する際に有効です。

Style プロパティが tbsCheck で設定され，Grouped プロパティが **true** に設定された隣接する一連のツールボタンは，1 つのグループを形成します。ツールボタンのグループを解除するには，ボタンの間に次のいずれかを入れます。

- Grouped プロパティに **false** が設定されたツールボタン

- Style プロパティに tbsCheck を設定していないツールボタン。ツールバーにスペースを空けたり，区切ったりするには，Style プロパティに tbsSeparator や tbsDivider を設定されたツールボタンを追加する
- 別のコントロール

切り替えツールボタンとして設定する

グループ内のツールボタンを切り替えボタンにするには，AllowAllUp を使用します。切り替えボタンは 1 度クリックすると押し下げられ，もう一度クリックすると押し上げられます。グループ内のツールボタンを切り替えボタンにするには，そのボタンの AllowAllUp プロパティに **true** を設定します。

スピードボタンと同じように，グループ内でいずれかのツールボタンの AllowAllUp プロパティに **true** を設定すると，グループ内のすべてのボタンに同じプロパティ値が自動的に設定されます。

クールバーコンポーネントの追加

メモ TCoolBar コンポーネントを使うには，COMCTL32.DLL のバージョン 4.70 以上が必要です。また，TCoolBar コンポーネントは CLX には含まれていません。

クールバー（TCoolBar）はリバーとも呼ばれるコンポーネントで，個々に移動可能でサイズ変更可能なバンドの上にウィンドウコントロールを表示します。ユーザーは各バンドの左側でサイズ変更グリップをドラッグして位置を決めることができます。

Windows アプリケーションでフォームにクールバーを追加する方法は次のとおりです。

1. コンポーネントパレットの［Win32］ページから，フォームにクールバーコンポーネントを追加します。クールバーが自動的にフォームの上端に配置されます。
2. コンポーネントパレットから，クールバーにウィンドウコントロールを追加します。

TWinControl から派生した VCL コンポーネントだけがウィンドウコントロールになります。クールバーには，ラベルやスピードボタンのようなグラフィックコントロールも追加できますが，別々のバンドに表示されるのではありません。

クールバーの外観の設定

クールバーコンポーネントには，次のような便利な構成オプションがあります。表 8.8 に，ツールボタンの外観を変えるアクションを示します。

表 8.8　クールボタンの外観の設定

クールバーの外観	プロパティの設定
バンドの大きさに合うように自動的にサイズ変更させる	AutoSize プロパティを **true** に設定する
バンドが一定の高さを維持するようにする	FixedSize プロパティを **true** に設定する
表示方向を水平から垂直に変更する	Vertical プロパティを **true** に設定する。これにより，FixedSize プロパティの効果が変更される
実行時にバンドの Text プロパティを表示しないようにする	ShowText プロパティを **false** に設定する。クールバーの各バンドには独自の Text プロパティがある
バーの周囲の境界を取り除く	BandBorderStyle を bsNone に設定する

表 8.8　クールボタンの外観の設定（つづき）

クールバーの外観	プロパティの設定
実行時にユーザーがバンドの順序を変更できないようにする（バンドの移動やサイズ変更はできる）	FixdedOrder を **true** に設定する
クールバーの背景イメージを作成する	Bitmap プロパティを TBitmap オブジェクトに設定する
バンドの左側に表示できるイメージのリストを選択する	Images プロパティを TImageList に設定する

イメージを個々のバンドに割り当てるには，クールバーを選択して，オブジェクトインスペクタで Bands プロパティをダブルクリックします。次にバンドを選択し，その ImageIndex プロパティに値を割り当てます。

クリックへの応答

ユーザーがツールバーボタンなどのコントロールをクリックすると，イベントハンドラを使って応答させることのできる OnClick イベントがアプリケーションで発生します。OnClick はボタンのデフォルトイベントなので，設計時にボタンをダブルクリックすると，スケルトンのイベントハンドラが生成されます。

ツールボタンにメニューを割り当てる

ツールボタン（TToolButton）を持つツールバー（TToolBar）を使用すると，特定のボタンにメニューを関連付けることができます。

1. ツールボタンを選択します。
2. オブジェクトインスペクタで，ツールボタンの DropDownMenu プロパティにポップアップメニュー（TPopupMenu）を割り当てます。

メニューの AutoPopup プロパティに **true** が設定されていると，ボタンが押されたときメニューが自動的に表示されます。

非表示のツールバーを追加する

ツールバーは常に表示されているとは限りません。複数のツールバーが存在する場合，実際に使用するツールバーだけを表示する方が便利です。複数のツールバーを持つフォームを作成したら，一部または全部のツールバーを非表示にできるように考慮します。

非表示のツールバーを作成する手順は次のとおりです。

1. フォームにツールバー，クールバー，またはパネルコンポーネントを追加します。
2. コンポーネントの Visible プロパティを **false** に設定します。

ツールバーは，設計時には常に表示されていて変更できますが，実行時にはアプリケーションで明示的に表示しない限り表示されません。

ツールバーを表示 / 非表示にする

アプリケーションで複数のツールバーを使用する場合，すべてを同時に表示するとフォームが混雑します。また，ツールバーをまったく使用しないユーザーもいます。すべてのコンポーネントに共通することですが，ツールバーも必要に応じて表示，非表示を実行時に切り替えられます。

実行時にツールバーを表示または非表示にするには，Visible プロパティにそれぞれ **true** または **false** を設定します。この設定は，特定のユーザーイベントやアプリケーションの操作モードの変更に対応して行います。設定するには，一般的に，各ツールバーにクローズボタンを追加します。ユーザーがクローズボタンをクリックすると，アプリケーションによりその対応するツールバーが非表示になります。

あるいは，ツールバーを切り替える手段を作成することもできます。次の例では，メインツールバーのボタンからペンのツールバーに切り替えられています。クリックするとボタンが押されたり放されたりするため，OnClick イベントハンドラは，ボタンが上か下かによって Pen ツールバーを表示または非表示にします。

```
void __fastcall TForm1::PenButtonClick(TObject *Sender)
{
  PenBar->Visible = PenButton->Down;
}
```

第9章

コントロールの種類

コントロールとは，ユーザーインターフェースを設計しやすくするビジュアルコンポーネントです。

この章では，テキストコントロール，入力コントロール，ボタン，リストコントロール，グループ化コントロール，表示コントロール，グリッド，値リストエディタ，グラフィックコントロールなど，さまざまなコントロールについて説明します。

グラフィックコントロールの作成については，第 54 章「グラフィックコントロールの作成」を参照してください。コントロールの実装方法については，第 6 章「コントロールの利用」を参照してください。

テキストコントロール

多くのアプリケーションは，テキストコントロールを使ってユーザーにテキストを表示します。以下のテキストコントロールが使用可能です。

- 編集コントロール：ユーザー側でテキストを追加できる

コンポーネント	用途
TEdit	単一行テキストの編集
TMemo	複数行テキストの編集
TMaskEdit	郵便番号や電話番号のような特定の書式を持つテキストの編集
TRichEdit	リッチテキスト形式の複数行テキストの編集（VCL のみ）

- テキスト表示コントロールおよびラベル：ユーザーはテキストを追加できない

コンポーネント	用途
TTextBrowser	テキストファイルまたはシンプルな HTML ページを表示する。ユーザーはスクロール表示ができる
TTextViewer	テキストファイルまたはシンプルな HTML ページを表示する。ユーザーはスクロール表示ができ，リンクをクリックして他のページや画像を表示することもできる
TLCDNumber	デジタル表示形式で数値情報を表示する
TLabel	非ウィンドウコントロールにテキストを表示する
TStaticText	ウィンドウコントロールにテキストを表示する

編集コントロール

編集コントロールは，実行時にテキストを表示し，ユーザーによるテキスト入力を可能にします。情報のサイズと形式に応じて数種類のテキストコントロールがあります。

TEdit と TMaskEdit は，単一行テキストの編集ボックスを持つシンプルな編集コントロールです。この編集ボックスにユーザーが情報を入力します。編集ボックスがフォーカスを受け取ると，挿入ポイントが点滅します。

編集ボックスにテキストを入れるには，編集ボックスの Text プロパティに文字列値を割り当てます。編集ボックスに入れたテキストの外観を変えるには，Font プロパティに値を割り当てます。フォントは，書体，サイズ，色，属性を指定できます。指定したフォント属性は，編集ボックス内のテキスト全体に影響します。個別の文字ごとにフォント属性を変えることはできません。

フォントサイズに応じてサイズが変わる編集ボックスを設計することもできます。フォントサイズによって編集ボックスのサイズが変わるようにするには，AutoSize プロパティを **true** に設定します。編集ボックスに入る文字の数を制限するには，MaxLength プロパティに値を割り当てます。

TMaskEdit は，テキストがとりうる有効な形式にコード化したマスクに対して入力テキストを検証する特殊な編集コントロールです。このマスクは，ユーザーに表示するテキストの形式を設定することもできます。

TMemo と TRichEdit は，ユーザー側でテキストを複数行追加できます。

編集コントロールのプロパティ

編集コントロールの主なプロパティは次のとおりです。

表 9.1　編集コントロールのプロパティ

プロパティ	説明
Text	編集ボックスやメモコントロールに表示されるテキストを指定する
Font	編集ボックスやメモコントロールに入力されたテキストの属性を制御する
AutoSize	現在選択されているフォントに応じて，編集ボックスの高さを動的に変化させる
ReadOnly	ユーザーがテキストを変更できるかどうか指定する
MaxLength	簡易編集コントロール内の文字数を制限する
SelText	現在選択されている（強調表示している）テキスト部分を保持する
SelStart，SelLength	選択されているテキスト部分の位置と長さを示す

メモコントロールと書式付きテキスト編集コントロール

TMemo コントロールも TRichEdit コントロールも，複数行のテキストを処理します。

VCL　書式付きテキスト編集コントロールは，VCL でのみ使用できます。

TMemo も編集ボックスの一種ですが，TEdit と違って複数行のテキストを処理できます。メモコントロールでは，テキスト行が編集ボックスの右端を超えることが可能です。あるいは，超えた分を次の行に折り返すこともできます。次の行に折り返すかどうかを指定するには，WordWrap プロパティを使います。

TrichEdit は，リッチテキストの書式設定，印刷，検索，ドラッグアンドドロップをサポートするメモコントロールです。つまり，フォント属性，位置揃え，タブ，インデント，番号付けの指定ができます。

メモコントロールと書式付きテキスト編集コントロールには，他の編集コントロールが持っているプロパティのほかに，次のようなプロパティがあります。

- Alignment プロパティ：コンポーネント内でのテキストの揃え方（左揃え，右揃え，中央揃え）を指定します。
- Text プロパティ：コントロール内にテキストを保持します。Modified プロパティを使用すると，テキストが変更されたかどうかをアプリケーションで判定できます。
- Lines プロパティ：テキストを文字列リストとして保持します。
- OEMConvert プロパティ：コントロールに入力されたテキストを一時的に ANSI から OEM に変換するかどうかを指定します。ファイル名を検証するのに便利です（VCL のみ）。
- WordWrap プロパティ：右マージンでテキストをワードラップするかどうかを指定します。
- WantReturns プロパティ：ユーザーがテキストに改行文字を挿入できるかどうかを指定します。
- WantTabs プロパティ：ユーザーがテキストにタブを挿入できるかどうかを指定します。
- AutoSelect プロパティ：オブジェクトがアクティブになったときにテキストを自動的に選択する（強調表示する）かどうかを指定します。

SelectAll メソッドを使うと，実行時にメモ内のすべてのテキストを選択できます。

テキスト表示コントロール（CLX のみ）

テキスト表示コントロールは，読み取り専用のテキストを表示します。TTextViewer は簡易ビューアとして機能します。TTextViewer を使うと，ユーザーはスクロールしながらテキストを読むことができます。TTextBrowser を使えば，ユーザーはリンクをクリックして他のドキュメントに移動したり，同じドキュメント内の他の部分を表示したりできます。閲覧されたドキュメントは履歴リストに保存され，Backward，Forward，Home の各メソッドで移動できます。HTML テキストを表示したり，HTML ベースのヘルプシステムを実装するには，TTextViewer と TTextBrowser を使うのがベストです。

TTextBrowser には，TTextViewer が持っているプロパティのほかに，Factory というプロパティがあります。Factory は，埋め込み画像のファイルタイプを決定するのに使われる MIME ファクトリオブジェクトを指定します。たとえば，.txt，.html，.xml などの拡張子と MIME 型を関連付けて，ファクトリにそのデータをコントロールへロードさせることができます。

FileName プロパティを使ってテキストファイル（.html など）を追加すると，実行時にコントロールに表示されます。

ラベル

ラベル（TLabel と TStaticText（VCL のみ））は，テキストを表示するコンポーネントです。通常は他のコントロールの隣に置きます。編集ボックスなどのコンポーネントを識別したり注釈を付けたりする場合や，フォームにテキストを表示したい場合，フォームにラベルを配置します。標準のラベルコンポーネント TLabel は非ウィンドウコントロール（CLX では非ウィジェットベース）なので，フォーカスを受け取ることができません。ウィンドウハンドルを持つラベルを作成するには，TLabel ではなく TStaticText を使います。

ラベルの主なプロパティは次のとおりです。

- Caption プロパティ：ラベルのテキスト文字列を保持します。
- Font プロパティ，Color プロパティなどのプロパティ：ラベルの外観を決定します。各ラベルに設定できる書体，サイズ，色はそれぞれ 1 種類です。
- FocusControl プロパティ：ラベルをフォーム上の別のコントロールとリンクさせます。キャプションにアクセラレータキーが含まれる場合，ユーザーがアクセラレータキーを使うと，FocusControl プロパティで指定したコントロールにフォーカスが移動します。
- ShowAccelChar プロパティ：下線付きアクセラレータ文字をラベルに表示するかどうかを指定します。ShowAccelChar を true にすると，&（アンド記号）で始まる文字に下線が表示され，有効なアクセラレータ文字になる
- Transparent プロパティ：ラベルの下に重なっている項目（グラフィックなど）が隠れないように表示するかどうかを指定します。

通常，ラベルは読み出し専用の静的テキストとして表示されるので，アプリケーションのユーザーはラベルを変更できません。アプリケーションの実行時にラベルのテキストを変更するには，Caption プロパティに別の値を割り当てます。ユーザーがスクロールしたり編集できるテキストオブジェクトをフォームに追加する場合は，TEdit を使います。

特殊入力コントロール

特殊な入力を扱う以下のコンポーネントが使えます。

コンポーネント	用途
TScrollBar	連続した範囲内の値を選択する
TTrackBar	連続した範囲内の値を選択する（ScrollBar よりも視覚効果が高い）
TUpDown	Edit コンポーネントに結び付けた増減ボタンを使って値を選択する（VCL のみ）
THotKey	〔Ctrl〕，〔Shift〕，〔Alt〕のキーシーケンスを入力する（VCL のみ）
TSpinEdit	スピンウィジェットから値を選択する（CLX のみ）

スクロールバー

ScrollBar コンポーネントはスクロールバーを作成します。スクロールバーを使うと，ウィンドウ，フォーム，コントロールの内容をスクロールできます。ユーザーがスクロールバーを動かしたときに応答するコントロールの動作を指定するコードは，OnScroll イベントハンドラに作成します。

多くのビジュアルコンポーネントには，コードを追加せずに使える独自のスクロールバーがあります。このため，ScrollBar コンポーネントはさほど頻繁に使われません。たとえば TForm の VertScrollBar プロパティと HorzScrollBar プロパティを使うと，スクロールバーが自動的にフォームに配置されます。フォーム内にスクロール可能な領域を作成するには，TScrollBox を使います。

トラックバー

トラックバーを使うと，連続した範囲の整数値を設定できます。ボリュームや明るさなどの調整に使える便利なコントロールです。ユーザーが値を調整するには，スライドインジケータを特定の位置までドラッグするか，トラックバーの内側をクリックします。

- トラックバーの範囲の上端と下端を設定するには，Max プロパティと Min プロパティを使います。
- 選択範囲を強調表示するには，SelEnd プロパティと SelStart プロパティを使います（図 9.1 を参照してください）。
- トラックバーの向きを水平または垂直に設定するには，Orientation プロパティを使います。
- デフォルトでは，トラックバーの下に 1 行の目盛りが表示されます。目盛りの位置を変えるには，TickMarks プロパティを使います。目盛りの間隔を制御するには，TickStyle プロパティと SetTick メソッドを使います。

図 9.1　トラックバーコンポーネントの 3 つの例

- Position プロパティはトラックバーのデフォルトの位置を設定します。実行時にユーザーが選択した値の追跡も行います。
- デフォルトでは，ユーザーが上下の矢印キーを押すと 1 目盛りずつ移動できます。1 回の移動分を変更するには，LineSize プロパティを設定します。
- ユーザーが【PageUp】と【PageDown】を押したときに移動する刻みの数を指定するには，PageSize プロパティを設定します。

アップダウンコントロール（VCL のみ）

アップダウンコントロール（TUpDown）は上下の矢印ボタンから成り，一定の値ごとに整数値を増減させます。現在の値は Value プロパティで設定し，増減値（デフォルトは 1）は Increment プロパティで指定します。アップダウンコントロールを別のコンポーネント（テキストコントロールなど）と結び付けるには，Associate プロパティを使います。

スピンエディットコントロール（CLX のみ）

スピンエディットコントロール（TSpinEdit）は，「アップダウンウィジェット」，「リトルアローウィジェット」，「スピンボタン」とも呼ばれます。スピンエディットコントロールは，一定の値ごとに整数値を増減させます。実際に値を変えるには，アプリケーションのユーザーが上下の矢印ボタンをクリックして値を増減させるか，スピンボックスに直接値を入力します。

現在の値は Value プロパティで設定し，増減値（デフォルトは 1）は Increment プロパティで指定します。

ホットキーコントロール（VCL のみ）

コンポーネントにフォーカスを移動するキーボードショートカットを指定するには，ホットキーコンポーネント（THotKey）を使います。HotKey プロパティは現在のキーの組み合わせを保持します。Modifiers プロパティは，HotKey に使用できるキーを決定します。

ホットキーは，メニュー項目の ShortCut プロパティとして割り当てることもできます。HotKey プロパティと Modifier プロパティでキーの組み合わせを指定しておくと，ユーザーがそのキーを使ったとき，Windows が目的のメニュー項目を起動します。

スプリッタコントロール

位置を揃えた 2 つのコントロールの間にスプリッタ（TSplitter）を追加すると，ユーザーがその 2 つのコントロールのサイズを変更できるようになります。パネルやグループボックスなどのコンポーネントと一緒にスプリッタを使うと，フォームを複数のペインに分割し，各ペインに複数のコントロールを置くことができます。

フォームにパネルなどのコントロールを置いてから，そのコントロールと同じ位置合わせ（Align プロパティ）が設定されたスプリッタを配置します。最後に配置したコントロールの Align プロパティはクライアント基準（alClient）にします。これにより，ほかのコントロールをサイズ変更したとき，最後のコントロールが残りの領域を占めるようになります。たとえば，まずフォームの左端にパネルを置き，その Align を alLeft に設定します。次に，パネルの右にスプリッタを置き，これも alLeft に位置揃えします。最後に，スプリッタの右に別のパネルを置き，alLeft か alClient に設定します。

MinSize プロパティは，隣り合うコントロールのサイズが変更されたときに，スプリッタの表示を残さなければならない最小サイズを指定します。Beveled プロパティを **true** に設定すると，スプリッタの端に 3D の外観が付加されます。

ボタンおよび類似のコントロール

アプリケーションでコマンドを起動するときは，メニューから選択するかわりにボタンを使うのが一般的方法です。C++Builder にはボタンコントロールが用意されています。

コンポーネント	用途
TButton	テキストを使ってコマンドの選択肢を表す
TBitBtn	テキストとグリフを使ってコマンドの選択肢を表す
TSpeedButton	グループ化されたツールバーボタンを作成する
TCheckBox	オン / オフのオプションを表す
TRadioButton	相互に排他的な（同時に 1 つしか選べない）選択肢を表す
TToolBar	ツールボタンとその他のコントロールを横一列に配置し，サイズと位置を自動的に調整する
TCoolBar	移動可能，サイズ変更可能なバンド内にウィンドウコントロールを表示する（VCL のみ）

ボタンコントロール

ユーザーはマウスでボタンコントロールをクリックして操作を開始できます。ボタンには，アクションを表すテキストのラベルが付きます。テキストを指定するには，Caption プロパティに文字列値を割り当てます。たいていの場合，キーボードショートカットとしてキーボードのキーを押してもボタンを選択できます。ボタンに表示されている下線付きの文字がショートカットです。

ユーザーはボタンコントロールをクリックして操作を開始できます。TButton コンポーネントにアクションを割り当てるには，そのアクションの OnClick イベントハンドラを作成します。設計時にボタンをダブルクリックすると，そのボタンの OnClick イベントハンドラがコードエディタに表示され編集できる状態になります。

- ユーザーが【Esc】を押したときにボタンの OnClick イベントを発生させるようにするには，Cancel プロパティを **true** に設定します。
- ユーザーが【Enter】を押したときにボタンの OnClick イベントを発生させるようにするには，Default プロパティを **true** に設定します。

ビットマップボタン

ビットマップボタンコンポーネント（BitBtn）は，ボタン面にビットマップのイメージを表示するボタンコントロールです。

- ボタン用のビットマップを選ぶには，Glyph プロパティを設定します。
- ボタンのグリフとデフォルト動作を自動的に設定するには，Kind プロパティを使います。
- デフォルトでは，グリフはテキストの左側に表示される。位置を移動するには，Layout プロパティを使います。
- グリフとテキストは自動的にボタンの中央に揃えられる。この位置を移動するには，Margin プロパティを使います。Margin プロパティは，イメージの端とボタンの端のピクセル数を設定します。
- デフォルトでは，イメージとテキストは 4 ピクセルだけ離れています。この距離を増減するには，Spacing プロパティを使います。
- ビットマップボタンには，上がった状態，下がった（押された）状態，使用不可の状態の 3 つがあります。状態ごとに異なるビットマップイメージを表示するには，NumGlyphs プロパティを 3 に設定します。

スピードボタン

スピードボタンは，通常，ボタン面にイメージが表示されたボタンです。グループとして動作させることができるという特性があります。通常は，パネルと一緒に使ってツールバーを作成します。

- 複数のスピードボタンをグループにまとめるには，グループ内のすべてのボタンの GroupIndex プロパティを 0 以外の同じ値に設定します。
- デフォルトでは，スピードボタンは上がった（選択されていない）状態で表示されます。最初からスピードボタンを選択した状態で表示するには，Down プロパティを **true** に設定します。

- AllowAllUp プロパティが **true** の場合，同じグループのすべてのスピードボタンは選択されていない状態になります。ラジオグループのように動作するボタングループとして使いたい場合は，AllowAllUp を **false** に設定します。

スピードボタンの詳細については，8-44 ページの「パネルコンポーネントを使ってツールバーを追加する」と 8-16 ページの「ツールバーとメニューのアクションを構成する」を参照してください。

チェックボックス

チェックボックスを作成すると，ユーザーはオン，オフのどちらかの状態を選択できます。オンが選択されると，チェックボックスをチェックした状態になります。オンが選択されなければ，チェックボックスはブランクになります。チェックボックスを作成するには，TCheckBox を使います。

- デフォルトでチェックボックスをチェックした状態にするには，Checked プロパティを true に設定します。
- チェックボックスの状態を 3 種類（チェックマークが付く，チェックマークが付かない，淡色表示）にするには，AllowGrayed プロパティを true に設定します。
- State プロパティは，チェックボックスにチェックマークが付いているか（cbChecked），付いていないか（cbUnchecked），それとも淡色表示されているか（cbGrayed）を表します。

メモ チェックボックスコントロールは，2 つのバイナリ状態のどちらか一方を表示します。ほかの設定の関係でチェックボックスの現在値を決定できないときは，不確定の状態を使用します。

ラジオボタン

ラジオボタンは，相互に排他的な（一度に 1 つしか選べない）選択肢の集合です。単独のラジオボタンを作成するには，TRadioButton を使います。ラジオグループコンポーネント TRadioGroup を使うと，ラジオボタンのグループが自動的に配置されます。ラジオボタンをグループ化しておけば，ユーザーはいくつかのラジオボタンの中から選択できます。詳細は，9-12 ページの「グループ化コントロール」を参照してください。

ラジオボタンが選択されると，円の中央が塗りつぶされた状態になります。ラジオボタンが非選択であれば，空白の円になります。ラジオボタンの表示状態を変更するには，Checked プロパティの値を **true** または **false** にします。

ツールバー

ツールバーを使うと，ビジュアルコントロールの配置と管理を簡単に行うことができます。ツールバーは ToolBar 使って作成します。パネルコンポーネントとスピードボタンを組み合わせて作ることもできます。ToolBar コンポーネントを使う場合は，右クリックして［ボタン新規作成］を選択すると，ボタンをツールバーに追加できます。

TToolBar コンポーネントにはいくつか利点があります。ToolBar を使うと，ツールバー上のボタンは一定のサイズと間隔を保ち，その他のコントロールは相対的な位置と高さを保ちます。また，コントロールがツールバーの横幅に収まらない場合は自動的に次の行に折り返します。さらに TToolBar は，

透過性，ポップアップボーダー，スペースなどの表示オプションのほか，コントロールをグループ化するディバイダなども提供します。

アクションリストかアクションバンドを使えば，一連のアクションを使ってツールバーとメニューを一元管理できます。アクションリストを使ったボタン，ツールバーの操作方法については，8-23 ページの「アクションリストの使い方」を参照してください。

ツールバーは，別のコントロール（編集ボックス，コンボボックスなど）の親になることもできます。

クールバー（VCL のみ）

クールバーには，個別に移動とサイズ変更が行える子コントロールが配置されます。各コントロールはそれぞれのバンド上に常駐します。ユーザーは，各バンドの左側のサイズ変更グリップをドラッグして位置を設定できます。

クールバーを使用するためには，設計時および実行時にバージョン 4.70 以上の COMCTL32.DLL（通常は Windows \ System または Windows \ System32 ディレクトリに格納されている）が必要です。クールバーは，クロスプラットフォームアプリケーションでは使用できません。

- Bands プロパティは TCoolBand オブジェクトの集合を保持する。設計時には，バンドエディタを使ってバンドの追加，削除，変更が行える。バンドエディタを開くには，オブジェクトインスペクタで Bands プロパティを選択して右側の欄をダブルクリックするか，省略記号（...）ボタンをクリックする。または，スピードメニューの［バンドの設定］を選択する。バンドの作成は，パレットから新しいウィンドウコントロールを追加することによっても行える
- FixedOrder プロパティは，ユーザーがバンドの順序を変更できるかどうかを指定する
- FixedSize プロパティは，バンドを統一した高さに保つかどうかを指定する

リストコントロール

リストは，選択する項目の集合をユーザーに表示します。リストを表示するコンポーネントは以下のとおりです。

コンポーネント	表示内容
TListBox	テキスト文字列のリスト
TCheckListBox	各項目の前にチェックボックスの付いたリスト
TComboBox	スクロール可能なドロップダウンリストを持つ編集ボックス
TTreeView	階層状のリスト
TListView	オプションでアイコン，カラム，見出しの付いた（ドラッグ可能な）項目のリスト
TIconView（CLX のみ）	大きいアイコンか小さいアイコンの形で行と列に並べて表示される項目またはデータのリスト
TDateTimePicker	日付または時刻入力用のリストボックス（VCL のみ）
TMonthCalendar	日付を選択するためのカレンダー（VCL のみ）

文字列リストとイメージリストを管理するには，非ビジュアルコンポーネントの TStringList と TImageList を使います。文字列リストについての詳細は，4-15 ページの「文字列リストの操作」を参照してください。

リストボックス / チェックリストボックス

リストボックス（TListBox）とチェックリストボックスは，ユーザーが選択できる項目の一覧を表示します。

- Items プロパティは，TStrings オブジェクトを使ってコントロールに値を入れます。
- ItemIndex プロパティは，リストのどの項目が選択されているかを表します。
- MultiSelect プロパティは，ユーザーが一度に複数の項目を選択できるかどうかを指定します。
- Sorted プロパティは，リストをアルファベット順に並べるかどうかを決定します。
- Columns プロパティは，リストコントロール内の列数を指定します。
- IntegralHeight プロパティは，リストボックスの縦のスペースに完全に入る項目だけを表示するかどうか指定する（VCL のみ）
- ItemHeight プロパティは，各項目の高さをピクセル単位で指定します。Style プロパティの設定によっては，ItemHeight の値が無視されることがあります。
- Style プロパティは，リストコントロールに項目を表示する方法を指定します。デフォルトでは，項目は文字列として表示されます。Style プロパティの値を変更すると，オーナー描画リストボックスを作成できます。つまり，項目をグラフィックにしたり高さを可変にしたりできます。オーナー描画コントロールについての詳細は，6-12 ページの「コントロールへのグラフィックの追加」を参照

単純なリストボックスを作成する手順は次のとおりです。

1. プロジェクトを作成し，コンポーネントパレットにあるリストボックスコンポーネントをフォームにドラッグします。
2. 必要に応じてリストボックスのサイズと位置揃えを設定します。
3. Items プロパティの右側をダブルクリックするか省略記号ボタンをクリックして，文字列リストエディタを表示します。
4. 文字列リストエディタを使って，リストボックスに表示する自由形式テキストを入力し，順に並べます。
5. ［OK］をクリックします。

ユーザーがリストボックス内の複数の項目を選択できるようにするには，ExtendedSelect プロパティと MultiSelect プロパティを使います。

コンボボックス

コンボボックス（TComboBox）は，スクロール可能なリストと編集ボックスを組み合わせたものです。テキストを入力するかリストの項目を選択してユーザーがコントロールにデータを入力すると，

Text プロパティの値が変化します。AutoComplete を有効にした場合，アプリケーションは，ユーザーが入力したデータにもっとも近いものをリストから探して表示します（AutoComplete プロパティは SBCS 文字に対してのみ有効です）。

3 種類のコンボボックスがあります（標準，ドロップダウン，ドロップダウンリスト）。デフォルトはドロップダウンです。

- コンボボックスの種類を選択するには，Style プロパティを使います。
- 編集ボックス付きのドロップダウンリストを作成するには，csDropdown を使う。編集ボックスを読み出し専用にする（ユーザーに強制的にリストから選択させる）には，csDropDownList を使います。リストに表示する項目の数を変えるには，DropDownCount プロパティを設定します。
- リストが閉じない編集ボックスを作成するには csSimple を使う。リスト項目が表示されるようにコンボボックスのサイズを調整すること
- オーナー描画コンボボックスを作成して項目をグラフィカル表示にするか高さを可変にするには，csOwnerDrawFixed または csOwnerDrawVariable を使います。オーナー描画コントロールについての詳細は，6-12 ページの「コントロールへのグラフィックの追加」を参照

実行時，CLX と VCL ではコンボボックスの動作が異なります。CLX では，コンボボックスの編集フィールドにテキストを入力して〔Enter〕を押すと，ドロップダウンリストに項目を追加できます（VCL のコンボボックスではできません）。InsertMode を ciNone に設定すれば，この機能がオフになります。コンボボックスのリストには，空の（文字列のない）項目を追加することもできます。また，〔↓〕を押し続けるとコンボボックスのリストの最後の項目では止まらず，一番上に戻ります。

ツリービュー

ツリービュー（TTreeView）は，項目の全体構造をインデントして表示します。このコントロールは，ノードを展開するボタンと折りたたむボタンを提供します。各項目のテキストラベルが付いたアイコンを入れられるので，ノードが展開しているか折りたたんでいるかをアイコンで表示できます。チェックボックスなどのグラフィックを入れて，項目の状態を示すこともできます。

- Indent プロパティは，親項目と項目を水平方向で分割するときの距離をピクセル数で設定する
- ShowButtons プロパティは，項目が展開可能かどうかを示す［+］ボタンと［-］ボタンを表示させる
- ShowLines プロパティは，階層関係を示す接続線を表示するかどうかを指定する（VCL のみ）
- ShowRoot プロパティは，最上位項目を接続する線を表示するかどうか指定する（VCL のみ）

設計時にツリービューコントロールに項目を追加するには，コントロール上をダブルクリックして，ツリービュー項目エディタを表示します。ここで追加した項目が Items プロパティの値になります。実行時に項目を変更するには，Items プロパティ（TTreeNodes 型のオブジェクト）のメソッドを使います。TTreeNodes には，ツリービュー内で項目を追加，削除，移動するメソッドがあります。

ツリービューでは，リストビューの vsReport モードに似た形で列と下位項目を表示できます。

リストビュー

リストビューを使うと，さまざまな形式でリストを表示できます。リストビューを作成するには TListView を使います。リストの種類を選択するには，ViewStyle プロパティを使います。以下に ViewStyle の設定値を示します。

- vsIcon と vsSmallIcon は，各項目をラベル付きアイコンとして表示する。ユーザーはリストビューウィンドウ内で項目をドラッグできる（VCL のみ）
- vsList は，項目をラベル付きアイコンとして表示します。項目をドラッグすることはできません。
- vsReport は，各項目に 1 行を割り当て，列形式で情報を配置します。一番左の列に小さいアイコンとラベルを表示し，それに続く列にアプリケーションで指定した下位項目を入れます。列にヘッダーを表示するには，ShowColumnHeaders プロパティを使う

DateTimePicker と MonthCalendar（VCL のみ）

DateTimePicker コンポーネントは，日付または時刻を入力するためのリストボックスを表示します。MonthCalendar コンポーネントは，日付または日付の範囲を入力するためのカレンダーを表示します。これらのコンポーネントを使用するには，設計時および実行時にバージョン 4.70 以上の COMCTL32.DLL（通常は Windows \ System または Windows \ System32 ディレクトリに格納）が必要です。DateTimePicker と MonthCalendar は，クロスプラットフォームアプリケーションでは使用できません。

グループ化コントロール

関連のあるコントロールと情報をグループに分けて表示すると，グラフィカルインターフェースがいっそう使いやすくなります。C++Builder には，コンポーネントをグループ化するコンポーネントがいくつかあります。

コンポーネント	用途
TGroupBox	タイトルが付いた標準のグループボックスを作成する
TRadioGroup	ラジオボタンだけの単純なグループを作成する
TPanel	視覚的に融通のきくコントロールグループを作成する
TScrollBox	コントロールを入れるスクロール可能領域を作成する
TTabControl	互いに排他的なノートブックスタイルのタブを作成する
TPageControl	互いに排他的なノートブックスタイルのタブとそれに対応するページを作成して各ページにほかのコントロールを入れる
THeaderControl	列ヘッダーのサイズを変更する

グループボックス / ラジオグループ

グループボックス（TGroupBox）は，関連のコントロールをフォーム上に並べるコンポーネントです。一般に，複数のラジオボタンをグループボックスにまとめます。最初にグループボックスを配置

し，その後でコンポーネントパレットからコンポーネントを選択し，グループボックスに貼り付けます。Caption プロパティで，実行時に表示するグループボックスのラベルを指定します。

ラジオグループコンポーネント（TRadioGroup）を使うと，ラジオボタンのグループ化とそのグループに協調動作をさせる処理を簡略化できます。ラジオグループにボタンを追加するには，オブジェクトインスペクタで Items プロパティを編集します。Items に入力した文字列がキャプションとなり，ラジオグループボックスにラジオボタンとキャプションが表示されます。ItemIndex プロパティの値は，現在どのボタンが選択されているかを示します。Columns プロパティは，ラジオボタンを何列で表示するかを指定します。ボタンの間隔を変更するには，ラジオグループコンポーネントのサイズを変更します。

パネル

TPanel は，ほかのコントロールの汎用コンテナになるコンポーネントです。一般に，複数のコンポーネントを視覚的にグループ化してフォームに配置するときにパネルを使います。フォームと位置合わせできるので，フォームのサイズを変更してもパネルの相対的な位置を維持できます。パネル回りの境界線の幅を指定するには，BorderWidth プロパティを使ってピクセル数で設定します。

パネルの中に複数のコントロールを格納できます。Align プロパティを使えば，グループ内のコントロールの位置を適切に保つことができます。パネルの Align プロパティを alTop に設定すると，フォームのサイズを変えてもパネルの位置は適正に維持されます。

パネルの外観を変える（もり上がった形かへこんだ形にする）には，BevelOuter プロパティと BevelInner プロパティを使います。これらのプロパティの値を変えると，3D のさまざまな視覚効果が得られます。ただし，単にベベルをもり上がった形かへこんだ形のどちらかにする場合には，TBevel を使った方がリソースが少なくてすみます。

また，1 つ以上のパネルを使って，さまざまなステータスバーや情報表示エリアを構築することもできます。

スクロールボックス

スクロールボックス（TScrollBox）は，フォームの中にスクロール領域を作成します。アプリケーションでは，特定領域に収まらない情報を表示する必要が生じることがよくあります。一部のコントロール（リストボックス，メモ，フォーム自身など）では，自動的に中身をスクロールできます。

スクロールボックスには別の使い方もあります。たとえば，同じウィンドウに複数のスクロール領域（ビュー）を作成することができます。一般に，市販のワードプロセッサ，スプレッドシート，プロジェクト管理アプリケーションなどでビューがよく使われています。スクロールボックスを使うと，より柔軟にフォーム内に任意のスクロール領域を定義できます。

パネルやグループボックスと同様に，スクロールボックス内に別のコントロール（TButton, TCheckBox など）を配置することができます。ただし，通常は，スクロールボックスは表示されません。しかし，スクロールボックスの可視領域にコントロールが表示しきれなくなると，自動的にスクロールバーが表示されます。

スクロールボックスには別の使い方もあります。たとえば，ウィンドウ内のツールバーやステータスバーなどの領域（TPanel コンポーネント）でスクロールを制限することができます。ツールバーとステータスバーがスクロールしないようにするには，スクロールバーを非表示にしてから，ウィンドウ内のツールバーとステータスバーの間のクライアント領域にスクロールボックスを置きます。このスクロールボックスに関連付けられたスクロールバーは，ウィンドウに属しているように見えますが，スクロールボックスの内部でしかスクロールしません。

タブコントロール

タブコントロールコンポーネント（TTabControl）は，ノートの仕切りに似たタブを作成します。オブジェクトインスペクタで Tabs プロパティに文字列を追加するごとに，新しいタブが作成されます。タブコントロールとは，パネル上にひとまとまりのコンポーネントが入った単一のパネルです。タブをクリックしたときの外観を変えるには，OnChange イベントハンドラを記述する必要があります。複数ページのダイアログボックスを作成するには，タブコントロールでなくページコントロールを使います。

ページコントロール

PageControl コンポーネント（TPageControl）は，複数ページのダイアログボックスを作成するのに適したページセットです。1 つのページコントロールに，複数の重なり合ったページ（TTabSheet オブジェクト）が表示されます。ユーザーインターフェースでページを選択するには，コントロールの上部に表示されたタブをクリックします。

設計時にページコントロールに新しいページを作成するには，ページコントロールを右クリックして［ページ新規作成］を選択します。実行時に新規ページを追加するには，新規ページのオブジェクトを作成し，その PageControl プロパティを設定します。

```
TTabSheet *pTabSheet = new TTabSheet(PageControl1);
pTabSheet->PageControl = PageControl1;
```

コードでアクティブページにアクセスするには，ActivePage プロパティを使います。プログラミングによってアクティブページを変更するには，ActivePage プロパティか ActivePageIndex プロパティを設定します。

ヘッダーコントロール

ヘッダーコントロール（THeaderControl）は列ヘッダーをまとめたものです。実行時には，ユーザーはヘッダーの選択とサイズ変更ができます。ヘッダーを追加，修正するには，Sections プロパティを使います。列やフィールドの上にヘッダーセクションを置きます。たとえば，リストボックス（TListBox）の上にヘッダーセクションを置くことがあります。

表示コントロール

ユーザーにアプリケーションの状態を知らせる方法は多数あります。たとえば，一部のコンポーネント（TForm を含む）が持っている Caption プロパティは，実行時にも設定できます。また，メッセージを表示するダイアログボックスを作成することもできます。特に，以下のコンポーネントは実行時に情報を表示する便利な方法です。

コンポーネントとプロパティ	用途
TStatusBar	ステータス領域を表示する（通常はウィンドウ下部に表示）
TProgressBar	特定の作業の進行状況を表示する
Hint と ShowHint	ヘルプヒントをアクティブにする
HelpContext と HelpFile	状況感知型オンラインヘルプにリンクする

ステータスバー

パネルを使ってステータスバーを作成することもできますが，ステータスバーコンポーネントを使った方が簡単です。デフォルトでは，ステータスバーの Align プロパティが alBottom に設定され，位置とサイズが管理されています。

ステータスバーに一度に表示する文字列を 1 行にするには，SimplePanel プロパティを true に設定し，ステータスバーに表示する文字列を SimpleText プロパティで設定します。

また，ステータスバーは複数のテキスト領域に分割して使用できます。パネルを作成するには Panels プロパティを使います。オブジェクトインスペクタからパネルエディタを開き，パネルの Width プロパティ，Alignment プロパティ，Text プロパティをそれぞれ設定します。各パネルの Text プロパティは，パネルに表示するテキストを保持します。

プログレスバー

アプリケーションが何か時間のかかる操作を実行する場合，プログレスバーを使って進行状況を表示することができます。プログレスバーで進行状況を表示すると，点線状のバーが左から右へ伸びていきます。

図 9.2　プログレスバー

Position プロパティは，点線状のバーの長さをトラッキングします。Max プロパティと Min プロパティは，Position プロパティの範囲を指定します。バーを進めるには，StepBy メソッドと StepIt メソッドを呼び出して Position プロパティの値を増やします。Step プロパティは，StepIt で使用する増分を指定します。

ヘルプとヒントプロパティ

ほとんどのビジュアルコントロールは，実行時に状況感知型ヘルプとヘルプヒントを表示できます。HelpContext プロパティはヘルプのコンテキスト番号を指定し，HelpFile プロパティはヘルプのファイル名を指定します。

Hint プロパティは，マウスポインタをコントロールまたはメニュー項目に置いたときに表示するテキスト文字列を保持します。ヘルプヒントを表示するには，ShowHint を **true** に設定します。ParentShowHint を **true** に設定しておくと，そのコントロールの ShowHint プロパティは親と同じ値になります。

グリッド（表）

グリッドとは，情報を行と列に並べて表示したものです。データベースアプリケーションを作成する場合は，TDBGrid コンポーネントか TDBCtrlGrid コンポーネントを使います。TDBGrid と TDBCtrlGrid についての詳細は，第 19 章「データコントロールの使い方」を参照してください。データベース以外のアプリケーションを作成する場合は，標準の描画グリッドか文字列グリッドを使います。

描画グリッド

描画グリッド（TDrawGrid）は，任意のデータを表形式で表示します。グリッド内のセルを描画するには，OnDrawCell イベントを記述します。

- CellRect メソッドはセルの画面座標を返し，MouseToCell メソッドは画面座標の位置にある行と列を返します。Selection プロパティは，現在選択されているセルの境界を指定します。
- TopRow プロパティは，現在グリッドの一番上にある行を指定します。LeftCol プロパティは，一番左に表示されている列を指定します。VisibleColCount プロパティはグリッド内に表示する列の数を示し，VisibleRowCount プロパティはグリッド内に表示する行の数を示します。
- 列の幅と行の高さを変更するには，ColWidths プロパティと RowHeights プロパティを使います。グリッド線の幅を設定するには GridLineWidth プロパティを使います。スクロールバーをグリッドに追加するには ScrollBars プロパティを使います。
- 行と列を固定またはスクロール不可にするには，FixedCols プロパティと FixedRows プロパティを使います。固定行と固定列に色を指定するには，FixedColor プロパティを使います。
- Options プロパティ，DefaultColWidth プロパティ，DefaultRowHeight プロパティもグリッドの外観と動作に影響します。

文字列グリッド

文字列グリッドは TDrawGrid の下位コンポーネントで，文字列表示を簡易化する特殊機能を備えています。Cells プロパティはグリッドの各セルの文字列をリストし，Objects プロパティは各文字列に関連付けられたオブジェクトをリストします。各列の文字列と関連オブジェクトのリストにアクセス

するには，Cols プロパティを使います。各行の文字列とその関連オブジェクトのリストにアクセスするには，Rows プロパティを使います。

値リストエディタ（VCL のみ）

TValueListEditor は，名前 / 値のペアを「Name=Value」の形で格納する文字列リストを編集するための専用グリッドです。名前と値は TStrings の下位オブジェクトとして保存され，この下位オブジェクトが Strings プロパティの値を表します。名前の値を調べるには，Values プロパティを使います。TValueListEditor は，クロスプラットフォームプログラミングでは使用できません。

グリッドは，名前と値の 2 つの列で構成されます。デフォルトでは，名前の列は「Key」，値列は「Value」という名称になっています。このデフォルト名を変更するには，TitleCaptions プロパティを設定します。これらの列タイトルを省略するには，DisplayOptions プロパティを使います（コントロールのサイズが変更されたときも DisplayOptions プロパティで調整します）。

ユーザーが名前列を編集するのを可能または不可にするには，KeyOptions プロパティを使います。KeyOptions プロパティには複数のオプションがあり，名前の変更，追加，削除，新しい名前の重複について，それぞれ可能 / 不可を設定します。

値列の入力内容をユーザーがどのように変更できるかを指定するには，ItemProps プロパティを使います。各項目にそれぞれ TItemProp オブジェクトがあり，以下の設定ができます。

- 編集マスクにより，有効な入力を制限する
- 値の最大長を指定する
- 値を読み出し専用にする
- 値リストエディタにドロップダウン矢印または省略記号ボタンを表示する。矢印の場合，矢印から選択リストが開いて，ユーザーに値を選択させる。省略記号ボタンの場合，ユーザーが値を入力するダイアログを表示するイベントをトリガーする

  ドロップダウン矢印を表示する場合は，ユーザーが選択する値のリストを用意しておく必要があります。静的リストにすることも（TItemProp オブジェクトの PickList プロパティ），実行時に OnGetPickList イベントにより動的に追加されるリストにすることもできます。さらに，これらの方法を組み合わせて，OnGetPickList イベントハンドラにより変更される静的リストにすることもできます。

  省略記号ボタンを表示させる場合は，ユーザーが省略記号ボタンをクリックしたときに発生する応答を用意しておく必要があります（適宜，値の設定も行う）。応答を用意するには，OnEditButtonClick イベントハンドラを作成します。

グラフィックコントロール

以下のコンポーネントを使うと，グラフィックを簡単にアプリケーションに挿入できます。

コンポーネント	表示内容
TImage	グラフィックファイル
TShape	幾何学図形
TBevel	3 次元の線と枠
TPaintBox	実行時にプログラムが描画するグラフィックス
TAnimate	AVI ファイル（VCL のみ）

これらのコントロールには共通のペイントルーチン（Repaint，Invalidate など）が含まれていますが，これらのルーチンはフォーカスを受け取る必要がありません。

イメージ

イメージコンポーネントを使うと，ビットマップ，アイコン，メタファイルなどのグラフィックイメージを表示できます。Picture プロパティは，表示するグラフィックスを指定します。表示オプションを設定するには，Center プロパティ，AutoSize プロパティ，Stretch プロパティ，および Transparent プロパティを使います。詳細については，10-1 ページの「グラフィックプログラミングの概要」を参照してください。

図形

図形（Shape）コンポーネントは幾何学図形を表示します。図形コンポーネントは非ウィンドウコントロール（CLX では非ウィジェットベース）なので，ユーザーの入力を受け取ることができません。Shape プロパティは，表示する図形を指定します。図形の色を変えたりパターンを追加したりするには，Brush プロパティを使います。Brush プロパティは TBrush オブジェクトを保持しています。ペイント方法は，TBrush オブジェクトの Color プロパティと Style プロパティで設定します。

ベベル

ベベルコンポーネント（TBevel）を使うと，3 次元効果を持つ線（へこんだように見える線と飛び出たように見える線）を表示できます。一部のコンポーネント（TPanel など）には，ベベル線を作成するプロパティがあります。ベベル線を作成するプロパティがない場合は，TBevel を使ってベベル線，ベベルボックス，ベベル枠を作成します。

ペイントボックス

ペイントボックスコンポーネント（TPaintBox）を使うと，アプリケーションに対してフォーム上に線描させることができます。ペイントボックスの Canvas に直接イメージを描画するには，OnPaint イ

ベントハンドラを記述します。ペイントボックスの外側には描画できません。詳細については，10-1 ページの「グラフィックプログラミングの概要」を参照してください。

アニメーションコントロール（VCL のみ）

アニメーションコンポーネントは，AVI（Audio Video Interleaved）クリップを音声なしで表示するウィンドウです。AVI クリップは，映画のような連続したビットマップフレームです。AVI クリップには音声も入れられますが，このコンポーネントでは音声のない AVI クリップしか扱えません。使用するファイルは，圧縮されていない AVI ファイルか，RLE（Run-Length Encoding）で圧縮された AVI クリップでなければなりません。アニメーションコントロールは，クロスプラットフォームプログラミングでは使用できません。

アニメーションコンポーネントには以下のようなプロパティがあります。

- ResHandle プロパティは，AVI クリップをリソースとして保持するモジュールの Windows ハンドル。実行時に，アニメーションリソースを含むモジュールのインスタンスハンドルまたはモジュールハンドルを ResHandle プロパティで設定する。ResHandle プロパティを設定したら，ResID プロパティまたは ResName プロパティを設定して，モジュール内のどのリソースをアニメーションコントロールで表示するかを指定する
- AutoSize プロパティを **true** に設定すると，アニメーションコントロールのサイズが AVI クリップのフレームのサイズに調整される
- StartFrame プロパティおよび StopFrame プロパティは，クリップの開始フレームと終了フレームを指定する
- CommonAVI プロパティを設定すると，Shell32.DLL で提供される Windows のコモン AVI クリップの 1 つを表示できる
- アニメーションの開始と中断は，Active プロパティで指定する プロパティを **true** または **false** に設定することにより指定する。再生する回数は Repetitions プロパティで指定する
- Timers プロパティを使うと，タイマーを使ってフレームを表示できる。これは，音声トラックの再生などのほかのアクションをアニメーションシーケンスに同期させるときに有用

第10章

グラフィックとマルチメディアの処理

グラフィックとマルチメディアの要素を取り入れて，アプリケーションをグレードアップすることができます。C++Builder は，お使いのアプリケーションにグラフィックとマルチメディアの機能を取り入れるさまざまな方法を提供しています。グラフィック要素を取り入れるには，設計時に描画済みのグラフィックを挿入する，設計時にグラフィックコントロールで作成する，実行時に動的に描画するなどの方法があります。C++Builder には，マルチメディア機能を取り入れるために，オーディオクリップとビデオクリップを再生する特別なコンポーネントが用意されています。

CLX　マルチメディアコンポーネントは，VCL でのみ使用できます。

グラフィックプログラミングの概要

Graphics ユニットに定義されている VCL グラフィックコンポーネントは GDI（Windows Graphics Device Interface）をカプセル化しているため，とても簡単に Windows アプリケーションにグラフィックを加えることができます。QGraphics ユニットに定義されている CLX グラフィックコンポーネントは Qt グラフィックをカプセル化しているので，クロスプラットフォームアプリケーションにグラフィックを加えることができます。

C++Builder で作成したアプリケーションでのグラフィックの描画においては，オブジェクトに直接描画するのではなく，オブジェクトの「キャンバス」に描画します。キャンバスはオブジェクトのプロパティで，同時にそれ自身がひとつのオブジェクトでもあります。キャンバスオブジェクトを使用する主な利点は，資源の有効利用ができることと，デバイスコンテキストについて自動的に考慮がなされることです。そのため，画面，プリンタ，ビットマップ，メタファイル（CLX ではドローイング）のいずれに対して描画を行う場合でも，同じメソッドを使うことができます。キャンバスを使用できるのは実行時だけです。したがって，キャンバスで行う作業のすべてはコードで記述します。

VCL　TCanvas は Windows デバイスコンテキストのラッパーリソースマネージャなので，キャンバス上であらゆる Windows GDI 関数を使うことができます。キャンバスの Handle プロパティは，デバイスコンテキストの Handle です。

CLX　TCanvas は Qt ペインタのラッパーリソースマネージャです。キャンバスの Handle プロパティは，Qt オブジェクトのインスタンスを指す型付きポインタです。これをエクスポーズ（公開）することにより，QPainterH を必要とする低レベルの Qt グラフィックライブラリ関数を使用できます。

アプリケーション内でグラフィックがどう表示されるかは，描画するキャンバスがどのオブジェクト型かによって異なります。コントロールの Canvas プロパティに直接描画すると，画像が即座に表示されます。TBitmap のようなメモリ中の Canvas プロパティに描画した場合，コントロールがビットマップから Canvas にコピーするまでは，イメージは表示されません。また，メモリ中で描画したものをコントロールにコピーしても，コントロールが OnPaint メッセージ（VCL）または OnPaint イベント（CLX）を受け取るまでは画面には表示されません。

グラフィック処理のプログラミングをするとき「描画」および「ペイント」という言葉をしばしば使います。

- 描画とは，線や図形などの個々のグラフィック要素をコードで作成することです。キャンバスの描画メソッドを呼び出して，指定のグラフィックを指定の場所に描画することをコードでオブジェクトに指示します。
- ペイントとは，オブジェクト全体の外観を作成することです。通常，ペイントには描画が伴います。つまり，OnPaint イベントへの応答では一般にオブジェクトは何らかのグラフィックを描画します。たとえば編集ボックスは，長方形を描画し，さらに長方形内にテキストを描画することで編集ボックス全体をペイントします。一方，図形コントロールは，単一のグラフィックを描画することで図形コントロール全体をペイントします。

この章の最初の例ではさまざまなグラフィックを描画する方法を示していますが，すべてを OnPaint イベントへの応答の中で描画しています。後半の例では，ほかのイベントに応答して描画する方法を示します。

画面の更新

オペレーティングシステムは，あるタイミングで画面上のオブジェクトの再表示を行う必要があると判断すると，Windows 上で WM_PAINT メッセージを発行します。VCL はこのメッセージを OnPaint イベントにルーティングします（CLX を使ってクロスプラットフォーム開発を行っている場合は，ペイントイベントが生成され，CLX はこのペイントイベントを OnPaint イベントにルーティングします）。そのオブジェクトの OnPaint イベントハンドラを作成した場合は，Refresh メソッドを使用すると OnPaint イベントハンドラが呼び出されます。フォーム内の OnPaint イベントハンドラ用に生成されるデフォルト名は，FormPaint です。Refresh メソッドを，プログラマが明示的に使用してフォームやコンポーネントを再描画することもできます。たとえば，フォームの OnResize イベントハンドラ中で Refresh を呼び出すと，任意のグラフィックを再描画できます。また，VCL を使用している場合は，フォームの背景をペイントすることもできます。

いくつかのオペレーティングシステムは，無効になっているウィンドウのクライアント領域の再描画を自動的に処理しますが，Windows はこれを行いません。Windows オペレーティングシステムでは，

画面に描画されたものすべてが一時的なものです。フォームまたはコントロールが一時的に覆い隠されている場合，たとえばウィンドウをドラッグしている場合など，それを再表示するときには，フォームまたはコントロールが隠されている領域を再ペイントしなければなりません。WM_PAINT メッセージについての詳細は，Windows のヘルプを参照してください。

TImage コントロールを使ってグラフィックイメージを表示する場合，TImage に含まれているグラフィックのペイントと更新は自動的に処理されます。TImage が実際にどのグラフィック（ビットマップ，ドローイング，または別のグラフィックオブジェクト）を表示するかを指定するプロパティは，Picture プロパティです。また，Proportional プロパティを設定すると，イメージを変形せずに完全にイメージコントロール内に表示できます。TImage 上で描画すると，永続的なイメージが作成されます。したがって，その中のイメージを再描画するための操作は何も必要ありません。それに対して，TPaintBox のキャンバスはスクリーンデバイス（VCL）またはペインタ（CLX）に直接マップされるので，PaintBox のキャンバスに描画されたすべてのものが一時的なものです。これは，フォーム自身も含め，コントロールのほとんどに当てはまります。したがって，TPaintBox 上にコンストラクタで描画またはペイントする場合，クライアント領域が無効になるたびにイメージが再ペイントされるように OnPaint イベントハンドラにコードを追加する必要があります。

グラフィックオブジェクトの型

VCL/CLX は，表 10.1 に示すグラフィックオブジェクトを提供しています。これらのオブジェクトには，キャンバス上に描画するメソッドがあります。そのメソッドについては 10-9 ページの「キャンバスのメソッドを使ってグラフィックオブジェクトを描画する」で説明しています。また，10-18 ページの「グラフィックファイルのロードと保存」で説明しているように，グラフィックオブジェクトはグラフィックファイルのロードと保存を行います。

表 10.1　グラフィックオブジェクトの型

オブジェクト型	説明
Picture	あらゆるグラフィックイメージを保存する。グラフィックファイルの形式を追加するには，Picture の Register メソッドを使用する。このオブジェクトは，イメージコントロール内へのイメージの表示など，任意のファイルを処理するときに使用する
Bitmap	イメージの作成，処理（拡大縮小，スクロール，回転，ペイント），ディスクへの保存を行うための強力なグラフィックオブジェクト。イメージそのものではなく，ハンドルをコピーするため，ビットマップの高速コピーが可能
Clipboard	アプリケーションから切り取り，コピーまたは貼り付けされるテキストやグラフィックに対するコンテナを表す。クリップボードに対して，適切な形式でのデータの保存や取り出し，参照カウントの処理，クリップボードのオープン / クローズ，クリップボード内のオブジェクトの形式の処理を行うことができる
Icon	アイコンファイル（.ICO ファイル）からロードした値を表す
Metafile（VCL のみ） Drawing（CLX のみ）	イメージの実際のビットマップピクセルを含むのではなく，イメージを構築するために必要な操作を記録するファイルを含む。メタファイルもドローイングも非常にスケーラブルで，イメージの詳細を失わず，かつビットマップよりもメモリを必要としない。これはプリンタなどの高解像度の装置に対して顕著である。ただし，メタファイルとドローイングはビットマップよりも描画速度が遅くなる。パフォーマンスよりも多様性や精度が重要な場合には，メタファイルかドローイングを使用する

キャンバスの共通プロパティおよびメソッド

表 10.2 に，共通に使用される Canvas オブジェクトのプロパティをリストします。プロパティとメソッドの完全なリストは，オンラインヘルプの TCanvas コンポーネントを参照してください。

表 10.2　Canvas オブジェクトの共通プロパティ

プロパティ	説明
Font	イメージにテキストを書くときの字体を指定。TFont オブジェクトのプロパティを設定することによって，書体，色，サイズ，スタイルを指定
Brush	グラフィック図形と背景の塗りつぶしにキャンバスが使う色と模様を指定。TBrush オブジェクトのプロパティを設定して，キャンバスの空間を塗りつぶすとき使用する色と模様，あるいはビットマップを指定する
Pen	キャンバス上での線の描画および図形の輪郭に使用するペンの種類を指定。TPen オブジェクトを使ってペンの色，スタイル，幅およびモードを指定
PenPos	ペンの現在の描画位置を指定
Pixels	現在の ClipRect 内のピクセルの色を指定

これらのプロパティについての詳細は，10-5 ページの「キャンバスオブジェクトのプロパティの使い方」を参照してください。

表 10.3 に使用可能なメソッドを示します。

表 10.3　Canvas オブジェクトの共通メソッド

メソッド	説明
Arc	指定された長方形に内接する楕円に沿って弧を描画
Chord	直線と楕円の交わりで表現される図形を描画
CopyRect	別のキャンバスのイメージの一部分をキャンバスにコピー
Draw	キャンバス上の点 (X, Y) に Graphic パラメータで指定されたグラフィックを描画
Ellipse	キャンバスに指定された長方形に内接する楕円を描画する
FillRect	現在のブラシを使ってキャンバスの指定された長方形の内部を塗りつぶす
FloodFill（VCL のみ）	現在のブラシを使ってキャンバスのある領域を塗りつぶす
FrameRect	キャンバスの Brush を使って長方形の境界を描画
LineTo	キャンバス上の PenPos から点 (X, Y) まで線分を描画し，点 (X, Y) に PenPos を移動する
MoveTo	現在の描画位置を点 (X, Y) に移動する
Pie	キャンバス上の点 (X1, Y1) を左上隅とし，(X2, Y2) を右下隅とする長方形に内接する楕円のパイ型の扇型領域を描画する
Polygon	キャンバス上の渡された点群を線分でつないでいき，最初の点と最後の点も結んで閉じた図形を描画する
Polyline	キャンバス上に現在のペンを使って Points オブジェクトで渡された点群を線分でつなぐ
Rectangle	キャンバス上の点 (X1, Y1) を左上隅とし，(X2, Y2) を右下隅とする長方形を描画。Rectangle を使うと，Pen で長方形の輪郭を描画し，Brush でそれを塗りつぶす
RoundRect	キャンバス上に角が丸い長方形を描画する
StretchDraw	キャンバス上の指定された長方形に合うようにグラフィックを描画する。この操作の結果，グラフィックイメージの大きさや縦横比が変更する場合がある

表 10.3 Canvas オブジェクトの共通メソッド（つづき）

メソッド	説明
TextHeight, TextWidth	現在のフォントの文字列での高さと幅をそれぞれ返す。高さは線の間の枠も含む
TextOut	キャンバス上の点 (X, Y) から文字列を書き，PenPos を文字列の終端に移動する
TextRect	文字列をある領域の中に書く。領域からはみ出した文字列部分は表示されない

これらのメソッドについての詳細は，10-9 ページの「キャンバスのメソッドを使ってグラフィックオブジェクトを描画する」を参照してください。

キャンバスオブジェクトのプロパティの使い方

キャンバスオブジェクトを使うと，線を描画するためのペン，空間を塗りつぶすためのブラシ，テキストを表示するためのフォント，イメージを表すピクセルの配列のプロパティを設定できます。

このセクションでは，以下について説明します。

- ペンの使い方
- ブラシの使い方
- ピクセルの読み出しと設定

ペンの使い方

キャンバスの Pen プロパティでは，線の表示方法を指定します。図形の輪郭線も線の一種です。実際には，直線を描画することは 2 点間を結ぶピクセル群を変更することと同じです。

ペンには，変更可能な 4 つのプロパティがあります。

- Color プロパティ：ペンの色の変更
- Width プロパティ：ペン幅の変更
- Style プロパティ：ペンスタイルの変更
- Mode プロパティ：ペンモードの変更

これらのプロパティの値は，線上のピクセルをどのように変更するかを指定します。デフォルトでは，ペンの色（Color）は黒，幅（Width）は 1 ピクセル，スタイル（Style）は実線，モード（Mode）はキャンバスにあるものすべてを上書きするコピーモードです。

TPenRecall を使うと，ただちに保存を中止してペンのプロパティを復元できます。

ペンの色の変更

ペンの色を設定する方法は，ほかのオブジェクトで Color プロパティを実行時に設定する方法と同じです。ペンの色はペンで描画する線の色を決定します。図形の境界として描画される線や多角線の色も，ペンの色で決定されます。ペンの色を変更するには，ペンの Color プロパティに値を代入します。

ユーザーに色を選ばせるには TColorDialog コンポーネントを使います。Execute メソッドを呼び出すと，標準の色選択ダイアログが現れます。TColorDialog コンポーネントをフォーム上の適当な位置に配置してください。次に，色選択ダイアログを表示させるために，フォーム上のペンのツールバーにスピードボタンを追加します。

スピードボタンがクリックされたら色選択ダイアログを表示させます。OnClick イベントに以下のコードを記述してください。

```
void __fastcall TForm1::PenColorClick(TObject *Sender)
{
  if (ColorDialog1->Execute())
    Canvas->Pen->Color = ColorDialog1->Color;
}
```

ペン幅の変更

ペン幅はペンで描画する線の太さをピクセル単位で指定します。

メモ 線幅が 1 より大きい場合，Windows はペンの Style プロパティの値に関係なく，常に実線を描きます。

ペン幅を変更するには，ペンの Width プロパティに値を代入します。

フォーム上のペンのツールバーにペン幅を表すための編集ボックスを用意します。編集ボックスの脇に，値を簡単に増減させるためのアップダウンコントロールを配置します。アップダウンコントロールの Associate プロパティにはペンの幅のための編集ボックスを指定します。こうすることで編集ボックスの値とアップダウンコントロールの値が関連付けられ，編集ボックスの文字列が常にアップダウンコントロールの値を表すようになります。

編集ボックスで表示される文字列が変更された場合，OnChange イベントが発生します。このイベントハンドラでペンの太さをアップダウンコントロールの Position プロパティの値に設定します。

```
void __fastcall TForm1::PenWidthChange(TObject *Sender)
{
  Canvas->Pen->Width = PenWidth->Position; // ペン幅はアップダウンコントロールから直接取得する
  PenSize->Caption = IntToStr(PenWidth->Position); // 文字列に変換する
}
```

ペンスタイルの変更

ペンの Style プロパティを使うと，実線，破線，点線などを指定できます。

VCL **メモ** Windows で配布されるクロスプラットフォームアプリケーションの場合，Windows は，1 ピクセルを超える線幅の破線や点線をサポートせず，このような線はスタイルの設定に関わらずすべて実線になります。

ペンのプロパティ設定は，異なるコントロールが同じイベントハンドラを使ってイベントを処理する事例の典型的な例です。実際にどのコントロールがイベントを受け取ったのかを確認するには，Sender パラメータをチェックします。

ツールバーにある 6 つのボタンに対応した共通のクリックイベントハンドラを作成するには，次のようにします。

1. 6 つのペンスタイルボタンをすべて選択して，オブジェクトインスペクタの［イベント］ページで OnClick イベントを選んだ後，ハンドラ列に「SetPenStyle」を入力します。

   C++Builder は，SetPenStyle という空のクリックイベントハンドラを生成し，6 種類すべてのボタンの OnClick イベントに結び付けます。

2. Sender の値に応じてペンのスタイルを設定するコードを，クリックイベントハンドラ内に記述します。Sender はクリックイベントを送ったコントロールを示しています。

```
void __fastcall TForm1::SetPenStyle(TObject *Sender)
{
  if (Sender == SolidPen)
    Canvas->Pen->Style = psSolid;
  else if (Sender == DashPen)
    Canvas->Pen->Style = psDash;
  else if (Sender == DotPen)
    Canvas->Pen->Style = psDot;
  else if (Sender == DashDotPen)
    Canvas->Pen->Style = psDashDot;
  else if (Sender == DashDotDotPen)
    Canvas->Pen->Style = psDashDotDot;
  else if (Sender == ClearPen)
    Canvas->Pen->Style = psClear;
}
```

ペンスタイル定数をペンスタイルボタンの Tag プロパティに組み込むことにより，上記のイベントハンドラコードをさらにかなり減らすことができます。このとき，このイベントコードは以下のようになります。

```
void __fastcall TForm1::SetPenStyle(TObject *Sender)
{
  if (Sender->InheritsFrom (__classid(TSpeedButton))
    Canvas->Pen->Style = (TPenStyle) ((TSpeedButton *)Sender)->Tag;
}
```

ペンモードの変更

ペンの Mode プロパティを使用すると，ペンの色とキャンバス上の色を組み合わせるためのさまざまな方法を指定できます。たとえば，ペンを常に黒にする，キャンバスの背景色と逆の色にする，ペンの色と逆の色にするなどの指定が可能です。詳しくは，オンラインヘルプで TPen の説明を参照してください。

ペン位置の取得

現在の描画位置（ペンを使って次の図形を描画するときの開始位置）を，ペン位置と呼びます。キャンバスは，ペン位置を PenPos プロパティ内に保存します。ペン位置が影響するのは線の描画だけであり，図形とテキストの描画では任意の座標を指定できます。

ペン位置を設定するには，キャンバスの MoveTo メソッドを呼び出します。たとえば，ペン位置をキャンバスの左上隅に移動するコードは次のようになります。

```
Canvas->MoveTo(0, 0);
```

メモ LineTo メソッドで線を描画すると，ペン位置は線の終端に移動します。

ブラシの使い方

キャンバスの Brush プロパティでは，領域を塗りつぶす方法を指定します。図形の内部を領域とすることもできます。ブラシで領域を塗りつぶすことは，隣接ピクセル群を特定の方法で変更することです。

ブラシには，Color，Style，Bitmap という変更可能な 3 種類のプロパティがあります。

- Color プロパティ：塗りつぶし色の変更
- Style プロパティ：ブラシスタイルの変更
- Bitmap プロパティ：ビットマップをブラシパターンとして使用

各プロパティの値は，キャンバスで図形などの領域をどのように塗りつぶすかを指定します。デフォルトのブラシの色（Color）は白，スタイル（Style）は実線，ビットマップ（Bitmap）はパターンなしです。

TBrushRecall を使うと，ただちに保存を中止してブラシのプロパティを復元できます。

ブラシ色の変更

ブラシ色はキャンバスで図形を塗りつぶすための色を決定します。塗りつぶしの色を変更するには，ブラシの Color プロパティに値を代入します。ブラシの色は，テキストや線の背景色としても使用されるため，背景色を設定するときにも使用します。

ブラシの色は，ペンの色と同じ方法で設定できます。ブラシの色は，ペンの色と同じ方法で設定できます。フォーム上のブラシのツールバーに配置したスピードボタンをクリックした場合の動作を記述します（10-5 ページの「ペンの色の変更」を参照してください）。

```
void __fastcall TForm1::BrushColorClick(TObject *Sender)
{
  Canvas->Brush->Color = BrushColor->BackgroundColor;
}
```

ブラシスタイルの変更

ブラシスタイルでは，キャンバスで図形を塗りつぶすためのパターンを指定します。このスタイルを使って，キャンバス上にすでに配置されている色とブラシの色を組み合わせるためのさまざまな方法が指定できます。定義済みのスタイルとして，塗りつぶし，無色，各種の線パターンと格子パターンがあります。

ブラシのスタイルを変更するには，Style プロパティをあらかじめ定義されている値（bsSolid, bsClear, bsHorizontal, bsVertical, bsFDiagonal, bsBDiagonal, bsCross, bsDiagCross, bsDense1, bsDense2, bsDense3, bsDense4, bsDense5, bsDense6, bsDense7）のいずれかに設定します。

次の例では，8 つのブラシスタイルボタンに対して，共通のクリックイベントハンドラを共有してスタイルを設定しています。15 個のボタンをすべて選択して，オブジェクトインスペクタの［イベント］ページで OnClick を選び，OnClick ハンドラの名前を「SetBrushStyle」に設定します。ハンドラのコードを次に示します。

```
void __fastcall TForm1::SetBrushStyle(TObject *Sender)
{
  if (Sender == SolidBrush)
    Canvas->Brush->Style = bsSolid;
  else if (Sender == ClearBrush)
    Canvas->Brush->Style = bsClear;
  else if (Sender == HorizontalBrush)
    Canvas->Brush->Style = bsHorizontal;
  else if (Sender == VerticalBrush)
    Canvas->Brush->Style = bsVertical;
  else if (Sender == FDiagonalBrush)
    Canvas->Brush->Style = bsFDiagonal;
  else if (Sender == BDiagonalBrush)
    Canvas->Brush->Style = bsBDiagonal;
  else if (Sender == CrossBrush)
    Canvas->Brush->Style = bsCross;
  else if (Sender == DiagCrossBrush)
```

```
    Canvas->Brush->Style = bsDiagCross;
}
```

ブラシスタイル定数をブラシスタイルボタンの Tag プロパティに組み込むことにより，上記のイベントハンドラコードをかなり減らすことができます。このとき，このイベントコードは以下のようになります。

```
void __fastcall TForm1::SetBrushStyle(TObject *Sender)
{
  if (Sender->InheritsFrom (__classid(TSpeedButton))
    Canvas->Brush->Style = (TBrushStyle) ((TSpeedButton *)Sender)->Tag;
}
```

ブラシの Bitmap プロパティの設定

ブラシの Bitmap プロパティを使うと，図形の内部などの領域を塗りつぶすためのパターンとして使用するビットマップイメージを設定できます。

以下の例では，ファイルからビットマップをロードし，Form1 の Canvas の Brush にそのビットマップを代入します。

```
BrushBmp->LoadFromFile("MyBitmap.bmp");
Form1->Canvas->Brush->Bitmap = BrushBmp;
Form1->Canvas->FillRect(Rect(0,0,100,100));
```

メモ ブラシは，Bitmap プロパティに割り当てられたビットマップオブジェクトの所有権を管理しません。Bitmap オブジェクトが Brush の存続期間中有効であることを確認して，後で Bitmap オブジェクトをユーザーが解放する必要があります。

ピクセルの読み出しと設定

すべてのキャンバスには，Pixels というインデックス付きのプロパティがあり，このプロパティによってキャンバス上のイメージを構成する個々の色付きの点を表します。Pixels プロパティへの直接アクセスが必要になることはほとんどありません。ピクセルの色の検索や設定のような小さな動作が必要になったときの便宜を考えて用意されているものです。

メモ ピクセルの直接操作は，通常のグラフィックス操作に比べて数千倍も時間がかかります。Pixels プロパティを一般的な配列のように使うことは避けてください。画像情報を直接扱う場合は TBitmap::ScanLine プロパティを使うと性能を向上できます。

キャンバスのメソッドを使ってグラフィックオブジェクトを描画する

このセクションでは，グラフィックオブジェクトを描画するためのいくつかの一般的なメソッドについて説明します。以下の項目について説明します。

- 直線と多角線の描画
- 図形の描画
- 角丸四角形の描画
- 多角形の描画

直線と多角線の描画

キャンバスには，直線と多角線を描画できます。直線は単に 2 点間を結ぶピクセルの線です。多角線は，複数の直線が接続されて，端点が接続された一連の直線です。キャンバスではすべての線をペンで描画します。

線の描画

キャンバス上に直線を描くには，キャンバスの LineTo メソッドを使用します。

LineTo では現在のペン位置から指定した点まで線を描画し，線の終端を現在のペン位置にします。キャンバスでは直線をペンで描画します。

たとえば，フォームを表示するときフォーム全体に交差する斜線を描画するメソッドは次のようになります。

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  Canvas->MoveTo(0,0);
  Canvas->LineTo(ClientWidth, ClientHeight);
  Canvas->MoveTo(0, ClientHeight);
  Canvas->LineTo(ClientWidth, 0);
}
```

多角線の描画

キャンバスでは，個別の線だけでなく，多角線も描画できます。多角線とは，複数の線分をつなげたものです。

キャンバス上に多角線を描くには，キャンバスの PolyLine メソッドを呼び出します。

PolyLine メソッドにはパラメータとして点の配列を渡します。多角線は，最初の点に対して MoveTo を実行し，後続の各点に対して LineTo を実行するものと考えることもできます。多くの線を描画するには，MoveTo メソッドと LineTo メソッドを使用するよりも，Polyline を使った方が呼び出しのオーバーヘッドが少なくなるので，処理時間が短縮できます。

たとえば，フォームに平行四辺形を描画するメソッドは次のとおりです。

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  TPoint vertices[5];
  vertices[0] = Point(0, 0);
  vertices[1] = Point(50, 0);
  vertices[2] = Point(75, 50);
  vertices[3] = Point(25, 50);
  vertices[4] = Point(0, 0);
  Canvas->Polyline(vertices, 4);
}
```

Polyline に対する最後のパラメータは，最後のポイントに対するインデックスであり，ポイントの数ではないので注意してください。

図形の描画

キャンバスには，図形を描画するメソッドがあります。キャンバスでは図形の輪郭線をペンで描画し，図形内部をブラシで塗りつぶします。図形の輪郭を構成する線は，現在の Pen オブジェクトによって制御されます。

このセクションでは，以下について説明します。

- 四角形と楕円の描画
- 角丸四角形の描画
- 多角形の描画

四角形と楕円の描画

キャンバス上に四角形や楕円を描くには，キャンバスの Rectangle メソッドまたは Ellipse メソッドを呼び出し，境界長方形の座標を渡します。

Rectangle メソッドは境界長方形をそのまま描画し，Ellipse メソッドは境界長方形の 4 辺に内接する楕円を描画します。

次のメソッドは，フォームの左上 1/4 の領域に四角形を描き，同じ領域に楕円を描きます。

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  Canvas->Rectangle(0, 0, ClientWidth/2, ClientHeight/2);
  Canvas->Ellipse(0, 0, ClientWidth/2, ClientHeight/2);
}
```

角丸四角形の描画

キャンバス上に角が丸い四角形を描くには，キャンバスの RoundRect メソッドを呼び出します。

RoundRect に渡す最初の 4 つのパラメータは，Rectangle メソッドや Ellipse メソッドのときと同様に境界長方形です。RoundRect では，丸い角の描画方法を指示する 2 つのパラメータも指定します。

たとえば，フォームの左上半分に角の丸い四角形を描画し，角は直径 10 ピクセルの円弧とするメソッドは次のとおりです。

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  Canvas->RoundRect(0, 0, ClientWidth/2, ClientHeight/2, 10, 10);
}
```

多角形の描画

キャンバスに任意の数の辺を持った多角形を描くには，キャンバスの Polygon メソッドを呼び出します。

Polygon は点の配列だけをパラメータとして受け取り，各点をペンでつなげます。さらに最後の点を最初の点に接続して，閉じた多角形を作ります。Polygon は多角形の各線を描画した後で，内部の領域をブラシで塗りつぶします。

たとえば，フォームの左下半分に直角三角形を描画するメソッドは次のとおりです。

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  TPoint vertices[3];
  vertices[0] = Point(0, 0);
  vertices[1] = Point(0, ClientHeight);
  vertices[2] = Point(ClientWidth,ClientHeight);
  Canvas->Polygon(vertices,2);
}
```

アプリケーション内の複数の描画オブジェクトの処理

多様な描画機能（長方形，形，線，など）は，多くのアプリケーションではツールバーとボタンパネルで選択できるようになっています。こうしたアプリケーションでは，スピードボタンのクリックに応答して，目的の描画オブジェクトを設定できます。ここでは次のことについて説明します。

- 使用する描画ツールの追跡
- スピードボタンによるツールの変更
- 描画ツールの使い方

使用する描画ツールの追跡

グラフィックプログラムでは，ある時点でユーザーがどの描画ツール（線，四角形，楕円，角丸四角形など）を使用しているのかを検出し，追跡する必要があります。通常は，使用可能なツールのリストを返すには C++ の列挙型を使用します。列挙型は型宣言でもあるので，C++ の型チェックを使って，確実に適切な値だけを割り当てられます。

たとえば，次のコードは，グラフィックアプリケーション内で使用可能な描画ツールに対する列挙型を宣言します。

```
typedef enum {dtLine, dtRectangle, dtEllipse, dtRoundRect} TDrawingTool;
```

TDrawingTool 型の変数には，定数 dtLine，dtRectangle，dtEllipse，または dtRoundRect の 1 つしか割り当てられません。

慣例として型識別子は文字 T で始まります。また，1 つの列挙型に属する各定数の名前は，先頭に同一の 2 文字を付加します。たとえば，描画ツール（drawing tool）では dt を付けることにします。

次のように，描画ツールを追跡するためのフィールドをフォームに追加します。

```
enum TDrawingTool {dtLine, dtRectangle, dtEllipse, dtRoundRect};
class TForm1 : public TForm
{
__published:  // IDE 管理コンポーネント
  void __fastcall FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y);
  void __fastcall FormMouseMove(TObject *Sender, TShiftState Shift, int X, int Y);
  void __fastcall FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y);
private:  // ユーザー宣言
public:   // ユーザー宣言
  __fastcall TForm1(TComponent* Owner);
  bool Drawing;  // ボタンが押されたかどうかを追跡するフィールド
  TPoint Origin, MovePt;  // ポイントを保存するフィールド
  TDrawingTool DrawingTool;  // 現在のツールを保持するフィールド
};
```

スピードボタンによるツールの変更

各描画ツールには対応する OnClick イベントハンドラが必要です。アプリケーションに次の 4 つの描画ツール（線，四角形，楕円，および角丸四角形）それぞれに対してツールバーボタンがあったとします。ツールバーの 4 つの描画ツールボタンの OnClick イベントに対して，次のイベントハンドラを記述し，各 DrawingTool に適切な値を設定します。

```
void __fastcall TForm1::LineButtonClick(TObject *Sender)  // LineButton
{
  DrawingTool = dtLine;
}
void __fastcall TForm1::RectangleButtonClick(TObject *Sender) // RectangleButton
{
  DrawingTool = dtRectangle;
}
void __fastcall TForm1::EllipseButtonClick(TObject *Sender)  // EllipseButton
{
  DrawingTool = dtEllipse;
}
void __fastcall TForm1::RoundedRectButtonClick(TObject *Sender) // RoundRectBtn
{
  DrawingTool = dtRoundRect;
}
```

描画ツールの使い方

これでどのツールを使うのかは明らかになりますが，次に，さまざまな図形をどのように描くのかを指定しなければなりません。描画を行うメソッドは，マウスムーブハンドラとマウスアップハンドラです。

さまざまな描画ツールを使用するには，ユーザーが指定したツールに対応した描画方法を選択するように，指定する必要があります。この命令を，各ツールのイベントハンドラに追加します。

このセクションでは，以下について説明します。

- 図形の描画
- イベントハンドラ間でのコードの共有

図形の描画

図形の描画は，直線の描画と同じように簡単です。図形ごとに 1 つの文を記述するだけです。すべての必要な座標はすでに取得済みとします。

OnMouseDown および OnMouseUp イベントハンドラを修正して，図形を描画できるようにします。

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Origin  = Point(X, Y);
  MovePt  = Origin;
  Drawing = true;
}
void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  switch (DrawingTool)
  {
    case dtLine:
      Canvas->MoveTo(Origin.x, Origin.y);
      Canvas->LineTo(X, Y);
      break;
    case dtRectangle:
      Canvas->Rectangle(Origin.x, Origin.y, X, Y);
```

```
        break;
      case dtEllipse:
        Canvas->Ellipse(Origin.x, Origin.y, X, Y);
        break;
      case dtRoundRect:
        Canvas->Rectangle(Origin.x, Origin.y, X, Y, (Origin.x - X)/2,
                          (Origin.y - Y)/2);
        break;
    }
    Drawing = false;
  }
```

もちろん，図形を描画するには OnMouseMove ハンドラも更新する必要があります。

```
  void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
    TShiftState Shift, int X, int Y)
  {
    if (Drawing)
    {
      Canvas->Pen->Mode = pmNotXor;        // XOR モードで描画または消去する
      switch (DrawingTool)
      {
        case dtLine:
          Canvas->MoveTo(Origin.x, Origin.y);
          Canvas->LineTo(MovePt.x, MovePt.y);
          Canvas->MoveTo(Origin.x, Origin.y);
          Canvas->LineTo(X, Y);
          break;
        case dtRectangle:
          Canvas->Rectangle(Origin.x, Origin.y, MovePt.x, MovePt.y);
          Canvas->Rectangle(Origin.x, Origin.y, X, Y);
          break;
        case dtEllipse:
          Canvas->Ellipse(Origin.x, Origin.y, MovePt.x, MovePt.y);
          Canvas->Ellipse(Origin.x, Origin.y, X, Y);
          break;
        case dtRoundRect:
          Canvas->Rectangle(Origin.x, Origin.y, MovePt.x, MovePt.y,
                            (Origin.x - MovePt.x)/2,(Origin.y - MovePt.y)/2);
          Canvas->Rectangle(Origin.x, Origin.y, X, Y,
                            (Origin.x - X)/2, (Origin.y - Y)/2);
          break;
      }
      MovePt = Point(X, Y);
    }
    Canvas->Pen->Mode = pmCopy;
  }
```

一般に，上記の例に示されているようなコードは，個々のルーチン内に繰り返し記述することになります。次のセクションでは，すべてのマウスイベントハンドラの呼び出しを単一のルーチンにまとめた，すべての図形描画コードの例を紹介します。

イベントハンドラ間でのコードの共有

イベントハンドラ内では，しばしば別のハンドラと同じコードを使用します。こうした場合，繰り返されるコードをすべてのイベントハンドラが共有できるルーチンにまとめておくと，アプリケーションをさらに効率化できます。

フォームにメソッドを追加するには,

1. フォームオブジェクトにメソッド宣言を追加します。

   メソッド宣言は，フォームオブジェクトの宣言の最後にある **public** 部にも **private** 部にも追加できます。イベントの処理を部分的に共有するだけならば，メソッドは **private** 部に入れる方が安全です。

2. フォームのユニットの .cpp ファイルにメソッドの実装を記述します。

メソッドの実装は，宣言と正確に一致させなければなりません。パラメータも同じものを同じ順序で記述します。

マウスイベントハンドラから重複する図形描画コードを除去するために，DrawShape というメソッドをフォームに追加し，このメソッドをすべてのイベントハンドラから呼び出します。まず，フォームオブジェクトの宣言に，DrawShape の宣言を追加します。

```
enum TDrawingTool {dtLine, dtRectangle, dtEllipse, dtRoundRect};
class TForm1 : public TForm
{
__published:  // IDE 管理コンポーネント
  void __fastcall FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y);
  void __fastcall FormMouseMove(TObject *Sender, TShiftState Shift, int X, int Y);
  void __fastcall FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y);
private:  // ユーザー宣言
  void __fastcall DrawShape(TPoint TopLeft, TPoint BottomRight, TPenMode AMode);
public:   // ユーザー宣言
  __fastcall TForm1(TComponent* Owner);
  bool Drawing;  // ボタンが押されたかどうかを追跡するフィールド
  TPoint Origin, MovePt;  // ポイントを保存するフィールド
  TDrawingTool DrawingTool;  // 現在のツールを保持するフィールド
};
```

次に，ユニットの .cpp ファイルに DrawShape の実装を記述します。

```
void __fastcall TForm1::DrawShape(TPoint TopLeft, TPoint BottomRight, TPenMode AMode)
{
  Canvas->Pen->Mode = AMode;
  switch (DrawingTool)
  {
    case dtLine:
      Canvas->MoveTo(TopLeft.x, TopLeft.y);
      Canvas->LineTo(BottomRight.x, BottomRight.y);
      break;
    case dtRectangle:
      Canvas->Rectangle(TopLeft.x, TopLeft.y, BottomRight.x, BottomRight.y);
      break;
    case dtEllipse:
      Canvas->Ellipse(TopLeft.x, TopLeft.y, BottomRight.x, BottomRight.y);
      break;
    case dtRoundRect:
      Canvas->Rectangle(TopLeft.x, TopLeft.y, BottomRight.x, BottomRight.y,
                (TopLeft.x - BottomRight.x)/2,(TopLeft.y - BottomRight.y)/2);
      break;
  }
}
```

最後に，DrawShape を呼び出すように，ほかのイベントハンドラを変更します。

```
void __fastcall TForm1::FormMouseUp(TObject *Sender)
{
  DrawShape(Origin, Point(X,Y), pmCopy); // 確定した座標で図形を描画する
  Drawing = false;
}
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  if (Drawing)
  {
    DrawShape(Origin, MovePt, pmNotXor); // 前に描いた図形を消去する
    MovePt = Point(X, Y);
    DrawShape(Origin, MovePt, pmNotXor); // 現在の図形を描画する
  }
}
```

グラフィックへの描画

作成するアプリケーションのグラフィックオブジェクトを操作するために特別なコンポーネントは必要ありません。グラフィックオブジェクトは，画面への描画をまったく行わずに，作成，追加描画，保存，破棄できます。実際問題として，アプリケーションがフォームに直接描画することはほとんどありません。どちらかといえば，アプリケーションがグラフィックを処理した上で，イメージコントロールのコンポーネントを使ってフォーム上にグラフィックを表示します。

アプリケーションでの描画をイメージコントロール内のグラフィックオブジェクトに対して行うようにすると，印刷，クリップボード，およびグラフィックオブジェクトの操作のロードと保存が簡単に追加できます。このグラフィックオブジェクトは，ビットマップファイル，ドローイング，アイコン，あるいは JPEG グラフィックとしてインストールされた他のグラフィッククラスなどが使われます。

メモ TBitmap のようなメモリ中の Canvas プロパティに描画した場合，コントロールの Canvas にコピーされるまでは表示されることはありません。また，メモリ中で描画したものをコントロールにコピーしても，コントロールがペイントメッセージを受け取るまでは画面には表示されません。これに対して，コントロールの Canvas プロパティに対して直接描画した場合には，即座に表示されます。

スクロール可能なグラフィックの作成

グラフィックは，フォームと同じサイズにする必要はありません。フォームにスクロールボックスコントロールを追加し，その中にグラフィックイメージを配置すると，フォームよりも大きな，さらには画面よりも大きなグラフィックを表示できます。スクロール可能なグラフィックを追加するには，まず TScrollBox コンポーネントを追加し，次にイメージコントロールを追加します。

イメージコントロールの追加

イメージコントロールは，ビットマップオブジェクトを表示するためのコンテナコンポーネントです。イメージコントロールは，常時表示しておく必要のないビットマップや，アプリケーションがほかの画像を生成するために使用するビットマップを格納するためにも利用できます。

メモ 6-12 ページの「コントロールへのグラフィックの追加」に，コントロールでのグラフィックの使用方法が示されています。

コントロールの配置

イメージコントロールはフォームのどこにでも配置できます。画像に応じて自分自身のサイズを変更するイメージコントロールの機能を活用する場合には，AutoSize プロパティを **true** に設定してください。非表示のビットマップを保存するためだけにイメージコントロールを使用する場合は，イメージコントロールをどこに配置してもかまいません。これは，非ビジュアルコンポーネントの場合と同じです。

すでにフォームのクライアント領域に配置されているスクロールボックス上にイメージコントロールをドロップすると，スクロールボックスに，イメージ画像の画面外の部分へのアクセスに必要なスクロールバーを確実に追加できます。次に，イメージコントロールのプロパティを設定します。

ビットマップの初期サイズの設定

配置したイメージコントロールは，単純なコンテナとなります。ただし，設計時には，静的なグラフィックを格納するようにイメージコントロールの Picture プロパティを設定できます。コントロールは，実行時にファイルから画像を読み込むこともできます。10-18 ページの「グラフィックファイルのロードと保存」を参照してください。

アプリケーションの起動時に空のビットマップを作成するには，次のように処理します。

1. イメージを含むフォームの OnCreate イベントに対してハンドラを結び付けます。
2. ビットマップオブジェクトを作成し，このオブジェクトをイメージコントロールの Picture->Graphic プロパティに割り当てます。

このサンプルの場合，イメージはアプリケーションのメインフォーム Form1 にあるので，Form1 の OnCreate イベントにハンドラを結び付けます。

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Graphics::TBitmap *Bitmap = new Graphics::TBitmap();   // ビットマップオブジェクトを作成
  Bitmap->Width = 200;                // 初期状態の幅を代入し ...
  Bitmap->Height = 200;               // 初期状態の高さを代入
  Image->Picture->Graphic = Bitmap;   // イメージコントロールにビットマップを設定する
  delete Bitmap;                      // ビットマップオブジェクトを解放する
}
```

画像オブジェクトの Graphic プロパティにビットマップを設定すると，ビットマップのコピーが作成されます。ただし，画像オブジェクトはビットマップのオーナーにならないので，Graphic プロパティにビットマップを設定したあとは，ビットマップの解放はプログラマが行う必要があります。

この時点でアプリケーションを実行すると，フォームのクライアント領域にビットマップを示す白い領域が表示されます。もしウィンドウのサイズを変更して，クライアント領域がイメージ全体を表示できなくなったときは，スクロールボックスが自動的にスクロールバーを表示するので，これを使って残りのイメージを表示させることができます。このとき，イメージ上で描画しようとしてもグラフィックを描くことはできません。というのは，アプリケーションがまだフォーム上で描画しており，そのフォームがイメージとスクロールボックスの陰に隠れているからです。

ビットマップへの描画

ビットマップへの描画を行うには，イメージコントロールのキャンバスを使用し，そのイメージコントロールの適切なイベントにマウスイベントハンドラを接続します。一般的には領域操作（塗りつぶし，長方形，多角線など）を使用します。領域操作は，高速で効率的な描画メソッドです。

個々のピクセルにアクセスする必要があるようなイメージを描画するには，ビットマップの ScanLine プロパティを使用すると効率的です。用途を汎用化するために，ビットマップピクセル形式を 24 ビットに設定し，ScanLine から返されたポインタを RGB の配列として扱うことができます。これ以外の設定では，ScanLine プロパティの本来の形式がわかっていないと処理できません。次の例は，一度に 1 行分のピクセルを取得するときの ScanLine の使用方法を示しています。

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Graphics::TBitmap *pBitmap = new Graphics::TBitmap();
// これは，Bitmap への直接描画の例
  Byte *ptr;
  try
  {
    pBitmap->LoadFromFile(
    "C: \\ Program Files \\ Borland \\ CBuilder6 \\ Images \\ Splash \\ 256color \\ factory.bmp ");
    for (int y = 0; y < pBitmap->Height; y++)
    {
      ptr = pBitmap->ScanLine[y];
      for (int x = 0; x < pBitmap->Width; x++)
        ptr[x] = (Byte)y;
    }
    Canvas->Draw(0,0,pBitmap);
  }
  catch (...)
  {
    ShowMessage("Could not load or alter bitmap");
  }
  delete pBitmap;
}
```

CLX　クロスプラットフォームアプリケーションの場合，Linux で動作できるようにするには，Windows と VCL 固有のコードを変更する必要があります。たとえば Linux のパス名は，区切り文字としてスラッシュを使用します。CLX およびクロスプラットフォームアプリケーションに関する詳細は，第 14 章「クロスプラットフォームアプリケーションの開発」を参照してください。

グラフィックファイルのロードと保存

アプリケーションの実行中にだけ存在するようなグラフィックイメージは，あまり価値のあるものとはいえません。同じ画像を毎回使用したり，後で使用するために保存しなければならないことがしばしばあります。イメージコンポーネントを使用すると，ファイルからイメージをロードしたり，もう一度ファイルに保存し直すという処理を簡単に行うことができます。

グラフィックイメージのロード，保存および置換に使用するコンポーネントは，ビットマップファイル，メタファイル，グリフなどを含む多くのグラフィック形式をサポートしています。インストール可能なグラフィッククラスもサポートしています。

グラフィックファイルのロードと保存の方法は，ほかのファイルの場合と類似しています。以下の節で説明します。

- ファイルからの画像の読み込み
- ファイルへの画像の保存
- 画像の置換

ファイルからの画像の読み込み

作成中のアプリケーションを使って画像を修正する必要がある場合や，ほかのユーザーやアプリケーションが画像を修正できるように，アプリケーションの外に画像を保存する必要がある場合は，ファイルから画像をロードする機能をアプリケーションに追加しなければなりません。

イメージコントロールにグラフィックをロードするには，イメージコントロールの Picture オブジェクトのメソッド LoadFromFile を呼び出します。

次のコードは，ピクチャーファイルを開くためのダイアログボックスからファイル名を取得し，Image という名前のイメージコントロールにそのファイルをロードします。

```
void __fastcall TForm1::Open1Click(TObject *Sender)
{
  if (OpenPictureDialog1->Execute())
  {
    CurrentFile = OpenPictureDialog1->FileName;
    Image->Picture->LoadFromFile(CurrentFile);
  }
}
```

ファイルへの画像の保存

画像オブジェクトは，いくつかの形式でグラフィックをロードし，保存できます。また，グラフィックオブジェクトが独自の形式のグラフィックのロードや保存ができるように，独自のグラフィックファイル形式の作成や登録を行うこともできます。

イメージコントロールの内容をファイルに保存するには，そのイメージコントロールの Picture オブジェクトの SaveToFile メソッドを呼び出します。

SaveToFile メソッドには，保存先のファイル名を指定する必要があります。画像を新たに作成した場合，その画像にファイル名が設定されていないことがあります。また，ユーザーが既存の画像を別のファイルに保存したいと思うこともあるでしょう。どちらの場合も，アプリケーションは，次のセクションに記載されている方法で，保存する前にユーザーからファイル名を取得する必要があります。

以下の一対のイベントハンドラは，それぞれ，［ファイル｜上書き保存］と［ファイル｜名前を付けて保存］というメニュー項目に結び付けられています。これらのイベントハンドラは，すでに名前が付けられているファイルへの再保存，名前のないファイルの保存，既存のファイルの別名での保存を処理します。

```
void __fastcall TForm1::Save1Click(TObject *Sender)
{
  if (!CurrentFile.IsEmpty())
    Image->Picture->SaveToFile(CurrentFile);   // 名前が付けられている場合は保存する
else SaveAs1Click(Sender);                     // 名前がない場合は名前を取得
}
void __fastcall TForm1::SaveAs1Click(TObject *Sender)
{
  if (SaveDialog1->Execute())               // ファイル名を取得
  {
    CurrentFile = SaveDialog1->FileName;  // ユーザー指定名で保存
    Save1Click(Sender);                   // 通常の方法で保存
  }
}
```

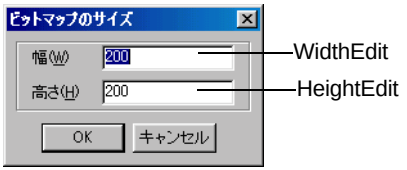
画像の置換

イメージコントロール内の画像は，いつでも置換できます。すでにグラフィックが格納されている画像オブジェクトに新しいグラフィックを代入すると，既存のグラフィックは新しいグラフィックで置き換えられます。

イメージコントロール内の画像を置換するには，イメージコントロールの Picture オブジェクトに新しいグラフィックを代入します。

画像の置換処理は，初期グラフィックの作成処理と同じです（10-17 ページの「ビットマップの初期サイズの設定」を参照してください）。ただし，初期グラフィックの作成時に使用されたデフォルトサイズ以外のサイズをユーザーが指定できるようにしなければなりません。このオプションを提供する簡単な方法は，図 10.1 にあるようなダイアログボックスを表示させることです。

図 10.1　BMPDlg ユニットの［ビットマップのサイズ］ダイアログボックス



この特別なダイアログボックスは，GraphEx プロジェクト（Examples \ Doc \ GraphEx ディレクトリ）に含まれる BMPDlg ユニット内に作成されます。

このようなダイアログボックスをプロジェクト内に入れる場合は，メインフォーム用の .cpp ファイルに BMPDlg.hpp の include 文を追加します。この文を追加すると，メニュー項目［ファイル｜新規作成］の OnClick イベントにイベントハンドラを接続できるようになります。以下に例を示します。

```
void __fastcall TForm1::New1Click(TObject *Sender)
{
     Graphics::TBitmap *Bitmap;
  // width フィールドに確実にフォーカスが当たるように
     NewBMPForm->ActiveControl = NewBMPForm->WidthEdit;
  //現在のサイズをデフォルトとして初期化
     NewBMPForm->WidthEdit->Text = IntToStr(Image->Picture->Graphic->Width);
     NewBMPForm->HeightEdit->Text = IntToStr(Image->Picture->Graphic->Height);
     if (NewBMPForm->ShowModal() != IDCANCEL){    // ダイアログがキャンセルされないときは ...
        Bitmap = new Graphics::TBitmap();         // 新しいビットマップオブジェクトを作成
             // 指定のサイズを使用
        Bitmap->Width = StrToInt(NewBMPForm->WidthEdit->Text);
        Bitmap->Height = StrToInt(NewBMPForm->HeightEdit->Text);
        Image->Picture->Graphic = Bitmap;         // グラフィックを新しいビットマップと交換
        CurrentFile = EmptyStr;                   // ファイル名が未定であることを示す
       delete Bitmap;
     }
}
```

メモ　画像オブジェクトの Graphic プロパティに新しいビットマップを代入すると，画像オブジェクトは新規ビットマップをコピーしますが，新規ビットマップのオーナーにはなりません。画像オブジェクトは，自分自身の内部グラフィックオブジェクトを管理します。このため，前のコードでは，ビットマップを設定したあとにビットマップオブジェクトを解放しています。

グラフィックに対するクリップボードの使い方

Windows のクリップボードを使ってアプリケーションにグラフィックをコピーしたり，貼り付けたり，ほかのアプリケーションとグラフィックを交換したりできます。VCL のクリップボードオブジェクトを使うと，グラフィックを含めた異なる種類の情報を簡単に扱えます。

アプリケーションでクリップボードオブジェクトを使用する前に，クリップボードデータにアクセスする各 .cpp ファイルに Clipbrd.hpp の include 文を追加します。

クロスプラットフォームアプリケーションの場合，CLX の使用時にクリップボードに保存されるデータは，関連の TStream オブジェクトと一緒に MIME 型として保存されます。CLX には，以下の MIME 型に対して，以下の定数があらかじめ定義されています。

表 10.4　CLX の MIME 型と定数

MIME 型	CLX の定数
'image/delphi.bitmap'	SDelphiBitmap
'image/delphi.component'	SDelphiComponent
'image/delphi.picture'	SDelphiPicture
'image/delphi.drawing'	SDelphiDrawing

クリップボードへのグラフィックのコピー

クリップボードへは，イメージコントロールの内容を含めて，あらゆる画像をコピーできます。クリップボードへコピーした画像は，すべてのアプリケーションが利用できるようになります。

画像をクリップボードへコピーするには，Assign メソッドを使って画像を Clipboard オブジェクトに代入します。

次のコードは，メニュー項目［編集｜コピー］への応答として，Image という名前のイメージコントロールからクリップボードへ画像をコピーする方法を示しています。

```
void __fastcall TForm1::Copy1Click(TObject *Sender)
{
  Clipboard()->Assign(Image->Picture);
}
```

クリップボードへのグラフィックの切り取り

クリップボードへのグラフィックの切り取りは，コピーとまったく同じ処理ですが，ソースからグラフィックが削除されるという点が異なります。

画像からクリップボードへグラフィックを切り取る場合は，最初にクリップボードへのコピーを行い，次に元のグラフィックを消去します。

ほとんどの場合，切り取りに関しては，消去された元のイメージをどのように表示するのかが唯一の問題になります。次のコードのように，その領域を白にするのが一般的な解決策です。次のコードは，メニュー項目［編集｜切り取り］の OnClick イベントにイベントハンドラを結び付けます。

```
void __fastcall TForm1::Cut1Click(TObject *Sender)
{
  TRect ARect;
  Copy1Click(Sender);          // クリップボードへ画像をコピー
```

```
  Image->Canvas->CopyMode = cmWhiteness; // すべてを白色でコピー
  ARect = Rect(0, 0, Image->Width, Image->Height); // イメージのサイズを取得
  Image->Canvas->CopyRect(ARect, Image->Canvas, ARect); // ビットマップを自身にコピー
  Image->Canvas->CopyMode = cmSrcCopy; // 通常のモードに戻る
}
```

クリップボードからのグラフィックの貼り付け

クリップボードにグラフィックが格納されているときには，そのグラフィックをイメージコントロールやフォームを含むイメージオブジェクトに貼り付けることができます。

クリップボードからグラフィックを貼り付けるには，次のように処理します。

1. クリップボードの HasFormat メソッド（VCL を使用している場合）か Provides メソッド（CLX を使用している場合）を呼び出して，クリップボードにグラフィックがコピーされているかどうかを確認します。

   HasFormat（CLX の場合は Provides）は論理関数です。パラメータで指定された種類のデータがクリップボードにあれば **true** を返します。Windows プラットフォームの場合，グラフィックかどうかをチェックするには，CF_BITMAP を渡します。クロスプラットフォームアプリケーションでは，SDelphiBitmap を渡します。

2. 貼り付け先にクリップボードを割り当てます。

VCL　次の VCL コードは，メニュー項目［編集｜貼り付け］のクリックへの応答として，クリップボードからイメージコントロールに画像を貼り付ける方法を示しています。

```
void __fastcall TForm1::Paste1Click(TObject *Sender)
{
  Graphics::TBitmap *Bitmap;
  if (Clipboard()->HasFormat(CF_BITMAP)){
    Image1->Picture->Bitmap->Assign(Clipboard());
  }
}
```

CLX　CLX を使ったクロスプラットフォーム開発で同じ例を示すと，次のようになります。

```
void __fastcall TForm1::Paste1Click(TObject *Sender)
{
  QGraphics::TBitmap *Bitmap;
  if (Clipboard()->Provides(SDelphiBitmap)){
    Image1->Picture->Bitmap->Assign(Clipboard());
  }
}
```

クリップボードへは，このアプリケーションからグラフィックをコピーすることも，Microsoft Paint などの別のアプリケーションからコピーすることもできます。この場合，クリップボードにサポートされているフォーマットがないときは［貼り付け］メニューを使用不可にする必要があるので，クリップボードのフォーマットをチェックする必要はありません。

ラバーバンドの例

この例では，実行時にユーザーがグラフィックを描く際にマウスの動作を追跡する「ラバーバンド」の効果の実装方法について詳しく説明します。実際のコードの例については，Examples\Doc\GraphEx

ディレクトリに保存されているサンプルアプリケーションを参照してください。このサンプルアプリケーションでは，クリックとドラッグに応じて直線や図形を描画します。マウスボタンが押されたら描画を開始し，マウスボタンが放されたら描画を終了します。

最初に，メインフォームの表面にどのように描画するのかを示し，次にビットマップへの描画を紹介します。

以下のトピックで例を解説しています。

- マウスへの応答
- マウス動作を追跡するためのフィールドのフォームへの追加
- より洗練された線の描画

マウスへの応答

アプリケーションは，マウスボタンを押す，マウスを移動する，マウスボタンを放す，というマウス固有の操作に応答できます。また，クリック（同じ場所でマウスボタンを押して放す動作）にも応答できます。クリックは，モード付きダイアログボックスに対して〔Enter〕キーを押すなど，いくつかのキーを使って生成することもできます。

このセクションでは，以下について説明します。

- マウスイベントのハンドラ
- マウスボタンが押されたことを示すイベントへの応答
- マウスボタンが放されたことを示すイベントへの応答
- マウスの移動への応答

マウスイベントのハンドラ

C++Builder には，OnMouseDown，OnMouseMove，OnMouseUp の 3 つのマウスイベントがあります。

アプリケーションがマウス動作を検出すると，対応するイベントに対して定義されているイベントハンドラを呼び出し，5 種類のパラメータを渡します。各パラメータの情報は，イベントに対する応答をカスタマイズするために使用します。この 5 つのパラメータを次の表に示します。

表 10.5　マウスイベントパラメータ

パラメータ	説明
Sender	マウスイベントを検出したオブジェクト
Button	関与したマウスボタンを示す。mbLeft，mbMiddle，または mbRight
Shift	マウス操作時の〔Alt〕，〔Ctrl〕，および〔Shift〕の各キーの状態を示す
X, Y	イベントが発生した座標

ほとんどの場合，マウスイベントハンドラに返される座標だけが必要になりますが，場合によっては，Button をチェックして，どのボタンが押されたのかを確認する必要があります。

メモ　C++Builder では，押されたマウスボタンの特定は Microsoft Windows と同じ判定条件で行われます。したがって，デフォルトの「一次」ボタンと「二次」ボタンの設定を切り替えて，右マウスボタンを一次ボタンとした場合には，一次ボタン（この場合右）をクリックしたときに Button パラメータの値に記録されるのは mbLeft です。

マウスボタンが押されたことを示すイベントへの応答

ユーザーがマウスボタンを押すと，マウスポインタの位置にあるオブジェクトで OnMouseDown イベントが発生します。イベントに応答できるのは，イベントの発生した場所にあるオブジェクトです。

マウスを押す動作に応答するには，OnMouseDown イベントにイベントハンドラを結び付けます。

C++Builder は，フォームのマウスボタンが押されたことを示すイベントに対して空のハンドラを生成します。

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{

}
```

マウスボタンが押されたことを示すイベントへの応答

次のコードは，マウスでクリックしたフォーム上の位置に文字列「Here!」を表示しています。

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->TextOut(X, Y, "Here!");  //  (X, Y) にテキストを表示
}
```

ここでアプリケーションを実行し，フォーム上にマウスポインタを置いてマウスボタンを押すと，クリック位置に「Here!」という文字列が表示されます。次のコードは，ペンの描画開始位置をユーザーがボタンを押した位置に移動します。

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->MoveTo(X, Y);  // ペンの位置を設定
}
```

マウスボタンを押すと，ペン位置が設定されて直線の始点になります。さらに，ユーザーがマウスボタンを放した座標を直線の終点とすることにします。このためには，マウスボタンが放されたことを示すイベントに応答しなければなりません。

マウスボタンが放されたことを示すイベントへの応答

ユーザーがマウスボタンを放すと，OnMouseUp イベントが発生します。このイベントは，ユーザーがボタンを押したときのマウスポインタ位置にあったオブジェクトにおいて発生します。このオブジェクトと，ボタンを放したときのポインタ位置にあるオブジェクトとは，必ずしも一致するとは限りません。このような仕様になっているので，たとえば，フォームの境界外まで延長した線でも描画できます。

マウスボタンが放されたことに応答するには，OnMouseUp イベントのハンドラを定義します。

マウスボタンを放した位置まで直線を描画する単純な OnMouseUp イベントハンドラを次に示します。

```
void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->LineTo(X, Y);  // PenPos から (X, Y) まで線を描画
}
```

このコードを使用すると，ユーザーは，マウスボタンを押し，マウスを動かしてボタンを放すという操作で線が描画できるようになります。この場合，マウスボタンを放すまで線は表示されません。

マウスの移動への応答

ユーザーがマウスを移動すると，移動中は OnMouseMove イベントが発生します。このイベントは，ユーザーがボタンを押したときのマウスポインタ位置にあったオブジェクトにおいて発生します。このイベントを使用して一時的な直線を描くと，マウスが動いている間の中間的なフィードバックが可能になります。

マウスの動作に応答するには，OnMouseMove イベントに対するイベントハンドラを定義してください。以下の例では，マウスボタンを押したままマウスを移動したときのイベントを利用して一時的な直線を描画し，ユーザーに動作がわかるようにします。フォームで OnMouseMove イベントの発生した位置まで線を描画する単純な OnMouseMove イベントハンドラを次に示します。

```
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->LineTo(X, Y);  // 現在の位置まで直線を描画
}
```

このコードでは，マウスボタンを押さなくても，マウスの動くとおりに描画が行われます。

マウスの移動イベントはマウスボタンを押さなくても発生します。

マウスボタンが押されているかどうかを追跡するには，フォームオブジェクトにオブジェクトフィールドを追加します。

マウス動作を追跡するためのフィールドのフォームへの追加

マウスボタンが押されたかどうかをチェックするには，フォームオブジェクトにオブジェクトフィールドを追加する必要があります。C++Builder では，フォームにコンポーネントを追加すると，そのフォームオブジェクトに対してそのコンポーネントを表現するフィールドが追加されるため，そのフィールドの名前を使ってコンポーネントを参照できるようになります。また，フォームユニットの型宣言を編集して，独自のフィールドを追加することもできます。

次の例では，ユーザーがマウスボタンを押しているかどうかを判定する必要があります。判定するには，論理フィールドを追加してユーザーがマウスボタンを押したときにその値を設定するようにします。

オブジェクトにフィールドを追加するには，オブジェクトの型宣言を編集し，宣言部の末尾にある **public** 指令の後にフィールド識別子と型を追加します。

public 指令の前にある宣言は C++Builder で作成されたものです。C++Builder はこの部分に，コントロールを示すフィールドとイベントに応答するメソッドを宣言します。

次のコードは，フォームオブジェクトの宣言内に論理型の Drawing というフィールドを追加します。ここでは，TPoint 型の 2 つのフィールド，Origin と MovePt も追加しています。

```
class TForm1 : public TForm
{
__published:  // IDE 管理コンポーネント
  void __fastcall FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y);
```

```
    void __fastcall FormMouseMove(TObject *Sender, TShiftState Shift, int X, int Y);
    void __fastcall FormMouseUp(TObject *Sender, TMouseButton Button,
    TShiftState Shift, int X, int Y);
  private:  // ユーザー宣言
  public:   // ユーザー宣言
    __fastcall TForm1(TComponent* Owner);
    bool Drawing;  // ボタンが押されたかどうかを追跡するフィールド
     TPoint Origin, MovePt;  // ポイントを保存するフィールド
  };
```

描画を行うかどうかを指定するための Drawing フィールドができたら，ユーザーがマウスボタンを押したときにはそのフィールドに **true** を設定し，ボタンを放したときに **false** を設定します。

```
  void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
    TShiftState Shift, int X, int Y)
  {
    Drawing = true;            // 描画フラグをセット
    Canvas->MoveTo(X, Y);      // ペンの位置を設定
  }
  void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button,
    TShiftState Shift, int X, int Y)
  {
    Canvas->LineTo(X, Y);      // PenPos から (X, Y) まで線を描画
    Drawing = false;           // 描画フラグをクリア
  }
```

OnMouseMove イベントハンドラを変更し，Drawing が **true** の場合にのみ描画させます。

```
  void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
    TShiftState Shift, int X, int Y)
  {
    if (Drawing)
      Canvas->LineTo(X, Y);    // マウスボタンが押し下げられている場合にのみ描画する
  }
```

このように記述すると，マウスボタンを押して，移動し，マウスボタンを放すまで線が描画されます。しかし，これでもマウスの動作に追従する線は，きれいな直線にはならず，いたずら書きのようになってしまいます。

問題は，マウスを移動するたびにマウスの移動イベントハンドラが LineTo を呼び出して，ペン位置を移動させてしまうことです。マウスボタンを放すときには，直線を描こうとしても始点の位置がわからなくなっています。

より洗練された線の描画

すでに追加したフィールドでさまざまな点を記録できるので，目的の直線を描けるようになります。

始点の記録

線を描画する場合は，Origin フィールドを使って線の開始位置を保存します。

OnMouseDown イベントの発生位置を Origin に記録すれば，OnMouseUp イベントハンドラはこの Origin を参照して線の始点を特定できます。

```
  void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
    TShiftState Shift, int X, int Y)
  {
    Drawing = true;            // 描画フラグをセット
```

```
  Canvas->MoveTo(X, Y);       // ペンの位置を設定
  Origin = Point(X, Y);       // 線の開始位置を記録
}
void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->MoveTo(Origin.x, Origin.y);  // ペンを始点に移動する
  Canvas->LineTo(X, Y);                // PenPos から (X, Y) まで線を描画
  Drawing = false;                     // 描画フラグをクリア
}
```

この変更を行うと，アプリケーションは最終的な線を描画できるようになりますが，中間的な動作は描画できません。「ラバーバンド」の機能がサポートされていないという点に注意してください。

移動の追跡

この例での OnMouseMove イベントハンドラとしての問題点は，最初の位置からではなく，直前のマウス位置から現在のマウス位置への描画を行うという点にあります。この問題は，描画位置を原点に戻してから現在の位置まで線を描くという方法で修正できます。

```
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  if (Drawing)
  {
    Canvas->MoveTo(Origin.x, Origin.y);  // ペンを始点に移動する
    Canvas->LineTo(X, Y);
  }
}
```

上記のコードは，現在のマウス位置を追跡しますが，中間的な線は消去しないため，どれが最終的な線なのかが判断できなくなります。したがって，直前の点がどこであったのかを保存しておいて，次の線を描く前に直前の線を消去する必要があります。この処理を行うために，MovePt フィールドを使用します。

MovePt と Origin を使って直前の線を消去できるように，MovePt に中間的な各線の終点を設定する必要があります。

```
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Drawing = true;             // 描画フラグをセット
  Canvas->MoveTo(X, Y);       // ペンの位置を設定
  Origin = Point(X, Y);       // 線の開始位置を記録
  MovePt = Point(X, Y);       // 線の終了位置を記録
}
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  if (Drawing)
  {
    Canvas->Pen->Mode = pmNotXor;         // XOR モードで描画または消去する
    Canvas->MoveTo(Origin.x, Origin.y);   // ペンを始点に移動する
    Canvas->LineTo(MovePt.x, MovePt.y);   // 前の線を消去する
    Canvas->MoveTo(Origin.x, Origin.y);   // ペンを始点に移動する
    Canvas->LineTo(X, Y);                 // 新しい線を描画する
```

```
    }
    MovePt = Point(X, Y);       // 線の新しい終点を記録
    Canvas->Pen->Mode = pmCopy;
  }
```

これで，描画時のラバーバンド効果が実現できました。ペンのモードを pmNotXor にすると，ペン色と背景色を排他演算した結果（つまりペンとも背景とも一致しない色）をさらに反転した色で描画します。このペンモードの特性は，一度目の描画は背景ともペンとも一致しないユニークな色で描画し，同じ部分をもう一度描画すると，元々の背景色に戻ることです。この例では，この特性を利用してマウス移動中の線を描画し，そして消去しています。線を描画した後でペンモードを pmCopy（デフォルト値）に戻すと，マウスボタンを放したときに最終的な線を描画できる状態が保証されます。

マルチメディアの利用

C++Builder を使うと，Windows アプリケーションにマルチメディアコンポーネントを追加できます（CLX/Linux アプリケーションには追加できません）。コンポーネントを追加するには，コンポーネントパレットの［Win32］ページの TAnimate コンポーネント，または［System］ページの TMediaPlayer コンポーネントのいずれかを利用できます。アプリケーションにサイレント（音のない）ビデオクリップを追加するには，TAnimate コンポーネントを使用します。また，アプリケーションにオーディオまたはビデオクリップを追加するには，TMediaPlayer コンポーネントを使用します。

TAnimate コンポーネントと TMediaPlayer コンポーネントについての詳細は，VCL オンラインヘルプを参照してください。

このセクションでは以下のことを説明します。

- アプリケーションへのサイレントビデオクリップの追加
- アプリケーションへのオーディオまたはビデオクリップの追加

アプリケーションへのサイレントビデオクリップの追加

C++Builder のアニメーションコントロールを使用すると，アプリケーションにサイレントビデオクリップを追加できます。

アプリケーションにサイレントビデオクリップを追加するには，次の手順に従います。

1. コンポーネントパレットの［Win32］ページの Animate コンポーネントをダブルクリックします。ビデオクリップを表示させるフォームウィンドウに自動的にアニメーションコントロールが追加されます。
2. オブジェクトインスペクタを使用して，Name プロパティを選択し，アニメーションコントロールに新しい名前を付けます。追加したアニメーションコントロールを呼び出す場合には，この名前が使用されます。新しい名前は C++Builder の識別子として有効なものでなければなりません。

   設計時プロパティを設定するときとイベントハンドラを作成するときには，常にオブジェクトインスペクタを使用して操作します。
3. 次のいずれかを選んでください。

- CommonAVI プロパティを選択し，ドロップダウンボタンをクリックします。リスト中の利用可能な AVI の中から 1 つを選びます。
- FileName プロパティを選択し，省略記号（...）ボタンをクリックし，［AVI を開く］ダイアログを表示させます。ファイルの一覧から表示したい任意の AVI ファイルを選択し［開く］ボタンで確定します。
- ResName または ResID プロパティを指定することで AVI リソースを選択できます。ResHandle プロパティを指定することでアプリケーション以外のモジュールからリソースを読み出すこともできます。

これによって AVI ファイルがメモリ内に読み込まれ，AVI クリップの最初のフレームが画面に表示されます。Active プロパティまたは Play メソッドにより再生が始まるまで最初のフレームを表示させたくない場合には，実行時に Open プロパティを **true** に設定します。

4. Repetition プロパティは AVI クリップを再生する繰り返し回数を設定します。この値が 0 の場合は，Stop メソッドが呼び出されるまで自動的に繰り返し再生されます。
5. その他のアニメーションコントロールの設定を変更します。たとえば，アニメーションコントロールを開いたときに表示される最初のフレームを変更する場合は，StartFrame プロパティの値を設定します。
6. ドロップダウンリストを使って Active プロパティを **true** に設定するか，または実行中に特定のイベントが発生すると AVI クリップを再生するイベントハンドラを作成します。たとえば，ボタンを押したら AVI の再生を開始させるには，ボタンの OnClick イベントの中で Active プロパティを **true** にします。Play メソッドを呼び出しても同じです。

メモ Active プロパティを **true** に設定した後にフォームまたはフォームのコンポーネントを変更すると，Active プロパティが **false** になるので，**true** に再設定する必要があります。これは実行直前か実行時に行ってください。

サイレントビデオクリップの追加の例

アプリケーションが起動するときに表示される最初の画面にアニメーションロゴを表示させる場合を考えてみます。ロゴの表示が終了すると，画面が消えます。

この例を実行するには，新しいプロジェクトを作成して Unit1.cpp ファイルを Frmlogo.cpp として保存し，Project1.bpr ファイルを Logo.bpr として保存します。その後，次の手順を実行します。

1. コンポーネントパレットの［Win32］ページの［Animate］アイコンをダブルクリックします。
2. オブジェクトインスペクタを使用して，Name プロパティを Logo1 に設定します。
3. FileName プロパティを選択して，省略記号（...）ボタンを選択し，.. \ Examples \ MFC \ General \ Cmnctrls ディレクトリから dillo.avi ファイルを選びます。［開く］をクリックします。

   これによって dillo.avi ファイルがメモリに読み込まれます。
4. フォーム上のアニメーションコントロールをクリックし，フォームの右上部に移動します。
5. Repetitions プロパティを 5 に設定します。

6. フォームをクリックしてアクティブにし，Name プロパティを LogoForm1 に設定してさらに Caption プロパティを Logo Window に設定します。フォームの高さを調整して，横に長いフォームにします。
7. オブジェクトインスペクタの［イベント］タグをクリックし，さらに OnActivate イベントをダブルクリックします。コードエディタがアクティブになっているので，その位置に次のコードを書き込みます。このコードにより，実行時にフォームがアクティブになったときに AVI クリップが実行されるようにします。

```
Logo1->Active = true;
```

8. コンポーネントパレットの［Standard］ページの［Label］アイコンをダブルクリックします。Caption プロパティを選択して，「Welcome to Armadillo Enterprises 4.0」と入力します。Font プロパティを選択して，省略記号（...）ボタンを選択し，［フォントの指定］ダイアログでスタイルを太字，サイズを「18」，色を「濃紺」に設定して［OK］を選択します。ラベルコントロールをドラッグし，アニメーションコントロールの左側に置きます。
9. アニメーションコントロールをクリックしてアクティブにします。［イベント］タブの OnStop イベントをダブルクリックして，次のコードを記述します。これにより，AVI ファイルが停止したときにフォームを閉じるようにします。

```
LogoForm1->Close();
```

10. ［実行｜実行］を選択してアニメーションロゴウィンドウを実行します。

アプリケーションへのオーディオまたはビデオクリップの追加

C++Builder のメディアプレイヤーコンポーネントを使用すると，アプリケーションにオーディオクリップやビデオクリップを追加できます。これはメディアデバイスを開いて，オーディオまたはビデオクリップの再生，停止，一時停止，録音などを行います。メディアデバイスはハードウェアまたはソフトウェアになります。

メモ オーディオ / ビデオクリップは，クロスプラットフォームプログラミングではサポートされていません。

アプリケーションにオーディオまたはビデオクリップを追加する手順は，以下のとおりです。

1. コンポーネントパレットの［System］ページにある［MediaPlayer］アイコンをダブルクリックします。これにより，メディア機能を持たせたいフォームウィンドウに自動的にメディアプレイヤーコントロールが配置されます。
2. オブジェクトインスペクタを使用して，Name プロパティを選択し，メディアプレイヤーコントロールに新しい名前を付けます。メディアプレイヤーコントロールを呼び出す場合に，この名前を使用します新しい名前は C++Builder の識別子として有効なものでなければなりません。

   設計時プロパティを設定するときとイベントハンドラを作成するときには，常にオブジェクトインスペクタを使用して操作します。
3. DeviceType プロパティを選択し，AutoOpen プロパティまたは Open メソッドで開くメディアデバイスの種類を指定します（DeviceType が dtAutoSelect の場合，デバイスの種類は FileName プロパティで指定されたメディアファイルのファイル拡張子に基づいて選択されます）。デバイスの種類と機能の詳細については，表 10.6 を参照してください。

4. ファイルにメディアデバイスのデータが格納されている場合は，FileName プロパティを使用してメディアファイルを指定できます。FileName プロパティを選択して，省略記号（...）ボタンをクリックし，利用可能なローカルまたはネットワークディレクトリからメディアファイルを選んで，［開く］ダイアログの［開く］をクリックします。メディアデバイスがハードウェアの場合は，実行する前に選択したメディアデバイスにメディアが格納されているハードウェア（ディスク，カセットなど）を挿入します。

5. AutoOpen プロパティを **true** に設定します。この設定では，実行時にメディアプレイヤーコントロールを持つフォームが作成されると，メディアプレイヤーが自動的に指定された装置を開きます。AutoOpen プロパティが **false** の場合，装置は Open メソッドを呼び出して開かなくてはなりません。

6. AutoEnable プロパティを **true** に設定すると，実行時にメディアプレイヤーの各ボタンが自動的に使用可能または使用不可能になります。また，EnabledButtons プロパティをダブルクリックすると，各ボタンごとに使用可能または使用不可能のどちらにするかを任意に設定できます。

   ユーザーがメディアプレイヤーコンポーネントの対応するボタンをクリックするとマルチメディア装置の再生，一時停止，停止などが行われます。装置は，ボタン（Play，Pause，Stop，Next，Previous など）に対応するメソッドを呼び出して制御することもできます。

7. メディアプレイヤーのコントロールバーをドラッグして，フォームの適切な場所に置きます。

   実行時にメディアプレイヤーを非表示にする場合には，Visible プロパティを **false** に設定し，適切なメソッド（Play，Pause，Stop，Next，Previous，Step，Back，StartRecording，Eject）を呼び出して装置を制御します。

8. メディアプレイヤーコントロールの他の設定を変更します。たとえば，メディアに表示ウィンドウが必要な場合，メディアを表示するための Display プロパティをコントロールに設定します。装置がマルチトラックを使用しているときは，目的のトラックに Tracks プロパティを設定します。

表 10.6　マルチメディア装置の種類と機能

装置の種類	使用ソフトウェアまたはハードウェア	再生可能メディア	トラックの使用	表示ウィンドウの使用
dtAVIVideo	Windows AVI Video Player	AVI ビデオファイル	不使用	使用
dtCDAudio	CD Audio Player for Windows または CD Audio Player	CD オーディオディスク	使用	不使用
dtDAT	DAT Player	DAT	使用	不使用
dtDigitalVideo	Digital Video Player for Windows	AVI，MPG，MOV ファイル	不使用	使用
dtMMMovie	MM Movie Player	MM フィルム	不使用	使用
dtOverlay	オーバーレイ装置	アナログビデオ	不使用	使用
dtScanner	Image Scanner	再生用ではない（記録のイメージをスキャン）	不使用	不使用
dtSequencer	MIDI Sequencer for Windows	MIDI ファイル	使用	不使用
dtVCR	VCR	ビデオカセット	不使用	使用
dtWaveAudio	Wave Audio Player for Windows	WAV ファイル	不使用	不使用

オーディオまたはビデオクリップの追加の例（VCL のみ）

この例では，C++Builder のマルチメディア広告の AVI ビデオクリップを実行します。この例を実行するには，新しいプロジェクトを作成して Unit1.cpp ファイルを FrmAd.cpp として保存し，Project1.bpr ファイルを MmediaAd.bpr として保存します。その後で，次の手順を実行します。

1. コンポーネントパレットの［System］ページにある［MediaPlayer］アイコンをダブルクリックします。
2. オブジェクトインスペクタを使用して，メディアプレイヤーの Name プロパティを VideoPlayer1 に設定します。
3. FileName プロパティを選択して，省略記号（...）ボタンを選び，.. \ Examples \ Coolstuf ディレクトリからファイルを選びます。［開く］をクリックします。
4. AutoOpen プロパティを **true** に，Visible プロパティを **false** に設定します。
5. コンポーネントパレットの［Win32］ページの Animate アイコンをダブルクリックします。AutoSize プロパティを **false** に設定し，Height プロパティを 175 に，Width プロパティを 200 に設定します。アニメーションコントロールをクリックして，フォームの左上隅までドラッグします。
6. メディアプレイヤーをクリックしてアクティブにします。Display プロパティを選択して，ドロップダウンリストから Animate1 を選びます。
7. フォームをクリックしてアクティブにし，Caption プロパティを選択して C++_Ad と入力します。フォームのサイズをアニメーションコントロールが十分入るように調整します。
8. フォームの OnActivate イベントをダブルクリックして次のコードを記述します。これにより，フォームがアクティブになったときに AVI ビデオが実行されるようにします。

   ```
   VideoPlayer1->Play();
   ```

9. ［実行｜実行］を選択して AVI ビデオを実行します。

第11章

マルチスレッドアプリケーションの作成

C++Builder には，マルチスレッドアプリケーションを簡単に開発するためのオブジェクトが用意されています。マルチスレッドアプリケーションとは，同時に複数の実行経路を持つアプリケーションのことです。複数のスレッドを使用する場合には注意が必要ですが，以下のような点でプログラムを強化できます。

- **ボトルネックを回避する。**スレッドが 1 つしかないと，遅いプロセスを待っている間は，プログラムの実行をすべて停止しなければなりません。遅いプロセスには，ディスク上のファイルへのアクセス，ほかのマシンとの通信，マルチメディアコンテンツの表示などがあります。CPU は，このプロセスが完了するまで何もせずに待っています。複数のスレッドを使用すると，遅いプロセスの結果を待っている間でも，別のスレッドでアプリケーションの実行を継続できます。
- **プログラムの動作を体系化する。**プログラムの動作は，独立で機能するいくつかの並列プロセスに体系化できます。スレッドを利用して，並列プロセスごとに一群のコードを同時に実行できます。スレッドを使用すると，複数のプログラムタスクに優先順位を付けることができるので，より重要なタスクにより多くの CPU 時間を割り当てることができます。
- **マルチプロセッシング。**プログラムを実行するシステムが複数のプロセッサを持つ場合は，プログラムを複数のスレッドに分割して別々のプロセッサ上で同時に実行すれば，パフォーマンスを向上させることができます。

メモ　使用するハードウェアがマルチプロセッシングをサポートしている場合でも，真の意味でのマルチプロセッシングを実現しないオペレーティングシステムもあります。たとえば Windows 9x では，たとえハードウェアが複数のプロセッサを持っていても 1 つのプロセッサしか使わず，タイムスライスを使ってマルチプロセッシングをシミュレーションするだけです。

スレッドオブジェクトの定義

たいていのアプリケーションでは，スレッドオブジェクトを使って，アプリケーションの実行スレッドを表すことができます。スレッドオブジェクトには一般的に必要となるスレッドの使い方がカプセル化されているので，マルチスレッドアプリケーションの作成が簡単にできます。

メモ スレッドオブジェクトでは，スレッドのセキュリティ属性やスタックサイズを制御できません。これらを制御する必要がある場合には，Windows API の CreateThread 関数または BeginThread 関数を使わなければなりません。Windows のスレッド API または BeginThread を使用する場合でも，スレッドの同期をとるオブジェクトおよび 11-7 ページの「スレッドの調整」に説明されているメソッドの機能を利用することができます。CreateThread または BeginThread の使い方の詳細については，Windows のオンラインヘルプを参照してください。

アプリケーションの中でスレッドオブジェクトを使用するには，TThread を継承した新しいクラスを作成しなければなりません。TThread の派生クラスを作成するには，メインメニューの［ファイル｜新規作成｜その他］を選択します。［新規作成］ダイアログボックスで［スレッドオブジェクト］（クロスプラットフォームアプリケーションの場合は［CLX スレッドオブジェクト］）をダブルクリックし，TMyThread などのクラス名を入力します。この新規スレッドに名前を付けるには，［名前付きスレッド］チェックボックスをチェックしてスレッド名を入力します（VCL のみ）。スレッドに名前を付けておくと，デバッグ中にスレッドを追跡するのが簡単になります。［OK］をクリックすると，C++Builder は新しい .cpp およびヘッダーファイルを作成してスレッドを実装します。スレッド名の設定についての詳細は，11-13 ページの「スレッドに名前を付ける」を参照してください。

メモ クラス名を要求する通常の IDE のダイアログボックスと違って，［スレッドオブジェクトの作成］ダイアログボックスでは，入力したクラス名の先頭に自動的に 'T' が付加されません。

自動生成された .cpp ファイルには，新しいスレッドクラスのスケルトンが含まれています。スレッド名が TMyThread の場合，そのコードは次のようになります。

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit2.h"
#pragma package(smart_init)
//---------------------------------------------------------------------------
__fastcall TMyThread::TMyThread(bool CreateSuspended): TThread(CreateSuspended)
{
}
//---------------------------------------------------------------------------
void __fastcall TMyThread::Execute()
{
  // ---- スレッドのコードをここに挿入 ----
}
//---------------------------------------------------------------------------
```

コンストラクタおよび Execute メソッドのコードを記述する必要があります。この手順を以降のセクションで説明していきます。

スレッドの初期化

コンストラクタを使用して新しいスレッドクラスを初期化します。ここで，このスレッドにデフォルトの優先度を割り当てます。また，このスレッドの実行が終了したときに，自動的にスレッドを解放するかどうかもここで指定します。

デフォルトの優先度を割り当てる

優先度とは，オペレーティングシステムがアプリケーション中のすべてのスレッド間の CPU 時間のスケジュールを作成する場合に，どのスレッドをどのぐらい優先するかを表します。処理速度に敏感なタスクを処理するスレッドには高い優先度を使い，それ以外のタスクを実行するスレッドには低い優先度を使います。スレッドオブジェクトの優先度を表すには，Priority プロパティを設定します。

Windows アプリケーションを作成する場合，Priority の値は 7 ポイントの範囲で表します。表 11.1 にその値を示します。

表 11.1　スレッドの優先度

値	優先度
tpIdle	システムの空き時間にだけ，このスレッドが実行される。tpIdle の優先度を持つスレッドを実行するために，Windows がほかのスレッドに割り込むことはない
tpLowest	スレッドの優先度が標準より 2 ポイント低い
tpLower	スレッドの優先度が標準より 1 ポイント低い
tpNormal	標準の優先度
tpHigher	スレッドの優先度が標準より 1 ポイント高い
tpHighest	スレッドの優先度が標準より 2 ポイント高い
tpTimeCritical	スレッドの優先度がもっとも高い

メモ　クロスプラットフォームアプリケーションを作成する場合は，Windows と Linux でそれぞれ別のコードを使って優先度を割り当てる必要があります。Linux では，Priority はスレッド規則によって決まる数値で，root によってのみ変更できます。詳細については，オンラインヘルプで CLX バージョンの TThread と Priority を参照してください。

注意　CPU に処理が集中するスレッドの優先度を高くすると，アプリケーション中のほかのスレッドが実行できなくなる可能性があります。外部イベントを待つことにほとんどの時間を費やすようなスレッドの優先度だけを高くするようにしてください。

次のコードは，バックグラウンドタスクを実行する優先度の低いスレッドのコンストラクタです。バックグラウンドタスクはアプリケーションのほかの処理のパフォーマンスを妨げてはいけません。

```
//---------------------------------------------------------------------------
__fastcall TMyThread::TMyThread(bool CreateSuspended): TThread(CreateSuspended)
{
  Priority = tpIdle;
}
//---------------------------------------------------------------------------
```

スレッドを解放するタイミングを指定する

通常は，スレッドの動作が終了した時点で簡単にスレッドを解放できます。この場合は，スレッドオブジェクトに自分自身を解放させるのがもっとも簡単です。それには，FreeOnTerminate プロパティを **true** に設定します。

一方，複数のスレッドの間で終了のタイミングを調整しなければならない場合もあります。たとえば，あるスレッドが値を返すのを待ってから別のスレッドを実行する場合などです。この場合は，2 番目のスレッドが戻り値を受け取るまで，1 番目のスレッドを解放できません。これを処理するには，まず FreeOnTerminate を **false** に設定し，次に 2 番目のスレッドから明示的に 1 番目のスレッドを解放します。

スレッド関数の書き方

Execute メソッドはスレッドの関数です。つまり，アプリケーションによって起動されるプログラムの 1 つと考えることができますが，同じプロセス空間を共有する点が異なります。スレッド関数を記述するには，独立のプログラムを記述する場合に比べて少しだけこつが必要です。それは，アプリケーション中のほかのスレッドによって使われるメモリを上書きしないようにする必要があるからです。一方，スレッドはほかのスレッドと同じプロセス空間を共有するので，共有メモリを使用して，スレッド間の通信ができます。

メイン VCL/CLX スレッドを使用する

VCL または CLX のオブジェクト階層にあるオブジェクトを使用する場合は，そのプロパティおよびメソッドは，スレッドセーフであることが保証されていません。つまり，プロパティへのアクセスやメソッドの実行のために使用するメモリは，ほかのスレッドの動作から保護されていない可能性があります。そのため，VCL オブジェクトと CLX オブジェクトへのアクセス用にメインスレッドが確保されています。このスレッドは，アプリケーション中のコンポーネントが受け取った Windows メッセージをすべて処理します。

すべてのオブジェクトが，プロパティへのアクセスおよびメソッドの実行を，この 1 つのスレッド内で行う場合は，オブジェクト間の干渉を心配する必要はありません。メインスレッドを使用するには，必要な動作を実行するための独立したルーチンを作成します。そして，自分のスレッドの Synchronize メソッド内から，この独立したルーチンを呼び出します。次に例を示します。

```
void __fastcall TMyThread::PushTheButton(void)
{
  Button1->Click();
}
...
void __fastcall TMyThread::Execute()
{
  ...
  Synchronize((TThreadMethod)PushTheButton);
  ...
}
```

Synchronize は，メインスレッドがメッセージループに入り，渡したメソッドが実行され終了するまで待ちます。

メモ Synchronize はメッセージループを使用するため，コンソールアプリケーションでは動作しません。コンソールアプリケーションでは，クリティカルセクションなどのほかのメカニズムを使用して VCL/CLX オブジェクトへのアクセスを保護する必要があります。

必ずしも常にメインスレッドを使う必要はありません。スレッド対応のオブジェクトもあります。オブジェクトのメソッドがスレッドセーフであることがわかっている場合は，Synchronize メソッドを使用しなくてもかまいません。Synchronize メソッドを使わない方が性能が向上します。これは，VCL スレッドまたは CLX スレッドがメッセージループに入るのを待つ必要がなくなるからです。以下のオブジェクトでは，Synchronize メソッドを使う必要はありません。

- データアクセスコンポーネントは次のとおりスレッドセーフです。BDE 対応データセットの場合，スレッドごとに各自のデータベースセッションコンポーネントを持たなければなりません。ただし，Microsoft の Access ドライバを使用する場合は例外です。Access ドライバは，Microsoft のライブラリを使って作成されています。このライブラリはスレッドセーフではありません。dbExpress の場合，ベンダーのクライアントライブラリがスレッドセーフである限り，dbExpress コンポーネントはスレッドセーフです。ADO コンポーネントと InterBase Express コンポーネントはスレッドセーフです。

  データアクセスコンポーネントを使用している場合でも，データベース対応のビジュアルコントロールを含む呼び出しはすべて Synchronize メソッドでラップしなければなりません。たとえば，データソースオブジェクトの DataSet プロパティを設定して，データコントロールをデータセットにリンクする呼び出しは，同期させる必要があります。ただし，そのデータセットのフィールド内のデータにアクセスするだけの場合は，同期させる必要はありません。

  BDE 対応アプリケーションでのスレッドを使ったデータベースセッションの使用については，24-28 ページの「複数のセッションの管理」を参照してください。

- DataCLX オブジェクトはスレッドセーフですが，VisualCLX オブジェクトはスレッドセーフではありません。
- グラフィックオブジェクトはスレッドセーフです。TFont，TPen，TBrush，TBitmap，TMetafile（VCL のみ），TDrawing（CLX のみ），または TIcon にアクセスする場合，メイン VCL/CLX スレッドを使用する必要はありません。Canvas オブジェクトは，ロックすれば，Synchronize メソッド以外で使用できます（11-8 ページの「オブジェクトをロックする」を参照）。
- リストオブジェクトはスレッドセーフではありませんが，TList のかわりにスレッドセーフなバージョンである TThreadList を使用できます。

アプリケーションのメインスレッド内で CheckSynchronize ルーチンを呼び出すと，バックグラウンドスレッドの実行をメインスレッドと同期させることができます。CheckSynchronize を呼び出す最適のタイミングは，アプリケーションがアイドルのときです（たとえば，OnIdle イベントハンドラから呼び出す）。これにより，バックグラウンドスレッドで安全にメソッドを呼び出すことができます。

スレッドローカル変数を使用する

Execute メソッド，およびそれから呼び出されるルーチンは，どれもその他の C++ ルーチンと同様にローカル変数を持ちます。また，これらのルーチンは任意のグローバル変数にアクセスできます。実際，グローバル変数は，スレッド間通信のための強力なメカニズムを提供します。

しかし，場合によっては，スレッド内で実行されるすべてのルーチンに対してはグローバルでも，同じスレッドクラスのほかのインスタンスとは共有しないような変数を使いたいこともあります。そのような場合には，スレッドローカル変数を宣言します。変数宣言に **__thread** 修飾子を追加して，変数をスレッドローカルにします。下に例を示します。

```
int __thread x;
```

上の宣言文は，アプリケーション内のスレッドごとにはプライベートですが，各スレッド内ではグローバルな整数型変数を宣言しています。

__thread 修飾子は，グローバル変数（ファイルスコープ）および静的変数にのみ使用できます。ポインタおよび関数へのポインタは，スレッド変数にはできません。AnsiStrings などのコピーオンライト手法を使用する型も，スレッド変数として動作しません。実行時に初期化または終了処理を必要とするプログラム要素は，**__thread** 型として宣言することはできません。

以下の宣言は，実行時に初期化を必要とするので無効となります。

```
int f( );
int __thread x = f( );   // 無効
```

ユーザー定義コンストラクタまたはデストラクタを使ってクラスをインスタンス化すると，実行時に初期化を必要とするので無効となります。

```
class X  {
   X( );
   ~X( );
};
X __thread myclass;   // 無効
```

ほかのスレッドによる終了をチェックする

スレッドは，Execute メソッドの呼び出しによって起動されます（11-11 ページの「スレッドオブジェクトの実行」を参照）。そして Execute が終了するまで継続します。これは，スレッドが特定のタスクを実行して，それが終了すると停止するというモデルを反映しています。しかし，アプリケーションによっては，ある外的な基準が満たされている間中，実行されるスレッドが必要な場合もあります。

スレッドを終了させるタイミングをほかのスレッドから管理することもできます。それには，Terminated プロパティをチェックします。ほかのスレッドからスレッドを終了させるには，Terminate メソッドを呼び出します。Terminate メソッドは，終了させるスレッドの Terminated プロパティを **true** にします。Terminate メソッドによる要求に正しく対応するように Execute メソッドの中で Terminated プロパティをチェックするコードを書くのはプログラマの責任です。次の例は，これを実現する方法の一例です。

```
void __fastcall TMyThread::Execute()
{
  while (!Terminated)
    PerformSomeTask();
}
```

スレッド関数で例外を処理する

Execute メソッドは，スレッドで発生する例外をすべて捕捉しなければなりません。スレッド関数で例外を捕捉できないと，アプリケーションにアクセス違反が生じることがあります。アプリケーショ

ンの開発時は，このことが明白に分からない場合があります。その理由は，IDE が例外を捕捉しても，デバッガの外でアプリケーションを実行すると，例外により実行時エラーが生じ，アプリケーションの動作が停止することがあるからです。

スレッド関数の内側で発生する例外を捕捉するには，Execute メソッドの実装部に **try**...**catch** ブロックを追加します。

```
void __fastcall TMyThread::Execute()
{
  try
  {
    while (!Terminated)
      PerformSomeTask();
  }
  catch (...)
  {
    // 例外を処理する
  }
}
```

クリーンアップコードの書き方

スレッドの実行を終了するときの後始末をするコードは，集中させることができます。スレッドがシャットダウンされる直前に，OnTerminate イベントが発生します。OnTerminate イベントハンドラ内にクリーンアップコードを置けば，Execute メソッドが従っている実行経路にかかわらず，必ずそのコードが実行されます。

OnTerminate イベントハンドラは，スレッドの一部としては実行されません。そのかわりに，アプリケーションのメイン VCL/CLX スレッドのコンテキスト内で実行されます。これには，以下に示すような 2 つの意味があります。

- OnTerminate イベントハンドラ内では，（メイン VCL/CLX スレッドの値を要求しない限り）スレッドローカル変数は一切使えない
- OnTerminate イベントハンドラからは，任意のコンポーネントおよび VCL/CLX オブジェクトに安全にアクセスできる。ほかのスレッドを破壊する心配はない

メイン VCL/CLX スレッドについての詳細は，11-4 ページの「メイン VCL/CLX スレッドを使用する」を参照してください。

スレッドの調整

スレッドが実行されるときに動作するコードを書く場合は，同時に実行されているほかのスレッドの動作を考慮しなければなりません。特に，2 つのスレッドが，同時に同じグローバルオブジェクトまたはグローバル変数を使うことのないように注意する必要があります。さらに，あるスレッド内のコードが，ほかのスレッドによって実行されるタスクの結果に依存する場合もあります。

同時アクセスの回避

グローバルオブジェクトまたはグローバル変数にアクセスする場合に，ほかのスレッドを破壊することを避けるには，そのスレッドのコードが処理を終了するまで，ほかのスレッドの実行を遮断する必要があります。ほかの実行スレッドを不必要に遮断しないように注意してください。深刻なパフォーマンスの低下を招き，マルチスレッドを使用するメリットがなくなってしまいます。

オブジェクトをロックする

オブジェクトの中には，ほかのスレッドの実行がそのオブジェクトインスタンスを使用することを抑制する組み込みのロック機能を持つものがあります。

たとえば，キャンバスオブジェクト（TCanvas およびその派生オブジェクト）は，Unlock メソッドが呼び出されるまでほかのスレッドがキャンバスにアクセスするのを防ぐ Lock メソッドを持っています。

VCL と CLX には，スレッドセーフなリストオブジェクト TThreadList もあります。TThreadList::LockList を呼び出すと，UnlockList メソッドが呼び出されるまで，ほかのスレッドはそのリストを使用できません。TCanvas::Lock または TThreadList::LockList の呼び出しは，安全にネストできます。最後のロック呼び出しが，同じスレッド内の対応するアンロック呼び出しに一致するまで，ロックは解除されません。

クリティカルセクションを使用する

オブジェクトに組み込みのロック機能がない場合は，クリティカルセクションを使用できます。クリティカルセクションは，1 度に 1 つのスレッドしか入れないゲートの役割をします。クリティカルセクションを使用するには，TCriticalSection のグローバルインスタンスを作成します。TCriticalSection には，Acquire（ほかのスレッドがセクションを実行するのを防ぐ）と Release（ブロックを取り除く）の 2 つのメソッドがあります。

各クリティカルセクションは，保護する対象のグローバルメモリと関連付けられています。それらのグローバルメモリにアクセスする各スレッドは，最初に Acquire メソッドを使用して，ほかのスレッドがそれを使用していないことを確認する必要があります。確認が終了すると，スレッドは Release メソッドを呼び出して，ほかのスレッドが Acquire を呼び出してグローバルメモリにアクセスできるようにします。

注意 クリティカルセクションが機能するのは，すべてのスレッドが，クリティカルセクションを使ってそれに関連付けられたグローバルメモリにアクセスする場合だけです。クリティカルセクションを無視して，Acquire を呼び出さずにグローバルメモリにアクセスするスレッドがある場合は，同時アクセスの問題が発生します。

たとえば，アプリケーションがグローバルなクリティカルセクション変数 pLockXY を持ち，グローバル変数 X および Y へのアクセスを遮断する場合を考えます。X または Y を使用するスレッドはすべて，次のようにクリティカルセクションへの呼び出しで囲まなければなりません。

```
pLockXY->Acquire(); // ほかのスレッドをロックアウトする
try
{
  Y = sin(X);
}
```

```
__finally
{
  pLockXY->Release();
}
```

複数読み取り時の排他書き込みシンクロナイザを使用する

クリティカルセクションを使用してグローバルメモリを保護しているときには，1 度に 1 つのスレッドしかメモリを使用できません。これはユーザーが必要とする保護よりも過剰な保護となる場合があります。特に，頻繁に読み取るけれどほとんど書き込まないオブジェクトや変数に対して効果がある場合があります。複数のスレッドが同じメモリを同時に読み取っても，書き込みを行うスレッドがない限りは危険はありません。

頻繁に読み取るけれどほとんど書き込まないグローバルメモリがある場合，TMultiReadExclusiveWriteSynchronizer を使用して保護できます。このオブジェクトはクリティカルセクションのように動作しますが，書き込みを行うスレッドがない限り，複数のスレッドが保護されているメモリを読み取ることを許可します。スレッドは，TMultiReadExclusiveWriteSynchronizer によって保護されているメモリに書き込むためには，排他的アクセスを持っていなければなりません。

複数読み取り時の排他書き込みシンクロナイザを使用するには，保護したいグローバルメモリと関連付けられている TMultiReadExclusiveWriteSynchronizer のグローバルインスタンスを作成します。このメモリから読み取りを行なうすべてのスレッドは，最初に BeginRead メソッドを呼び出さなければなりません。BeginRead は，現在ほかのスレッドがメモリに書き込みを行っていないことを確認します。スレッドが保護されているメモリからの読み取りを終了すると，EndRead メソッドを呼び出します。保護されているメモリに書き込みを行うすべてのスレッドは，最初に BeginWrite を呼び出す必要があります。BeginWrite は，現在ほかのスレッドがメモリの読み取りまたは書き込みを行っていないことを確認します。スレッドは，保護されているメモリへの書き込みを終了すると，EndWrite メソッドを呼び出します。これにより，メモリの読み取りを待っているスレッドが処理を実行できるようになります。

注意 クリティカルセクションと同様に，複数読み取り時の排他書き込みシンクロナイザが機能するのは，すべてのスレッドが，このシンクロナイザを使って関連付けられたグローバルメモリにアクセスする場合だけです。シンクロナイザを無視して，BeginRead または BeginWrite を呼び出さずにグローバルメモリにアクセスするスレッドがある場合は，同時アクセスの問題が発生します。

メモリを共有するためのその他のテクニック

VCL または CLX のオブジェクトを使用するときには，メインスレッドを使用してコードを実行します。メインスレッドを使用すると，ほかのスレッド内の VCL/CLX オブジェクトで使用されているメモリにオブジェクトが間接的にアクセスしないことを確実にします。メインスレッドについての詳細は，11-4 ページの「メイン VCL/CLX スレッドを使用する」を参照してください。

複数のスレッドがグローバルメモリを共有する必要がないときは，グローバル変数のかわりにスレッドローカル変数を使うことを考慮してみてください。スレッドローカル変数を使うと，スレッドが他のスレッドを待ったり，ロックアウトしたりする必要がなくなります。スレッドローカル変数についての詳細は，11-5 ページの「スレッドローカル変数を使用する」を参照してください。

ほかのスレッドを待つ

別のスレッドが何らかのタスクを終了するまで待たなければならない場合は，一時的にスレッドの実行を停止します。別のスレッドが完全に実行を終了するのを待つか，または別のスレッドがタスクを完了したというシグナルが出るまで待つことができます。

スレッドの実行が終了するまで待つ

別のスレッドの実行が終了するまで待つには，そのスレッドの WaitFor メソッドを使います。WaitFor は，そのスレッドが終了するまで戻りません。スレッドが終了するのは，そのスレッドの Execute メソッドが終了するか，または例外によって終了した場合です。たとえば次のコードは，別のスレッドが実行を終了するまで（スレッドリストオブジェクトを埋めるまで）待ってから，そのリストのオブジェクトにアクセスします。

```
if (pListFillingThread->WaitFor())
{
  TList *pList = ThreadList1->LockList();
  for (int i = 0; i < pList->Count; i++)
    ProcessItem(pList->Items[i]);
  ThreadList1->UnlockList();
}
```

先の例では，リスト項目にアクセスできるのは，リストが正常に埋められたことが WaitFor メソッドによって示された場合だけです。このときの戻り値は，待たせているスレッドの Execute メソッドから与えなければなりません。ただし，WaitFor を呼び出したスレッドは，Execute を呼び出したコードの実行結果ではなく，スレッドの実行結果が知りたいので，Execute メソッドが値を返すのではありません。そのかわりに，Execute メソッドは ReturnValue プロパティを設定します。ほかのスレッドが WaitFor を呼び出した場合は，WaitFor は ReturnValue を返します。この戻り値は整数です。戻り値の意味はアプリケーションが決定します。

タスクが完了するまで待つ

特定のスレッドの実行が終了するのを待つのではなく，ある処理が完了するのを待つ必要がある場合があります。それには，イベントオブジェクトを使います。Event オブジェクト（TEvent）は，すべてのスレッドから見えるシグナルのように動作するように，グローバルスコープで作成する必要があります。

ほかのスレッドが依存している操作をスレッドが完了したら，TEvent::SetEvent を呼び出します。SetEvent はシグナルをオンにします。ほかのスレッドは，このシグナルをチェックして処理が完了したことを知ります。シグナルをオフにするには，ResetEvent メソッドを使用します。

たとえば，単一のスレッドではなく複数のスレッドが実行を完了するのを待たなければならない状態を考えてみましょう。どのスレッドが最後に終了するかがわからないため，スレッドのいずれかの WaitFor メソッドを単純に使用することはできません。そのかわりに，各スレッドが終了したときにカウンタをインクリメントさせ，最後のスレッドにイベントを設定させてすべての処理が終了したことを示すシグナルを出させることができます。

次のコードは，完了しなければならないすべてのスレッド用の OnTerminate イベントハンドラの例を示します。CounterGuard は，グローバルなクリティカルセクションオブジェクトであり，複数のス

レッドが同時にカウンタを使用するのを防ぎます。Counter は，完了したスレッドの数をカウントするグローバル変数です。

```
void __fastcall TDataModule::TaskThreadTerminate(TObject *Sender)
{
  ...
  CounterGuard->Acquire();     // カウンタをロックする
  if (--Counter == 0)          // グローバルカウンタをデクリメントする
    Event1->SetEvent();        // 最後のスレッドの場合にシグナルを出す
  CounterGuard->Release();     // カウンタのロックを解放する
}
```

メインスレッドは，Counter 変数を初期化し，タスクスレッドを起動して，WaitFor メソッドを呼び出してすべてが終了したことを知らせるシグナルを待ちます。WaitFor はシグナルが設定されるまで一定の時間待ちます。そして，表 11.2 に示す値のいずれかを返します。

表 11.2　TEvent.WaitFor の戻り値

値	説明
wrSignaled	イベントのシグナルが設定された
wrTimeout	シグナルが設定されずに指定された時間が経過した
wrAbandoned	タイムアウトの時間が経過する前に，イベントオブジェクトが破棄された
wrError	待機中にエラーが発生した

次のコードは，メインスレッドがどのようにタスクスレッドを起動し，すべてが完了したときにどのように再開するかを示します。

```
Event1->ResetEvent(); // スレッドを起動する前にイベントをクリアする
for (int i = 0; i < Counter; i++)
  new  TaskThread(false);   // タスクスレッドを作成し，起動する
if (Event1->WaitFor(20000) != wrSignaled)
  throw Exception;
// すべてのタスクスレッドが終了し，メインスレッドの実行を継続する
```

メモ　指定した時間が経過した後もイベントを待ち続けたい場合は，WaitFor メソッドの引数の値に INFINITE を渡します。INFINITE 使用する場合には注意が必要です。期待したシグナルが受け取れないと，スレッドがハングアップします。

スレッドオブジェクトの実行

Execute メソッドを記述してスレッドクラスを実装したら，アプリケーションでスレッドクラスを使用して，Execute メソッドのコードを起動できます。スレッドを使用するには，まずそのスレッドクラスのインスタンスを作成します。スレッドインスタンスを作成して，すぐに実行を開始させることもできますが，一時停止の状態でスレッドを作成しておいて，Resume メソッドが呼び出されたときに実行を開始させることもできます。すぐに実行を開始するスレッドを作成するには，そのコンストラクタの CreateSuspended パラメータに **false** を設定します。たとえば，次の行はスレッドを作成してすぐに実行を開始します。

```
TMyThread *SecondThread = new TMyThread(false); // スレッドを作成して実行
```

警告 アプリケーション内にスレッドをたくさん作りすぎてはいけません。複数のスレッドを管理するために必要となるオーバーヘッドが増加し，システム全体に悪影響を与えることがあります。シングルプロセッサシステムの場合は，プロセス当たりのスレッド数を 16 個以下に抑えてください。この制限は，ほとんどのスレッドが外部イベント待ち状態であることを想定した場合です。同時に実行されるスレッドが増える場合はさらに小さい値にしなければなりません。

同じ型のスレッドのインスタンスを複数作成して，並列にコードを実行させることもできます。たとえば，あるユーザー動作に応答してスレッドのインスタンスを新たに作成し，それぞれのスレッドに対応する応答を処理させることができます。

デフォルトの優先度のオーバーライド

スレッドが使える CPU 時間の量をスレッドのタスク内で暗黙のうちに指定するには，そのコンストラクタで優先度を設定します。これについては，11-3 ページの「スレッドの初期化」で説明しています。もし，実行時に状況に応じてスレッドの優先順位を変更したい場合には，次に示すように，スレッドを一時停止状態で作成し，優先度を変更してから実行させます。

```
TMyThread *SecondThread = new TMyThread(true); // 作成するが実行しない
SecondThread->Priority = tpLower;  // 優先度を標準よりも低く設定する
SecondThread->Resume();  // ここでスレッドを実行
```

メモ クロスプラットフォームアプリケーションを作成する場合は，Windows と Linux でそれぞれ別のコードを使って優先度を割り当てる必要があります。Linux では，Priority はスレッド規則によって決まる数値で，root によってのみ変更できます。詳細については，オンラインヘルプで TThread と Priority の CLX 版を参照してください。

スレッドの停止と再開

スレッドは，その実行が終了するまでの間に，何度でも停止や再開ができます。スレッドを一時的に停止するには，Suspend メソッドを呼び出します。スレッドを安全に再開するには，Resume メソッドを呼び出します。Suspend は内部カウンタをインクリメントするので，Suspend および Resume 呼び出しをネストさせることができます。すべての Suspend 呼び出しが Resume 呼び出しと対応するまで，スレッドは再開されません。

Terminate メソッドを呼び出して，途中でスレッドの実行を終了させることもできます。Terminate メソッドは，終了させるスレッドの Terminated プロパティを **true** にします。Execute メソッドが適切に実装されていれば，定期的に Terminated プロパティをチェックして，Terminated が **true** の場合には実行を停止します。

マルチスレッドアプリケーションのデバッグ

マルチスレッドアプリケーションをデバッグする場合は，同時に実行中のすべてのスレッドのステータスを追跡しようとすると混乱します。ブレークポイントで停止したときに，どのスレッドが実行されているかを判断することですら困難です。アプリケーション中のすべてのスレッドを追跡したり操

作するには，［スレッドの状態］ボックスが役立ちます。［スレッドの状態］ボックスを表示するには，メインメニューから［表示｜デバッグ｜スレッド］を選択します。

デバッグイベント（ブレークポイント，例外，一時停止）が発生すると，スレッドの状態ビューに各スレッドのステータスが表示されます。［スレッドの状態］ボックスを右クリックすると，対応するソース位置へ移動したり，他のスレッドをカレントスレッドに設定するコマンドにアクセスできます。次のステップまたは実行操作は，カレントスレッドに対応しています。

［スレッドの状態］ボックスには，アプリケーション中のすべてのスレッドの ID のリストが表示されます。スレッドオブジェクトを使用している場合は，スレッド ID が ThreadID プロパティの値になります。スレッドオブジェクトを使用していない場合は，各スレッドのスレッド ID は CreateThread または BeginThread への呼び出しからの戻り値になります。

［スレッドの状態］ボックスの詳細については，オンラインヘルプを参照してください。

スレッドに名前を付ける

［スレッドの状態］ボックスでスレッドを追跡するとき，スレッド ID でスレッドを識別するのは困難です。このため，スレッドクラスに名前を付けると便利です。［スレッドオブジェクトの作成］ダイアログボックスでスレッドクラスを作成するとき，クラス名を入力するだけでなく，［名前付きスレッド］にチェックを付けてスレッド名を入力し，［OK］をクリックします。

スレッドクラスに名前を付けると，SetName というメソッドがスレッドクラスに追加されます。スレッドが実行し始めると，最初に SetName メソッドを呼び出します。

CLX　VCL アプリケーションでのみ，スレッドに名前を付けることができます。

名前のないスレッドを名前付きのスレッドに変換する

名前のないスレッドを名前付きのスレッドに変換することができます。たとえば，C++Builder 5 以前のバージョンで作成したスレッドクラスがある場合，以下の手順で名前付きのスレッドに変換できます。

1. スレッドクラスに SetName メソッドを追加します。

```
//---------------------------------------------------------------------------
void TMyThread::SetName()
{
        THREADNAME_INFO info;
        info.dwType = 0x1000;
        info.szName = "MyThreadName";
        info.dwThreadID = -1;
        info.dwFlags = 0;

        __try
        {
              RaiseException( 0x406D1388, 0, sizeof(info)/sizeof(DWORD),(DWORD*)&info );
        }
        __except (EXCEPTION_CONTINUE_EXECUTION)
        {
        }
}
//---------------------------------------------------------------------------
```

メモ　info.szName にスレッドクラスの名前を設定します。

デバッガは例外を確認し，構造体内に指定されているスレッド名を調べます。デバッグ時には，［スレッドの状態］ボックスのスレッド ID フィールドにスレッド名が表示されます。

2. スレッドの Execute メソッドの先頭に，この SetName メソッドへの呼び出しを追加します。

```
//---------------------------------------------------------------------------
void __fastcall TMyThread::Execute()
{
        SetName();
        // ---- ここに既存の Execute メソッドを置く ----
}
//---------------------------------------------------------------------------
```

類似スレッドに個別の名前を割り当てる

同じスレッドクラスに属するスレッドインスタンスは，すべて同じ名前になります。しかし，実行時に各スレッドインスタンスに個別の名前を割り当てることができます。手順は次のとおりです。

1. スレッドクラスに ThreadName プロパティを追加します（クラス定義に以下を追加する）。

```
__property AnsiString ThreadName = {read=FName, write=FName};
```

2. SetName メソッドで，次のように変更します。

```
info.szName = "MyThreadName";
```

を以下に変更

```
info.szName = ThreadName;
```

3. スレッドオブジェクトを作成するとき，次のようにします。
   1. スレッドを一時停止の状態で作成します。11-11 ページの「スレッドオブジェクトの実行」を参照してください。
   2. MyThread.ThreadName="SearchForFiles"; のように名前を割り当てます。
   3. スレッドを再開します。11-12 ページの「スレッドの停止と再開」を参照してください。

第12章

例外処理

C++Builder は，C++ 例外処理，C ベースの構造化例外処理，および VCL/CLX 例外処理をサポートしています。

ANSI 標準 C++ 例外および VCL タイプの例外を生成できます。VCL タイプの例外には，組み込みのエラー処理ルーチンが含まれています。

さらに C ベースの Win32 構造化例外もサポートしているので，作成したコードは，Windows OS が生成した例外にも適切に対応できます。

C++ 例外処理

例外とは，特別な処理が必要な例外的な状況を言います。その中には，ゼロによる除算やメモリ不足など，実行時に発生するエラーも含まれます。例外処理は，予測できる問題と予測できない問題の両方を発見し，エラーを処理するための標準的な方法を提供します。開発者は，例外処理を利用して，バグの認識，追跡，および修正ができます。

エラーが発生すると，プログラムは例外を送出（throw）します。送出された例外は，通常，何が起こったかについての情報を格納しています。これにより，プログラムの別の部分で例外の原因を診断することができます。

プログラムに例外発生の準備をさせるには，例外を送出する可能性のある文を try ブロックに記述します。ここで記述した文のどれかが実際に例外を送出すると，制御が例外ハンドラに移り，例外ハンドラはこの種の例外を処理します。例外ハンドラが例外を処理することを「例外を捕捉（catch）する」と言います。例外ハンドラには，プログラムを終了する前に実行すべきアクションを指定します。

例外処理構文

例外処理には，**try**，**throw**，および **catch** の 3 つのキーワードを使う必要があります。キーワード **throw** は，例外を生成させるのに使います。**try** ブロックには，例外を送出する可能性のある文を記

述します。**try** ブロックのあとに，**catch** 文を 1 つ以上置きます。**catch** 文 1 つ 1 つが個々の例外を処理します。

メモ C プログラムでは，キーワード **try**，**catch**，および **throw** は使用できません。

try ブロック

try ブロックには，例外を送出する可能性のある文を 1 つ以上記述します。**throw** 文が実行されると，プログラムは例外を送出します。一般に，**throw** 文は関数内に置かれます。次に例を示します。

```
void SetFieldValue(DF *dataField, int userValue)
{
    if ((userValue < 0) || userValue > 10)
      throw EIntegerRange(0, 10, userValue);
      . . .
}
```

プログラムの別の部分が，送出された例外オブジェクトを捕捉し，それに対応した処理を行います。次に例を示します。

```
try
{
    SetFieldValue(dataField, userValue);
}
catch (EIntegerRange &rangeErr)
{
    printf("Expected value between %d and %d, but got %d \ n",
         rangeErr.min, rangeErr.max, rangeErr.value);
}
```

上の例では，SetFieldValue 関数が入力パラメータを無効と判断した場合に，例外を送出してそのことを知らせます。SetFieldValue が送出した例外を，**try**/**catch** ブロック（SetFieldValue を囲む部分）で捕捉し，printf 文を実行します。例外がまったく送出されなければ，printf 文は実行されません。

try で指定された **try** ブロックの直後には，**catch** で指定されたハンドラを続けなければなりません。**try** ブロックは，プログラム実行時の制御の流れを指定する文です。**try** ブロックで例外が送出されると，プログラムの制御は適切な例外ハンドラに移されます。

例外ハンドラは，例外を処理するように設計されたコードブロックです。C++ 言語では，**try** ブロックのすぐ後ろに，少なくとも 1 つのハンドラが必要です。プログラムには，プログラムが生成する例外ごとにハンドラが 1 つずつ必要です。

throw 文

throw 文は，さまざまな種類のオブジェクトを送出できます。C++ のオブジェクトは通常，値，参照，ポインタのいずれかで送出されます。次に例を示します。

```
// 値または参照で捕捉するオブジェクトを送出
throw EIntegerRange(0, 10, userValue);

// ポインタで捕捉するオブジェクトを送出
throw new EIntegerRange(0, 10, userValue);
```

次の 2 つの例は，主に **throw** 文の基準を示す目的で特徴を示したものです。実際は，もっと分かりやすい例外を送出するのがベターです。整数型などの組み込み型を送出する特殊なケースもあります。また，ポインタで例外を送出するのはできるだけ避けるようにしてください。

```
// 整数を送出する
throw 1;

// char* を送出する
throw "foo";
```

たいていは，参照（特に **const** 参照）で例外を捕捉することになります。オブジェクトを値で捕捉する場合は，注意を要することがあります。オブジェクトを値で捕捉するには，オブジェクトを **catch** パラメータに割り当てる前に，オブジェクトをコピーしなければなりません。ユーザーがコピーコンストラクタを指定しているとそれが呼び出されるので，効率が悪くなることがあります。

catch 文

catch 文にはいくつかの形があります。オブジェクトは値，参照，ポインタのいずれかで捕捉できます。加えて，**const** 修飾子を **catch** パラメータに適用することもできます。1 つの **try** ブロックに複数の **catch** 文を置けるので，ブロック 1 つでさまざまな例外を捕捉できます。また，送出される可能性のある例外 1 つ 1 つに対してそれぞれ **catch** 文を記述します。次に例を示します。

```
try
   CommitChange(dataBase, recordMods);
catch (const EIntegerRange &rangeErr)
   printf("Got an integer range exception");
catch (const EFileError &fileErr)
   printf("Got a file I/O error");
```

CommitChange 関数が複数のサブシステムを使用していて，そのサブシステムから種々の例外が送出される可能性がある場合は，各タイプの例外を別々に処理することもできます。1 つの **try** 文に複数の **catch** 文を置くと，各タイプの例外ごとにハンドラを割り当てることができます。

例外オブジェクトが何らかの基本クラスから派生している場合には，派生した例外に対する専用ハンドラだけでなく，基本クラス用の汎用ハンドラも追加できます。そのためには，例外が送出されたときの検索順に **catch** 文を置きます。たとえば下のコード例では，最初に EIntegerRange を処理し，次に ERange（EIntegerRange の派生元）を処理します。

```
try
   SetFieldValue(dataField, userValue);
catch (const EIntegerRange &rangeErr)
   printf("Got an integer range exception");
catch (const ERange &rangeErr)
   printf("Got a range exception");
```

try ブロックを通り越して送出される可能性のある例外をすべてハンドラで捕捉するには，**catch**(...) という特殊な形を使います。**catch**(...) は，どんな例外でも例外ハンドラを起動するように例外処理システムに指示します。次に例を示します。

```
try
   SetFieldValue(dataField, userValue);
catch (...)
   printf("Got an exception of some kind");
```

例外の再送出

例外ハンドラが例外を処理したあと，その例外をもう一度送出するか，または別の例外を送出することもあります。

例外ハンドラに現在の例外を再送出させるには，パラメータのない **throw** 文を置きます。これで，コンパイラ / ランタイムライブラリ に対して，現在の例外オブジェクトを取り出して再送出するように指示したことになります。次に例を示します。

```
catch (EIntegerRange &rangeErr)
{
    // 例外に対してローカルな処理をするコードをここに記述
    throw;  // この例外を再送出する
}
```

例外ハンドラに別の例外を送出させるには，通常の方法で **throw** 文を記述します。

例外指定

関数から送出される可能性のある例外を指定しておくことができます。関数を通り越して間違った型の例外を送出すると，実行時エラーになります。例外指定（exception specification）の構文は次のとおりです。

```
exception-specification:
   throw (type-id-list)    // type-id-list は任意
   type-id-list:
      type-id
      type-id-list, type-id
```

次に，例外指定のある関数の例を示します。

```
void f1();                // 関数はあらゆる例外を送出できる
void f2() throw();        // 例外を送出しない
void f3() throw( A, B* ); // A を pubic 基本クラスとする例外,
                          // または，B を pubic 基本クラスとするポインタを送出できる
```

このような関数の定義とすべての宣言には，同じ型 ID のセットを含む例外指定がなければなりません。関数が例外指定にない例外を送出すると，プログラムは unexpected を呼び出します。

例外指定には以下の短所があるので，使用を避けた方がよい場合もあります。

第 1 に，関数の例外指定を追加するため，Windows に実行時のパフォーマンスヒットが生じます。

第 2 に，実行時に予期せぬエラーが発生することがあります。たとえば，システムに例外指定が指定され，その実装部に別のサブシステムが使われているとします。サブシステムから別のタイプの例外を送出するようにサブシステムを変更したとします。このサブシステムの新しいコードを使うと，コンパイラからは何も指摘されないまま，実行時エラーが発生する可能性があります。

第 3 に，仮想関数を使用した場合，プログラム設計に影響を与える可能性があります。理由は，例外指定が関数型の 1 つとみなされないためです。たとえば次のコードでは，派生クラス BETA::vfunc を定義して，元の関数宣言から逸脱するような例外を一切送出しないようにしています。

```
class ALPHA {
public:
  struct ALPHA_ERR {};
  virtual void vfunc(void) throw (ALPHA_ERR){}; // 例外指定
};
class BETA : public ALPHA {
  void vfunc (void) throw(){}  // 例外指定の変更
};
```

例外の解放

例外が送出されると，ランタイムライブラリ（RTL）が送出オブジェクトを受け取り，そのオブジェクトの型を取得します。そして，呼び出しスタックを上方向に調べ，送出されたオブジェクトの型と一致する型のハンドラを探します。ハンドラが見つかると，RTL はハンドラのポイントまでスタックを解放し，そのハンドラを実行します。

スタックの解放プロセス中，RTL は，例外の送出場所から捕捉場所の間にあるスタックフレーム内の全ローカルオブジェクトのデストラクタを呼び出します。スタックの解放中にデストラクタによって例外が生成されて，それが処理できない場合は，terminate が呼び出されます。デストラクタはデフォルトで呼び出されますが，**-xd** コンパイラオプションを使ってデフォルトの設定を無効にすることもできます。

安全なポインタ

オブジェクトを指すポインタであるローカル変数がある場合，例外が送出されても，ポインタは自動的には削除されません。それは，コンパイラから見て，この関数だけのために割り当てられたポインタと，他のポインタを区別する良い方法がないからです。ローカル使用に割り当てたオブジェクトを例外発生時に確実に破棄するには，auto_ptr というクラスを使用できます。次の例のように，関数内で割り当てられているポインタのためにメモリを解放する，という特殊なケースがあります。

```
TMyObject *pMyObject = new TMyObject;
```

この例では，TMyObject のコンストラクタが例外を送出した場合，RTL が例外を解放した時点で，RTL は，TMyObject で割り当てられているオブジェクトを指すポインタを削除します。このタイミングが唯一，プログラマに代わってコンパイラが自動的にポインタを削除するときです。

例外処理におけるコンストラクタ

クラスのコンストラクタは，オブジェクトを正常に構築できなかった場合，例外を送出できます。コンストラクタが例外を送出した場合は，そのオブジェクトのデストラクタは必ずしも呼び出されません。基本クラスと，**try** ブロックに入った時点から後でクラス内で作成が完了しているオブジェクトに限って，デストラクタが呼び出されます。

捕捉されない例外と予期せぬ例外の処理

例外が送出されても例外ハンドラが見つからない場合（つまり，例外が捕捉されない場合），プログラムは何らかの終了関数を呼び出します。独自の終了関数を指定するには，set_terminate を使います。終了関数を指定しないと，terminate 関数が呼び出されます。下のコード例では，どのハンドラにも捕捉されない例外を，my_terminate 関数を使って処理しています。

```
void SetFieldValue(DF *dataField, int userValue)
{
  if ((userValue < 0) || (userValue) > 10));
    throw EIntegerRange(0, 10, userValue);
  . . .
}
```

```
void my_terminate()
{
  printf("Exception not caught");
  abort();
}
// my_terminate() を終了関数として設定
set_terminate(my_terminate);
// SetFieldValue を呼び出す。ユーザー値は 10 より大きいので，例外が生成される
// 呼び出しが try ブロックにないので，my_terminate が呼び出される
SetFieldValue(DF, 11);
```

送出する例外を指定した関数が予期せぬ例外を送出した場合，予期せぬ例外を処理する関数が呼び出されます。独自の関数を指定するには，set_unexpected を使います。自分で関数を指定しないと，unexpected 関数が呼び出されます。

Win32 における構造化例外

Win32 は C++ の例外と類似の C ベースの構造化例外をサポートしています。ただし，重要な違いがあるので，例外に対応した C++ のコードと併用する場合には，注意が必要です。

C++Builder アプリケーションで処理する構造化例外を使う場合，次のことに注意してください。

- C の構造化例外は C++ プログラムで使うことができる。
- C++ 例外は C プログラムで使うことができない。C++ 例外ではハンドラを **catch** キーワードで指定する必要があるが，C プログラムでは **catch** は使えない。
- RaiseException 関数の呼び出しによって生成された例外は，C++ の場合は **try**/**__except**，C の場合は **__try**/**__except** ブロックによって処理される（**try**/**__finally** または **__try**/**__finally** ブロックも使用できる。12-7 ページの「構造化例外の構文」を参照）。RaiseException が呼び出された場合は，**try**/**catch** ブロックのすべてのハンドラは無視される。
- アプリケーションによって処理されなかった例外は，オペレーティングシステムに渡される（通常は，その結果としてプロセスが終了する）。terminate() が呼び出されるわけではない。
- 例外ハンドラが例外オブジェクトを要求しない限り，例外ハンドラは例外オブジェクトのコピーを受け取らない。

C または C++ のプログラムで，次の C 例外ヘルパー関数を使うことができます。

- GetExceptionCode
- GetExceptionInformation
- SetUnhandledExceptionFilter
- UnhandledExceptionFilter

C++Builder は，UnhandledExceptionFilter 関数の使用を **__try**/**__except** または **try**/**__except** ブロックの例外フィルタに限定してはいません。ただし，**__try**/**__except** または **try**/**__except** ブロックの外でこの関数が呼び出された場合は，プログラムの動作は未定義になります。

CLX　Linux で動作している CLX アプリケーションは，C/C++ の構造化例外をサポートしません。

構造化例外の構文

C プログラムでは，構造化例外を実現するために使う ANSI 準拠のキーワードは，**__except**，**__finally**，および **__try** です。

メモ **__try** キーワードが使えるのは，C プログラムの中だけです。移植性のあるコードを書きたい場合は，C++ プログラムの中で構造化例外処理を使わないでください。

「try して，例外が起きたら処理をする」場合の構文

「try して，例外が起きたら処理をする」場合の構文は，次のとおりです。

```
try ブロック：
    __try 複合文      (C モジュールの場合 )
    try 複合文      (C++ モジュールの場合 )
ハンドラ：
    __except ( 式 ) 複合文
```

「try して，最後に終了処理をする」場合の構文

「try して，最後に終了処理をする」場合の構文は，次のとおりです。

```
try ブロック：
    __try 複合文      (C モジュールの場合 )
    try 複合文      (C++ モジュールの場合 )
終了：
    __finally 複合文
```

構造化例外の処理

構造化例外は，C++ 例外処理の拡張を使って処理できます。

```
try {
        foo();
    }
    __except(__expr__) {
        // ここにハンドラを記述する
    }
```

__expr__ の評価結果は，次の 3 つの値のいずれかになります。

値	説明
EXCEPTION_CONTINUE_SEARCH(0)	ハンドラに入っていない。OS が例外ハンドラの検索を継続中
EXCEPTION_CONTINUE_EXECUTION(-1)	例外の場所で実行を継続中
EXCEPTION_EXECUTE_HANDLER(1)	ハンドラに入った。デストラクタによるクリーンアップを有効にして（**-xd**，デフォルトでオン）コードをコンパイルした場合は，例外の場所と例外ハンドラの間で作成されたローカルオブジェクトのデストラクタが，スタックの解放時に呼び出される。スタックの解放は，ハンドラに入る前に完了している

Win32 には，アクティブな例外の情報を問い合わせるために使用できる関数として，GetExceptionCode() と GetExceptionInformation() の 2 つがあります。上の「フィルタ」式の一部とし

て関数を呼び出す場合は，次のように **__except()** のコンテキスト内から直接これらの関数を呼び出さなければなりません。

```
#include <Windows.h>
#include <excpt.h>
int filter_func(EXCEPTION_POINTERS *);
...
EXCEPTION_POINTERS *xp = 0;
    try {
        foo();
    }
    __except(filter_func(xp = GetExceptionInformation())) {
        //...
    }
```

あるいは，次の例のようにカンマ演算子を使って，代入を関数呼び出しの中にネストします。

```
__except((xp = GetExceptionInformation()), filter_func(xp))
```

例外フィルタ

フィルタ式ではフィルタ関数を呼び出すことができます。ただし，フィルタ関数では GetExceptionInformation を呼び出すことができません。そこで，GetExceptionInformation の戻り値をフィルタ関数のパラメータとして渡します。

EXCEPTION_POINTERS 情報を例外ハンドラに渡すには，フィルタ式またはフィルタ関数を使って，そのポインタまたはデータを，GetExceptionInformation から，例外ハンドラが後でアクセスできる場所にコピーしなければなりません。

ネストされた **try-except** 文の場合は，各文のフィルタ式は，EXCEPTION_EXECUTE_HANDLER または EXCEPTION_CONTINUE_EXECUTION の位置になるまで評価されます。フィルタ式では GetExceptionInformation を呼び出して，例外情報を取得できます。

__except に渡される式の中で直接 GetExceptionInformation または GetExceptionCode が呼び出される限りは，関数を使って例外の処理方法を決定できます。複雑な C++ の式を作成する必要はありません。例外を処理するために必要な情報は，ほとんどすべて GetExceptionInformation() から抽出できます。GetExceptionInformation() は EXCEPTION_POINTERS 構造体へのポインタを返します。

```
struct EXCEPTION_POINTERS {
        EXCEPTION_RECORD *ExceptionRecord;
        CONTEXT *Context;
    };
```

EXCEPTION_RECORD には，機種に依存しない状態が含まれています。

```
struct EXCEPTION_RECORD {
        DWORD ExceptionCode;
        DWORD ExceptionFlags;
        struct EXCEPTION_RECORD *ExceptionRecord;
        void *ExceptionAddress;
        DWORD NumberParameters;
        DWORD ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
    };
```

典型的なフィルタ関数は，ExceptionRecord 内の情報を見て，応答のしかたを決定します。もっと特殊な情報が必要な場合もあります。特に，とるべきアクションが EXCEPTION_CONTINUE_EXECUTION の場合は，何もしなければ，例外が発生したコードが再度実行されます。このような場合は，EXCEPTION_POINTERS 構造体の別のフィールドから，例外が発生したときのプロセッサの状態を取得できます。この構造体が変更されて，フィルタが EXCEPTION_CONTINUE_EXCEPTION を返した場合は，実行を継続する前に，これを使ってスレッドの状態を設定します。次に例を示します。

```
static int xfilter(EXCEPTION_POINTERS *xp)
    {
        int rc;

        EXCEPTION_RECORD *xr = xp->ExceptionRecord;
        CONTEXT *xc = xp->Context;

        switch (xr->ExceptionCode) {
            case EXCEPTION_BREAKPOINT:
                // なんと，ブレークポイントが埋め込まれていた
                // 単純にブレークポイントを飛び越す（x86 の場合は 1 バイト）
                ++xc->Eip;
                rc = EXCEPTION_CONTINUE_EXECUTION;
                break;

            case EXCEPTION_ACCESS_VIOLATION:
                rc = EXCEPTION_EXECUTE_HANDLER;
                break;

            default:
                // あきらめる
                rc = EXCEPTION_CONTINUE_SEARCH;
                break;
        };

        return rc;
    }
    ...

    EXCEPTION_POINTERS *xp;

    try {
        func();
    }
    __except(xfilter(xp = GetExceptionInformation())) {
        abort();
    }
```

C++ と構造化例外を併用する

構造化例外を C++ プログラムで使用する場合は，いくつかの問題に注意する必要があります。まず第 1 に，C++Builder は Win32 の構造化例外を使って，C++ 例外を実装しています。したがって，C++ 例外は **__except** に対して透過的です。

try ブロックの後には，ただ 1 つの **except** ブロック，または少なくとも 1 つの **catch** ブロックを続けることができます。この 2 つを混在させると，コンパイルエラーになります。両方の種類の例外を処理する必要があるコードは，次に示すように，2 つの **try** ブロックを単純にネストさせます。

```
try {
        EXCEPTION_POINTERS *xp;

        try {
            func();
        }
        __except(xfilter(xp = GetExceptionInformation())) {
            //...
        }
    }
    catch (...) {
        //...
    }
```

関数の **throw()** 指定は，Win32 例外に関連したプログラムの動作には影響しません。また，処理されなかった例外は，最終的にオペレーティングシステムによって処理されます（ただし，デバッガによって先に例外が処理されなかった場合）この点が，terminate() を呼び出す C++ プログラムとは異なります。

-xd コンパイラオプション（デフォルトでオン）でコンパイルしたモジュールは，**auto** で確保されたすべてのオブジェクトのデストラクタを呼び出します。例外が送出された場所から，例外が捕捉された場所までスタックが解放されます。

C++ プログラムの C ベースの例外の例

```
/* プログラムの結果：
Another exception:
Caught a C-based exception.
Caught C++ exception[Hardware error: Divide by 0]
C++ allows __finally too!
*/
#include <stdio.h>
#include <string.h>
#include <windows.h>

class Exception
{
  public:
  Exception(char* s = "Unknown"){ what = strdup(s);      }
  Exception(const Exception& e ){ what = strdup(e.what); }
 ~Exception()                   { delete[] what;         }
  char* msg() const             { return what;           }
private:
  char* what;
};

int main()
{
  float e, f, g;
  try
  {
```

```
    try
    {
      f = 1.0;
      g = 0.0;
      try
      {
        puts("Another exception:");
        e = f / g;
      }
      __except(EXCEPTION_EXECUTE_HANDLER)
      {
        puts("Caught a C-based exception.");
        throw(Exception("Hardware error: Divide by 0"));
      }
    }
    catch(const Exception& e)
    {
      printf("Caught C++ Exception: %s :\n", e.msg());
    }
  }
  __finally
  {
    puts("C++ allows __finally too!");
  }
  return e;
}
```

例外の定義

同じプログラム内で処理される例外を生成する場合に，Win32 例外を生成するのは，一般にあまり有意義ではありません。この場合は，移植性を高く保つことができ，構文も単純な C++ 例外の方が優れています。ただし，異なるコンパイラでコンパイルされた複数のコンポーネントの間で例外を処理する場合には，Win32 例外の方が有利です。

まず最初に，例外を定義します。例外は，次のような形式（ビット 0 から始まる）の 32 ビットの整数にします。

ビット	説明
31-30	11 = エラー（標準） 00 = 成功 01 = 情報提供 10 = 警告
29	1 = ユーザー定義
28	予備
27-0	ユーザー定義

例外コードを定義することのほかに，例外に付加情報（例外レコードからフィルタまたはハンドラにアクセス可能）を含めるかどうかも決定する必要があります。付加情報のパラメータを例外コードにエンコードするために，決まりきった方法はありません。詳細は，C++Builder オンラインヘルプの Win32 の関連トピックを参照してください。

例外の生成

Win32 例外は，RaiseException() 呼び出しによって生成されます。これは，次のように宣言します。

```
void RaiseException(DWORD ec, DWORD ef, DWORD na, const DWORD *a);
```

ここでの引数の内容は次のとおりです。

ec	例外コード
ef	例外フラグ。0 または EXCEPTION_NONCONTINUABLE （例外が継続不能となっているにもかかわらず，フィルタがそれを継続しようとすると，EXCEPTION_NONCONTINUABLE_EXCEPTION が生成される）
na	引数配列の要素数
a	引数配列の先頭の要素へのポインタ。これらの引数の意味は，特定の例外に依存する

終了ブロック

構造化例外処理モデルは「終了ブロック」をサポートしています。終了ブロックは，囲まれたブロックが正常に終了したか，例外によって終了したかにかかわらず，実行されます。C++Builder のコンパイラでは，これを次のような C の構文でサポートします。

```
__try {
        func();
    }
    __finally {
        // func() が例外を生成するかどうかにかかわらず実行される
    }
```

終了ブロックは C++ の拡張でもサポートされています。この場合は，**__finally** ブロックでクリーンアップ処理ができます。

```
try {
        func();
    }
    __finally {
        // func() が例外を生成するかどうかにかかわらず実行される
    }
```

次の例は，終了ブロックを示しています。

```
/* プログラムの結果：
An exception:
Caught an exception.
The __finally is executed too!
No exception:
No exception happened, but __finally still executes!
*/
#include <stdio.h>
#include <windows.h>

int main()
{
  float e, f, g;
```

```
try
{
  f = 1.0;
  g = 0.0;
  try
  {
    puts("An exception:");
    e = f / g;
  }
  __except(EXCEPTION_EXECUTE_HANDLER)
  {
    puts("Caught an exception.");
  }
}
__finally
{
  puts("The __finally is executed too!");
}
try
{
  f = 1.0;
  g = 2,0;
  try
  {
    puts("No exception:");
    e = f / g;
  }
  __except(EXCEPTION_EXECUTE_HANDLER)
  {
    puts("Caught an exception.");
  }
}
__finally
{
  puts("No exception happened, but __finally still executes!");
}
return e;
}
```

C++ のコードでも，「終了ブロック」を処理できます。ローカルオブジェクトを作成したら，スコープを抜けるときに，これらのオブジェクトのデストラクタが呼び出されます。C++Builder の構造化例外は，デストラクタによるクリーンアップをサポートしているので，これは，生成された例外の型に関係なく動作します。

メモ 例外が発生したが，それをプログラムで処理できなかった場合に起こることに関しては，特別な場合があります。C++ 例外に対しては，C++Builder のコンパイラはローカルオブジェクトのデストラクタを呼び出します（言語の定義で要求されているわけではありません）。一方，Win32 例外が処理されなかった場合は，デストラクタによるクリーンアップは実行されません。

C++Builder の例外処理オプション

C++Builder コンパイラの例外処理オプションを以下に示します。

表 12.1 例外処理のコンパイラオプション

コマンドライン スイッチ	説明
-x	C++ 例外処理を有効にする
-xd	デストラクタによる後片付けを有効にする。例外が送出された場合，**catch** 文と **throw** 文のスコープの間で自動的に宣言されたすべてのオブジェクトのデストラクタが呼び出される（デフォルトはオン）
-xp	例外位置情報を有効にする。例外が発生した場所のソースコード番号が提供されるので，実行時に例外を識別できる。これによりプログラムは，except.h で定義されているグローバル変数 __ThrowFileName と __ThrowLineNumber を使って，C++ 例外が発生した場所のファイルと行番号を問い合わせることができる

VCL/CLX 例外処理

アプリケーションで VCL コンポーネントか CLX コンポーネントを使用する場合は，VCL/CLX 例外処理メカニズムについて理解する必要があります。たくさんのクラスに例外が組み込まれていて，何か予期せぬ事態が発生したときには，自動的に例外が送出されるからです。アプリケーション側で例外を処理しない場合は，VCL/CLX がデフォルトの方法で例外を処理します。通常は，発生したエラーの種類を説明するメッセージを表示します。

例外が発生し，送出された例外の種類を示すメッセージが表示された場合は，オンラインヘルプの例外クラスを調べてください。エラーが発生した場所と，その原因を突き止めるのに役立つ情報が載っている場合がよくあります。

さらに，第 13 章「VCL と CLX のための C++ 言語サポート」では，例外の原因となり得る微妙な言語間の違いについて説明しています。13-13 ページの「コンストラクタから送出された例外」には，オブジェクトの構築中に例外が送出された場合にどうなるかを示す例があります。

C++ と VCL/CLX 例外処理の違い

C++ と VCL/CLX 例外処理機構には注意すべき違いがいくつかあります。

コンストラクタから送出された例外：

- C++ のデストラクタは，完全に構築されたメンバーと基本クラスに対して呼び出される
- VCL/CLX の基本クラスのデストラクタは，オブジェクトまたは基本クラスが完全には構築されていない場合でも呼び出される

捕捉，送出される例外：

- C++ 例外は，参照，ポインタ，または値によって捕捉される。TObject から派生した VCL/CLX 例外だけが参照またはポインタによって捕捉される。TObject 例外を値で捕捉しようとするとコンパ

イルエラーとなる。EAccessViolation のようなハードウェアやオペレーティングシステムの例外は参照によって捕捉されなくてはならない

- VCL/CLX 例外は参照によって捕捉される
- VCL コード内または CLX コード内で捕捉された例外を，**throw** を使って再送出することはできない

オペレーティングシステム例外の処理

C++Builder では，オペレーティングシステムが送出した例外を処理することができます。オペレーティングシステム例外には，アクセス違反，整数演算エラー，浮動小数点演算エラー，スタックオーバーフロー，および【Ctrl】+【C】割り込みが含まれます。これらは，アプリケーションにディスパッチされる前に，C++ RTL で処理され，VCL/CLX 例外クラスのオブジェクトに変換されます。したがって，C++ コード内では以下のようなコードを記述することができます。

```
try
{
  char * p = 0;
  *p = 0;
}
// 必ず参照によって捕捉する
catch (const EAccessViolation &e)
{
  printf("You can't do that!\n");
}
```

C++Builder が使用しているクラスは Delphi が使用しているクラスそのものであり，C++Builder VCL/CLX アプリケーションでしか使用できません。これらは TObject から派生しており，VCL と CLX によるサポートを必要とします。

C++Builder の例外処理機構の特徴を以下に示します。

- 例外オブジェクトの解放については責任は**ない**
- オペレーティングシステム例外は，参照によって捕捉しなければならない
- オペレーティングシステム例外の捕捉フレームがすでに存在し，その VCL/CLX 捕捉フレームの介在によって例外がすでに捕捉されている場合には，その例外を再送出することはできない
- オペレーティングシステム例外の捕捉フレームがすでに終了し，その捕捉フレームの介在によって例外がすでに捕捉されている場合には，その例外を再送出することはできない

最後の 2 つの事項をまとめて言うと次のようになります。オペレーティングシステム例外が C++ 例外として捕捉された場合，捕捉を行うスタックフレームの中からでなければ，その例外をオペレーティングシステム例外か VCL/CLX 例外であるかのように再送出することはできません。

VCL/CLX 例外の処理

C++Builder は，例外処理のセマンティクスを拡張して，VCL/CLX から送出されるソフトウェア例外，あるいはそれと同等な，TObject から派生した例外クラスから送出される C++ のソフトウェア例

外を処理できるようになっています。このような例外を処理する場合，VCL 形式のクラスはヒープ上にしか割り当てられないことから生じる規則がいくつかあります。

- VCL 形式の例外クラスだけが，ポインタによって捕捉でき，ソフトウェアの例外の場合は，参照によって捕捉できる（参照が推奨される）
- VCL 形式の例外は「値渡し」構文で送出しなくてはならない

VCL/CLX の例外クラス

C++Builder には，大規模な一連の組み込みの例外クラスがあります。これらのクラスは，ゼロによる除算，ファイル入出力エラー，無効な型キャスト，その他の例外条件を自動的に処理します。VCL/CLX の例外クラスは，すべて Exception という 1 つのルートオブジェクトから派生しています。Exception は，VCL 型のすべての例外に必要な基本的なプロパティおよびメソッドをカプセル化して，アプリケーションが例外を処理する際に一貫性のあるインターフェイスを提供します。

Exception 型のパラメータをとる **catch** ブロックに例外を渡すことができます。VCL/CLX 例外を捕捉するには，次の構文を使います。

```
catch (exception_class &exception_variable)
```

例外クラスには，捕捉したい例外と，その例外を参照するために必要な変数を指定します。

次のコードは，VCL/CLX 例外を送出する方法の例です。

```
void __fastcall TForm1::ThrowException(TObject *Sender)
{
  try
  {
    throw Exception("An error has occurred");
  }
  catch(const Exception &E)
  {
    ShowMessage(AnsiString(E.ClassName())+ E.Message);
  }
}
```

前の例の **throw** 文は，Exception クラスのインスタンスを作成して，そのコンストラクタを呼び出します。Exception から派生した例外は，すべてメッセージを表示します。このメッセージは，コンストラクタを介して渡されて，Message プロパティから取得できます。

表 12.2 に，主要な VCL/CLX 例外クラスを示します。

表 12.2　主要な例外クラス

例外クラス	説明
EAbort	エラーメッセージのダイアログボックスを表示せずに，イベントのシーケンスを停止する
EAccessViolation	無効なメモリアクセスエラーをチェックする
EBitsError	論理型配列への無効なアクセスを防止する
EComponentError	コンポーネントの無効な登録および名前変更を知らせる
EConvertError	文字列またはオブジェクトの変換エラーを表す
EDatabaseError	データベースアクセスエラーを表す

表 12.2 主要な例外クラス（つづき）

例外クラス	説明
EDBEditError	指定されたマスクと互換性のないデータを捕捉する
EDivByZero	ゼロによる除算エラーを捕捉する
EExternalException	認識できない例外コードを表す
EInOutError	ファイル入出力エラーを表す
EIntOverflow	整数型の計算結果が，割り当てられているレジスタより大きくなったことを表す
EInvalidCast	不正な型キャストをチェックする
EInvalidGraphic	認識できないグラフィックファイル形式を処理しようとしたことを表す
EInvalidOperation	コンポーネントに対して不正な操作を行おうとしたときに発生する
EInvalidPointer	不正なポインタ操作の結果発生する
EMenuError	メニュー項目に関する問題が含まれる
EOleCtrlError	ActiveX コントロールに関連した問題を検出する
EOleError	OLE オートメーションエラーを表す
EPrinterError	印刷エラーを知らせる
EPropertyError	プロパティ値の設定に失敗したときに発生する
ERangeError	整数の値が，割り当てられた型より大きいことを表す
ERegistryException	レジストリエラーを表す
EZeroDivide	浮動小数点のゼロによる除算を捕捉する

上のリストを見てもわかるとおり，組み込みの VCL/CLX 例外クラスは，例外処理の多くを処理してくれるので，アプリケーションのコードを単純にできます。場合によっては，特殊な状況を処理するために独自の例外クラスを作成する必要があります。そのような場合は，新しい例外クラスを Exception 型から派生させます。そして，必要なコンストラクタを作成します（または，Sysutils.hpp にある既存のクラスのコンストラクタをコピーします）。

移植性についての注意

C++Builder には数種類のランタイムライブラリ（RTL）が含まれています。ほとんどは C++Builder アプリケーションに付属するものですが，1 つ（cw32mt.lib）は，VCL/CLX への参照をまったく行わない通常のマルチスレッド用 RTL です。この RTL は，VCL/CLX に依存しないプロジェクトの一部となる従来形式のアプリケーションをサポートするために提供されています。この RTL はオペレーティングシステム例外の捕捉をサポートしていません。そのような例外オブジェクトは TObject から派生しており，VCL と CLX をアプリケーションにリンクするする必要があるためです。

cp32mt.lib ライブラリ（マルチスレッド RTL）は，VCL/CLX を使ったメモリ管理と例外処理を提供します。

RTL DLL を使う場合，2 つのインポートライブラリ（cw32mti.lib と cp32mti.lib）を使用できます。VCL/CLX 例外のサポートには cp32mti.lib を使用してください。

第13章

VCL と CLX のための C++ 言語サポート

C++Builder には，VCL（ビジュアルコンポーネントライブラリ）と CLX（クロスプラットフォーム用コンポーネントライブラリ）を利用した RAD（Rapid Application Development）機能があります。VCL，CLX の両方とも Object Pascal で記述されています。この章では，VCL と CLX をサポートするために Object Pascal 言語の機能，構造，および概念が，C++Builder でどのように実現されているかを説明します。この章は，アプリケーションで VCL/CLX オブジェクトを使用するプログラマ，および VCL/CLX クラスを継承して新しいクラスを作成する開発者を対象としています。

この章の前半では，C++ と Object Pascal のオブジェクトモデルを比較し，これら 2 つのアプローチが C++Builder 内でどのように組み合わされているかについて説明します。この章の後半では，Object Pascal の言語構造が，C++Builder 内でそれに相当する C++ の言語構造にどのように変換されるかを説明します。ここでは VCL と CLX をサポートするために追加された拡張キーワードの詳細についても説明します。このような拡張の中には，クロージャやプロパティのように，VCL/CLX に基づくコードのサポートを抜きにしても，役立つものがあります。

メモ 「TObject から派生した C++ クラス」という表現は，TObject を最上位の親にもつクラスを意味しています。しかし，必ずしも TObject がすぐ上の親であるとは限りません。コンパイラの診断との一貫性を保つために，このようなクラスを「VCL スタイルクラス」とも呼びます。

C++ と Object Pascal のオブジェクトモデル

C++ と Object Pascal では，クラスモデルが明確に異なる点もあれば，微妙に違う点もあります。最も明確に異なる点は，C++ が多重継承を許可するのに対し，Object Pascal は単一継承モデルに制限していることです。加えて，C++ と Object Pascal では，オブジェクトの作成，初期化，参照，コピー，および破棄の方法が微妙に異なります。ここでは，これらの違いと，その違いが C++Builder の VCL スタイルクラスに及ぼす影響について説明します。

継承とインターフェース

C++ と異なり，Object Pascal 言語は多重継承をサポートしません。VCL か CLX を派生元にして作成されたクラスも，この制約を継承します。つまり，VCL/CLX クラスから直接派生したクラスでなくても，VCL スタイルの C++ クラスは複数の基本クラスを使用できません。

多重継承のかわりにインターフェースを使う

C++ で多重継承を使うケースの多くは，Object Pascal コードではインターフェースで代用できます。C++ には，Object Pascal のインターフェースという概念に直接マッピングされる構造がありません。Object Pascal のインターフェースは，実装を持たないクラスのような働きをします。つまり，インターフェースとは，関数がすべて純粋仮想で，かつデータメンバーを持たないクラスに似ています。Object Pascal のクラスは，それぞれ親クラスを 1 つしか持てませんが，インターフェースはいくつでもサポートできます。Object Pascal のコードは，どのインターフェース型の変数にもクラスインスタンスを代入できます。これはちょうど，上位のどのクラス型の変数にもクラスインスタンスを代入できるのと同様です。その結果，複数のクラスの派生元が同じでなくても，同じインターフェースを共有していれば多態性のある動作が可能になります。

純粋仮想関数だけを持ち，データメンバーを持たないクラスがある場合，C++Builder のコンパイラは，これを Object Pascal のインターフェースに対応するクラスとして認識します。このように，VCL スタイルクラスを作成して，多重継承を使用することが可能です，ただし，基本クラス（VCL クラス，CLX クラス，VCL スタイルクラスである基本クラスを除く）のすべてがデータメンバーを持たず，純粋仮想関数だけを持っていることが条件です。

メモ インターフェースクラスは VCL スタイルクラスでなくてもかまいません。唯一の要件は，データメンバーを持たず，純粋仮想関数だけを持つことです。

インターフェースクラスの宣言

インターフェースを表すクラスを宣言する方法は，他の C++ クラスの宣言と同じです。ただし，いくつかの規則を適用すると，そのクラスをインターフェースとして機能させることがいっそう明確になります。規則は以下のとおりです。

- クラスキーワードでなく，**__interface** を使ってインターフェースを宣言します。**__interface** は，クラスキーワードにマッピングするマクロです。**__interface** は必ずしも必要ではありませんが，**__interface** を使えば，そのクラスをインターフェースとして機能させるということが明確になります。
- インターフェースの名前は通常，「I」で始まります。たとえば，IComponentEditor，IDesigner などです。この規則を守っていれば，クラスがいつインターフェースの機能をしているかを確認するためにクラス宣言を調べ直す必要がなくなります。
- 一般に，インターフェースは関連の GUID を持っています。GUID を持つことは絶対条件ではありませんが，インターフェースをサポートするコードの大部分は，GUID を見つけることを想定しています。インターフェースと GUID を関連付けるには，**__declspec** 修飾子と引数 **uuid** を使います。インターフェースの場合，INTERFACE_UUID マクロがこの操作をマッピングします。

次の例は，上記の規則を使ってインターフェースを宣言しています。

```
__interface INTERFACE_UUID("{C527B88F-3F8E-1134-80e0-01A04F57B270}") IHelloWorld :
    public IInterface
{
public:
  virtual void __stdcall SayHelloWorld(void) = 0 ;
};
```

通常，インターフェースクラスを宣言すると，C++Builder のコードでは，それに対応する DelphiInterface クラスも宣言します。DelphiInterface クラスはインターフェースの操作をより簡単にします。

```
typedef System::DelphiInterface< IHelloWorld >  _di_IHelloWorld;
```

DelphiInterface クラスについての詳細は，13-20 ページの「Delphi のインターフェース」を参照してください。

IUnknown と IInterface

Object Pascal のインターフェースはすべて，IInterface という単一の派生元から派生します。VCL スタイルクラスが別の基本クラスを使えるという意味では，C++ のインターフェースクラスは必ずしも IInterface を基本クラスにする必要はありませんが，インターフェースを処理する VCL/CLX コードの側では，IInterface のメソッドが存在するものと想定します。

COM プログラミングでは，インターフェースはすべて IUnknown から派生します。VCL における COM サポートが基本としていることは，IUnknown の定義が直接 IInterface にマッピングされるということです。すなわち，Object Pascal では，IUnknown と IInterface は同一です。

ただし，IUnknown に関する Object Pascal の定義と，C++Builder が使用する IUnknown の定義は一致していません。unknwn.h ファイルに定義されている IUnknown には，以下の 3 つのメソッドが含まれています。

```
virtual HRESULT STDCALL QueryInterface( const GUID &guid, void ** ppv ) = 0;
virtual ULONG STDCALL AddRef() = 0;
virtual ULONG STDCALL Release() = 0;
```

この定義は，Microsoft が COM 仕様の 1 つとして規定している IUnknown の定義に一致します。

メモ IUnknown とその使い方については，第 38 章「COM テクノロジーの概要」または Microsoft のドキュメントを参照してください。

Object Pascal と違って，C++Builder では IInterface の定義と IUnknown の定義は同等ではありません。C++Builder では，IUnknown から IInterface が派生します。また，この IInterface には Supports という追加のメソッドが含まれています。

```
template <typename T>
HRESULT __stdcall Supports(DelphiInterface<T>& smartIntf)
{
  return QueryInterface(__uuidof(T),
        reinterpret_cast<void**>(static_cast<T**>(&smartIntf)));
}
```

Supports の働きは，IInterface を実装するオブジェクトから，別のインターフェースの DelphiInterface を取得することです。たとえば，インターフェース IMyFirstInterface の DelphiInterface があり，実装オブジェクトが IMySecondInterface も実装している（DelphiInterface 型が _di_IMySecondInterface である）場合，次のようにして 2 番目のインターフェースの DelphiInterface を取得できます。

```
_di_IMySecondInterface MySecondIntf;
MyFirstIntf->Supports(MySecondIntf);
```

VCL と CLX は，IUnknown から Object Pascal へのマッピングを使用します。このマッピングにより，IUnknown のメソッドで使われる型が Object Pascal の型に変換され，メソッド名 AddRef と Release が _AddRef と _Release に変更されます。このようにして，IUnknown のメソッドを直接呼び出すのを禁止することを示します（Object Pascal では，IUnknown のメソッドへの必要な呼び出しはコンパイラが自動的に生成します）。VCL/CLX オブジェクトがサポートする IUnknown（または IInterface）のメソッドを C++ にマッピングし直すと，以下のメソッドシグネチャになります。

```
virtual HRESULT __stdcall QueryInterface(const GUID &IID, void *Obj);
int __stdcall _AddRef(void);
int __stdcall _Release(void);
```

つまり，VCL/CLX オブジェクトが Object Pascal の IUnknown または IInterface をサポートしても，C++Builder に現れる IUnknown と IInterface はサポートしないということです。

IUnknown をサポートするクラスの作成

VCL/CLX の多くのクラスがインターフェースをサポートしています。実際，VCL スタイルクラスの基本クラスである TObject は GetInterface メソッドを持っているので，オブジェクトインスタンスがサポートするどのインターフェースでも DelphiInterface クラスを取得できます。TComponent クラスと TInterfacedObject クラスは，どちらも IInterface（IUnknown）のメソッドを実装しています。したがって，TComponent または TInterfacedObject から作成できる子孫はすべて，Object Pascal の全インターフェースに共通するメソッドのサポートを自動的に継承します。Object Pascal では，TComponent または TInterfacedObject の子孫を作成する場合，新しいインターフェースから導入されるメソッドを実装するだけで，その新規インターフェースをサポートできます。あとは，継承元である IInterface のメソッドのデフォルト実装を使用します。

残念ながら，C++Builder と VCL/CLX では IUnknown と IInterface のメソッドシグネチャが異なるため，VCL スタイルクラスを作成した場合（TComponent または TInterfacedObject から直接または間接的に派生させたとしても），IInterface と IUnknown は自動的にはサポートされません。つまり，C++ では，IUnknown または IInterface から派生させて定義したインターフェースをサポートするには，IUnknown のメソッドの実装を自分で追加しなければなりません。

TComponent または TInterfacedObject から派生したクラスで IUnknown のメソッドを実装する最も簡単な方法は，関数シグネチャに違いはありますが，組み込みの IUnknown サポートを利用することです。C++ 版の IUnkown メソッドを追加実装するだけで，あとは Object Pascal 版の継承メソッドに任せます。次に例を示します。

```
virtual HRESULT __stdcall QueryInterface(const GUID& IID, void **Obj)
{
  return TInterfacedObject::QueryInterface(IID, (void *)Obj);
}
virtual ULONG __stdcall AddRef()
{
  return TInterfacedObject::_AddRef();
}
virtual ULONG __stdcall Release()
{
  return TInterfacedObject::_Release();
}
```

前の 3 つのメソッドの実装を TInterfacedObject の子孫に追加することにより，作成したクラスは IUnknown または IInterface を完全にサポートします。

インターフェースクラスと存続期間管理

インターフェースクラスの IUnknown のメソッドは，インターフェースを実装しているオブジェクトをどう割り当て，解放するかに影響します。COM オブジェクトが IUnknown を実装すると，IUnknown のメソッドを使って，インターフェースへの参照がいくつ使われているかを追跡します。参照カウントがゼロになると，この COM オブジェクトは自動的に自身を解放します。TInterfacedObject も IUnknown のメソッドを実装して，同様の存続期間の管理を行います。

しかし，IUnknown のメソッドのこの解釈は，必ずしも必要ではありません。たとえば TComponent クラスでは，IUnknown メソッドのデフォルト実装はインターフェースに対する参照カウントを無視します。このため，参照カウントがゼロになってもコンポーネントは自分自身を解放しません。その理由は，TComponent はオブジェクトの Owner プロパティを使ってオブジェクトを解放するからです。

この 2 つを混合したハイブリッドモデルを使うコンポーネントもあります。Owner プロパティがヌルの場合はインターフェースに対する参照カウントを使って存続期間を管理し，参照カウントがゼロになると自身を解放します。オーナーを持つコンポーネントであれば，参照カウントを無視してオーナーに自身を解放させます。なお，こうしたハイブリッドオブジェクトに限らず，参照カウントを使って存続期間を管理するオブジェクトはすべて，アプリケーションがオブジェクトを作成してもそのオブジェクトからインターフェースを取得しなければ，そのオブジェクトは自動的には解放されません。

オブジェクトの識別とインスタンス化

C++ では，クラスのインスタンスが実際のオブジェクトです。このオブジェクトを直接操作できます。また，参照やポインタを使って間接的にオブジェクトにアクセスできます。たとえば，C++ のクラス CPP_class が，引数なしのコンストラクタを持つとします。次のコードは，すべてこのクラスの有効なインスタンス変数です。

```
CPP_class by_value;               // CPP_class 型のオブジェクト
CPP_class& ref = by_value;        // 上で定義したオブジェクト by_value の参照
CPP_class* ptr = new CPP_class(); // CPP_class 型オブジェクトへのポインタ
```

一方，Object Pascal では，オブジェクト型の変数は，常にそのオブジェクトを間接的に参照します。すべてのオブジェクトのメモリ領域は，動的に割り当てられます。たとえば，Object Pascal のクラス OP_class を考えます。

```
ref: OP_class;
ref := OP_class.Create;
```

ref は，OP_class 型のオブジェクトの「参照」です。これは，次のような C++Builder のコードに変換されます。

```
OP_class* ref = new OP_class;
```

C++ と Object Pascal の参照を区別する

ドキュメント内では，多くの場合 Object Pascal クラスのインスタンスの変数が参照として扱われますが，その動作はポインタのように説明されていることがあります。これは Object Pascal クラスの変数

が両方の側面を持つためです。Object Pascal におけるオブジェクトの参照は，以下の点を除いて C++ のポインタに似ています。

- Object Pascal の参照は暗黙のうちに逆参照される（この場合は，C++ の参照のように動作する）
- Object Pascal の参照には，定義済み演算としてのポインタ演算がない

Object Pascal の参照を C++ の参照と比較すると，似ている部分と異なる部分があります。両方の言語の参照は，暗黙に逆参照されていますが，以下の点が異なります。

- Object Pascal の参照はバインドし直すことができるが，C++ の参照はできない
- Object Pascal の参照は **nil** になり得るが，C++ の参照は必ず有効なオブジェクトを参照しなければならない

VCL/CLX フレームワークの基本的な設計の中には，この種のインスタンス変数の利用に基づいている部分があります。C++ の言語構造の中で，ポインタは最も Object Pascal の参照に似ています。したがって，ほとんどすべての VCL/CLX オブジェクト識別子が，C++Builder では C++ のポインタに翻訳されます。

メモ Object Pascal の **var** パラメータ型は，C++ の参照によく似ています。**var** パラメータについての詳細は，13-16 ページの「var パラメータ」を参照してください。

オブジェクトをコピーする

C++ とは違って，Object Pascal コンパイラにはオブジェクトをコピーするための組み込みのサポート機能がありません。ここでは，この違いが VCL スタイルクラスの代入演算子およびコピーコンストラクタに与える影響について説明します。

代入演算子

Object Pascal の代入演算子（:=）は，クラスの代入演算子（**operator=()**）ではありません。代入演算子はオブジェクトではなく，参照をコピーします。次のコードでは，B と C はともに同じオブジェクトを参照しています。

```
B, C: TButton;
B := TButton.Create(ownerCtrl);
C := B;
```

上の例は，C++Builder では次のように変換されます。

```
TButton *B = NULL;
TButton *C = NULL;
B = new TButton(ownerCtrl);
C = B;            // オブジェクトではなく，ポインタをコピーする
```

C++Builder の VCL スタイルクラスは，Object Pascal 言語の代入演算子の規則に従います。したがって，次のコードでは，逆参照されたポインタ間の代入は無効です。これは，ポインタではなくオブジェクトをコピーしようとしているからです。

```
TVCLStyleClass *p = new TVCLStyleClass;
TVCLStyleClass *q = new TVCLStyleClass;
*p = *q;    // VCL スタイルクラスのオブジェクトに対しては無効
```

メモ VCL スタイルクラスに対して，C++ の構文を使って参照をバインドすることはできます。たとえば，次のコードは有効です。

```
TVCLStyleClass *ptr = new TVCLStyleClass;
TVCLStyleClass &ref = *ptr;  // VCL スタイルクラスに対しても有効
```

これは代入演算子を使用することとは異なりますが，構文がよく似ているので，比較のためにここで説明しました。

コピーコンストラクタ

Object Pascal には，組み込みのコピーコンストラクタはありません。したがって，C++Builder の VCL スタイルクラスにも組み込みのコピーコンストラクタはありません。次のコードは，コピーコンストラクタを使って TButton ポインタを作成しようとしています。

```
TButton *B = new TButton(ownerCtrl);
TButton *C = new TButton(*B);  // VCL スタイルクラスのオブジェクトに対しては無効
```

VCL クラスと CLX クラスに対しては，組み込みのコピーコンストラクタに依存したコードを書いてはいけません。C++Builder で VCL スタイルクラスのオブジェクトのコピーを作成するには，オブジェクトをコピーするためのメンバー関数のコードを書きます。あるいは，VCL/CLX の TPersistent クラスの派生クラスで，オブジェクトのデータを別のオブジェクトにコピーするように Assign メソッドをオーバーライドします。これは，リソースイメージを持つ TBitmap や TIcon などのグラフィッククラスではよく行われます。最終的なオブジェクトのコピーのしかたは，プログラマ（コンポーネントの作成者）が決定します。ただし，VCL スタイルクラスに対しては，標準的な C++ で使われるコピー方法のいくつかが利用できない点に注意してください。

関数の引数としてのオブジェクト

前述したように，C++ と Object Pascal のインスタンス変数は同じではありません。オブジェクトを引数として関数へ渡すときにこの点に気づくはずです。C++ では，オブジェクトを値，参照，またはポインタのいずれかで関数に渡すことができます。Object Pascal でオブジェクトが値で関数に渡された場合は，そのオブジェクト引数はすでにそのオブジェクトへの参照になっていることに注意してください。したがって，実際には，値で渡されるのはオブジェクトそのものではなく参照です。Object Pascal では，C++ のように実際のオブジェクトを値で渡すことはできません。関数に渡された VCL スタイルのオブジェクトは，Object Pascal の規則に従います。

C++Builder の VCL/CLX クラスのためのオブジェクトの構築

C++ と Object Pascal では，オブジェクトの構築方法が異なります。ここではこのトピックの概要と，これら 2 つのアプローチが C++Builder 内でどのように組み合わされているかについて説明します。

C++ におけるオブジェクトの構築

標準的な C++ では，仮想基本クラス，基本クラス，最後に派生クラスの順に構築されます。C++ の構文では，コンストラクタの初期化リストを使って基本クラスのコンストラクタを呼び出します。そのオブジェクトの実行時の型は，現在呼び出されているコンストラクタのクラスの型になります。仮想メソッドのディスパッチは，オブジェクトの実行時の型に従ううので，構築中に変化します。

Object Pascal におけるオブジェクトの構築

Object Pascal では，インスタンス化されたクラスのコンストラクタだけが呼び出されることが保証されています。ただし，基本クラスのメモリ領域は割り当てられます。派生クラスのコンストラクタ内

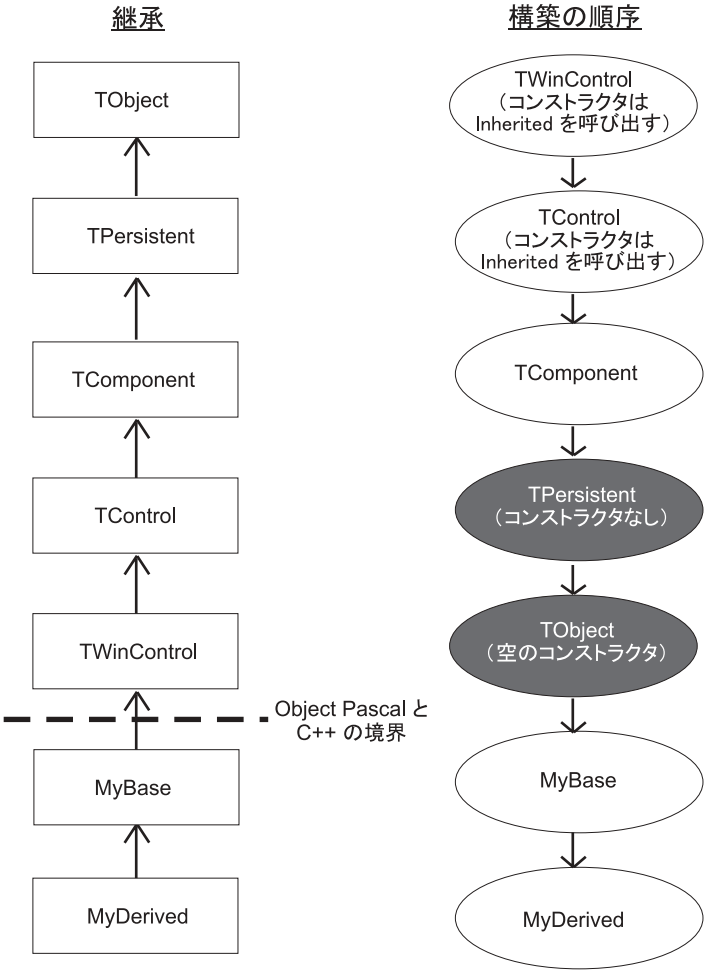
で **inherited** が呼び出されると，すぐ上の基本クラスが構築されます。慣例的に，VCL クラスと CLX クラスでは **inherited** を使って（空でない）基本クラスのコンストラクタを呼び出します。ただし，これは言語の要求からくるものではありません。オブジェクトの実行時の型は，インスタンス化されたクラスの型としてすぐに構築されるので，基本クラスのコンストラクタが呼び出されいる間も変化しません。仮想メソッドのディスパッチは，オブジェクトの実行時の型に従うので，構築中でも変化しません。

C++Builder におけるオブジェクトの構築

VCL スタイルのオブジェクトは，Object Pascal のオブジェクトと同様に構築されます。ただし，C++ の構文を使います。したがって，VCL/CLX でないすべての基本クラスと，すぐ上の VCL/CLX 上位クラスに対しては，基本クラスのコンストラクタを呼び出す方法および順序は，初期化リストを使った C++ 構文に従います。すぐ上の VCL/CLX 基本クラスは最初に構築されます。これによって，このクラスの基本クラスが構築されます。このときに，Object Pascal の方法に従って，**inherited** を使うこともできます。したがって，VCL/CLX の基本クラスは C++ の場合とは逆の順序で構築されます。C++ の基本クラスは，最上位のクラスから派生クラスの順に構築されます。VCL オブジェクトの実行時の型，および仮想メソッドのディスパッチは，Object Pascal と同じです。

図 13.1 は，VCL スタイルクラスのインスタンスの構築を表しています。MyDerived は MyBase から派生しています。MyBase は TWinControl のすぐ下のクラスです。MyDerived と MyBase は C++ で実装されています。TWinControl は Object Pascal で実装された VCL クラスです。

図 13.1 VCL スタイルオブジェクトの構築の順序



構築の順序は，C++ プログラマにとっては逆向きに見えるかもしれません。元々の VCL/CLX クラスに対しては，最も末端の上位クラスから始まって TObject の方向に構築されます。次に MyBase が構築され，最後に派生クラスが構築されます。

メモ TPersistent にはコンストラクタがないので，TComponent は **inherited** を呼び出しません。TObject のコンストラクタは空なので，呼び出されません。これらのクラスのコンストラクタが呼び出されたとすると，図に示した順序に従います。これらのクラスは，図では灰色で表されています。

表 13.1 に C++，Object Pascal，および C++Builder で使われるオブジェクト構築モデルの概要を示します。

表 13.1 オブジェクトモデルの比較

C++	Object Pascal	C++Builder
構築の順序		
仮想基本クラス，次に基本クラス，最後に派生クラス	インスタンス化されたクラスのコンストラクタのみが最初に自動的に呼び出される。それに続くクラスがある場合は，末端のクラスから上位クラスの方向に構築される	すぐ上の VCL/CLX 基本クラス，次に Object Pascal のモデルに従って構築される。最後に C++ のモデルに従って構築される（ただし，仮想基本クラスは許されない）
基本クラスのコンストラクタの呼び出し方法		
コンストラクタの初期化リストから自動的に呼び出される	派生クラスのコンストラクタ内で **inherited** キーワードを使って，いつでも選択的かつ明示的に呼び出せる	コンストラクタの初期化リストから，すぐ上の VCL/CLX 基本クラスのコンストラクタが自動的に呼び出される。次に Object Pascal の方法に従い，**inherited** を使ってコンストラクタが呼び出される
構築中のオブジェクトの実行時の型		
現在構築されているクラスの型を反映して変化する	インスタンス化されたクラスの型としてすぐに構築される	インスタンス化されたクラスの型としてすぐに構築される
仮想メソッドのディスパッチ		
基本クラスのコンストラクタが呼び出されている場合は，オブジェクトの実行時の型に応じて変化する	オブジェクトの実行時の型に従う。実行時の型は，どんなクラスのコンストラクタを呼び出している間も変化しない	オブジェクトの実行時の型に従う。実行時の型は，どんなクラスのコンストラクタを呼び出している間も変化しない

次の節では，これらの違いの重要性について説明します。

基本クラスのコンストラクタ内での仮想メソッドの呼び出し

VCL/CLX 基本クラス（Object Pascal で実装されているクラス）のコンストラクタ内から呼び出された仮想メソッドも，C++ の場合と同様に，オブジェクトの実行時の型に応じてディスパッチされます。C++Builder では，オブジェクトの実行時の型がただちに設定されるという Object Pascal のモデルと，派生クラスが構築される前に基本クラスが構築されるという C++ のモデルを組み合わせています。したがって，VCL スタイルクラスの基本クラスのコンストラクタから仮想メソッドを呼び出すと，微妙な副作用が発生する可能性があります。この影響について次に説明し，少なくとも 1 つの基本クラスから派生したクラスをインスタンス化する場合の例を示します。ここでは，インスタンス化されたクラスとは派生クラスのことを指します。

Object Pascal のモデル

Object Pascal では，プログラマは **inherited** キーワードを使用できます。inherited キーワードを使用すると，派生クラスのコンストラクタ内の任意の場所から，基本クラスのコンストラクタを柔軟に呼び出すことができます。したがって，オブジェクトのセットアップやデータメンバーの初期化に依存す

る仮想メソッドを，派生クラスでオーバーライドしている場合は，基本クラスのコンストラクタが呼び出される前に，この仮想メソッドが呼び出されてしまう可能性があります。

C++ のモデル

C++ の構文にはキーワード **inherited** がないので，派生クラスのコンストラクタ内の任意の場所から基本クラスのコンストラクタを呼び出すことができません。しかし，C++ のモデルでは必ずしも **inherited** を使う必要性はありません。なぜなら，オブジェクトの実行時の型は，現在構築されているクラスの型であって，派生クラスの型ではないからです。このため，呼び出される仮想メソッドは，派生クラスのメソッドではなく，現在のクラスのメソッドになります。したがって，仮想メソッドが呼び出される前に，データメンバーを初期化したり，派生クラスのオブジェクトをセットアップする必要はありません。

C++Builder のモデル

C++Builder では，VCL スタイルオブジェクトの実行時の型は，基本クラスのコンストラクタを呼び出している間でも，派生クラスの型になります。したがって，基本クラスのコンストラクタが仮想メソッドを呼び出すと，派生クラスがそのメソッドをオーバーライドしている場合には，派生クラスのメソッドが呼び出されます。この仮想メソッドが初期化リストや派生クラスのコンストラクタ本体に依存している場合は，これらが実行される前にメソッドが呼び出されます。たとえば，CreateParams という仮想メンバー関数は，TWinControl のコンストラクタ内で直接呼び出されています。

TWinControl からクラスを派生させて，CreateParams をオーバーライドした場合を考えます。オーバーライドした CreateParams が派生クラスのコンストラクタに依存していると，CreateParams が呼び出された後でコンストラクタのコードが処理されることになります。このような状況は，基本クラスから派生させたクラスすべてに当てはまります。クラス C がクラス B から派生し，クラス B がクラス A から派生している場合を考えます。B でメソッドがオーバーライドされいて，C ではオーバーライドされていない場合は，C のインスタンスを作成すると，A はオーバーライドされた B のメソッドを呼び出します。

メモ CreateParams のように，コンストラクタ内で直接呼び出されるのではなく，間接的に呼び出される仮想メソッドに注意にしてください。

仮想メソッドの呼び出し例

次の例では，オーバーライドされた仮想メソッドを持つ C++ クラスと VCL スタイルクラスを比較します。この例は，両方のケースで，このような仮想メソッドが基本クラスのコンストラクタからどのように呼び出され，解決されるかを示しています。MyBase と MyDerived は標準的な C++ のクラスです。MyVCLBase と MyVCLDerived は，TObject を継承した VCL スタイルクラスです。仮想メソッド what_am_I() は，両方のクラスでオーバーライドされています。ただし，what_am_I() は基本クラスのコンストラクタでのみ呼び出されます。派生クラスのコンストラクタでは呼び出されません。

```
#include <sysutils.hpp>
#include <iostream.h>
// VCL スタイルクラスでないクラス
class MyBase {
public:
   MyBase() { what_am_I(); }
   virtual void what_am_I() { cout << " 基本クラスです " << endl; }
};
```

```
class MyDerived : public MyBase {
public:
   virtual void what_am_I() { cout << " 派生クラスです " << endl; }
};
// VCL スタイルクラス
class MyVCLBase : public TObject {
public:
   __fastcall MyVCLBase() { what_am_I(); }
   virtual void __fastcall what_am_I() { cout << " 基本クラスです " << endl; }
};
class MyVCLDerived : public MyVCLBase {
public:
   virtual void __fastcall what_am_I() { cout << " 派生クラスです " << endl; }
};
int main(void)
{
   MyDerived d;    // C++ クラスのインスタンス化
   MyVCLDerived *pvd = new MyVCLDerived;  // VCL スタイルクラスのインスタンス化
   return 0;
}
```

この例の出力結果は次のようになります。

```
基本クラスです
派生クラスです
```

この違いは，MyDerived と MyVCLDerived では，それぞれの基本クラスのコンストラクタが呼び出されている間の実行時の型が異なるために生じます。

仮想関数で使われるデータメンバーをコンストラクタで初期化する

データメンバーは，仮想関数の中でも使われる可能性があるので，データメンバーが初期化されるタイミングと方法について理解することは重要です。Object Pascal では，初期化されていないデータは，すべてゼロに初期化されます。たとえば，このことは **inherited** によってコンストラクタが呼び出されない基本クラスにも当てはまります。標準的な C++ では，初期化されていないデータメンバーの値は保証されません。次の型のデータメンバーは，クラスのコンストラクタの初期化リスト内で初期化しなければなりません。

- 参照
- デフォルトのコンストラクタを持たないデータメンバー

それでも，基本クラスのコンストラクタが呼び出されている間は，これらのデータメンバーの値や，コンストラクタ本体で初期化される値は未定義です。C++Builder では，VCL スタイルクラスのメモリ領域はゼロで初期化されます。

メモ 技術的には，ゼロになるのは VCL/CLX クラスのメモリ領域です。つまり，ビットがゼロになるのであって，実際の値は未定義です。たとえば，参照はゼロになります。

コンストラクタ本体や初期化リスト内で初期化されるメンバー変数の値に依存する仮想関数は，それらの変数がゼロに初期化されているかのように動作します。これは，初期化リストが処理されたり，コンストラクタ本体に入る前に，基本クラスのコンストラクタが呼び出されるためです。これを，次の例で示します。

```
#include <sysutils.hpp>
class Base : public TObject {
public:
   __fastcall Base() { init(); }
   virtual void __fastcall init() { }
};
class Derived : public Base {
public:
   Derived(int nz) : not_zero(nz) { }
   virtual void __fastcall init()
   {
      if (not_zero == 0)
         throw Exception("not_zero is zero!");
   }
private:
   int not_zero;
};
int main(void)
{
   Derived *d42 = new Derived(42);
   return 0;
}
```

この例では，Base のコンストラクタ内で例外が送出されます。これは，Base が Derived の前に構築されるからです。not_zero は，コンストラクタに渡された 42 という値でまだ初期化されていません。基本クラスのコンストラクタが呼び出される前に，VCL スタイルクラスのデータメンバーを初期化することはできない点に注意してください。

オブジェクトの破棄

C++ と Object Pascal では，以下の 2 つの点に関してオブジェクトの破棄動作が異なります。次に相違点を示します。

- コンストラクタからの例外送出によって呼び出されたデストラクタ
- デストラクタから呼び出された仮想メソッド

VCL スタイルクラスでは，これら 2 つの言語の方法を組み合わせています。この問題について次にに説明します。

コンストラクタから送出された例外

オブジェクトの構築中に例外が送出された場合，デストラクタが呼び出される方法が C++ と Object Pascal で少し異なります。次に例を示します。クラス C はクラス B から派生し，クラス B はクラス A から派生しています。

```
class A
{
   // コードの本体
};
class B: public A
{
   // コードの本体
};
```

```
class C: public B
{
   // コードの本体
};
```

C のインスタンスの構築中に，クラス B のコンストラクタ内で例外が発生した場合を考えます。C++，Object Pascal，および VCL スタイルクラスにおける結果を以下に説明します。

- C++ では，まず B のオブジェクトデータメンバーのうち，完全に構築されたものすべてのデストラクタが呼び出されます。次に，A のデストラクタが呼び出されます。次に，A のデータメンバーのうち，完全にコンストラクタされたものすべてのデストラクタが呼び出されます。だたし，B と C のデストラクタは呼び出されません。
- Object Pascal では，インスタンス化されたクラスのデストラクタだけが自動的に呼び出されます。それは C のデストラクタです。コンストラクタの場合と同様に，デストラクタ内で **inherited** を呼び出すかどうかを決めるのはプログラマです。この例では，すべてのデストラクタが **inherited** を呼び出していると仮定すると，C，B，A の順にデストラクタが呼び出されます。例外が発生する前に，すでに B のコンストラクタ内で **inherited** が呼び出されていたかどうかには無関係です。A のデストラクタが呼び出されるのは，B のデストラクタ内で **inherited** が呼び出されているからです。A のデストラクタは，A のコンストラクタが実際に呼び出されたかどうかとは無関係に呼び出されます。さらに重要なことは，通常コンストラクタはただちに **inherited** を呼び出すので，C のコンストラクタ本体が完全に実行されたかどうかにかかわらず，C のデストラクタが呼び出されます。
- VCL スタイルクラスでは，（Object Pascal で実装されている）VCL/CLX 基本クラスは，Object Pascal のデストラクタ呼び出し方法に従います。（C++ で実装された）派生クラスは，厳密にはどちらの言語の方法にも従いません。この場合，すべてのデストラクタが呼び出されますが，C++ 言語の規則に従って呼び出されたのではないデストラクタの本体は実行されません。

Object Pascal で実装されたクラスには，デストラクタ本体に書いておいたクリーンアップコードを実行する機会があります。クリーンアップコードには，コンストラクタで例外が発生する前にコンストラクタされたサブオブジェクト（オブジェクトデータメンバー）のメモリ領域を解放するコードが含まれます。VCL スタイルクラスの場合，インスタンス化されたクラスや C++ で実装された基本クラスのデストラクタが呼び出されたとしても，そのクリーンアップコードは実行されない点に注意してください。

C++Builder の例外処理についての詳細は，12-14 ページの「VCL/CLX 例外処理」を参照してください。

デストラクタから呼び出された仮想メソッド

デストラクタ内での仮想メソッドのディスパッチは，コンストラクタの場合と同じです。VCL スタイルクラスの場合は，まず派生クラスが破棄されます。ただし，オブジェクトの実行時の型は，基本クラスのデストラクタが呼び出されている間も，派生クラスの型のままです。したがって，VCL/CLX 基本クラスのデストラクタ内で仮想メソッドが呼び出されると，すでに破棄されているクラスのメソッドをディスパッチすることになります。

AfterConstruction と BeforeDestruction

TObject は，BeforeDestruction と AfterConstruction の 2 つの仮想メソッドを導入しています。これを利用して，プログラマはオブジェクトが破棄される前と，オブジェクトが作成された後に実行するコードをそれぞれ書くことができます。AfterConstruction は，最後のコンストラクタが呼び出された後に呼び出されます。BeforeDestruction は，最初のデストラクタが呼び出される前に呼び出されます。これらのメソッドはパブリックで，自動的に呼び出されます。

クラス仮想関数

Object Pascal には，クラス仮想関数という概念があります。これは，もしあるとすれば C++ の「静的仮想関数」に当たるものですが，C++ にはこの種の関数に正確に対応する概念はありません。これらの関数は，VCL と CLX から内部で安全に呼び出されます。ただし，C++Builder で，この種の関数を呼び出してはいけません。この種の関数には，ヘッダーファイル内に次のようなコメントが付加されています。

```
/* 仮想クラスメソッド */
```

Object Pascal のデータ型および言語概念のサポート

C++Builder では，VCL と CLX をサポートするために Object Pascal のデータ型，構造，および言語概念の多くを C++ 言語に実装，変換，またはマップしています。このために，次の機能が使われています。

- C++ 固有の型への **typedef** 宣言
- クラス，構造体，およびクラステンプレート
- Object Pascal 言語に相当する C++ 言語の機能
- マクロ
- ANSI の規則に従った言語拡張としてのキーワード

Object Pascal 言語のすべての機能が C++ に明確にマップされているわけではありません。C++ にマップされない Object Pascal の部分を使用すると，アプリケーションが予想外の動作をする場合があります。次に例を示します。

- Object Pascal と C++ の両方に存在する型で，その定義が異なるものがある。しがたって，2 つの言語間でコードを共有する場合には，注意が必要である
- C++Builder をサポートする目的で，Object Pascal にはいくつかの拡張が加えられている。しばしば，これらの拡張が 2 つの言語の相互利用に微妙な影響を与える
- Object Pascal の型および言語構造の中には，C++ 言語にマップできないものがある。2 つの言語間でコードを共有する場合は，C++Builder ではこれらの使用を避けなければならない

この節では，Object Pascal 言語が C++Builder にどう実装されているかを簡単に説明し，注意点を挙げます。

typedef 宣言

Object Pascal 固有のデータ型のほとんどが，C++Builder では **typedef** 宣言を使って C++ 固有の型に実装されています。これらの宣言は sysmac.h にあります。できる限り，Object Pascal の型ではなく，C++ 固有の型を使用してください。

Object Pascal 言語をサポートするクラス

Object Pascal のデータ型および言語構造の中には，それに相当するものが C++ にはないものもあります。これらは，クラスまたは構造体として実装されています。また，集合（**set**）のような Object Pascal の言語構造やデータ型を実装するために，クラステンプレートも使われています。このテンプレートから特定の型を宣言できます。これらの宣言は，以下のヘッダーファイルにあります。

- dstring.h
- wstring.h
- sysclass.h
- syscomp.h
- syscurr.h
- sysdyn.h
- sysopen.h
- sysset.h
- systdate.h
- systobj.h
- systvar.h
- sysvari.h

これらのヘッダーファイルで実装されているクラスは，Object Pascal ルーチンで使われる固有の型をサポートするために作られました。VCL/CLX ベースのコード内で Object Pascal ルーチンを呼び出す場合に，これらを使います。

Object Pascal 言語に相当する C++ 言語の機能

Object Pascal の **var** パラメータと型なしパラメータは，本来 C++ にはありません。C++Builder では，C++ 言語のこれらに相当するものを使用します。

var パラメータ

C++ と Object Pascal には，ともに「参照渡し」の概念があります。これらは変更可能な引数です。Object Pascal では，これらを **var** パラメータと呼びます。**var** パラメータを取る関数の構文は次のとおりです。

```
procedure myFunc(var x : Integer);
```

C++ では，この型のパラメータを参照で渡します。

```
void myFunc(int& x);
```

C++ の参照とポインタは，両方ともオブジェクトを修正するために使われます。ただし，ポインタよりも参照の方が **var** パラメータに似ています。ポインタとは違って，参照をバインドし直すことはできません。**var** パラメータも代入し直すことはできません。ただし，どちらも，参照されている値を変更することはできます。

型なしパラメータ

Object Pascal では，特定の型を持たないパラメータも許されます。これらのパラメータは，型が定義されないまま関数に渡されます。これらのパラメータを受け取った関数は，それを使用する前に既知の型にキャストしなければなりません。C++Builder では，型なしパラメータを **void** へのポインタ（**void ***）であると解釈します。これらのパラメータを受け取った関数は，**void** ポインタを適切な型のポインタにキャストしなければなりません。次に例を示します。

```
int myfunc(void* MyName)
{
   // ポインタを適切な型にキャストしてから逆参照する
   int* pi = static_cast<int*>(MyName);
   return 1 + *pi;
}
```

オープン配列

Object Pascal には「オープン配列」という構造があり，サイズ指定のない配列を関数に渡すことができます。C++ には，直接この種の配列をサポートする機能はありません。オープン配列パラメータを持つ Object Pascal の関数には，配列の最初の要素へのポインタと最後のインデックスの値（配列の要素数 - 1）の両方を明示的に渡します。たとえば，math.hpp の中の Mean 関数は，Object Pascal では次のように宣言されています。

```
function Mean(Data: array of Double): Extended;
```

C++ では次のように宣言されます。

```
Extended __fastcall Mean(const double * Data, const int Data_Size);
```

次のコードは，C++ から Mean 関数を呼び出す処理を表しています。

```
double d[] = { 3.1, 4.4, 5.6 };
// 明示的に最後のインデックスを指定する
long double x = Mean(d, 2);
// より優れた方法 : sizeof を使って，確実に正しい値が渡されるようにする
long double y = Mean(d, (sizeof(d) / sizeof(d[0])) - 1);
// sysopen.h のマクロを使用する
long double z = Mean(d, ARRAYSIZE(d) - 1);
```

メモ 上の例と同様なケースで，Object Pascal の関数が **var** パラメータを持つ場合は，C++ での関数宣言のパラメータは **const** をはずします。

配列の要素数を計算する

sizeof()，ARRAYSIZE マクロ，または EXISTINGARRAY マクロを使って配列の要素数を計算する場合には，誤って配列へのポインタを使わないように注意してください。かわりに，配列自身の名前を渡します。

```
double d[] = { 3.1, 4.4, 5.6 };
int n = ARRAYSIZE(d); // sizeof(d)/sizeof(d[0]) => 24/8 => 3
double *pd = d;
int m = ARRAYSIZE(pd); // sizeof(pd)/sizeof(pd[0]) => 4/8 => 0 => Error!
```

配列の「sizeof」を求めることと，ポインタの「sizeof」を求めることは同じではありません。たとえば，次のような宣言を考えます。

```
double d[3];
double *p = d;
```

配列のサイズを求めるには，次のようにします。

```
sizeof(d)/sizeof d[0]
```

これは，次のようにポインタのサイズを求めることと等価ではありません。

```
sizeof(p)/sizeof(p[0])
```

以下の例では，**sizeof()** 演算子のかわりに ARRAYSIZE マクロを使用しています。ARRAYSIZE マクロについての詳細は，オンラインヘルプを参照してください。

一時変数

Object Pascal では，名前のない一時的なオープン配列を関数に渡すことができます。C++ には，これを実行するための構文がありません。ただし，変数定義にほかの文と含めることができるので，1 つの解決方法は，単純に一時変数に名前を付けることです。

Object Pascal の次のようなコードを考えます。

```
Result := Mean([3.1, 4.4, 5.6]);
```

C++ では，次のように名前付きの「一時変数」を使います。

```
double d[] = { 3.1, 4.4, 5.6 };
return Mean(d, ARRAYSIZE(d) - 1);
```

「一時変数」の名前のスコープを制限して，ほかのローカル変数との衝突を避けるには，次のように新しいスコープを指定します。

```
long double x;
{
   double d[] = { 4.4, 333.1, 0.0 };
   x = Mean(d, ARRAYSIZE(d) - 1);
}
```

もう 1 つの解決方法については，13-19 ページの「OPENARRAY マクロ」を参照してください。

array of const

Object Pascal は，**array of const** と呼ばれる言語構造をサポートしています。この引数型は，TVarRec 型のオープン配列を値で渡すことと同じです。

次に **array of const** を受け取るように宣言した Object Pascal のコードの一部を示します。

```
function Format(const Format: string; Args: array of const): string;
```

C++ では，プロトタイプは次のようになります。

```
AnsiString __fastcall Format(const AnsiString Format,
      TVarRec const *Args, const int Args_Size);
```

この関数の呼び出し方は，オープン配列を引数に取るほかの関数とまったく同じです。

```
void show_error(int error_code, AnsiString const &error_msg)
{
   TVarRec v[] = { error_code, error_msg };
   ShowMessage(Format("%d: %s", v, ARRAYSIZE(v) - 1));
}
```

OPENARRAY マクロ

OPENARRAY マクロは sysopen.h で定義されています。オープン配列を引数に取る関数に，一時的なオープン配列を渡す場合，名前付きの変数を渡すかわりにこのマクロを使用します。このマクロは，次のように使います。

```
OPENARRAY(T, (value1, value2, value3)) // 最大 19 個の値をとれる
```

T は，構築するオープン配列の型を表します。また，値パラメータを使って配列をうめます。値引数は丸カッコで囲む必要があります。次に例を示します。

```
void show_error(int error_code, AnsiString const &error_msg)
{
   ShowMessage(Format("%d: %s", OPENARRAY(TVarRec, (error_code, error_msg))));
}
```

OPENARRAY マクロには，最大 19 個まで値を渡すことができます。それより大きな配列が必要な場合は，明示的に変数を定義しなければなりません。さらに，OPENARRAY マクロを使用すると，わずかながら実行時のコストが増加します。これは，このマクロで使用する配列を割り当てて，そこに値をコピーするためのコストです。

EXISTINGARRAY マクロ

EXISTINGARRAY マクロは sysopen.h で定義されています。オープン配列を期待している引数に既存の配列を渡す場合に，このマクロを使用します。このマクロは，次のように使います。

```
long double Mean(const double *Data, const int Data_Size);
double d[] = { 3,1, 3,14159, 2,17128 };
Mean(EXISTINGARRAY (d));
```

メモ 13-17 ページの「配列の要素数を計算する」にも EXISTINGARRAY マクロに関する説明があります。

オープン配列を引数に取る C++ 関数

Object Pascal からオープン配列を渡される C++ 関数を書く場合は，「値渡し」のセマンティクスを明示的に維持することが重要です。特に，関数の宣言が「値渡し」に相当する場合は，配列を変更する前に，必ず明示的にすべての要素をコピーしてください。Object Pascal ではオープン配列は組み込み型なので，値で渡すことができます。C++ では，オープン配列はポインタを使って実装されています。したがって，ローカルにコピーを作成しておかないと，元々の配列を変更してしまう可能性があります。

定義が異なる型

Object Pascal と C++ で定義が異なる型は，通常はそれほど問題を起こしません。まれに問題が発生するケースは微妙な場合があります。このため，このセクションでこれらの型について説明します。

論理データ型

Object Pascal の ByteBool，WordBool，および LongBool データ型の True の値は，Object Pascal 内部では -1 で表されます。False は 0 で表されます。

メモ Boolean データ型はそのままです（True = 1，False = 0）。

C++ の論理型は，これらの Object Pascal の型を正しく変換できます。ただし，WinAPI 関数，または Windows の BOOL 型を使用したその他の関数を共有する場合には，問題が発生します。Windows の BOOL 型は 1 で表されます。ある値が BOOL 型のパラメータに渡されると，その評価結果は Object Pascal では -1 に，C++ では 1 になります。したがって，2 つの言語間でコードを共有する場合は，2 つの識別子がともに 0（False，**false**）でない限り，比較演算はうまく動作しない可能性があります。対策として，次のような比較方法を使うことができます。

```
!A == !B;
```

表 13.2 に，この等価比較法の実行結果を示します。

表 13.2 BOOL 変数の等価比較 !A == !B

Object Pascal	C++	!A == !B
0 (False)	0 (**false**)	!0 == !0 (TRUE)
0 (False)	1 (**true**)	!0 == !1 (FALSE)
-1 (True)	0 (**false**)	!-1 == !0 (FALSE)
-1 (True)	1 (**true**)	!-1 == !1 (TRUE)

この比較方法を利用すれば，どのような値の組み合わせでも正しい評価結果が得られます。

char データ型

C++ の **char** 型は符号付きの型です。一方，Object Pascal の Char 型は符号なしの型です。コードを共有した場合に，この違いが問題になるような状況はほとんど発生しません。

Delphi のインターフェース

Object Pascal のコンパイラは，インターフェースに対する細かな操作の多くを自動的に処理します。アプリケーションコードがインターフェースポインタを取得すれば自動的に参照カウントを増やし，インターフェースがスコープの外に出れば参照カウントを減らします。

C++Builder では，DelphiInterface テンプレートクラスが C++ のインターフェースクラス用に同様の簡便さをいくつか提供します。Object Pascal のインターフェース型を使用する VCL/CLX の各種のプロパティとメソッドに関しては，C++ では，基礎となるインターフェースクラスを使って構築される DelphiInterface を C++ のラッパーが使用します。

DelphiInterface のコンストラクタ，コピーコンストラクタ，代入演算子，およびデストラクタが，必要に応じて参照カウントのインクリメントとデクリメントを行います。しかし，DelphiInterface は，Object Pascal でインターフェースをサポートするコンパイラと比べて簡便さが劣ります。基礎となるインターフェースポインタへのアクセスを提供する演算子の中には，参照カウントを処理しないものがあります。理由は，該当する場所を識別できないことがあるからです。参照カウントを正しく実行するには，場合によっては AddRef または Release を明示的に呼び出す必要があります。

リソース文字列

Pascal のユニット内にリソース文字列を使用するコードがある場合は，Pascal コンパイラ（DCC32）は，ヘッダーファイルを生成するときに，リソース文字列ごとに 1 つのグローバル変数と，それに対応するプロプロセッサマクロを作成します。このマクロは，リソース文字列を自動的にロードするために使われます。このマクロは，C++ コード内でリソース文字列を参照しているすべての場所で使うために作成されます。たとえば，Object Pascal コード内に，次のような **resourcestring** セクションがある場合を考えます。

```
unit borrowed;
interface
resourcestring
   Warning = 'Be careful when accessing string resources.';
implementation
begin
end.
```

C++Builder 用に Pascal コンパイラが生成したコードは次のようになります。

```
extern PACKAGE System::Resource ResourceString _Warning;
#define Borrowed_Warning System::LoadResourceString(&Borrowed::_Warning)
```

これによって，明示的に LoadResourceString を呼び出さなくても，Object Pascal のエクスポート済みのリソース文字列を利用できます。

デフォルト引数

現在の Pascal コンパイラは，C++ との互換性を保つために，コンストラクタに関してはデフォルト引数を受け付けます。C++ とは違って，Object Pascal のコンストラクタは，名前さえ異なれば，同じ型の引数を同じ数だけ持つことができます。このような場合は，Object Pascal のコンストラクタでダミーの引数を使います。こうすれば，C++ のヘッダーファイルが生成された場合でも，これらのコンストラクタを識別できます。たとえば，次のような TInCompatible という名前のクラスの Object Pascal におけるコンストラクタがあるとします。

```
constructor Create(AOwner: TComponent);
constructor CreateNew(AOwner: TComponent);
```

デフォルト引数がないと，上の両方のコンストラクタは，C++ では次のようにあいまいなコードに変換されます。

```
__fastcall TInCompatible(Classes::TComponent* Owner); // Pascal の Create
                                                      // コンストラクタの C++ バージョン
__fastcall TInCompatible(Classes::TComponent* Owner); // Pascal の CreateNew
                                                      // コンストラクタの C++ バージョン
```

デフォルト引数を使用すると，TCompatible という名前のクラスの Object Pascal におけるコンストラクタは次のようになります。

```
constructor Create(AOwner: TComponent);
constructor CreateNew(AOwner: TComponent; Dummy: Integer = 0);
```

今度は，C++Builder でもあいまい性のないコードに翻訳されます。

```
__fastcall TCompatible(Classes::TComponent* Owner); // Pascal の Create コンストラクタの
                                                    // C++ バージョン
```

```
__fastcall TCompatible(Classes::TComponent* Owner, int Dummy); // Pascal の CreateNew
                                                               // コンストラクタの C++ バージョン
```

メモ デフォルト引数に関する最大の問題は，DCC32 がデフォルト引数のデフォルト値を取り去ってしまうことです。デフォルト値を取り去らないと，曖昧性が生じてしまうためです。VCL/CLX クラスやサードパーティ製のコンポーネントを使用する場合には，この点に注意してください。

実行時型情報

Object Pascal には，RTTI を扱う言語構造があります。C++ にも，それに相当するものがいくつかあります。それらを表 13.3 に示します。

表 13.3 Object Pascal から C++ への RTTI のマッピング例

Object Pascal RTTI	C++ RTTI
`if Sender is TButton...`	`if (dynamic_cast <TButton*> (Sender)` `// 失敗した場合は dynamic_cast が NULL を返す`
`b := Sender as TButton;` `(* エラー時に例外を生成する *)`	`TButton& ref_b = dynamic_cast <TButton&> (*Sender)` `// エラー時に例外を送出する`
`ShowMessage(Sender.ClassName);`	`ShowMessage(typeid(*Sender).name());`

表 13.3 で，ClassName は TObject のメソッドであり，変数の宣言時の型に関係なく，オブジェクトの実際の型の名前を含む文字列を返します。TObject に導入されているその他の RTTI メソッドに相当するものは，C++ にはありません。これらのメソッドはすべてパブリックです。その一覧を以下に示します。

- ClassInfo は，実行時型情報（RTTI: runtime type information）テーブルへのポインタを返します。
- ClassNameIs は，オブジェクトが特定の型かどうか判断します。
- ClassParent は，そのクラスの 1 つ上位のクラスの型を返します。TObject には親クラスがないので，TObject の場合は ClassParent は nil を返します。これは，**is** 演算子，**as** 演算子，および InheritsFrom メソッドで使われます。
- ClassType は，オブジェクトの実際の型を動的に判断します。**is** 演算子および **as** 演算子で内部的に使われます。
- FieldAddress は RTTI を使って，公開されているフィールドのアドレスを取得します。これは，ストリームシステムで内部的に使われます。
- InheritsFrom は，2 つのオブジェクトの関係を判断します。**is** 演算子および **as** 演算子で内部的に使われます。
- MethodAddress は RTTI を使って，メソッドのアドレスを検索します。これは，ストリームシステムで内部的に使われます。

TObject のこれらのメソッドの中には，主にコンパイラやストリームシステムで内部的に使われるものもあります。これらのメソッドについての詳細はオンラインヘルプを参照してください。

マップされない型

6 バイトの Real 型

昔の Object Pascal の 6 バイトの浮動小数点形式は，現在は Real48 と呼ばれています。昔の Real 型は現在は **double** です。C++ には Real48 型に相当するものがありません。したがって，この型を含む Object Pascal のコードを C++ コードと共に使用しないでください。これを行おうとすると，ヘッダーファイルジェネレータによって警告が出されます。

関数の戻り値型としての配列

Object Pascal では，関数の引数や戻り値型に配列を取ることができます。たとえば，次の構文では関数 GetLine が 80 文字の配列を返します。

```
type
  Line_Data = array[0..79] of char;
function GetLine: Line_Data;
```

C++ には，この概念に相当するものがありません。C++ では，配列を関数の戻り値型に使うことはできませんし，関数の引数として配列を受け取ることもできません。

VCL と CLX には配列で表されるプロパティはありませんが，Object Pascal では配列プロパティが許されるという点に注意してください。プロパティの場合は，取得と設定を行う **read** および **write** メソッドを使って，指定したプロパティ型の値を取得できるので，C++Builder ではプロパティ型の配列は必要ありません。

メモ 配列プロパティは Object Pascal でも有効ですが，C++ でも問題ありません。これは，Get メソッドは引数にインデックスの値をとり，Set メソッドは配列が含む型のオブジェクトを返すからです。配列プロパティについての詳細は，47-8 ページの「配列プロパティの作成」を参照してください。

拡張キーワード

ここでは，VCL と CLX をサポートするために C++Builder に実装された拡張キーワードについて説明します。これらの拡張キーワードは ANSI の規則に従っています。C++Builder のキーワードと拡張キーワードの完全なリストは，オンラインヘルプを参照してください。

__classid

__classid 演算子は，指定した classname の仮想テーブルへのポインタを生成するために，コンパイラによって使用されます。この演算子は，あるクラスのメタクラスを取得するために使われます。

構文

```
__classid(classname)
```

たとえば，**__classid** は，プロパティエディタ，コンポーネント，およびクラスを登録する場合に，TObject の InheritsFrom とともに使われます。次のコードでは，TWinControl を継承して新しいコンポーネントを作成するために **__classid** を使用しています。

```
namespace Ywndctrl
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(MyWndCtrl)};
```

```
    RegisterComponents("Additional", classes, 0);
  }
}
```

__closure

キーワード **__closure** は，メンバー関数に特殊な型のポインタを宣言するために使用されます。標準の C++ では，メンバー関数を指すポインタを取得するには，完全に限定されたメンバー名を使う以外に方法がありません。下に例を示します。

```
class base
{
  public:
    void func(int x) { };
};

typedef void (base::* pBaseMember)(int);

int main(int argc, char* argv[])
{
  base        baseObject;
  pBaseMember m = &base::func; // メンバー 'func' へのポインタを取得

  // メンバーを指すポインタを通じて 'func' を呼び出す
  (baseObject.*m)(17);
  return 0;
}
```

ところが，基本クラスのメンバー関数を指すポインタに，派生クラスのメンバー関数を指すポインタを代入することはできません。次のコードはこのルール（コントラバリアンスと呼ばれる）を表しています。

```
class derived: public base
{
  public:
    void new_func(int i) { };
};

int main(int argc, char* argv[])
{
  derived         derivedObject;
  pBaseMember m = &derived::new_func; // 無効

  return 0;
}
```

拡張キーワード **__closure** を使うと，この制約を回避でき，さらにそれ以上のことができます。クロージャを使うと，オブジェクト（あるクラスの特定のインスタンス）のメンバー関数を指すポインタを取得できます。ここでいう「オブジェクト」は，どのオブジェクトでもかまいません。オブジェクトの継承階層は問いません。クロージャを介してメンバー関数を呼び出すとき，オブジェクトの **this** ポインタが自動的に使用されます。次の例は，クロージャーの宣言の仕方と使い方を示しています。基本クラスと派生クラスは，すでに定義されているものとします。

```
int main(int argc, char* argv[])
{
  derived         derivedObject;
```

```
    void (__closure *derivedClosure)(int);

       derivedClosure = derivedObject.new_func;// メンバー 'new_func' を指すポインタを取得する
                                               // クロージャが特定のオブジェクト 'derivedObject'
                                               // と関連付けられていることに注意
       derivedClosure(3);  // クロージャを介して 'new_func' を呼び出す
       return 0;
  }
```

クロージャは，オブジェクトを指すポインタも操作できます。下に例を示します。

```
  void func1(base *pObj)
  {
    // 引数に int を取り，void を返すクロージャ
    void ( __closure *myClosure )(int);
    // クロージャの初期化
    myClosure = pObj->func;
    // クロージャを使ってメンバー関数を呼び出す
    myClosure(1);
    return;
  }
  int main(int argc, char* argv[])
  {
    derived         derivedObject;
    void (__closure *derivedClosure)(int);

       derivedClosure = derivedObject.new_func; // 前の例と同じ
       derivedClosure(3);

       // ポインタを使ってクロージャの初期化もできる
       // 基本クラスにあるメンバー関数 'func'
       // を指すポインタも取得できる
       func1(&derivedObject);
       return 0;
  }
```

この例では，派生クラスのインスタンスを指すポインタを渡し，基本クラスにあるメンバー関数を指すポインタを取得しています。これは，標準の C++ ではできないことです。

クロージャは，C++Builder RAD 環境の重要な要素です。クロージャのおかげで，オブジェクトインスペクタでイベントハンドラを割り当てることができます。たとえば，TButton オブジェクトには OnClick というイベントがあります。TButton クラスでは，OnClick イベントは，宣言部で拡張キーワード **__closure** を使用するプロパティです。キーワード **__closure** を使うと，このプロパティに別のクラスのメンバー関数（一般的には TForm オブジェクトのメンバー関数）を代入することができます。フォームに TButton オブジェクトを置き，そのボタンの OnClick イベントハンドラを作成すると，C++Builder はボタンの親である TForm にメンバー関数を作成し，そのメンバー関数を TButton の OnClick イベントに代入します。このようにして，TButton の特定のインスタンスとイベントハンドラが関連付けられます（特定のインスタンス以外は関連付けされない）。

イベントとクロージャの詳細については，第 48 章「イベントの作成」を参照してください。

__property

キーワード **__property** は，クラスの属性を宣言します。プログラマにとって，プロパティはクラスの属性（フィールド）とまったく同じように見えます。しかし，Object Pascal のプロパティと同様に，キーワード **__property** は単に属性値を確認，変更するだけでなく，ほかにも多くの機能を持っています。プロパティの属性がプロパティへのアクセスを完全に制御するので，クラス自体の中でのプロパティの実装方法に制約はありません。

構文

```
__property type propertyName[index1Type index1][indexNType indexN] = { attributes };
```

ここで

- type は，組み込みのデータ型または前に宣言されているデータ型
- propertyName は，任意の有効な識別子
- indexNType は，組み込みのデータ型または前に宣言されているデータ型
- indexN は，プロパティの **read** 関数と **write** 関数に渡されるインデックスパラメータの名前
- attributes は，**read**，**write**，**stored**，**default**（または **nodefault**），**index** の順に一部または全部をカンマで区切って並べる

大カッコ [ ] で囲まれた indexN は省略可能です。indexN がある場合，配列プロパティを宣言します。このインデックスパラメータは，配列プロパティの **read** メソッドと **write** メソッドに渡されます。

いくつか簡単なプロパティ宣言を行っている例を下に示します。

```
class PropertyExample {
   private:
      int Fx,Fy;
      float Fcells[100][100];

   protected:
      int  readX()           { return(Fx); }
      void writeX(int newFx) { Fx = newFx;  }

      double computeZ() {
            // 計算を実行して浮動小数点値を返す
            return(0.0);
      }

      float cellValue(int row, int col) { return(Fcells[row][col]); }

   public:
      __property int    X = { read=readX, write=writeX };
      __property int    Y = { read=Fy };
      __property double Z = { read=computeZ };
      __property float Cells[int row][int col] = { read=cellValue };
};
```

この例は，いくつかのプロパティ宣言を示しています。プロパティ X は，メンバー関数 readX と writeX を介して読み書きアクセスをします。プロパティ Y はメンバー変数 Fy に直接対応するもので，読み出し専用です。プロパティ Z は，計算される読み出し専用値です。クラスのデータメンバーとして保存されることはありません。最後の Cells プロパティは，インデックスを 2 つ持つ配列プロパティを示しています。次の例で，これらのプロパティにアクセスするためのコードの書き方を示します。

```
PropertyExample myPropertyExample;
```

```
myPropertyExample.X = 42;            // 評価結果： myPropertyExample.WriteX(42);
int    myVal1 = myPropertyExample.Y; // 評価結果： myVal1 = myPropertyExample.Fy;
double myVal2 = myPropertyExample.Z;; // 評価結果： myVal2 =
myPropertyExample.ComputeZ();
float  cellVal = myPropertyExample[3][7]; // 評価結果：
                                          // cellVal = myPropertyExample.cellValue(3,7);
```

プロパティには，この例以外にも多くの機能やバリエーションがあります。プロパティは次のこともできます。

- index 属性を使って，同じ読み書きメソッドを複数のプロパティと関連付ける
- デフォルト値を持つ
- フォームファイルに保存される
- 基本クラスで定義されているプロパティを拡張できる

プロパティについての詳細は，第 47 章「プロパティの作成」を参照してください。

__published

__published キーワードを指定すると，クラスがコンポーネントパレット上にある場合に，そのセクション内のプロパティがオブジェクトインスペクタに表示されるようになります。**__published** セクションを持てるのは TObject から派生したクラスだけです。

パブリッシュされたメンバーとパブリックなメンバーの公開規則は同じです。唯一異なるのは，**__published** セクション内で宣言されたデータメンバーおよびプロパティに対しては，Object Pascal スタイルの実行時の型情報（RTTI）が生成される点です。アプリケーションは RTTI を使用して，未知のクラス型のデータメンバー，メンバー関数，およびプロパティを動的に問い合わせることができます。

メモ **__published** セクション内に，コンストラクタまたはデストラクタを置くことはできません。**__published** セクション内に置けるのは，プロパティ，Pascal 固有のデータメンバー，VCL/CLX から派生したデータメンバー，メンバー関数，およびクロージャです。**__published** セクション内で定義されたフィールドはクラス型でなければなりません。**__published** セクション内で定義されたプロパティは，プロパティ配列にはできません。**__published** セクション内で定義されたプロパティの型は，順序型，実数型，文字列型，小規模な集合型，クラス型，またはメソッドポインタ型でなければなりません。

__declspec 拡張キーワード

__declspec 拡張キーワードにいくつかの引数を渡すことによって，VCL と CLX をサポートできます。これらの引数を以下に示します。declspec 引数用のマクロおよびそれらの組み合わせは sysmac.h で定義されています。多くの場合，これらを指定する必要はありません。これらを使う必要がある場合は，マクロを使用してください。

__declspec(delphiclass)

delphiclass 引数は，TObject から派生したクラスの宣言に使用します。これらのクラスは，以下に示すような互換性を持つように作成されます。

- Object Pascal 互換の RTTI

- コンストラクタ / デストラクタの動作に VCL（CLX）と互換性がある
- VCL（CLX）互換の例外処理

VCL/CLX 互換クラスには以下の制約があります。

- 仮想基本クラスは許されない
- 多重継承はできない。ただし，13-2 ページの「継承とインターフェース」に記載されているケースを除く
- グローバルな **new** 演算子を使って動的に割り当てなければならない
- デストラクタを持たなければならない
- VCL/CLX の派生クラスのコピーコンストラクタおよび代入演算子は，コンパイラによって作成されない

クラスが TObject から派生していることをコンパイラに知らせる必要がある場合は，Object Pascal から翻訳されたクラス宣言にこの修飾子が必要です。

__declspec(delphireturn)

delphireturn 引数は，C++Builder の VCL と CLX 内部でのみ使用されます。この引数は，C++Builder で作成したクラスの宣言に使用し，C++ 本来の型がない Object Pascal の組み込みデータ型および言語構造をサポートできるようにします。これには，Currency，AnsiString，Variant，TDateTime，および Set が含まれます。**delphireturn** 引数は C++ クラスに印をつけて，関数呼び出しの際の引数および戻り値を VCL または CLX と互換性があるものとして扱います。Object Pascal と C++ の間で，構造体を関数に値で渡す場合は，この修飾子が必要です。

__declspec(delphirtti)

delphirtti 引数は，コンパイル時にクラスに実行時型情報を含める働きをします。この修飾子を使うと，コンパイラは，**published** 部で宣言されているすべてのフィールド，メソッド，プロパティに関する実行時型情報を生成します。インターフェースの場合は，インターフェースのすべてのメソッドに関する実行時型情報を生成します。あるクラスを実行時型情報と一緒にコンパイルすると，そのすべての子孫にも実行時型情報が含まれます。TPersistent クラスは実行時型情報と一緒にコンパイルされるので，TPersistent を上位に持つクラスを作成していれば，この修飾子を使う必要はありません。この修飾子は，主に Web サービスを実装または使用するアプリケーションのインターフェースで使用します。

__declspec(dynamic)

dynamic 引数は，動的関数の宣言で使用します。動的関数は仮想関数に似ていますが，それを定義しているオブジェクトの vtable にだけ保存され，下位オブジェクトの vtable には保存されない点が異なります。動的関数を呼び出した場合に，その関数がオブジェクト内で定義されていないときは，その関数が見つかるまで上位オブジェクトの vtable が検索されます。動的関数が有効なのは，TObject から派生したクラスだけです。

__declspec(hidesbase)

hidesbase 引数は，Object Pascal の仮想関数およびオーバーライド関数を C++Builder に移植する場合に，Object Pascal プログラムのセマンティクスを保持します。Object Pascal では，基本クラスの仮想

関数が，派生クラスに同じ名前の関数として現れる可能性があります。ただし，これは基本クラスの仮想関数とは無関係のまったく新しい関数を表しています。

コンパイラは sysmac.h で定義されている HIDESBASE マクロを使って，この種の関数宣言を完全に分離します。たとえば，基本クラス T1 で，引数なしの仮想関数 func が宣言されているとします。T1 の派生クラス T2 で，同じ名前とシグニチャを持つ関数が宣言されている場合は，DCC32 -jphn は次のようなプロトタイプを含む HPP ファイルを作成します。

```
virtual void T1::func(void);
HIDESBASE void T2::func(void);
```

HIDESBASE 宣言がないと，C++ プログラムのセマンティクスでは，仮想関数 T1::func() が T2::func() によってオーバーライドされることになります。

__declspec(package)

package 引数は，クラスを定義するコードが，1 つのパッケージ内でコンパイルできることを表します。IDE でパッケージを作成すると，この修飾子が自動的に作成されます。パッケージについての詳細は，第 15 章「パッケージとコンポーネントの操作」を参照してください。

__declspec(pascalimplementation)

pascalimplementation 引数は，クラスを定義するコードが Object Pascal で実装されていることを表します。この修飾子は，拡張子 .hpp を持つ Object Pascal 移植用ヘッダーファイルで使用されます。

__declspec(uuid)

uuid 引数は，クラスと GUID（globally unique identifier）を関連付ける働きをします。どのクラスでも使用できますが，通常は，Object Pascal のインターフェース（または COM インターフェース）を表すクラスで使います。この修飾子を使って宣言したクラスの GUID を取り出すには，**__uuidof** 指令を呼び出します。

第14章

クロスプラットフォームアプリケーションの開発

C++Builder を使うと，Windows と Linux の両方のプラットフォームで動作するクロスプラットフォームの 32 ビットアプリケーションを開発できます。クロスプラットフォームアプリケーションは CLX コンポーネントを使用し，オペレーティングシステム固有の API 呼び出しは一切行いません。クロスプラットフォームアプリケーションを開発するには，CLX アプリケーションを新規作成するか，既存の Windows アプリケーションを修正します。次に，アプリケーションが動作するプラットフォームでコンパイルおよび配布をします。Windows では，C++Builder を使ってコンパイルします。Linux の場合，Borland C++ ソリューションはまだ利用できませんが，前もって C++Builder を使ってアプリケーションを開発しておくことができます。

この章では，C++Builder アプリケーションを変更して Linux でコンパイル可能にする方法，および Windows と Linux 間で移植可能な，プラットフォームに依存しないコードの書き方について説明します。Windows と Linux 間のアプリケーション開発の違いについても説明します。

クロスプラットフォームアプリケーションの作成

クロスプラットフォームアプリケーションの作成方法は，C++Builder アプリケーションの作成と同じです。クロスプラットフォームアプリケーションを作成するには，CLX というビジュアル / 非ビジュアルコンポーネント群を使います。また，アプリケーションを完全にクロスプラットフォームにするには，オペレーティングシステム固有の API を使用してはいけません（14-15 ページの「移植可能なコードの記述」に，クロスプラットフォームアプリケーションの作成に関するヒントがあります）。

クロスプラットフォームアプリケーションを作成する手順は次のとおりです。

1. IDE で［ファイル｜新規作成｜ CLX アプリケーション］を選択します。コンポーネントパレットに，CLX アプリケーションで使用できるページとコンポーネントが表示されます。
2. IDE 内でアプリケーションを開発します。

3. アプリケーションのコンパイルとテストをします。エラーメッセージを確認し，他に変更する必要のある箇所がないかを調べます。

メモ C++Builder ソリューションが Linux で利用できるようになった時点で，Linux 上でもアプリケーションのコンパイルとテストが行えます。

アプリケーションを Linux に移植した後は，プロジェクトオプションを再設定する必要があります。その理由は，プロジェクトオプションを保存する .dof ファイルが Linux では別の拡張子で再作成されるからです（デフォルトのオプション設定）。

クロスプラットフォームアプリケーションでは，フォームファイルの拡張子は dfm でなく xfm になります。フォームファイルの拡張子が変わる理由は，CLX コンポーネントを使ったクロスプラットフォームフォームと，VCL コンポーネントを使ったフォームとを区別するためです。拡張子 .xfm の付いたフォームファイルは Windows と Linux の両方で動作しますが，.dfm の付いたフォームは Windows でしか動作しません。

プラットフォームに依存しないデータベースやインターネットアプリケーションの作成については，14-19 ページの「クロスプラットフォームのデータベースアプリケーション」と 14-26 ページの「クロスプラットフォームのインターネットアプリケーション」を参照してください。

Windows アプリケーションの Linux への移植

Windows 環境用に作成した C++Builder アプリケーションがある場合，Linux 環境でも使えるように準備しておくことができます。それが簡単にできるかどうかは，アプリケーションの性質，複雑さ，Windows の明示的な依存関係の数によって差があります。

以下の各セクションでは，Windows 環境と Linux 環境の違いを説明し，アプリケーションの移植を開始するための指針を示します。

移植の方法

以下に，あるプラットフォームから別のプラットフォームにアプリケーションを移植する方法を示します。

表 14.1 移植の方法

方法	説明
プラットフォーム固有の移植	特定のオペレーティングシステムとその基礎となる API を目的とする
クロスプラットフォームの移植	クロスプラットフォームの API を目的とする
Windows エミュレーション	コードはそのまま残し，コードで使う API を移植する

プラットフォームに特化した移植

プラットフォームに特化した移植には時間と経費がかかり，単一のプラットフォームを目的とした結果しか得られません。この種の移植ではさまざまなコードベースが作成されるので，保守が特に難しくなります。しかし，特定のオペレーティングシステムごとに移植を設計するので，プラットフォーム固有の機能を利用できます。したがって，アプリケーションは一般に高速で動作します。

クロスプラットフォーム移植

クロスプラットフォーム移植は，移植するアプリケーションが複数のプラットフォームをターゲットとするため，概して時間を節約できます。しかし，クロスプラットフォームのアプリケーションの開発に伴う作業量は，元のコードによって大きく異なります。プラットフォームに依存しないことを考慮せずにコードが書かれた場合，プラットフォームに依存しない「ロジック」とプラットフォームに依存する実装コードが混在しているという状況に陥ります。

ビジネスロジックはプラットフォームに依存しない言葉で表現されるので，クロスプラットフォームの方法が望まれます。一部のサービスは，外観はすべてのプラットフォームに共通でも実装はそれぞれに固有の内部インターフェースの背後で抽象化されます。C++Builder のランタイムライブラリがこれに当たります。どちらのプラットフォームでもインターフェースは非常に似かよっていますが，実装はまったく異なります。クロスプラットフォームの要素を分離してから，各サービスをその上に実装する必要があります。さらに，クロスプラットフォームの移植では，ソースベースの大部分の共有とアプリケーションアーキテクチャの改良によって保守コストが削減されるので，もっともコストが少ない方法です。

Windows エミュレーションの移植

Windows エミュレーションはもっとも複雑な方法であり，非常にコストがかかる場合もありますが，Linux アプリケーションの外観は Windows アプリケーションに非常に近くなります。この方法では，Windows の機能を Linux に実装する必要があります。エンジニアリングの観点から見て，これは非常に保守しにくい方法です。

Windows API をエミュレートしようとする場合，#ifdef を使うと，コードの中で Windows 専用の部分と Linux 専用の部分とを区別することができます。

ユーザーアプリケーションの移植

アプリケーションを移植して Windows と Linux の両方で実行する場合は，コードを変更するか，#ifdef を使って Windows 専用の部分と Linux 専用の部分を区別する必要があります。

次に，VCL アプリケーションを CLX に移植する一般的な手順を示します。

1. C++Builder で変更するアプリケーションの入ったプロジェクトを開きます。
2. .dfm ファイルを同名の .xfm ファイルにコピーします（たとえば，ファイル名 unit1.dfm を unit1.xfm に変更する）。ヘッダートファイル内の .dfm ファイルへの参照を，#pragma resource "*.dfm" から #pragma resource "*.xfm" に変更します（または #ifdef で処理）。なお，.xfm ファイルは C++Builder と Linux アプリケーション の両方で動作します。
3. ソースファイル内のすべてのヘッダーファイルを，CLX の正しいユニットを参照するように変更（または #ifdef を使用）します（詳細については，14-8 ページの「CLX ユニットと VCL ユニットの比較」を参照）。

   たとえば，Windows アプリケーションのヘッダーファイルで以下の #include 文を，

```
#include <vcl.h>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
```

次のように CLX アプリケーション用に変更します。

```
#include <clx.h>
#include <QControls.hpp>
#include <QStdCtrls.hpp>
#include <QForms.hpp>
```

4. プロジェクトをいったん保存し，開き直します。これで，CLX アプリケーションで使用できるコンポーネントがコンポーネントパレットに表示されます。

メモ Windows 専用の非ビジュアルコンポーネントのうち，CLX アプリケーションで使用できるものがいくつかありますが，それらには Windows CLX アプリケーションでしか機能しない機能が含まれています。Windows だけでなく Linux でもアプリケーションをコンパイルする場合には，VCL の非ビジュアルコンポーネントをアプリケーションで使用しないようにするか，または，#ifdef を使って Windows 専用のコード部分を区別する必要があります。VCL のビジュアルな部分を VisualCLX と合わせて 1 つのアプリケーションで使用することはできません。

5. Windows に依存する必要のあるコードを，プラットフォームに依存しないように書き換えます。この作業は，ランタイムライブラリルーチンと定数を使って行います（詳細については，14-15 ページの「移植可能なコードの記述」を参照）。

6. Linux 上で異なる機能に相当する機能を洗い出します。#ifdef を使って（多用はしない），Windows 固有の情報を区別します（詳細については，14-16 ページの「条件指令の使い方」を参照）。

   たとえば，次のようにソースファイル内のプラットフォーム固有のコードを #ifdef で区切ります。

```
#ifdef WINDOWS // C++Builder コンパイラを使用する場合，__WIN32__ を使う
IniFile->LoadfromFile("c: \\ x.txt");
#endif
#ifdef __linux__
IniFile->LoadfromFile("/home/name/x.txt");
#endif
```

7. すべてのプロジェクトファイル内でパス名への参照を検索します。

   - Linux のパス名では区切り文字としてスラッシュが使用され（たとえば /usr/lib），ファイルは Linux システムのさまざまなディレクトリに配置される。SysUtils 内の PathDelim 定数を使うと，システムに固有のパスの区切り文字を指定できる。Linux 上の各ファイルの正確な場所を決定する
   - ドライブを示す文字（たとえば C: \）への参照と，2 文字目のコロンを検索してドライブを示す文字を探すコードを変更する。SysUtils 内の DriveDelim 定数を使って，システムに固有の表現で場所を指定する
   - 複数のパスを指定する箇所では，パスセパレータをセミコロン（;）から コロン（:）に変更する。SysUtils 内の PathSep 定数を使うと，システムに固有のパスセパレータを指定できる。
   - Linux ではファイル名の大文字と小文字を区別するので，アプリケーションでファイル名の大文字と小文字を変更したり，大文字と小文字のいずれかを想定したりしない

8. アプリケーションのコンパイル，テスト，デバッグを行います。

CLX と VCL

CLX アプリケーションでは，VCL（ビジュアルコンポーネントライブラリ）のかわりに CLX（クロスプラットフォーム用コンポーネントライブラリ）を使用します。VCL では，Windows API ライブラリを呼び出すことにより，多くのコントロールが簡単に Windows コントロールにアクセスできます。同様に，CLX でも Qt 共有ライブラリ内の Qt ウィジェット（ウィンドウ＋ガジェット）にアクセスできます。C++Builder には，CLX と VCL の両方が同梱されています。

CLX の外観は VCL によく似ています。コンポーネントもプロパティも，ほとんどは VCL と CLX で同じ名前が付いています。さらに，VCL と同様に CLX も Windows で利用できます（C++Builder のバージョン（版）によっては利用できない場合もあります）。

CLX コンポーネントは以下のパーツに分類されます。

表 14.2　CLX のパーツ

パーツ	説明
VisualCLX	固有のクロスプラットフォーム GUI コンポーネントおよびグラフィック。
DataCLX	クライアントのデータアクセスコンポーネント。この領域のコンポーネントは，クライアントデータセットに基づくローカル，クライアント / サーバー，n 層の下位セットである。コードは Linux でも Windows でも同じである
NetCLX	Apache DSO や CGI WebBroker などのインターネットコンポーネント。Linux と Windows の間に違いはない
BaseCLX	Classes ユニットまでのランタイムライブラリ（Classes ユニットを含む）。コードは Linux でも Windows でも同じである

VisualCLX ウィジェットは Windows コントロールの代わりになります。たとえば，CLX の TWidgetControl は VCL の TWinControl の代わりになります。TScrollingWinControl などの VCL コンポーネントは，CLX では別の名前（TScrollingWidget など）になります。ただし，TWinControl を TWidgetControl に変更する必要はありません。たとえば次の型宣言は，

```
TWinControl = TWidgetControl;
```

QControls ユニットファイルに含まれるもので，ソースコードの共有が簡単になるようにしています。TWidgetControl とその下位オブジェクトには，いずれも Qt オブジェクトへの参照を表す Handle プロパティと，イベントメカニズムを操作するフックオブジェクトへの参照を表す Hooks プロパティがあります。

一部のクラスのユニット名と場所は，CLX では異なります。CLX に存在しないユニットへの参照をなくし，CLX ユニットの名前を変更するため，ソースファイルにインクルードするヘッダーファイルを変更する必要があります。

CLX で異なる機能

CLX の大部分は VCL と一致するように実装されていますが，異なる機能もいくつかあります。この節では，CLX と VCL の実装の相違点をいくつか紹介します。特にクロスプラットフォームアプリケーションの作成時に注意すべき相違点について説明します。

ルックアンドフィール

Linux のビジュアル環境は，Windows のそれとはやや異なっています。ダイアログの外観は，使用するウィンドウマネージャ（KDE，Gnome など）によって異なります。

スタイル

OwnerDraw プロパティのほかに，アプリケーション全体のスタイルを指定できます。アプリケーションのグラフィック要素全体のルックアンドフィールを設定するには，TApplication::Style プロパティを使います。スタイルを使用すると，ウィジェットまたはアプリケーションは全体の外観を変えることができます。Linux 上ではオーナー描画も使用できますが，スタイルの使用をお勧めします。

Variants

System ユニット にあったバリアントおよび SafeArray のコードは，すべて次の 2 つのユニット内にあります。

- Variants
- VarUtils

オペレーティングシステムに依存するコードは，現在は VarUtils ユニットに分離されています。これには，Variants で必要とするあらゆる汎用のバージョンも存在します。Windows 呼び出しのある VCL アプリケーションを CLX アプリケーションに変換する場合は，これらの呼び出しを VarUtils ユニットへの呼び出しに置き換える必要があります。

バリアントを使用する場合は，ソースファイル内のヘッダーファイルに Variants ユニットを追加しなければなりません。

VarIsEmpty は，varEmpty に対する簡単なテストを行い，バリアントがクリアされているかどうかを調べます。Linux 上で VarIsClear 関数を使うと，バリアントの値が未定義かどうかを確認できます。

Registry

Linux では，環境設定情報の保存にレジストリを使いません。レジストリのかわりに，テキストの環境設定ファイルと環境変数を使用します。Linux 上のシステム環境設定ファイルは，一般に /etc（/etc/hosts など）に配置されます。他のユーザープロファイルは，ドットファイル（ピリオドで始まる）に格納されます。ドットファイルには，bash シェルの設定値を保持する .bashrc や，X プログラムのデフォルト値を設定する .XDefaults などがあります。

レジストリに依存するコードは，代わりにローカル環境設定テキストファイルを使って変更できます。ユーザーが変更できる設定はユーザーのホームディレクトリに保存して，ユーザーが書き込めるようにします。root ユーザーが設定する必要のある環境設定オプションは，/etc に置きます。レジストリの機能をすべてユニットに記述し，すべての出力をローカル環境設定ファイルに転送するのも，元のレジストリ依存性を処理する 1 つの方法です。

Linux 上のグローバルな位置に情報を配置するには，グローバルな環境設定ファイルを /etc ディレクトリに保存するか，ユーザーのホームディレクトリに隠しファイルとして保存します。これにより，すべてのアプリケーションが同じ環境設定ファイルにアクセスできるようになります。ただし，ファイルの許可とアクセス権を正しく設定しておかなければなりません。

クロスプラットフォームアプリケーションでも ini ファイルを使用できます。ただし，CLX では，TRegIniFile のかわりに TMemIniFile を使う必要があります。

その他の違い

CLX と VCL とでは，ほかにもコンポーネントの動作に影響する実装の違いがあります。このセクションでは，その違いのいくつかについて説明します。

- CLX コンポーネントの場合，コンポーネントパレットで選択した後，左右どちらのマウスボタンをクリックしてもフォームに追加できます。一方，VCL コンポーネントはマウスの左ボタンしか使用できません。
- CLX の TButton コントロールには ToggleButton プロパティがありますが，VCL の TButton コントロールにはありません。
- CLX の TColorDialog には，TColorDialog::Options プロパティの設定がありません。したがって，色の選択ダイアログの外観と機能をカスタマイズすることはできません。また，Linux で使用するウィンドウマネージャによっては，TColorDialog は必ずしもモード付きで，サイズ変更不能ではありません。Windows では，TColorDialog は常にモード付きでサイズ変更不能です。
- 実行時には，CLX のコンボボックスの機能は VCL のそれとは異なります。CLX では，コンボボックスの編集フィールドにテキストを入力して〔Enter〕を押すと，ドロップダウンリストに項目を追加できます（VCL のコンボボックスではできません）。InsertMode を ciNone に設定すれば，この機能がオフになります。コンボボックスのリストには，空の（文字列のない）項目を追加することもできます。また，編集ボックスが閉じているときに〔↓〕を押し続けると，コンボボックスのリストの最後の項目では止まらず，一番上に戻ります。
- TCustomEdit では Undo，ClearUndo，CanUndo を実装できません。したがって，編集をプログラム的に取り消す方法はありません。しかし，アプリケーションユーザーは，編集ボックス（TEdit）を右クリックして［取り消し］を選択すれば，編集ボックスで編集を取り消すことができます。
- イベントに使われるキー値が VCL と CLX で異なる場合があります。たとえば，【Enter】キーのキー値は VCL が 13，CLX は 4100 です。CLX アプリケーションでキー値をハードコードする場合には，Windows と Linux の間で移植するときにキー値を変更する必要があります。

このほかにもいくつか違いがあります。CLX オブジェクトの詳細については，CLX オンラインドキュメントを参照してください。また，ソースコードが同梱されている C++Builder をお持ちの場合は，コードを参照できます（インストール先ディレクトリ \ CBuilder6 \ Source \ CLX にあります）。

CLX にない機能

VCL のかわりに CLX を使っても，オブジェクトの多くは違いがありません。しかし，オブジェクトの機能がいくつか欠けている場合があります（プロパティ，メソッド，イベントなど）。CLX には以下の一般的な機能がありません。

- 双方向プロパティ（BidiMode）による右から左へのテキスト出力または入力
- コモンコントロールの汎用ベベルプロパティ（一部のオブジェクトにはベベルプロパティがある）
- ドッキングプロパティおよびメソッド

- ［Win3.1］タブや Ctl3D のコンポーネントに関する下位互換性
- DragCursor と DragKind（ただしドラッグアンドドロップを含む）

直接的には移植されない機能

C++Builder でサポートする Windows 固有の機能には，Linux 環境にそのまま移植されないものがあります。たとえば COM，ActiveX，OLE，BDE，ADO は Windows のテクノロジーに依存しており，Linux では利用できません。次の表に，2 つのプラットフォーム間で異なる機能をリストアップし，対応する Linux/CLX 機能を利用できる場合はその機能を示します。

表 14.3　変更する機能または異なる機能

Windows/VCL の機能	Linux/CLX の機能
ADO コンポーネント	標準のデータベースコンポーネント
オートメーションサーバー	利用不可
BDE	dbExpress および標準のデータベースコンポーネント
COM+ コンポーネント（ActiveX など）	利用不可
DataSnap	利用不可
FastNet	利用不可
従来形式のコンポーネント（コンポーネントパレットの［Win 3.1］タブの項目など）	利用不可
Messaging Application Programming Interface（MAPI）。Windows メッセージング機能の標準ライブラリで構成される	SMTP と POP3。電子メールメッセージの送信，受信，保存が可能
Quick Report	利用不可
Web サービス（SOAP）	利用不可
WebSnap	利用不可
Windows API 呼び出し	CLX メソッド，Qt 呼び出し，libc 呼び出し，または他のシステムライブラリの呼び出し
Windows メッセージング	Qt イベント
Winsock	BSD ソケット

Windows dll に相当する Linux の機能は，共有オブジェクトライブラリ（.so ファイル）です。このライブラリには，位置に依存しないコード（PIC）があります。よって，グローバル変数参照と外部関数呼び出しは，EBX レジスタを基準として作成されます。このレジスタは，呼び出し間で一貫して保持しなければなりません。

グローバルメモリ参照と外部関数呼び出しについて心配が必要なのは，アセンブラを使用する場合に限ります。それ以外の場合は，C++Builder によって正しいコードが生成されます（詳細については，14-18 ページの「アセンブラコードのをインライン記述」を参照）。

CLX ユニットと VCL ユニットの比較

VCL および CLX のオブジェクトは，すべてヘッダーファイル内で定義されます。たとえば，System ユニットでは TObject が実装され，Classes ユニットでは TComponent 基本クラスが定義されます。フォームの上にオブジェクトをドロップするか，またはアプリケーションの中でオブジェクトを使用

すると，ソースファイルにインクルードされているヘッダーファイルにユニットの名前が追加されます。これにより，コンパイラはプロジェクトにどのユニットをリンクするかを判断します。

この節では，VCL と CLX のユニットを 3 つの表で示します（VCL ユニットとそれに対応する CLX ユニット，CLX のみのユニット，VCL のみのユニット）。

次の表は，VCL のユニットとそれに相当する CLX のユニットです。VCL と CLX で同一のユニット，およびサードパーティのユニットは記載していません。

表 14.4　VCL ユニットと CLX ユニットの対応表

VCL のユニット	CLX のユニット
ActnList	QActnList
Buttons	QButtons
CheckLst	QCheckLst
Clipbrd	QClipbrd
ComCtrls	QComCtrls
Consts	Consts，QConsts，RTLConsts
Controls	QControls
DBActns	QDBActns
DBCtrls	QDBCtrls
DBGrids	QDBGrids
Dialogs	QDialogs
ExtCtrls	QExtCtrls
Forms	QForms
Graphics	QGraphics
Grids	QGrids
ImgList	QImgList
Mask	QMask
Menus	QMenus
Printers	QPrinters
Search	QSearch
StdActns	QStdActns
StdCtrls	QStdCtrls
Types	Types，QTypes
VclEditors	ClxEditors

以下のユニットは，CLX にはありますが VCL にはありません。

表 14.5　CLX にしかないユニット

ユニット	説明
DirSel	ディレクトリの選択
QStyle	GUI のルックアンドフィール
Qt	Qt ライブラリへのインターフェース

以下の Windows VCL ユニットは，CLX にはありません。その理由の多くは，ADO，BDE，COM など，Linux では利用できない Windows 固有の機能に関連することです。

表 14.6 VCL にしかないユニット

ユニット	除外の理由
ADOConst	ADO 機能がない
ADODB	ADO 機能がない
AppEvnts	TApplicationEvent オブジェクトがない
AxCtrls	COM 機能がない
BdeConst	BDE 機能がない
Calendar	現時点ではサポートしていない
Chart	現時点ではサポートしていない
CmAdmCtl	COM 機能がない
ColorGrd	現時点ではサポートしていない
ComStrs	COM 機能がない
ConvUtils	利用不可
CorbaCon	Corba 機能がない
CorbaStd	Corba 機能がない
CorbaVCL	Corba 機能がない
CtlPanel	Windows コントロールパネルがない
CustomizeDlg	現時点ではサポートしていない
DataBkr	現時点ではサポートしていない
DBCGrids	BDE 機能がない
DBExcept	BDE 機能がない
DBInpReq	BDE 機能がない
DBLookup	使われなくなっている
DbOleCtl	COM 機能がない
DBPWDlg	BDE 機能がない
DBTables	BDE 機能がない
DdeMan	DDE 機能がない
DRTable	BDE 機能がない
ExtActns	現時点ではサポートしていない
ExtDlgs	画像ダイアログ機能がない
FileCtrl	使われなくなっている
ListActns	現時点ではサポートしていない
MConnect	COM 機能がない
Messages	Windows メッセージングがない
MidasCon	使われなくなっている
MPlayer	Windows メディアプレーヤーがない
Mtsobj	COM 機能がない
MtsRdm	COM 機能がない
Mtx	COM 機能がない
mxConsts	COM 機能がない
ObjBrkr	現時点ではサポートしていない

表 14.6 VCL にしかないユニット（つづき）

ユニット	除外の理由
OleConstMay	COM 機能がない
OleCtnrs	COM 機能がない
OleCtrls	COM 機能がない
OLEDB	COM 機能がない
OleServer	COM 機能がない
Outline	使われなくなっている
Registry	Windows のレジストリ機能がない
ScktCnst	Sockets に置き換え
ScktComp	Sockets に置き換え
SConnect	サポートする接続プロトコルがない
SHDocVw_ocx	ActiveX 機能がない
StdConvs	現時点ではサポートしていない
SvcMgr	Windows NT サービスの機能がない
TabNotbk	使われなくなっている
Tabs	使われなくなっている
ToolWin	ドッキング機能がない
ValEdit	現時点ではサポートしていない
VarCmplx	現時点ではサポートしていない
VarConv	現時点ではサポートしていない
VCLCom	COM 機能がない
WebConst	Windows の定数がない
Windows	Windows API 呼び出しがない

CLX オブジェクトコンストラクタの違い

CLX オブジェクトを作成すると，フォームに配置して暗黙的に作成したか，オブジェクトのコンストラクタを使ってコードで明示的に作成したかにかかわらず，その基礎となるウィジェットのインスタンスも作成されます。CLX オブジェクトは，ウィジェットのインスタンスを所有します。CLX オブジェクトが削除されると，その基礎となるウィジェットも削除されます。これは，Windows アプリケーションの VCL の機能と同様です。

QWidget_Create() のように，Qt インターフェースライブラリを呼び出してコードで明示的に CLX オブジェクトを作成すると，CLX オブジェクトに所有されない Qt ウィジェットのインスタンスが作成されます。これにより，構築中に使用する既存の Qt ウィジェットのインスタンスが CLX オブジェクトに渡されます。この CLX オブジェクトは，自分に渡された Qt ウィジェットを所有していません。したがって，この方法でオブジェクトを作成した後，CLX オブジェクトは破棄されますが，基礎となる Qt ウィジェットのインスタンスは残ります。これは VCL とは異なります。

TBrush，TPen などいくつかの CLX グラフィックオブジェクトでは，OwnHandle メソッドを使うことにより，基礎となるウィジェットの所有関係を管理できます。OwnHandle を呼び出してから CLX オブジェクトを削除すると，基礎となるウィジェットも破棄されます。

CLX での一部のプロパティ割り当ては，コンストラクタから InitWidget に移動しています。これで，Qt オブジェクトが実際に必要になるまで，構築を遅らせることができます。たとえば，Color というプロパティがあるとします。SetColor では，HandleAllocated を使って Qt に Handle があるかどうかを確認できます。ハンドルが割り当てられている場合は，Qt への適切な呼び出しを作成して Color を設定できます。Handle が割り当てられていない場合は，プライベートフィールド変数に値を保存して，InitWidget でプロパティを設定します。

オブジェクト構築の詳細については，13-7 ページの「C++Builder の VCL/CLX クラスのためのオブジェクトの構築」を参照してください。

システムイベントとウィジェットイベントの処理

システムイベントとウィジェットイベントは，コンポーネントを記述するときの大きなポイントになりますが，その処理の仕方は VCL と CLX で異なります。一番重要な違いは，CLX コントロールが Windows メッセージに直接的に応答しないことです。CLX コントロールが Windows 上で動作しているときも直接は応答しません（第 51 章「メッセージとシステム通知の処理」を参照）。直接応答するのでなく，基礎となるウィジェットレイヤからの通知に応答します。ウィジェットレイヤからの通知は別のシステムを使用しているため，CLX オブジェクトとそれに対応する VCL オブジェクトの間でイベントの順序やタイミングが異なることがあります。CLX アプリケーションが Linux でなく Windows で動作していても，この違いは発生します。VCL アプリケーションを CLX に移植する場合は，この違いに対処するため，必要に応じてイベントハンドラの応答の仕方を変更してください。

システムイベントとウィジェットイベント（CLX コンポーネントのパブリッシュイベントに反映されるイベント以外）に応答するコンポーネントの記述についての詳細は，51-10 ページの「CLX によるシステム通知への応答」を参照してください。

Windows と Linux のソースファイルの共有

Linux と Windows の両方でアプリケーションを実行する場合は，2 つのプラットフォーム間でソースファイルを共有し，両方のオペレーティング システムからアクセスすることができます。これには，両方のコンピュータにアクセスできるサーバー上にソースファイルを配置する，Linux マシン上で Samba を使って Linux と Windows 両方の Microsoft ネットワークファイル共有を介してソースファイルにアクセスするなど，いくつかの方法があります。Linux 上にソースを保持し，Linux 上に共有ドライブを作成することもできます。また，Windows 上にソースを保持し，Windows 上に Linux マシンがアクセスする共有を作成することもできます。

C++Builder で，VCL と CLX の両方がサポートするオブジェクトを使って開発とコンパイルを継続できます。終了すると，Windows でコンパイルできます。Linux C++ ソリューションが利用可能になれば，Linux でコンパイルすることもできます。

C++Builder で CLX アプリケーションを新規作成すると，.dfm ファイルでなく .xfm ファイルが作成されます。

Windows と Linux の環境の違い

現在のところ，「クロスプラットフォーム」とは，Windows と Linux の両プラットフォームでほとんど相違なくコンパイルできるアプリケーションを意味します。下の表に Linux と Windows の動作環境の主な相違点を示します。

表 14.7　Linux と Windows のオペレーティング環境の違い

違い	説明
ファイル名における大文字と小文字の区別	Linux では，ファイル名の大文字と小文字を区別します。Test.txt と test.txt は同じファイルではありません。Linux では，ファイル名の大文字小文字に十分注意する必要があります。
行の終了文字	Windows では CR/LF（すなわち ASCII 13 + ASCII 10）で行が終了しますが，Linux では LF で終了します。コードエディタはこの違いを処理できますが，Windows からコードをインポートする場合はこのことを忘れないでください。
ファイル終了文字	MS-DOS と Windows では，#26（Ctrl-Z）の値がテキストファイルの終わりとして処理され，この文字の後にデータがあっても終了と見なされます。Linux は Ctrl+D をファイル終了文字に使います。
バッチファイルとシェルスクリプト	Linux 環境で .bat ファイルに相当するのはシェルスクリプトです。スクリプトは命令を記述したテキストファイルであり，コマンド `chmod +x <scriptfile>` で実行できます。スクリプト言語は，Linux 上で使用するシェルによって異なります。通常は，Bash が使用されます。
コマンドの確認	MS-DOS または Windows では，ファイルまたはフォルダを削除しようとすると，確認メッセージ（「～してよろしいですか ?」など）が表示されます。一般に，Linux ではメッセージは表示されず，直ちに操作が行われます。したがって，ファイルやファイルシステム全体を誤って破壊する確率が高くなります。ファイルを別のメディアにバックアップしておかない限り，Linux 上で行った削除を取り消すことはできません。
コマンドのフィードバック	Linux 上でコマンドが正常に実行されると，ステータスメッセージなしでコマンドプロンプトが再表示されます。
コマンドスイッチ	Linux では，ダッシュ（-）でコマンドスイッチを表し，2 つのダッシュ（--）で複数文字のオプションを表します。DOS ではスラッシュ（/）またはダッシュ（-）を使います。
環境設定ファイル	Windows では，レジストリや autoexec.bat などのファイルで環境設定を行います。 Linux では，環境設定ファイルは隠しファイルとしてユーザーのホームディレクトリに作成されます。/etc ディレクトリにある環境設定ファイルは，通常は隠しファイルではありません。 Linux では，LD_LIBRARY_PATH（ライブラリの検索パス）などの環境変数も使用します。以下に，他の重要な環境変数を示します。 HOME　ユーザーのホームディレクトリ（/home/sam） TERM　端末の種類（xterm，vt100，コンソール） SHELL　シェルへのパス（/bin/bash） USER　ユーザーのログイン名（sfuller） PATH　プログラムを検索するパスのリスト 環境変数は，シェルまたは .bashrc などのファイルで指定します。
DLL	Linux では，共有オブジェクトファイル（.so）を使用します。Windows のダイナミックリンクライブラリ（DLL）に相当します。

表 14.7 Linux と Windows のオペレーティング環境の違い（つづき）

違い	説明
ドライブを示す文字	Linux にはドライブを示す文字がありません。たとえば，Linux のパス名は /lib/security のようになります。ランタイムライブラリの DriveDelim を参照してください。
例外	オペレーティングシステム例外は，Linux ではシグナルと呼ばれます。
実行形式ファイル	Linux では，実行形式ファイルに拡張子はありません。Windows では，実行形式ファイルに拡張子 exe が付きます。
ファイル名拡張子	Linux では，ファイルの種類の識別や，ファイルとアプリケーションの関連付けにファイル名拡張子を使用しません。
ファイルの許可	Linux では，ファイル（とディレクトリ）に対して，オーナー，グループ，その他のユーザーに関する読み出し，書き込み，実行の許可を割り当てます。下に例を示します。 -rwxr-xr-x は左から右へ以下のような意味を表します。 - はファイルの種類（- = 通常のファイル，d = ディレクトリ，l = リンク），rwx はファイルオーナーの許可（読み出し，書き込み，実行），r-x はファイルオーナーグループの許可（読み出し，実行），さらに r-x は他のすべてのユーザーの許可（読み出し，書き込み）です。root ユーザー（スーパーユーザー）は，以上の許可の制約を受けません。 アプリケーションが適切なユーザーによって実行され，必要なファイルへの適切なアクセス権があるようにしておく必要があります。
make ユーティリティ	Borland の make ユーティリティは，Linux プラットフォームでは利用できません。かわりに，Linux の GNU make ユーティリティを使用します。
マルチタスキング	Linux では，マルチタスキングを完全にサポートします。同時に複数のプログラム（Linux では プロセスと呼ぶ）を実行できます。プロセスをバックグラウンドで起動し（コマンドの後に & を付ける），そのまま作業を継続できます。Linux では，複数のセッションを使用できます。
パス名	DOS で円記号（\）を使う箇所は，Linux では必ずスラッシュ（/）を使います。PathDelim 定数を使うと，プラットフォームに固有の文字を指定できます。ランタイムライブラリの PathSep を参照してください。
検索パス	プログラムを実行するときに，Windows では必ずカレントディレクトリを最初にチェックしてから PATH 環境変数を参照します。Linux ではカレントディレクトリを参照せずに PATH に示すディレクトリのみを検索します。カレントディレクトリ内のプログラムを実行するには，通常はプログラムの前に ./ を入力する必要があります。 PATH に最初の検索パスとして ./ を追加することもできます。
検索パスセパレータ	Windows では，検索パスセパレータとしてセミコロンを使用します。Linux ではコロンを使用します。ランタイムライブラリの PathSep を参照してください。
シンボリックリンク	Linux では，シンボリックリンクはディスク上の別のファイルを指す特殊なファイルです。アプリケーションのメインファイルを指すシンボリックリンクをグローバルな bin ディレクトリに配置すると，システム検索パスを変更する必要はありません。シンボリックリンクは ln（link）コマンドで作成されます。 Windows には，GUI デスクトップのショートカットがあります。コマンドラインでプログラムを実行できるようにするには，通常，Windows のインストールプログラムでシステム検索パスを変更します。

Linux のディレクトリ構造

Linux のディレクトリは Windows とは異なります。ファイルまたはデバイスは，ファイルシステムのどこにでもマウントできます。

メモ Linux のパス名にはスラッシュを使用しますが，Windows のパス名は円記号（\）を使います。Linux の最初のスラッシュは，ルートディレクトリを表します。

以下に，Linux で一般的に使用するディレクトリをいくつか示します。

表 14.8 一般的な Linux ディレクトリ

ディレクトリ	内容
/	Linux ファイルシステム全体のルートすなわち最上位ディレクトリ
/root	ルートファイルシステムすなわちスーパーユーザーのホームディレクトリ
bin	コマンド，ユーティリティ
/sbin	システムユーティリティ
/dev	ファイルとして表されるデバイス
/lib	ライブラリ
/home/username	ユーザーが所有するファイル。ここで username はユーザーのログイン名
/opt	オプション
/boot	システム起動時に呼び出されるカーネル
/etc	環境設定ファイル
/usr	アプリケーション，プログラム。通常，/usr/spool，/usr/man，/usr/include，/usr/local などのディレクトリがある
/mnt	CD やフロッピーディスクドライブなどのシステムにマウントされた他のメディア
/var	ログ，メッセージ，スプールファイル
/proc	仮想ファイルシステムとレポートシステム統計
/tmp	一時ファイル

メモ Linux のディストリビューションが異なると，ファイルを別の場所に配置する場合があります。ユーティリティプログラムは，Red Hat ディストリビューションでは /bin に配置され，Debian ディストリビューションでは /usr/local/bin に配置されます。

UNIX/Linux 階層ファイルシステムの構造についての詳細は，www.pathname.com を参照し，「Filesystem Hierarchy Standard」をお読みください。

移植可能なコードの記述

Windows，Linux の両方で動作するクロスプラットフォームアプリケーションとして記述すれば，異なる条件下でコンパイルできるコードを作成できます。条件コンパイルを使うと，Windows のコーディングを保持したまま，Linux オペレーティングシステムの違いを考慮に入れることができます。

Windows と Linux との間で簡単に移植できるアプリケーションを作成するには，以下の注意が必要です。

- プラットフォーム固有の（Win32 または Linux）API 呼び出しを減らすかまたは分離する。かわりに，CLX のメソッドか Qt ライブラリの呼び出しを使う

- アプリケーション内の Windows メッセージング（PostMessage，SendMessage）構造をなくす。CLX では，かわりに QApplication_postEvent メソッドと QApplication_sendEvent メソッドを呼び出す。システムイベントとウィジェットイベントに応答するコンポーネントの記述についての詳細は，51-10 ページの「CLX によるシステム通知への応答」を参照
- TRegIniFile のかわりに TMemIniFile を使う
- ファイル名とディレクトリ名の大文字と小文字の区別に注意する
- 外部アセンブラ TASM コードがあればこれを移植する。GNU アセンブラ as では，TASM の構文はサポートされない（14-18 ページの「アセンブラコードのをインライン記述」を参照してください）。

プラットフォームに依存しないランタイムライブラリを使用し，System，SysUtils などのランタイムライブラリ ユニット内の定数を使用するコードを記述します。たとえば，PathDelim 定数を使って '/' と '\' というプラットフォーム間の違いに対応します。

他には，両方のプラットフォームでマルチバイト文字を使用するという例もあります。Windows のコードでは，2 バイトを 1 マルチバイト文字と見なします。Linux のマルチバイト文字の符号化では，1 文字が 2 バイトを超える場合があります（UTF-8 では最大 6 バイト）。SysUtils の StrNextChar 関数を使用すれば，両方のプラットフォームに対応できます。次のような既存の Windows コードは，

```
while(*p != 0)
{
   if(LeadBytes.Contains(*p))
      p++;
   p++;
}
```

次のようなプラットフォームに依存しないコードに置き換えられます。

```
while(*p != 0)
{
   if(LeadBytes.Contains(*p))
     p = StrNextchar(p);
   else
     p++;
}
```

この例はプラットフォーム間で移植できます。しかも，マルチバイト以外の環境の手続き呼び出しに関するパフォーマンスコストがかかりません。

ランタイムライブラリを使う方法では動作しない場合は，ルーチン内のプラットフォーム固有のコードをひとまとめにするか 1 つのサブルーチンとして分離します。#ifdef ブロックの数を制限し，ソースコードの読みやすさと移植可能性を保持します。条件シンボル WIN32 は Linux では定義されません。条件シンボル LINUX が定義されると，ソースコードが Linux プラットフォーム用にコンパイルされることを示します。

条件指令の使い方

#ifdef コンパイラ指令を使うのは，Windows プラットフォームと Linux プラットフォームに条件を設定する適切な方法です。ただし，#ifdef を使用するとソースコードがわかりにくくなり，保守も難しくなるので，どんな場合に #ifdef を使用するのが妥当かを理解しておく必要があります。#ifdef の使

用を考える場合の最初の質問は，「なぜこのコードに #ifdef が必要か」，そして「このコードを #ifdef なしで記述できないか」ということです。

クロスプラットフォームのアプリケーションで #ifdef を使用する場合は，以下の指針に従ってください。

- #ifdef は，どうしても必要な場合以外は使わないようにします。ソースファイル内の #ifdef は，ソースコードをコンパイルするときにだけ評価されます。C/C++ では，プロジェクトをコンパイルするのにユニットソース（ヘッダーファイル）が必要です。多くの C++Builder プロジェクトでは，ソースコード全体を完全に再構築するのは一般的ではない。
- パッケージ（.bpk）ファイルでは #ifdef を使わないようにし，ソースファイルのみで使用するようにします。コンポーネント開発者は，クロスプラットフォームの開発を行う場合には，1 つのパッケージで #ifdef を使うのではなく，2 つの設計時パッケージを作成します。
- 一般に，#ifdef WINDOWS は WIN32 を含む任意の Windows プラットフォームかどうかをテストするのに使用します。#ifdef WIN32 は，32 ビットと 64 ビットなど，特定の Windows プラットフォームを区別するのに使用します。コードが WIN64 で動作しないことが確実な場合以外は WIN32 に限定しないようにします。
- #ifndef のような否定的なテストは，どうしても必要でない限り使わないようにします。#ifndef__linux__ は，#ifdef WINDOWS と同じではありません。
- #ifndef / #else の組み合わせは使わないようにします。肯定的なテスト（#ifdef）を使った方が読みやすくなります。
- #else 節は，プラットフォームの影響を受ける #ifdef では使わないようにします。LINUX 固有のコードおよび Windows 固有のコードについては，#ifdef__linux__/#else や #ifdef WINDOWS/#else ではなく，別々の #ifdef ブロックを使うようにします。

  たとえば，以下のような古いコードがあるとします。

```
#ifdef WIN32
  { 32 ビット Windows のコード }
#else
  { 16 ビット Windows のコード } //!! 誤って，Linux でこのコードが実行されてしまう可能性がある
#endif
```

  #ifdef 内に移植できないコードがある場合は，プラットフォームが #else 節に入って不可解な実行時エラーになるよりも，ソースコードがコンパイルエラーになる方が，問題点を見つけやすくなります。
- 複雑なテストには #if 構文を使うようにします。#ifdef のネストは，#if 指令の論理式に置き換えます。#if 指令は，#endif で終了させます。これで，#if 式を #ifdef の中に記述でき，新しい #if 構文を以前のコンパイラから隠すことができます。

すべての条件指令について，オンラインヘルプに説明があります。また，オンラインヘルプの「条件コンパイル」も参照してください。

メッセージの生成

#pragma message コンパイラ指令を使用すると，コンパイラとまったく同様にソースコードで警告とエラーを生成できます。プログラムコードの中でユーザー定義メッセージを指定するには，#pragma message を使います。以下のフォーマットのどれかを使用してください。

文字列定数の数が可変の場合：

```
#pragma message( "hi there" )
#pragma message( "hi" " there" )
```

メッセージのあとにテキストを記述する場合：

```
#pragma message text
```

前に定義した値を拡張する場合：

```
#pragma message (text)
  #define text "a test string"
  #pragma message (text)
```

これらのメッセージをボタンに表示する例を下に示します。

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    #pragma message( "hi there 1" )
    #pragma message( "hi " "there 2" )

    #pragma message hi there 3

    #define text "a test string"
    #pragma message (text)
}
```

このメッセージを IDE で表示するには，［プロジェクト｜オプション］を選択し，［コンパイラ］タブをクリックして，［メッセージを表示］チェックボックスをチェックします。

アセンブラコードのをインライン記述

Windows アプリケーションでインラインにアセンブラコードを記述した場合，Linux で同じコードを使用できないかもしれません。Linux では位置に依存しないコード（PIC）が必要なためです。Linux 共有オブジェクトライブラリ（DLL に相当）では，すべてのコードを変更することなくメモリ内に再配置できる必要があります。このことは，主にインラインのアセンブラルーチンでグローバル変数などの絶対アドレスを使用するものや外部関数を呼び出すものに影響を及ぼします。

C++ コードのみで構成するユニットの場合は，必要であればコンパイラが自動的に PIC を生成します。ソースファイルは，それぞれ PIC 形式と PIC 以外の形式にコンパイルするとよいでしょう。-VP コンパイラオプションを使うと，PIC が生成されます。

実行形式にコンパイルするか共有ライブラリにコンパイルするかによって，異なるアセンブラルーチンのコードを作成し，#ifdef __PIC__ で分岐して 2 種類のアセンブラコードを使い分けてください。それ以外には，C++ でルーチンを記述し直せばこの問題を回避できます。

以下に，インラインアセンブラコードに関する PIC の規則を示します。

- PIC では，すべてのメモリ参照が EBX レジスタを基準として作成されていることが必要である。EBX レジスタには，現在のモジュールのベースアドレスポインタが入る（Linux では Global Offset Table または GOT と呼ばれる）。したがって，次のコードのかわりに

  ```
  MOV EAX,GlobalVar
  ```

  次のコードを使用する

  ```
  MOV EAX,[EBX].GlobalVar
  ```

- PIC では，アセンブリコードの呼び出し間で一貫して EBX レジスタを保持し（Win32 と同様），EBX レジスタを復元してから外部関数を呼び出す必要がある（Win32 と異なる）
- PIC コードは基本実行形式で機能するが，パフォーマンスが低下し，生成コードが大きくなることがある。共有オブジェクトでは選択の余地はないが，実行形式では可能な最高水準のパフォーマンスを得ることが望まれる

Linux 上のプログラミングの違い

Linux の wchar_t widechar は，1 文字が 32 ビットです。VCL と CLX がサポートする 16 ビット Unicode 標準は，Linux と GNU ライブラリがサポートする 32 ビット UCS 標準のサブセットです。WideString 型を wchar_t として OS 関数に渡す場合は，あらかじめ 1 文字 32 ビットに広げておく必要があります。

Linux では，WideStrings は長い文字列と同様に参照カウントで管理されます（Windows では異なる）。

Windows では，マルチバイト文字（MBCS）は 1 バイトか 2 バイトの文字コードで表されます。Linux では，1 ～ 3 バイトで表されます。

AnsiString は，ユーザーの環境設定によって，マルチバイト文字列を保持できます。Linux における日本語，中国語，ヘブライ語，アラビア語などのマルチバイト文字のエンコーディングは，Windows における同じロケールのエンコーディングとの互換性がないことがあります。Unicode は移植できますが，マルチバイト文字は移植できません。国際化対応アプリケーションの各種ロケールで使われる文字列の処理について，詳しくは 16-2 ページの「コードを多国語対応にする」を参照してください。

クロスプラットフォームのデータベースアプリケーション

Windows の C++Builder では，データベース情報にアクセスする方法を選択できます。ADO，ボーランドデータベースエンジン（BDE），および InterBase Express を使用する方法があります。C++Builder のバージョン（版）によっては，Windows と Linux の両方で dbExpress（クロスプラットフォームのデータアクセス技術）を使用できます。

データベースアプリケーションを Linux で実行するために dbExpress に移植する前に，dbExpress を使った方法と従来の方法との違いを理解する必要があります。さまざまなレベルの違いがあります。

- 最下位レベルには，アプリケーションとデータベースサーバーとの通信を行う層がある。これは，ADO，BDE，InterBase のいずれかのクライアントソフトウェアである。この層は，dbExpress で置き換えられる。dbExpress は，動的な SQL 処理を行う軽量ドライバの集まりである
- 低レベルのデータアクセスは，データモジュールまたはフォームに追加するコンポーネントの集まりとしてラッピングされている。これらのコンポーネントには，データベースサーバーへの接続を表すデータベース接続や，サーバーから取得したデータを表すデータセットがある。dbExpress カーソルが単一方向なので，非常に重要な違いはいくつかあるが，データベース接続コンポーネントと同様にデータセットもすべて派生元が同じなので，このレベルでは違いがあまり明らかにされない
- ユーザーインターフェースのレベルでは，違いがもっとも少ない。CLX のデータベース対応コントロールは，対応する Windows のコントロールにできるだけ類似するように設計されています。ユーザーインターフェースレベルでの主な違いは，更新のキャッシングに対応するための変更によるものである。

既存のデータベースアプリケーションの dbExpress への移植については，14-22 ページの「データベースアプリケーションの Linux への移植」を参照してください。新しい dbExpress アプリケーションの設計については，第 18 章「データベースアプリケーションの設計」を参照してください。

dbExpress の違い

Linux では，dbExpress を使ってデータベースサーバーとの通信を行います。dbExpress は，共通のインターフェース群を実装する軽量ドライバの集まりです。個々のドライバは共有オブジェクト（.so ファイル）であり，アプリケーションにリンクしなければなりません。dbExpress はクロスプラットフォームに設計されているので，Windows ではダイナミックリンクライブラリ（.dll）として利用できます。

データアクセス層と同様に，dbExpress にはデータベースベンダーが提供するクライアント用ソフトウェアが必要です。さらに，データベース固有のドライバの他に 2 つの環境設定ファイル dbxconnections と dbxdrivers を使用します。これは，たとえば BDE で必要なものよりも明らかに少なくなります。BDE では，BDE のメインライブラリ（Idapi32.dll）とデータベース固有のドライバ，および他の多くのサポートライブラリが必要です。

このほか，dbExpress には以下の特徴があります。

- リモートデータベースへの単純で迅速な経路を利用できる。したがって，単純で直接的なデータアクセスによって顕著なパフォーマンスの向上が期待される
- 問い合わせとストアドプロシージャを処理できるが，テーブルを開くという概念はサポートしない
- 単一方向カーソルのみを返す
- INSERT，DELETE，UPDATE の問い合わせを実行する機能以外に組み込みの更新サポートはない
- メタデータキャッシングを実行せず，コアデータアクセスインターフェースを使って設計時のメタデータアクセスインターフェースが実装されている
- ユーザーが要求した問い合わせのみを実行するので，他の問い合わせを追加しないことでデータベースアクセスを最適化する

- レコードバッファまたはレコードバッファのブロックを内部的に管理する。この点が BDE では，クライアントがレコードバッファに使用するメモリを割り当てる必要がある
- InterBase，Oracle など SQL ベースのローカルテーブルしかサポートしない
- ドライバは，DB2，Informix，InterBase，MySQL，Oracle 用の各ドライバを使用する。これら以外のデータベースサーバーを使っている場合には，サポートされているサーバーのデータベースにデータを変換するか，そのデータベースサーバー用の dbExpress ドライバを記述するか，またはそのデータベースサーバー用のサードパーティ製 dbExpress ドライバを入手する必要がある

コンポーネントレベルの違い

dbExpress アプリケーションを記述する場合は，既存のデータベースアプリケーションで使うものとは別のデータアクセスコンポーネント群が必要です。dbExpress コンポーネントは，他のデータアクセスコンポーネント（TDataSet と TCustomConnection）と同じ基本クラスを共有します。したがって，プロパティ，メソッド，イベントの多くは既存のアプリケーションで使用するものと同じです。

表 14.9 に，Windows 環境の InterBase Express，BDE，および ADO で使用する重要なデータベースコンポーネントをリストアップし，Linux とクロスプラットフォームアプリケーションで使用する同等の dbExpress コンポーネントを示します。

表 14.9　同等のデータアクセスコンポーネント

InterBase Express **コンポーネント**	BDE **コンポーネント**	ADO **コンポーネント**	dbExpress **コンポーネント**
TIBDatabase	TDatabase	TADOConnection	TSQLConnection
TIBTable	TTable	TADOTable	TSQLTable
TIBQuery	TQuery	TADOQuery	TSQLQuery
TIBStoredProc	TStoredProc	TADOStoredProc	TSQLStoredProc
TIBDataSet		TADODataSet	TSQLDataSet

ただし，dbExpress データセット（TSQLTable，TSQLQuery，TSQLStoredProc，TSQLDataSet）は編集をサポートせず，フォワードナビゲーションのみが可能なので，他の対応するデータセットよりも制限があります。dbExpress データセットと Windows で利用できる他のデータセットの違いについての詳細は，第 26 章「単方向データセットの使い方」を参照してください。

編集とナビゲーションをサポートしないので，dbExpress アプリケーションは dbExpress データセットを直接操作することはしません。dbExpress データセットをクライアントデータセットに接続すると，レコードがメモリに一時的に記憶され，編集とナビゲーションがサポートされます。このアーキテクチャについての詳細は，18-6 ページの「データベースアーキテクチャ」を参照してください。

メモ　単純なアプリケーションの場合は，クライアントデータセットに接続する dbExpress データセットでなく，TSQLClientDataSet を利用できます。これには，移植前のアプリケーションのデータセットと移植後のアプリケーションのデータセットが 1:1 に対応するので，単純であるという利点がありますが，明示的にクライアントデータセットに接続する dbExpress データセットほど柔軟ではありません。多くのアプリケーションでは，TClientDataSet コンポーネントに接続した dbExpress データセットの使用をお勧めします。

ユーザーインターフェースレベルの違い

CLX のデータベース対応コントロールは，対応する Windows のコントロールにできるだけ類似するように設計されています。したがって，データベースアプリケーションのユーザーインターフェースの部分を移植する場合は，Windows アプリケーションの CLX への移植で考慮した問題以外にはほとんどないでしょう。

ユーザーインターフェースレベルの主な違いは，dbExpress のデータセットとクライアントデータセットのデータを提供する方法の違いによるものです。

dbExpress のデータセットのみを使用する場合は，データベースが編集をサポートせず，フォワードナビゲーションのみをサポートすることに対応するように，ユーザーインターフェースを調整する必要があります。したがって，たとえば前のレコードに移動するためのコントロールを削除することが必要になることがあります。dbExpress のデータセットはデータをバッファに保存しないので，データベース対応のグリッドにはデータを表示できません。一度に表示できるレコードは 1 つだけです。

dbExpress データセットをクライアントデータセットに接続すると，編集とナビゲーションに関するユーザーインターフェース要素がそのまま機能します。ユーザーはクライアントデータセットに再接続するだけです。この場合の主な問題は，更新をデータベースに書き込む方法の処理です。Windows のデータセットの多くは，デフォルトで，更新が登録されると（たとえばユーザーがレコードを移動した場合）データベースサーバーにその更新が自動的に書き込まれます。一方，クライアントデータセットでは常に更新がメモリにキャッシングされます。この違いを処理する方法については，14-24 ページの「dbExpress アプリケーションのデータ更新」を参照してください。

データベースアプリケーションの Linux への移植

データベースアプリケーションを dbExpress に移植すると，Windows と Linux の両方で動作するクロスプラットフォームのアプリケーションを作成できます。テクノロジーが異なるので，移植のプロセスでアプリケーションの変更も行います。移植の難易度は，アプリケーションの種類と複雑さ，および移植に必要な作業によって変わります。ADO など，Windows 固有のテクノロジーを多用するアプリケーションは，C++Builder データベーステクノロジーを使ったアプリケーションよりも移植が難しくなります。

次に，Windows/VCL のデータベースアプリケーションを Linux/CLX に移植する一般的な手順を示します。

1. dbExpress のサポートするデータベース（DB2，Informix，InterBase，MySQL，Oracle など）にデータが保存されているか確認します。データは，上記のいずれかの SQL サーバー上に配置する必要があります。これら以外のデータベースにデータが保存されている場合は，データ転送を行うユーティリティを入手してください。

   たとえば，C++Builder のバージョン（版）によっては，Data Pump というユーティリティが付属しています。Data Pump ユーティリティを使うと，特定のデータベース（dBase，FoxPro，Paradox など）を dbExpress 対応データベースに変換できます。このユーティリティの使い方については，Program Files \ Common Files \ Borland Shared \ BDE の datapump.hlp ファイルを参照してください。

2. データセットと接続コンポーネントを格納するデータモジュールを作成します。これにより，ユーザーインターフェースのフォームおよびコンポーネントと分離します。こうして，アプリケーション内で完全に新しいコンポーネントセットが必要な部分を，データモジュールとして分離します。ユーザーインターフェースを表すフォームは，他のアプリケーションと同様に移植できます。詳細は，14-3 ページの「ユーザーアプリケーションの移植」を参照してください。

   その他のステップでは，データセットと接続コンポーネントは独自のデータモジュールとして分離されていることが仮定されます。

3. 新しいデータモジュールを作成し，CLX バージョンのデータセットと接続コンポーネントを置きます。

4. 元のアプリケーションのデータセットごとに，dbExpress データセット，TDataSetProvider コンポーネント，および TClientDataSet コンポーネントを追加します。表 14.9 の対応を使って，どの dbExpress データセットを使うかを決定します。これらのコンポーネントに意味のある名前を付けます。

   - TClientDataSet コンポーネントの ProviderName プロパティを TDataSetProvider コンポーネントの名前に設定する
   - TDataSetProvider の DataSet プロパティを dbExpress データセットに設定する
   - 元のデータセットを参照していたデータソースコンポーネントの DataSet プロパティが，クライアントデータセットを参照するように変更する

5. 新しいデータセットのプロパティを，元のデータセットに合わせて次のように設定します。

   - 元のデータセットが TTable，TADOTable，TIBTable のいずれかのコンポーネントの場合は，新しい TSQLTable の TableName プロパティを元のデータセットの TableName に設定する。マスター / 詳細の関係をセットアップするプロパティ，またはインデックスを指定するプロパティもコピーする。範囲とフィルタを指定するプロパティは，新しい TSQLTable コンポーネントでなく，クライアントデータセット上で設定する
   - 元のデータセットが TQuery，TADOQuery，TIBQuery のいずれかのコンポーネントの場合は，新しい TSQLQuery コンポーネントの SQL プロパティを元のデータセットの SQL プロパティに設定する。新しい TSQLQuery の Params プロパティを元のデータセットの Params プロパティか Parameters プロパティの値に合わせて設定する。DataSource プロパティを設定してマスター / 詳細の関係を確立するようにする場合は，これもコピーする
   - 元のデータセットが TStoredProc，TADOStoredProc，TIBStoredProc のいずれかのコンポーネントの場合は，新しい TSQLStoredProc コンポーネントの StoredProcName プロパティを元のデータセットの StoredProcName プロパティか ProcedureName プロパティに設定する。新しい TSQLStoredProc の Params プロパティを元のデータセットの Params プロパティか Parameters プロパティの値に合わせて設定する

6. 元のアプリケーションのデータベース接続コンポーネント（TDatabase，TIBDatabase，または TADOConnection）があれば，新しいデータモジュールに TSQLConnection コンポーネントを追加します。また，ADO データセットの ConnectionString プロパティを使用する場合や，BDE データセットの DatabaseName プロパティに BDE エリアスを設定する場合のように，接続コンポーネントなしで接続するデータベースサーバーについても，TSQLConnection コンポーネントを追加しなければなりません。

7. ステップ 4 で配置した dbExpress データセットごとに，その SQLConnection プロパティを，適切なデータベース接続に対応する TSQLConnection コンポーネントに設定します。
8. TSQLConnection コンポーネントごとに，データベース接続の確立に必要な情報を指定します。そのために，TSQLConnection コンポーネントをダブルクリックして接続エディタを表示し，パラメータの値を適切な設定値を表すように設定します。ステップ 1 で新しいデータベースサーバーにデータを転送した場合は，新しいサーバーに合わせた設定値を指定します。以前と同じサーバーを使用する場合は，以下のように元の接続コンポーネントの情報を調べます。
   - 元のアプリケーションで TDatabase を使っていた場合は，Params プロパティと TransIsolation プロパティの情報を転送しなければならない
   - 元のアプリケーションで TADOConnection を使っていた場合は，ConnectionString プロパティと IsolationLevel プロパティの情報を転送しなければならない
   - 元のアプリケーションで TIBDatabase を使っていた場合は，DatabaseName プロパティと Params プロパティの情報を転送しなければならない
   - 元の接続コンポーネントがない場合は，BDE エリアスに関連する情報かデータセットの ConnectionString プロパティ の情報を転送しなければならない

   このパラメータセットを新しい接続名で保存してください。このプロセスについての詳細は，21-2 ページの「接続の制御」を参照してください。

dbExpress アプリケーションのデータ更新

dbExpress アプリケーションでは，クライアントのデータセットを使って編集をサポートします。クライアントデータセットに編集を登録すると，クライアントデータセットのメモリ上のスナップショットにデータの変更が書き込まれますが，データベースサーバーに自動的に書き込まれるわけではありません。元のアプリケーションでクライアントデータセットを使って更新をキャッシングしていた場合は，Linux 上での編集をサポートするための変更は必要ありません。しかし，Windows データセットの多くのデフォルトの動作（レコードを登録するとデータベースサーバーに変更を書き込む）を使っていた場合は，クライアントデータセットを使用するように変更しなければなりません。

キャッシュアップデートを使っていないアプリケーションを変換するには，次の 2 通りの方法があります。

- レコードの更新が登録されるたびに直ちにデータベースサーバーに更新が適用されるようなコードを記述すれば，Windows のデータセットの動作をまねることができる。このために，次のようにデータベースサーバーに更新を適用する AfterPost イベントハンドラをクライアントデータセットに提供する

```
void __fastcall TForm1::ClientDataSet1AfterPost(TDataSet *DataSet)
{
  TClientDataSet *pCDS = dynamic_cast<TClientDataSet *>(DataSet);
  if (pCDS)
    pCDS->ApplyUpdates(1);
}
```

- キャッシュアップデートを使うようにユーザーインターフェースを調整する。この方法には，ネットワークトラフィックを縮小し，トランザクション時間を最小化するなどの利点がある。し

かし，キャッシュアップデートを使用するように切り換える場合は，更新をいつデータベースサーバーに適用するかを決定しなければならない。また，多くの場合にユーザーが更新のアプリケーションを起動できるように，または編集がデータベースに書き込まれたかどうかをユーザーに通知するようにユーザーインターフェースを変更する必要がある。さらに，ユーザーがレコードを登録したときには更新エラーが検出されないので，問題の原因となった更新と発生した問題の種類がわかるように，更新エラーをユーザーに報告する方法を変更する必要がある

元のアプリケーションで BDE または ADO が提供するキャッシュアップデートのサポートを使っていた場合は，コードを若干調整してクライアントデータセットを使用するように切り替える必要があります。次の表に，BDE データセットと ADO データセット上でキャッシュアップデートをサポートするプロパティ，イベント，メソッドと，TClientDataSet 上の対応するプロパティ，イベント，メソッドを示します。

表 14.10 キャッシュアップデートをサポートするプロパティ，イベント，メソッド

BDE データセット（または TDatabase）	ADO データセット	TClientDataSet	目的
CachedUpdates	LockType	不要。クライアントデータセットでは常にキャッシュアップデートを使用する	キャッシュアップデートが有効かどうかを調べる
サポートしない	CursorType	サポートしない	サーバー上の変更からのデータセットの分離のレベルを表す
UpdatesPending	サポートしない	ChangeCount	データベースへの適用を要する更新レコードがローカルキャッシュ内にあるかどうかを表す
UpdateRecordTypes	FilterGroup	StatusFilter	キャッシュアップデートがいつ適用されるかわかるように，更新されたレコードの種類を指定する
UpdateStatus	RecordStatus	UpdateStatus	レコードが変更されていない，変更された，挿入された，または削除されたかどうかを表す
OnUpdateError	サポートしない	OnReconcileError	レコードごとの更新エラーを処理するイベント
ApplyUpdates（データセット上またはデータベース上）	UpdateBatch	ApplyUpdates	ローカルキャッシュ内のレコードをデータベースに適用する
CancelUpdates	CancelUpdates または CancelBatch	CancelUpdates	ローカルキャッシュ内の更新を適用する前に削除する
CommitUpdates	自動的に処理する	Reconcile	更新のアプリケーションが正常に実行された後で更新キャッシュをクリアする
FetchAll	サポートしない	GetNextPacket（および PacketRecords）	ローカルキャッシュ内のレコードをデータベースにコピーして編集と更新を行う
RevertRecord	CancelBatch	RevertRecord	現在のレコードの更新が適用されていない場合に更新を取り消す

クロスプラットフォームのインターネットアプリケーション

インターネットアプリケーションは，標準のインターネットプロトコルを使ってクライアントをサーバーに接続するクライアント / サーバーアプリケーションです。アプリケーションでは標準のインターネットプロトコルを使ってクライアント / サーバー間の通信を行うので，アプリケーションをクロスプラットフォームにすることができます。たとえば，サーバー用のインターネットアプリケーションプログラムは，マシンの Web サーバーソフトウェアを介してクライアントと通信します。サーバーアプリケーションは，通常は Linux 用か Windows 用に記述されますが，クロスプラットフォームにすることもできます。クライアントはいずれのプラットフォームにも配置できます。

C++Builder を使うと，Linux 上で動作する CGI，Apache アプリケーションなどの Web サーバーアプリケーションを作成できます。Windows では，ISAPI（Microsoft Server DLL），NSAPI（Netscape Server DLL），Windows CGI アプリケーションなど，他の Web サーバーを作成できます。純粋な CGI アプリケーションと，WebBroker を使用した一部のアプリケーションだけが，Windows と Linux の両方で動作します。

インターネットアプリケーションの Linux への移植

既存のインターネットアプリケーションをクロスプラットフォームにするには，Web サーバーアプリケーションを Linux に移植するか，Borland の C++ ソリューションが利用できるようになった時点で Linux 上で新しいアプリケーションを作成します。Web サーバーの記述方法については，第 32 章「インターネットサーバーアプリケーションの作成」を参照してください。アプリケーションが WebBroker を使用し，WebBroker インターフェースに書き込み，かつネイティブの API 呼び出しを使っていなければ，Linux への移植はそれほど難しくはありません。

ISAPI，NSAPI，Windows CGI などの Web API に書き込みを行うアプリケーションは，移植がより難しくなります。ソースファイル内を検索し，上記の API 呼び出しを Apache または CGI 呼び出しに書き換える必要があります（Apache API の関数プロトタイプについては，.. \ Include \ Vcl \ httpd.hpp を参照してください）。14-2 ページの「Windows アプリケーションの Linux への移植」に記載されているその他の変更もすべて行う必要があります。

第15章

パッケージとコンポーネントの操作

パッケージは，C++Builder のアプリケーションおよび IDE またはその両方によって使用される特別なダイナミックリンクライブラリです。実行時パッケージはユーザーがアプリケーションを実行するときに使用します。設計時パッケージは，IDE にコンポーネントをインストールして，カスタムコンポーネント用の特別なプロパティエディタを使用するために使います。パッケージは設計時と実行時の両方で機能し，設計時パッケージは実行時パッケージを頻繁に呼び出して動作します。パッケージライブラリは，その他の DLL と区別するため，.bpl（Borland package library）という拡張子を持つファイルに格納されます。

ほかの実行時ライブラリと同様に，パッケージにはアプリケーション間で共有できるコードが含まれます。たとえば，もっとも頻繁に使われる C++Builder コンポーネントは，vcl と呼ばれるパッケージの中にあります。デフォルトのアプリケーションを新規作成するたびに，自動的に vcl が使われます。このようにして作成したアプリケーションを構築（ビルド）すると，その実行形式イメージにはアプリケーションに固有のコードとデータだけが含まれます。共通コードは vcl60.bpl という実行時パッケージに含まれています。そのため，別のコンピュータで実行するには vcl60.bpl をコピーする必要があります。vcl60.bpl は，すべてのアプリケーションおよび C++Builder IDE 自身の間で共有されます。

C++Builder には，VCL コンポーネントと CLX コンポーネントをカプセル化したコンパイル済みの実行時パッケージがいくつか用意されています。また C++Builder では，IDE でコンポーネントを操作するのに設計時パッケージを使います。

アプリケーションはパッケージの有無にかかわらず構築できます。ただし，IDE にカスタムコンポーネントを追加したいときは，カスタムコンポーネントを設計時パッケージとしてインストールしなければなりません。

独自の実行時パッケージを作成して，それをアプリケーション間で共有できます。C++Builder のコンポーネントを作成する場合は，コンポーネントを設計時パッケージに構築してからインストールします。

なぜパッケージを使うのか

設計時パッケージを使うことで，カスタムコンポーネントの配布とインストールが簡単になります。実行時パッケージは使うことも使わないこともできますが，従来のプログラミングよりもいくつかの利点があります。よく使うコードを実行時パッケージとしてコンパイルすることにより，複数のアプリケーションでそのコードを共有することができます。たとえば，C++Builder 自身もそうですが C++Builder で作成されたすべてのアプリケーションは，パッケージを介して標準コンポーネントにアクセスすることができます。個々のアプリケーションが，それぞれコンポーネントライブラリのコードをコピーする必要がなくなります。その結果，システムリソースとディスク記憶域の両方が劇的に節約されます。さらに，実行時パッケージを使うことで，構築時に再コンパイルされるのはアプリケーションに固有のコードだけとなるので，より速いコンパイルが可能になります。

パッケージと標準 DLL

IDE で使用可能なカスタムコンポーネントを作成したいときは，パッケージを作成します。アプリケーションの構築に使われる開発ツールの種類に関係なく，任意のアプリケーションから呼び出せるライブラリを構築したいときは，標準 DLL を作成します。

次の表に，パッケージに関連付けられているファイルの種類を示します。

表 15.1　パッケージファイル

ファイル拡張子	内容
bpk	プロジェクトオプションのソースファイル。パッケージプロジェクトの XML 部分を成すファイル。ProjectName.bpk と ProjectName.cpp の組み合わせにより，パッケージプロジェクトが使用する設定内容，オプション，およびファイルを管理する
bpl	実行時パッケージ。C++Builder 固有の機能を持つ Windows .dll。bpl の基準名は bpk ソースファイルの基準名
cpp	ProjectName.cpp は，パッケージのエントリポイントを格納する。加えて，一般に，パッケージ内に格納される各コンポーネントは .cpp ファイルの中にある
h	コンポーネントのヘッダーファイルまたはインターフェース ComponentName.h は，ComponentName.cpp とペアで使う
lib	静的ライブラリ（.obj ファイルの集まり）。アプリケーションが実行時パッケージを使用しないとき，.bpi のかわりに使われる。**-Gl**（.LIB ファイルの生成）を選択したときにのみ生成される
obj	パッケージに含まれるユニットファイルのバイナリイメージ。各 .cpp ファイルにつき，必要に応じて，1 つの obj ファイルが作成される
bpi	Borland パッケージインポートライブラリ。各パッケージに bpi が 1 つ作成される。bpl と bpi の関係は，dll とインポートライブラリの関係に似ている。アプリケーションがパッケージを使って .bpi ファイルをリンカに渡し，パッケージ内の関数への参照を解決する。bpi の基本名は，そのパッケージソースファイルの基本名と同じ

同じパッケージに VCL コンポーネントと CLX コンポーネントの両方を含めることもできます。ただし，クロスプラットフォームにするパッケージは CLX コンポーネントだけを含めます。

メモ　パッケージでは，そのグローバルデータをアプリケーション内のほかのモジュールと共有します。

実行時パッケージ

実行時パッケージは，C++ Builder で作成したアプリケーションと一緒に配布しなければなりません。実行時パッケージは，ユーザーがアプリケーションを実行するときに機能を提供します。

パッケージを使うアプリケーションを実行するには，アプリケーションの実行形式ファイルとアプリケーションによって使用されるすべてのパッケージ（.bpl ファイル）の両方がコンピュータに必要です。アプリケーションが .bpl ファイルを使うためには，.bpl ファイルがシステムパス上になければなりません。アプリケーションを配布するとき，必要な .bpl の適切なバージョンをユーザーが必ず受け取るようにします。

アプリケーションで実行時パッケージを使用する

アプリケーションでパッケージを使用する手順は次のとおりです。

1. IDE にプロジェクトを読み込むかまたは作成します。
2. ［プロジェクト｜オプション］を選択します。
3. ［パッケージ］タブを選択します。
4. ［実行時パッケージを使って構築］チェックボックスを選択し，その下の編集ボックスにパッケージ名を入力します（インストールされた設計時パッケージと関連付けられている実行時パッケージはすでに編集ボックスに表示されています）。
5. 既存のリストにパッケージを追加するには，［追加］ボタンをクリックして，［実行時パッケージの追加］ダイアログボックスで新しいパッケージの名前を入力します。使用可能なパッケージを参照して追加するには，［追加］ボタンをクリックして，［実行時パッケージの追加］ダイアログボックスの［パッケージ名］編集ボックスの隣の［参照］ボタンをクリックしパッケージを選択します。

   ［実行時パッケージの追加］ダイアログボックスの［検索パス］編集ボックスの値を変更した場合，［ツール｜環境オプション］の環境オプションダイアログの［ライブラリ］タブにある［ライブラリパス］を変更したことになります。

   パッケージ名にファイル拡張子（または C++Builder のリリース番号を表す数字）を付ける必要ありません。つまり，「vcl60.bpl」は「vcl」と入力します。［実行時パッケージ］編集ボックスに直接入力する場合，複数のファイルはセミコロンで区切って入力します。次に例を示します。

   ```
   rtl;vcl;vcldb;vclado;vclx;Vclbde;
   ```

［実行時パッケージ］編集ボックスに表示されているパッケージは，自動的にアプリケーションにリンクされます。重複するパッケージ名があると無視されます。［実行時パッケージを使って構築］チェックボックスがチェックされていない場合，アプリケーションはパッケージを含めずにリンクされます。

この変更は現在のプロジェクトだけに適用されます。現在の設定を新規プロジェクトのデフォルトとして設定するには，ダイアログボックスの一番下にある［デフォルト］チェックボックスをチェックします。

パッケージを使ってアプリケーションを構築する場合でも，そこで使用するパッケージユニットのヘッダーファイルをインクルードする必要があります。たとえば，データベース関連のコントロールを使用するアプリケーションには，

```
#include "vcldb.h"
```

という文が必要です。これは，vcldb パッケージを使うかどうかと関係ありません。C++Builder で生成されたソースファイルでは，**#include** 文が自動的に作成されます。

パッケージを動的にロードする

実行時にパッケージをロードするには，関数 LoadPackage を呼び出します。LoadPackage はパッケージをロードし，重複するユニットがないかチェックし，パッケージ内の全ユニットの初期化ブロックを呼び出します。たとえば，次のコードはファイル選択ダイアログでファイルを選択すると実行されます。

```
if (OpenDialog1->Execute())
  PackageList->Items->AddObject(OpenDialog1->FileName,
    (TObject *)LoadPackage(OpenDialog1->FileName));
```

パッケージを動的にアンロードするには UnloadPackage を呼び出します。この場合に，そのパッケージで定義されているクラスのインスタンスがあればそれを破棄し，そのパッケージによって登録されたクラスの登録を解除するのを忘れないでください。

どの実行時パッケージを使うか決定する

C++Builder は，基本的な言語サポートやコンポーネントを提供する rtl，vcl をはじめ，多くの実行時パッケージを同梱しています。

vcl パッケージには，よく使われるコンポーネントが入っています。これに対し，rtl パッケージはコンポーネント以外のシステム機能と Windows インターフェース要素が入っています。これには，データベースや他の特殊コンポーネントは含まれず，別のパッケージで提供されます。

パッケージを使ってクライアント / サーバー形式のデータベースアプリケーションを作成する場合は，vcl，vcldb，rtl，dbrtl などいくつかの実行時パッケージが必要です。アプリケーションでアウトラインコンポーネントを使いたい場合には，このほかに vclx も必要です。これらのパッケージを使うには，［プロジェクト｜オプション］を選択し，［パッケージ］タブを選んでから，次のリストを［実行時パッケージ］編集ボックスに入力します。

```
rtl;vcl;vcldb;vclx;
```

実際には，vcl と rtl は vcldb の Requires リストで参照されるため vcl と rtl は入力する必要がありません（15-9 ページの「Requires リスト」を参照してください）。このため，vcl と rtl が［実行時パッケージ］編集ボックスに含まれているかどうかに関わらず，アプリケーションコンパイルは同じように構築できます。

カスタムパッケージ

カスタムパッケージは，自分でコーディングし構築する bpl か，サードパーティベンダーが提供する既存パッケージのどちらかです。カスタムの実行時パッケージをアプリケーションで使うには，［プロジェクト｜オプション］を選択し，［パッケージ］ページの［実行時パッケージ］編集ボックスに，パッケージの名前を追加します。たとえば，stats.bpl という名前の統計パッケージを持っていると仮

定します。このパッケージをアプリケーション内で使うには，［実行時パッケージ］編集ボックスに入力する行は次のようになります。

```
rtl;vcl;vcldb;stats
```

独自のパッケージを作成する場合は，必要であればそれらをリストに追加できます。

設計時パッケージ

設計時パッケージを使って，IDE のコンポーネントパレットにコンポーネントをインストールし，カスタムコンポーネント用の特別なプロパティエディタを作成します。

C++Builder には多くの設計時コンポーネントパッケージが用意されています。これらのパッケージは IDE にあらかじめ組み込まれています。どのパッケージが組み込まれているかは，C++Builder のバージョンによって異なります。また，C++Builder がカスタマイズされているかによっても異なります。［コンポーネント｜パッケージのインストール］を選択すると，システムに組み込まれているパッケージの一覧が表示されます。

設計時パッケージは実行時パッケージを呼び出すことによって動作します。呼び出される実行時パッケージは，設計時パッケージの Requires リストで宣言されます（15-9 ページの「Requires リスト」を参照してください）。たとえば，dclstd は vcl を参照します。Dclstd 自身は，標準コンポーネントのほとんどをコンポーネントパレットで使用可能にする追加機能を持っています。

メモ 慣例的に，IDE の設計時パッケージは「dcl」で始まり，bin ディレクトリに格納されます。\ bin \ Dclstd などの設計時パッケージには，リンカ用の対応ファイル（.. \ lib \ vcl.lib, .. \ lib \ vcl.bpi など）および実行時パッケージ本体（Windows \ System \ vcl60.bpl）があります。

インストール済みのパッケージに加え，独自のコンポーネントパッケージやサードパーティベンダーのコンポーネントパッケージも IDE にインストールできます。新しいコンポーネント用のデフォルトのコンテナとして，dclusr 設計時パッケージが提供されています。

コンポーネントパッケージのインストール

すべてのコンポーネントはパッケージとして IDE にインストールされます。したがって，ユーザー自身がコンポーネントを作成するときには，まずそのコンポーネントを入れるパッケージを作成して構築する必要があります（15-6 ページの「パッケージの作成と編集」を参照してください）。コンポーネントのソースコードは，第 V 部「カスタムコンポーネントの作成」で説明されているモデルに従って記述する必要があります。複数のユニットを 1 つのパッケージに含め，しかも各ユニットに複数のコンポーネントがある場合は，パッケージの名前を持つ名前空間にすべてのコンポーネントの Register 関数を 1 つ作成します。

独自またはサードパーティベンダーのコンポーネントをインストール / アンインストールする手順は次のとおりです。

1. 新しいパッケージをインストールするときは，パッケージファイルをローカルディレクトリにコピーまたは移動します。パッケージに .bpl，.bpi，.lib，および .obj の各ファイルが同梱されている場合は，これらのファイルもすべてコピーしておく必要があります（これらのファイルについての情報は「パッケージと標準 DLL」を参照）。

.bpi ファイルとヘッダーファイル，および .lib または .obj ファイル（配布に含まれている場合）を格納するディレクトリは，C++Builder のライブラリパスに定義しておく必要があります。

2. IDE メニューの［コンポーネント｜パッケージのインストール］を選択するか，［プロジェクト｜オプション］を選択して，［パッケージ］タブをクリックします。
3. ［設計時パッケージ］の下に，使用可能なパッケージのリストが表示されます。
   - IDE にパッケージをインストールするには，パッケージの横にあるチェックボックスを選択します。
   - パッケージをアンインストールするには，このチェックボックスのチェックを外します。
   - インストールされているパッケージに含まれるコンポーネントのリストを表示するには，そのパッケージを選択して［コンポーネント］を選択します。
   - リストにパッケージを追加するには，［追加］を選択し，［実行時パッケージの追加］ダイアログボックスで，.bpl ファイルがあるディレクトリを参照します（ステップ 1 を参照）。.bpl ファイルを選択して，［開く］を選択します。
   - リストからパッケージを削除するには，そのパッケージを選択して［削除］を選択します。
4. ［OK］を選択します。

パッケージ内のコンポーネントは，そのコンポーネントの RegisterComponents 手続きで指定されたコンポーネントパレットページに，RegisterComponents 手続きに与えられたのと同じ名前でインストールされます。

デフォルトの設定を変更していなければ，インストールされて使用可能なすべてのパッケージを使って新しいプロジェクトが作成されます。インストールの現在の選択内容を新しいプロジェクトの自動デフォルトとするためには，［プロジェクトオプション］ダイアログボックスの［パッケージ］タブの一番下にある［デフォルト］チェックボックスにチェックを付けます。

パッケージをアンインストールせずに，コンポーネントパレットからコンポーネントを削除するには，［コンポーネント｜パレットの設定］を選択するか，［ツール｜環境オプション］を選択して，［パレット］タブをクリックします。［パレットオプション］タブには，インストール済みの各コンポーネントと，それが表示される［コンポーネントパレット］ページの名前の一覧が表示されます。任意のコンポーネントを選択して，［非表示］をクリックすると，そのコンポーネントがパレット上で非表示になります。

パッケージの作成と編集

パッケージを作成するには，以下の項目を指定する必要があります。

- パッケージの名前
- 新しいパッケージをリンクするために必要（requires）となるパッケージの名前の一覧
- 新しいパッケージに含まれる（contains）ユニットファイルの一覧。基本的にパッケージとは，ソースコードユニットを 1 つにまとめたものです。Contains リストには，パッケージへと構築するカスタムコンポーネントのソースコードを指定します。

パッケージは，C++ ソースファイル（.cpp）およびプロジェクトオプションファイル（.bpk）で定義します。この 2 つのファイルは，パッケージエディタを使って作成します。

パッケージの作成

パッケージを作成する手順は次のとおりです。ここで示した手順の詳細については，15-9 ページの「パッケージの構造を理解する」を参照してください。

メモ クロスプラットフォームの開発時などでは，パッケージファイル（.bpk）で ifdef を使用してはいけません。ただし，ソースコードでは ifdef を使用できます。

1. ［ファイル｜新規作成｜その他］を選択し，［パッケージ］アイコンを選んで［OK］をクリックします。
2. 作成されたパッケージがパッケージエディタに表示されます。
3. パッケージエディタは，新しいパッケージの Requires ノードと Contains ノードを表示します。
4. ユニットを Contains リストに追加するには［追加］スピードボタンを選択します。［ユニットの追加］ページで，［ユニットファイル名］編集ボックス内に .cpp ファイル名を入力するか，または［参照］をクリックしてファイルを指定し，［OK］をクリックします。選択したユニットが，パッケージエディタの Contains ノードの下に表示されます。ユニットをさらに追加するには，このステップを繰り返します。
5. ユニットを Requires リストに追加するには［追加］スピードボタンをクリックします。［必須］ページで，［パッケージ名］編集ボックス内に .bpi ファイル名を入力するか，または［参照］をクリックしてファイルを指定してから，［OK］をクリックします。選択したパッケージが，パッケージエディタの Requires ノードの下に表示されます。パッケージをさらに追加するには，このステップを繰り返します。
6. ［オプション］スピードボタンを選択し，構築したいパッケージの種類を決めます。
   - 設計時専用パッケージ（実行時に使うことができないパッケージ）を作成するには，［設計時のみ使用可能］ラジオボタンを選択します（または，式 `LFLAGS = ... -Gpd ...` を使って bpk ファイルに **-Gpd** リンカスイッチを追加します）。
   - 実行時専用パッケージ（インストールできないパッケージ）を作成するには，［実行時のみ使用可能］ラジオボタンを選択します（または，式 `LFLAGS = ... -Gpr ...` を使って bpk ファイルに **-Gpr** リンカスイッチを追加します）。
   - 設計時と実行時ともに利用可能なパッケージを作成するには，［設計 / 実行時とも使用可能］ラジオボタンを選択します。
7. パッケージエディタで，［コンパイル］スピードボタンをクリックして，パッケージをコンパイルします。

メモ ［インストール］ボタンをクリックして，強制的にメイクを実行することもできます。

既存のパッケージを編集する

既存のパッケージを開いて編集するには，いくつかの方法があります。

- ［ファイル｜開く］（または［ファイル｜開き直す］）を選択して，cpp ファイルか bpk ファイルを選択する
- ［コンポーネント｜パッケージのインストール］を選択して，［設計時パッケージ］リストからパッケージを選び，［編集］ボタンをクリックする
- パッケージエディタが開いているとき，Requires ノード内のパッケージを 1 つ選んで右クリックし，［開く］を選択する

パッケージの説明を変更したり，使用上のオプションを設定するには，パッケージエディタの［オプション］スピードボタンを選択し，［情報］タブを選択します。

［プロジェクトオプション］ダイアログの左下に［デフォルト］チェックボックスがあります。このボックスを選択して OK をクリックすると，選択したオプションが新しいパッケージオブジェクトのデフォルト設定値として保存されます。元のデフォルト値に戻すには，default.bpk ファイルを削除するか名前を変更します。

パッケージソースファイルとプロジェクトオプションファイル

パッケージソースファイルには，拡張子 .cpp が付きます。パッケージプロジェクトオプションファイルには，拡張子 .bpk（Borland パッケージ）が付きます。このファイルは XML フォーマットで作成されます。パッケージエディタでパッケージプロジェクトオプションファイルを表示するには，Contains 節か Requires 節上で右クリックして，［オプションソースの編集］を選択します。

メモ C++Builder が .bpk ファイルを管理します。通常は，手動で編集する必要はありません。変更を加える場合は，［プロジェクトオプション］ダイアログボックスの［パッケージ］タブを使用してください。

たとえば，MyPack というパッケージのプロジェクトオプションファイルは次のようになります。

```
<MACROS>
    <VERSION value="BCB.05.02"/>
    <PROJECT value="MyPack.bpl"/>
    <OBJFILES value="MyPack.obj Unit2.obj Unit3.obj"/>
    <RESFILES value="MyPack.res"/>
    <IDLFILES value=""/>
    <IDLGENFILES value=""/>
    <DEFFILE value=""/>
    <RESDEPEN value="$(RESFILES)"/>
    <LIBFILES value=""/>
    <LIBRARIES value=""/>
    <SPARELIBS value="Vcl60.lib"/>
    <PACKAGES value="Vcl60.bpi vcldbx60.bpi"/>
    ...
```

ここでは，MYPACK.cpp は次のコードをインクルードします。

```
USERES("MyPack.res");
USEPACKAGE("vcl60.bpi");
USEPACKAGE("vcldbx60.bpi");
USEUNIT("Unit2.cpp");
USEUNIT("Unit3.cpp");
```

MyPack の Contains リストには，MyPack のほかに Unit2 および Unit3 という 2 つのユニットが含まれます。MyPack の Requires リストには VCL と VCLDBX の 2 つのパッケージが含まれます。

コンポーネントのパッケージ化

［コンポーネント｜コンポーネントの新規作成］を使ってコンポーネントを作成すると，適当な位置に PACKAGE マクロが挿入されます。ただし，旧バージョンの C++Builder で作成したカスタムコンポーネントについては，2 ヶ所に手動で PACKAGE を追加する必要があります。

C++Builder コンポーネントのヘッダーファイル宣言には，次のように **class** の後に定義済みの PACKAGE マクロを含める必要があります。

```
class PACKAGE MyComponent : ...
```

また，次のように，コンポーネントを定義する .cpp ファイルの中の Register 関数の宣言に PACKAGE マクロを含めます。

```
void __fastcall PACKAGE Register()
```

この PACKAGE マクロは展開され，生成される bpl ファイルからクラスをインポートしたりエクスポートしたりできます。

パッケージの構造を理解する

パッケージは以下のパーツで構成されます。

- パッケージの名前
- Requires リスト
- Contains リスト

パッケージの名前

パッケージの名前はそのプロジェクト内で重複しないものでなくてはなりません。パッケージに Stats という名前を付けた場合，パッケージエディタはソースファイル Stats.cpp とプロジェクトオプションファイル Stats.bpk を作成します。コンパイラとリンカは，実行形式ファイル Stats.bpl，バイナリイメージ Stats.bpi，およびオプションで静的ライブラリ Stats.lib を作成します。このパッケージをアプリケーションで使うには，［プロジェクト｜オプション］を選択し，［パッケージ］タブを選択してから，［実行時パッケージ］編集ボックスに Stats を追加します。

パッケージ名に先頭文字，末尾文字，バージョン番号を追加することもできます。パッケージエディタが開いているときに［オプション］ボタンをクリックします。［プロジェクトオプション］ダイアログボックスの［情報］ページで，［プレフィクス］，［サフィックス］，［バージョン］のテキストまたは値を入力します。たとえばパッケージプロジェクトにバージョン番号を追加するには，［バージョン］の後に「6」を入力します。すると，Package1 から Package1.bpl.6 が生成されます。

Requires リスト

Requires リストは，現在のパッケージで使われる，その他の外部のパッケージを羅列します。Requires リストに含まれる外部パッケージは，アプリケーションのコンパイル時に自動的にリンクされます。リンクされるアプリケーションは，現在のパッケージ，および外部パッケージを両方とも使うアプリケーションです。

パッケージに含まれるユニットファイルがほかのパッケージのユニットを参照する場合，パッケージの Requires リストに参照するパッケージを含めておく必要があります。参照する別のパッケージが

Requires リストから省かれている場合，コンパイラはそれらをパッケージの「暗黙で含まれているユニット」にインポートします。

メモ ユーザーが作成するほとんどのパッケージには rtl が必要です。VCL コンポーネントを使用する場合は，vcl パッケージも含める必要があります。CLX コンポーネントを使ってクロスプラットフォームプログラミングを行う場合は，VisualCLX を含める必要があります。

パッケージの循環参照の回避

パッケージは，その Requires リストに循環参照を含めることはできません。これは以下のことを意味します。

- つまり，パッケージはその Requires リストで自分自身を参照できません。
- 参照のチェーンは，チェーン内のどのパッケージも再度参照することなく終了しなければなりません。パッケージ A がパッケージ B を要求する場合，パッケージ B がパッケージ A を要求することはできません。また，パッケージ A がパッケージ B を要求し，パッケージ B がパッケージ C を要求する場合，パッケージ C がパッケージ A を要求することはできません。

重複したパッケージ参照の処理

パッケージの Requires リスト（または，実行時パッケージ編集ボックス）の中で同じパッケージが複数回記述されている場合，リンカはそれを無視します。しかし，プログラミングの明確さと可読性が損なわれることがあるため，パッケージの重複参照は削除することをお勧めします。

Contains リスト

Contains リストは，そのパッケージにバインドされるユニットファイルを識別するためのものです。独自のパッケージを cpp ファイルに記述している場合，そのソースコードを Contains リストに含めます。

冗長ソースコード使用の回避

パッケージは，別のパッケージの Contains リストに入れることができません。

パッケージの Contains リストに直接含まれるすべてのユニット，またはそれらのユニットのいずれかに間接的に含まれるすべてのユニットは，リンク時にパッケージに結び付けられます。

1 つのアプリケーションにおいて，あるユニットは，直接的にも間接的にも 1 つのパッケージにしか含まれてはなりません。このことは C++Builder IDE にとっても当てはまります。たとえば，vcl に含まれているユニットを Contains 節で直接指定しているパッケージは，IDE に組み込むことができなくなります。既存のパッケージに含まれているユニットを別のパッケージから使うには，Contains 節に記述する代わりに Requires リストに，参照したいユニットを含む既存のパッケージを追加してください。

パッケージの構築

IDE またはコマンドラインからパッケージを構築（ビルド）することができます。IDE からパッケージを再構築する手順は次のとおりです。

1. ［ファイル｜開く］を選択し，パッケージのソースファイルまたはプロジェクトオプションファイルを選択して，［開く］をクリックします。

2. エディタが起動したら，［プロジェクト｜ Project をメイク］または［プロジェクト｜ Project を再構築］を選択します。

メモ　［ファイル｜新規作成｜その他］を選択し，［パッケージエディタ］をダブルクリックするという方法もあります。［インストール］ボタンをクリックすると，パッケージプロジェクトのメイクが行われます。パッケージプロジェクトのノードを右クリックすると，インストール，メイク，再構築から選択できます。

生成されていない .cpp ファイルを追加するには，パッケージソースコードにコンパイラ指令のどれか 1 つを追加します。詳細については，次の「パッケージ固有のコンパイラ指令」を参照してください。

コマンドラインでリンクする場合，パッケージ独自のリンカスイッチをいくつか使用できます。詳細については，15-12 ページの「コマンドラインコンパイラとリンカの使い方」を参照してください。

パッケージ固有のコンパイラ指令

次の表は，ソースコードに挿入できるパッケージ独自のコンパイラ指令のリストです。

表 15.2　パッケージ固有のコンパイラ指令

指令	目的
#pragma package(smart_init)	パッケージ内のユニットの初期化が依存関係に従って行われるようにする（デフォルトでパッケージソースファイルに含まれている）
#pragma package(smart_init, weak)	ユニットを「弱く」パッケージ化します。下の「弱いパッケージ」（ユニットソースファイルに指令を挿入する）を参照してください。

すべてのライブラリで使用できる他の指令については，7-10 ページの「パッケージと DLL の作成」を参照してください。

弱いパッケージ

#pragma package(smart_init, weak) 指令によって .obj ファイルがパッケージの .bpi ファイルや .bpl ファイルに格納されるかどうかが決まります（コンパイラおよびリンカによって作成されるファイルについては，15-12 ページの「構築により作成されるパッケージファイル」を参照してください）。ユニットファイルに **#pragma package(smart_init, weak)** が記述されていると，リンカは，可能であれば bpl ファイルにユニットを含めません。そして，他のアプリケーションやパッケージがそのユニットを必要としている場合は，ユニットのパッケージではないローカルコピーを作成します。この指令でコンパイルされたユニットは「弱いパッケージ」と呼ばれます。

たとえば，ユニットを 1 つだけ含む PACK と呼ばれるパッケージを作成するとします。さらに，含まれるユニット UNIT1 は追加のユニットは使わず，RARE.dll を参照するだけだとします。UNIT1.cpp に **#pragma package(smart_init, weak)** を指定してパッケージを構築すると，UNIT1 は PACK.bpi には含まれますが PACK.bpl には含まれません。RARE.dll を PACK パッケージと一緒に配布する必要性はなくなります。もちろん，PACK を使う別のパッケージやアプリケーションが UNIT1 を参照すると，UNIT1 が PACK.bpi からコピーされてプロジェクトに直接リンクされます。

第 2 のユニットである UNIT2 を PACK に追加したとします。UNIT2 は UNIT1 を使うとします。今度は，UNIT1.cpp で **#pragma package(smart_init, weak)** により PACK をコンパイルしても，リンカは PACK.bpl に UNIT1 を含めます。しかし，UNIT1 を参照するその他のパッケージやアプリケーションは，PACK.bpi からの（パッケージではない）コピーを使います。

メモ **#pragma package(smart_init, weak)** を含むユニットファイルでは，グローバル変数を使えません。

#pragma package(smart_init, weak) 指令は bpl をほかの C++Builder プログラマへ配布する開発者向けに拡張された機能です。頻繁に使われない DLL の配布を避け，同じ外部ライブラリに依存するパッケージ間の衝突をなくすのに役立ちます。

たとえば，C++Builder の PenWin ユニットは PenWin.dll を参照します。ほとんどのプロジェクトは PenWin を使いません。また，ほとんどのコンピュータには，PenWin.dll がインストールされていません。このため，PenWin ユニットは vcl では弱いパッケージにしています。PenWin と vcl パッケージ を使うプロジェクトをリンクすると，PenWin は vcl60.bpi からコピーされてプロジェクトに直接結び付けられます。その実行結果は PenWin.dll に静的にリンクされます。

PenWin が弱いパッケージではない場合，2 つの問題点が生じます。第 1 に，vcl 自体が PenWin.dll に静的にリンクされるので，PenWin.dll がインストールされていないコンピュータにロードできません。第 2 に，PenWin を含むパッケージを作成しようとすると，PenWin ユニットは vcl と新規パッケージの両方に含まれてしまうため，ビルドエラーが生じます。したがって，弱いパッケージでなければ PenWin を vcl に含めることはできません。

コマンドラインコンパイラとリンカの使い方

コマンドラインからリンクを行うときには，プロジェクトが確実にパッケージに作成されるように **-Tpp** リンカスイッチを使います。これ以外のパッケージ固有のスイッチを次の表に示します。

表 15.3 **パッケージ固有のコマンドラインコンパイラスイッチ**

スイッチ	目的
-Tpp	プロジェクトをパッケージに作成する（デフォルトでパッケージプロジェクトファイルに含まれている）
-Gi	生成された bpi ファイルを保存する（デフォルトでパッケージプロジェクトファイルに含まれている）
-Gpr	実行時専用パッケージを生成する
-Gpd	設計時専用パッケージを生成する
-Gl	.lib ファイルを生成する
-D "description"	指定した記述をパッケージとともに保存する

スイッチ **-Gpr** と **-Gpd** は，パッケージプロジェクトでのみ使用可能な［プロジェクトオプション］ダイアログの［情報］ページにある［実行時のみ使用可能］と［設計時のみ使用可能］の 2 つのラジオボタンに対応しています。**-Gpr** と **-Gpd** をどちらも使用しないと，生成されるパッケージは設計時にも実行時にも使用可能となります。**-D** スイッチは，同じページの［説明］編集ボックスに対応しています。**-Gl** スイッチは，［プロジェクトオプション］ダイアログの［リンカ］ページにある［.lib を作成］チェックボックスに対応しています。

メモ ［プロジェクト｜メイクファイルの作成］を選択すると，メイクファイルを生成してコマンドラインで使用できます。

構築により作成されるパッケージファイル

パッケージを作成するには，拡張子 .bpk が付いたプロジェクトオプションファイルを使ってソースファイル（.cpp）をコンパイルします。コンパイルするソースファイルの基本名は，コンパイラに

よって生成されるファイルの基本名と一致していなければなりません。たとえば，ソースファイル名が TRAYPAK.cpp であれば，プロジェクトオプションファイルの TRAYPAK.bpk には次の行が含まれている必要があります。

```
<PROJECT value="Traypak.bpl"/>
```

このプロジェクトをコンパイルしリンクすると，TRAYPAK.bpl というパッケージが作成されます。

パッケージをコンパイルしてリンクすると，bpi ファイル，bpl ファイル，obj ファイル，および lib ファイルが作成されます。デフォルトでは，［ツール｜環境オプション］ダイアログボックスの［ライブラリ］ページで指定したディレクトリに .bpi ファイル，.bpl ファイル，および .lib ファイルが作成されます。このデフォルト設定をオーバーライドするには，パッケージエディタで［オプション］スピードボタンをクリックして［プロジェクトオプション］ダイアログを表示し，［ディレクトリ / 条件］ページで変更します。

パッケージの配布

パッケージの配布方法は，アプリケーションの配布とよく似ています。パッケージと一緒にどのファイルを配布するかは，場合によって異なります。bpl，および bpl が必要とする パッケージと dll は必ず配布しなければなりません。

配布する必要のあるファイルをパッケージの用途別に下の表に示します。

表 15.4　パッケージと一緒に配布するファイル

ファイル	説明
ComponentName.h	エンドユーザーがクラスのインターフェースにアクセスできるようにする
ComponentName.cpp	エンドユーザーがコンポーネントソースにアクセスできるようにする
bpi，obj，lib	エンドユーザーがアプリケーション間をリンクできるようにする

配布の概要については，第 17 章「アプリケーションの配布」を参照してください。

パッケージを使うアプリケーションの配布

実行時パッケージを使うアプリケーションを配布するときは，そのアプリケーションの .exe ファイルだけでなく，そのアプリケーションに必要なすべてのライブラリ（.bpl または .dll）ファイルを配布しなくてはなりません。実行時ライブラリが .exe ファイルと異なるディレクトリにある場合，ユーザーのパスによりアクセスできる必要があります。また，実行時ライブラリを Windows の System ディレクトリに置く規則に従う必要があります。InstallShield Express を使うと，インストールスクリプトによって，ユーザーのシステムに必要な任意のパッケージを無条件に再インストールせずに，システム内に既存のパッケージがあるかどうかが，あらかじめチェックされます。

ほかの開発者にパッケージを配布する

実行時パッケージまたは設計時パッケージをほかの C++Builder 開発者に配布するときには，他の必要なファイルとともに必ず .bpi ファイルおよび .bpl ファイルを提供する必要があります。コンポーネ

ントをそのアプリケーションに静的にリンクする（つまり，実行時パッケージを使わないアプリケーションを作成する）ためには，提供するパッケージとともに .lib ファイルまたは .obj ファイルも必要となります。

パッケージコレクションファイル

パッケージコレクション（.dpc ファイル）は，ほかの開発者へパッケージを配布する便利な方法です。各パッケージコレクションは，bpl および bpl と一緒に配布したい追加ファイルを含む 1 つまたは複数のパッケージから構成されます。IDE へのインストール時にパッケージコレクションが選択されると，コレクションの構成ファイルが自動的に .pce コンテナから抽出されます。［インストール］ダイアログボックスでは，コレクション内のすべてのパッケージをインストールするか，パッケージを選択してインストールするかを指定できます。

パッケージコレクションを作成する手順は，以下のとおりです。

1. ［ツール｜パッケージコレクションエディタ］エディタを選んでパッケージコレクションエディタを開きます。
2. ［パッケージの追加］スピードボタンをクリックし，次に［パッケージの選択］ダイアログの中の［パッケージ］を選択し，［開く］をクリックします。さらにパッケージをコレクションに追加するには，［パッケージの追加］スピードボタンを再びクリックします。パッケージを追加すると，パッケージエディタの左側のツリー図に追加したパッケージが表示されます。パッケージを削除するには，そのパッケージを選択して［パッケージの削除］スピードボタンをクリックします。
3. ツリー図の一番上にある［コレクション］ノードを選択します。パッケージコレクションエディタの右側に 2 つのフィールドが表示されます。
   - ［作者 / ベンダー名］編集ボックスには，ユーザーがパッケージをインストールするときにインストールダイアログに表示するパッケージコレクションについてのオプション情報を入力できます。
   - ［ディレクトリリスト］の下には，ユーザーのパッケージコレクションのファイルをインストールするデフォルトディレクトリをリストします。［追加］，［編集］，および［削除］ボタンを使ってリストを編集します。たとえば，すべてのソースコードファイルを同じディレクトリにコピーしたいとします。この場合，ディレクトリ名 Source を，推奨パスと一緒に指定して C: \ MyPackage \ Source と入力します。［インストール］ダイアログボックスに，そのディレクトリの推奨パスとして C: \ MyPackage \ Source が表示されます。
4. パッケージのほかに，パッケージコレクションには，.bpi，.obj，.cpp（ユニット）ファイル，文書，およびその他の配布したいファイルを含むことができます。補助ファイルは指定パッケージ（bpl）に関連付けられるファイルグループに含まれます。グループ内のファイルはその関連するパッケージがインストールされるときだけインストールされます。補助ファイルをパッケージコレクションに含むには，ツリー図でパッケージを選択して，［ファイルグループの追加］スピードボタンをクリックし，ファイルグループの名前を入力します。必要であれば，同じ方法でファイルグループをさらに追加します。ファイルグループを選択すると，パッケージコレクションエディタの右側に次の新規フィールドが表示されます。

- ［インストールディレクトリ］リストボックスで，このグループのファイルをインストールしたいディレクトリを選択します。ドロップダウンリストには，上記手順 3 でディレクトリリストに入力したディレクトリが含まれています。
- このグループのファイルのインストールをオプションにしたい場合は，［オプショングループ］チェックボックスをチェックします。
- ［含まれるファイル］には，このグループに含みたいファイルをリストします。［追加］，［削除］，［自動］ボタンを使ってこのリストを編集します。［自動］ボタンを使うと，パッケージの Contains リストに含まれる指定拡張子を持つファイルがすべて選択できます。パッケージコレクションエディタは，C++Builder に設定されているライブラリパスを使ってこれらのファイルを検索します。

5. コレクションの中のパッケージの Requires リストに含まれているパッケージ用のインストールディレクトリを選択できます。ツリー図のパッケージを選択すると，パッケージコレクションエディタの右側に 4 つの新しいフィールドが表示されます。
   - ［実行可能ファイル］リストボックスで，Requires リストに含まれているパッケージの .bpl ファイルをインストールしたいディレクトリを選択します（ドロップダウンリストには，上記手順 3 でディレクトリリストに入力したディレクトリが含まれています）。パッケージコレクションエディタは，C++Builder に設定されているライブラリパスを使ってこれらのファイルを検索し，ライブラリファイルのリストに入れます。
   - ［ライブラリファイル］リストボックスで，Requires リストに含まれているパッケージの .obj ファイルおよび .bpi ファイルをインストールしたいディレクトリを選択します（ドロップダウンリストには，上記手順 3 でディレクトリリストに入力したディレクトリが含まれています）。パッケージコレクションエディタは，C++Builder に設定されているライブラリパスを使ってこれらのファイルを検索し，ライブラリファイルのリストに入れます。
6. パッケージコレクションソースファイルを保存するには，［ファイル｜上書き保存］を選択します。パッケージコレクションソースファイルは，拡張子 .pce を付けて保存しなければなりません。
7. パッケージコレクションを構築するには，［コンパイル］スピードボタンをクリックします。パッケージコレクションエディタは，ソース（.pce）ファイルと同じ名前の .dpc ファイルを生成します。ソースファイルが保存されていないと，構築する前にファイル名を要求されます。

既存の .pce ファイルを編集または再構築するには，パッケージコレクションエディタで［ファイル｜開く］を選択し，操作したいファイルを指定します。

第16章

国際化対応アプリケーションの作成

この章では，国際市場への配布を予定しているアプリケーションの開発についての指針を解説します。計画的に進めることにより，国内市場のみならず国際市場におけるアプリケーション作成に必要な時間とコーディングの労力が削減できます。

国際化対応とローカライズ

海外市場に配布できるアプリケーションを作成するには，次の 2 つの主要なステップの実行が必要です。

- 国際化対応
- ローカライズ

お使いの C++Builder にトランスレーションツールが付属している場合は，それを使ってローカライズを管理できます。詳細は，オンラインヘルプで「トランスレーションツール」（Etm.hlp）を参照してください。

国際化対応

国際化対応とは，プログラムが複数のロケールで動作できるようにするプロセスです。ここでいうロケールとは，ターゲットとする国の言語や文化的慣習も含めたユーザー環境です。Windows では多様な環境をサポートしており，それぞれの国の言語で記述されています。

ローカライズ

ローカライズとは，アプリケーションを特定のロケールで機能するように翻訳するプロセスです。ユーザーインターフェースの翻訳のほかに，ローカライズでは，機能のカスタマイズも行われます。たとえば，財務アプリケーションを変更して，各国のさまざまな税法に対応させることができます。

アプリケーションの国際化対応

国際化対応アプリケーションを作成するには，以下の手順を行う必要があります。

- 他の国の文字セットの文字列をコードで処理できるようにする
- ユーザーインターフェースをローカライズによる変更に適合できるように設計する
- ローカライズに必要なすべてのリソースを分離する

コードを多国語対応にする

アプリケーションのコードが，対象の国や地域で使用される文字列を処理できるようにする必要があります。

文字セット

Windows の欧米版（英語，フランス語，ドイツ語など）では，ANSI Latin-1（1252）文字セットを使用しています。しかし，その他の Windows では別の文字セットを使用しています。たとえば，日本語バージョンでは Shift-JIS 文字セット（コードページ 932）を使用しています。Shift-JIS 文字セットは日本語をマルチバイトの文字コードとして表します。

文字セットは，大きく分けて次の 3 種類があります。

- 1 バイト
- マルチバイト
- ワイド文字

Windows と Linux のどちらも，1 バイト文字セット，マルチバイト文字セット，および Unicode 文字をサポートしています。1 バイト文字セットは，文字列中の各 1 バイトが 1 文字を表します。欧米の多くのオペレーティングシステムで使われている ANSI 文字セットは，1 バイト文字セットです。

マルチバイト文字セットの場合，1 バイトで表す文字もあれば，2 バイト以上で表す文字もあります。マルチバイト文字の最初の 1 バイトを「リードバイト」といいます。一般に，マルチバイト文字セットのうち下位の 128 文字が 7 ビット ASCII 文字に対応し，序数が 127 より大きいバイトはすべて，マルチバイト文字のリードバイトです。1 バイト文字のみヌル値（#0）を持つことができます。マルチバイト文字（特に 2 バイト文字）は，アジア言語で広く使われています。

OEM と ANSI 文字セット

米国を含む多くの 1 バイトコードを常用している国では，歴史的理由から Windows 環境下の文字セットとファイルシステム上で使われる文字セットが異なる場合があります。Windows 上の文字

セットを ANSI 文字セットと呼び，動作環境に固有の文字セットを OEM 文字セットと呼ぶ場合があります。DOS 環境で作られたファイルと扱う場合に ANSI - OEM 変換が必要になる場合があります。日本語環境においては Windows 環境の文字セットと環境固有な文字セットは両方とも Shift-JIS である場合がほとんどなので，ANSI-OEM 変換は意味を持ちません。

マルチバイト文字セット

東アジア地域で使用される表意文字（漢字）は，1 バイト（8 ビット）の **char** 型の範囲では単純な 1 対 1 のマッピングが行えません。東アジア諸国の言語は多数の文字を使用するため，1 バイトの **char** ではすべての文字を表現できません。マルチバイト文字列では，1 文字あたり 1 バイト以上を格納できます。AnsiString なら，1 バイト文字とマルチバイト文字を混用できます。

マルチバイト文字コードの先頭バイトには，予約された一定範囲のコードが割り当てられていますが，予約範囲は個々の文字セットにより異なります。2 番目以降のバイトの範囲は，独立した 1 バイト文字の文字コード，および第 1 バイトの予約範囲にまたがっています。したがって，文字列中の特定のバイトが 1 バイト文字なのかマルチバイト文字の一部なのかを知るには，文字列の先頭から順に，第 1 バイトの予約範囲内のバイトがあったら，次の 1 バイトと合わせて 2 バイト以上の文字として解析しなければなりません。

東アジア地域用のコードを記述するときは，文字列をマルチバイト文字で解析できるような関数を使って，すべての文字列操作を処理するようにします。2 バイト文字を使用できる RTL 関数のリストについては，オンラインヘルプの「国際化対応 API」を参照してください。

文字列のバイト数は文字数と一致するとは限らないことに注意してください。マルチバイト文字を半分に分断して文字列を切り捨てないように注意してください。文字のサイズが不明であるため，文字をパラメータとして関数や手続きに渡してはなりません。文字または文字列は，常にポインタを渡すようにします。

ワイド文字

表意文字セットを扱うもう 1 つのアプローチは，すべての文字を Unicode のようなワイド文字に変換して符号化するという方法です。Unicode 文字，Unicode 文字列をワイド文字，ワイド文字列と呼ぶこともあります。Unicode 文字セットは，1 文字を 2 バイトで表します。したがって，Unicode 文字列は 1 バイト単位でなく 2 バイト文字を並べたものです。

Unicode の最初の 256 文字は ANSI 文字セットに対応します。Windows は Unicode（UCS-2）をサポートし，Linux は，UCS-2 の上位セットである UCS-4 をサポートしています。C++Builder は，Windows，Linux の両プラットフォームで UCS-2 をサポートします。ワイド文字は 1 バイトではなく 2 バイトのため，文字セットはより多くの種類の文字を表現できます。

1 バイト文字だけの文字列の処理では一般的であった仮定の多くは MBCS システムには適用できませんが，ワイド文字符号化方式を採用することで，それらが同じように適用できるという利点があります。文字列のバイト数と文字列の文字数の関係も一定したものになります。文字を分断してしまう心配もなく，文字の第 2 バイトを別の文字と間違える心配もありません。

ワイド文字を扱う際の最大の問題点は，Windows ではワイド文字に対応した API 関数呼び出しを少ししかサポートしていないことです。このため，VCL コンポーネントは，すべての文字列の値を MBCS 文字列として表します。文字列プロパティの設定や値の読み出しごとにワイド文字システムと MBCS システム間の翻訳を行うと，追加のコードが必要になり，アプリケーションの実行速度が

低下します。しかし，文字と WideChar の 1 対 1 のマッピングの利用を必要とする一部の特殊な文字列処理アルゴリズムを使用する場合には，ワイド文字への翻訳が必要なこともあります。

Unicode 文字を使用できる RTL 関数のリストについては，オンラインヘルプの「国際化対応 API」を参照してください。

アプリケーションに双方向機能性を含める

言語の中には西洋言語に見られるような左から右へと読む順番とは異なり，言葉を右から左に読んで，数字を左から右に読む言語があります。これらの言語は，この区別から双方向（BiDi）と呼ばれます。もっとも一般的な双方向言語は，アラビア語とヘブライ語ですが，その他の中東諸国の言語も双方向言語です。

TApplication の 2 つのプロパティ（BiDiKeyboard と NonBiDiKeyboard）により，キーボードレイアウトを指定できます。また，VCL は，BiDiMode プロパティと ParentBiDiMode プロパティで双方向ローカライズをサポートしています。次の表は，これらのプロパティを持つ VCL オブジェクトのリストです。

表 16.1　BiDi をサポートする VCL オブジェクト

コンポーネントパレットページ	VCL オブジェクト
Standard	TButton
	TCheckBox
	TComboBox
	TEdit
	TGroupBox
	TLabel
	TListBox
	TMainMenu
	TMemo
	TPanel
	TPopupMenu
	TRadioButton
	TRadioGroup
	TScrollBar
Additional	TActionMainMenuBar
	TActionToolBar
	TBitBtn
	TCheckListBox
	TColorBox
	TDrawGrid
	TLabeledEdit
	TMaskEdit
	TScrollBox
	TSpeedButton
	TStaticLabel
	TStaticText

表 16.1　BiDi をサポートする VCL オブジェクト（つづき）

コンポーネントパレットページ	VCL オブジェクト
	TStringGrid
	TValueListEditor
Win32	TComboBoxEx
	TDateTimePicker
	THeaderControl
	THotKey
	TListView
	TMonthCalendar
	TPageControl
	TRichEdit
	TStatusBar
	TTabControl
	TTreeView
Data Controls	TDBCheckBox
	TDBComboBox
	TDBEdit
	TDBGrid
	TDBListBox
	TDBLookupComboBox
	TDBLookupListBox
	TDBMemo
	TDBRadioGroup
	TDBRichEdit
	TDBText
QReport	TQRDBText
	TQRExpr
	TQRLabel
	TQRMemo
	TQRPreview
	TQRSysData
その他のクラス	TApplication（ParentBiDiMode はない）
	TBoundLabel
	TControl（ParentBiDiMode はない）
	TCustomHeaderControl（ParentBiDiMode はない）
	TForm
	TFrame
	THeaderSection
	THintWindow（ParentBiDiMode はない）
	TMenu
	TStatusPanel

メモ　THintWindow は，ヒントを起動させたコントロールの BiDiMode を使用します。

双方向プロパティ

16-4 ページの表 16.1「BiDi をサポートする VCL オブジェクト」に記載されているオブジェクトは，BiDiMode プロパティと ParentBiDiMode プロパティを持っています。この 2 つのプロパティと TApplication の BiDiKeyboard プロパティと NonBiDiKeyboard プロパティが双方向ローカライズをサポートしています。

メモ CLX でクロスプラットフォームプログラミングを行う場合，双方向プロパティは使用できません。

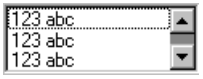
BiDiMode プロパティ

プロパティ BiDiMode は，4 つの状態を持つ列挙型 TBiDiMode です。4 つの状態は，bdLeftToRight，bdRightToLeft，bdRightToLeftNoAlign，および bdRightToLeftReadingOnly です。

bdLeftToRight

bdLeftToRight では，左から右への読み取り順序でテキストを描画します。位置合わせとスクロールバーは変更されません。たとえば，アラビア語またはヘブライ語など右から左へテキストを入力するときには，カーソルがプッシュモードになり，テキストは右から左へ入力されます。英語やフランス語などラテン系の文字は，左から右へ入力されます。bdLeftToRight はデフォルト値です。

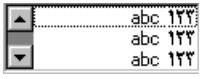
図 16.1 bdLeftToRight に設定された TListBox



bdRightToLeft

bdRightToLeft は，右から左への読み取り順序でテキストを描画します。位置合わせが変更され，スクロールバーが移動します。テキストはアラビア語またはヘブライ語など右から左へ読む言語を標準として入力されます。キーボードがラテン言語に変更されると，カーソルがプッシュモードになりテキストが左から右へ入力されます。

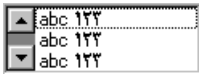
図 16.2 bdRightToLeft に設定された TListBox



bdRightToLeftNoAlign

bdRightToLeftNoAlign は，右から左への読み取り順序でテキストを描画します。位置合わせは変更されず，スクロールバーは移動します。

図 16.3 bdRightToLeftNoAlign に設定された TListBox



bdRightToLeftReadingOnly

bdRightToLeftReadingOnly では，右から左への読み取り順序でテキストを描画します。位置合わせとスクロールバーは変更されません。

図 16.4　bdRightToLeftReadingOnly **に設定された** TListBox

ParentBiDiMode プロパティ

ParentBiDiMode は Boolean プロパティです。**true**（デフォルト）の場合，コントロールは親を参照してどの BiDiMode を使用するかを決定します。コントロールが TForm オブジェクトの場合，フォームは TApplication の BiDiMode 設定を使用します。すべてのコントロールの ParentBiDiMode プロパティが **true** の場合，TApplication の BiDiMode プロパティを変更すると，プロジェクト内のすべてのフォームとコントロールが新しい設定に更新されます。

FlipChildren メソッド

FlipChildren メソッドを使用すると，コンテナコントロールの子の位置を反転できます。コンテナコントロールは，TForm，TPanel，および TGroupBox などのほかのコントロールを受け入れることができるコントロールです。FlipChildren は，論理型の AllLevels パラメータを持ちます。この値が **false** の場合，コンテナコントロールの直接の子だけが反転されます。**true** の場合，コンテナコントロールのすべてのレベルの子が反転されます。

C++Builder は，Left プロパティとコントロールの位置合わせを変更してコントロールを反転させます。コントロールの左側が親コントロールの左端から 5 ピクセルである場合，反転後は編集コントロールの右側が親コントロールの右端から 5 ピクセルになります。編集コントロールが左に位置合わせされていた場合，FlipChildren の呼び出しによりコントロールを右に位置合わせします。

設計時にコントロールを反転させるには，［編集｜左右入れ替え］を選択して，すべてのコントロールを反転させるかまたは選択したコントロールの子だけを反転させるかによって［すべて］または［選択部分］を選びます。フォーム上のコントロールを選択して右クリックし，コンテキストメニューから［左右入れ替え］を選んでコントロールを反転させることもできます。

メモ　編集コントロールを選択して［左右入れ替え｜選択部分］コマンドを発行しても何も起こりません。編集コントロールはコンテナではないからです。

追加メソッド

双方向ユーザーに対するアプリケーションを開発するときに役立つメソッドには，以下のようなものがあります。

メソッド	説明
OkToChangeFieldAlignment	データベースコントロールとともに使用する。コントロールの位置合わせが変更できるかをチェックする
DBUseRightToLeftAlignment	位置合わせをチェックするためのデータベースコントロール用のラッパー
ChangeBiDiModeAlignment	渡される位置合わせパラメータを変更する。BiDiMode 設定のチェックは行わず，中央揃えのコントロールはそのまま残して，左揃えを右揃えに，右揃えは左揃えに変換する
IsRightToLeft	右から左へのオプションが選択されると **true** を返す。**false** を返す場合はコントロールが左から右モードになっている

メソッド	説明
UseRightToLeftReading	コントロールが右から左への読み取りを使用している場合に **true** を返す
UseRightToLeftAlignment	コントロールが右から左への位置合わせを使用している場合に **true** を返す。これはカスタマイズしてオーバーライドできる
UseRightToLeftScrollBar	コントロールが左スクロールバーを使用している場合に **true** を返す
DrawTextBiDiModeFlags	コントロールの BiDiMode 用の描画テキストフラグを返す
DrawTextBiDiModeFlagsReadingOnly	コントロールの BiDiMode 用の描画テキストフラグのうち，読み取り方向のフラグのみを返す
AddBiDiModeExStyle	作成されるコントロールに適切な ExStyle フラグを追加する

ロケールに固有の機能

特定のロケール環境向けのアプリケーションに特別の機能を追加することができます。日本語の環境でアプリケーションを作成する場合には IME（インプットメソッドエディタ。FEP，かな漢字変換という場合もあります）の制御が必要になる場合があります。

VCL/CLX コンポーネントは IME の制御をサポートしています。文字入力用のキャレットが表示されるコントロールのほとんどは ImeMode プロパティを持っています。このプロパティによりコントロールが入力フォーカスを与えられたときに任意の変換モードに切り替えたり，IME そのものを使用不可能にすることができます。さらに IME 制御のためのプロテクトメソッドが TWinControl クラスで定義されているので，TWinControl クラスから継承したクラスでは IME の制御を簡単に行うことができます。また，実行環境で利用可能な IME に関する情報が Screen グローバル変数により提供されます。

メモ 日本以外の国では，複数の IME を切り替えることで変換モードを制御している環境があります。そのような環境に対応するために ImeName プロパティが用意されています。日本語環境でも指定できますが，通常は使用しません。

Screen グローバル変数（VCL，CLX の両方で使用可）は，ユーザーのシステムにインストールされているキーボードマッピングの情報も提供します。したがって，アプリケーションの実行環境に関する固有の情報が得られます。

ユーザーインターフェースの設計

アプリケーションを数ヶ国用に作成するには，翻訳の際に生じる変更に適合できるようにユーザーインターフェースを設計しておくことが重要です。

テキスト

ユーザーインターフェースに含まれるテキストはすべて翻訳しなければなりません。多くの場合，元のテキストよりも翻訳したものの方が長くなります。テキストを表示するユーザーインターフェース要素については，テキスト文字列が長くなっても表示できるように設計してください。ダイアログ，メニュー，ステータスバー，その他のテキスト表示用のユーザーインターフェースは，長い文字列を簡単に表示できるように設計します。一般的ではない省略語（たとえば，オブジェクトインスペクタを OI とするなど）は，ほかの言語に翻訳することが困難となるため避けてください。

長い文字列よりも短い文字列のほうが翻訳すると長くなる傾向があります。表 16.2 は，英語の文字列の長さが翻訳後にどのくらい増加するかの概算を示しています。

表 16.2　文字列の長さの概算

英語の文字列（文字数）	予想される増加率
1-5	100%
6-12	80%
13-20	60%
21-30	40%
31-50	20%
50 字以上	10%

グラフィックイメージ

翻訳を必要としないイメージを使用できれば理想的です。ただし，これはグラフィックイメージにテキストを含むことができないことを意味します。テキストは常に翻訳が必要だからです。テキストをイメージに含む必要がある場合は，イメージの一部としてのテキストを持つのではなく，イメージの上で透過となる背景を持つラベルオブジェクトを使用するのがよいでしょう。

グラフィックイメージを作成するときに，もう 1 つ考慮すべき点があります。特定の文化に固有のイメージの使用は避けるようにします。たとえば，メールボックス（郵便ポスト）は，各国ごとにさまざまです。また，さまざまな宗教を持つ国々用のアプリケーションであれば，宗教的なシンボルの使用は適切ではありません。色でさえも，文化によってその象徴する意味がさまざまです。

フォーマットとソート順序

アプリケーションで使用する日付，時刻，数字，および通貨形式は，対象の国に合わせてローカライズする必要があります。Windows で定義される形式しか使わない場合は，それらは Windows レジストリから取り出して使用されるため，表示形式を翻訳する必要はありません。ただし，独自の形式文字列を指定している場合には，これをリソース文字列定数として宣言し，ローカライズできるようにしてください。

文字列をソートする順序も国によってさまざまです。ヨーロッパ言語の多くには，国によってソート順序の異なる発音区別符号が含まれています。また，一部の国では 2 文字の組み合わせを 1 文字として扱ってソートします。たとえば，スペイン語では連字 ch は 1 つの文字として扱われ，c と d の間にソートされます。また，ドイツ語の eszett などは 1 つの文字を 2 つの別個の文字としてソートします。

キーボードマッピング

キーの組み合わせによるショートカットの割り当てには注意してください。英語キーボードの文字がすべてインターナショナル仕様のキーボードで使えるとは限りません。可能であれば，ショートカットには数字キーやファンクションキーを使用します。これらのキーは，実質的にすべてのキーボードにおいて使用できるからです。

リソースの分離

アプリケーションのローカライズの中で必ず発生する作業は，ユーザーインターフェースに表示される文字列の翻訳です。コードのいたるところを変更しなくても翻訳できるアプリケーションを作成するには，ユーザーインターフェースの文字列を単一モジュールに分離する必要があります。C++Builder では，自動的に .dfm ファイル（CLX では .xfm ファイル）が作成され，メニュー，ダイアログ，ビットマップ用のリソースがそこに格納されます。

このように明らかなユーザーインターフェース要素のほかに，ユーザーに送るエラーメッセージのような文字列もすべて分離しなければなりません。文字列のリソースはフォームファイルには含まれていませんが，このファイルから分離して .RC ファイルに含めることができます。

リソースモジュールの作成

リソースを分離することによって，翻訳のプロセスが単純化されます。リソース分離作業の次のレベルは，リソースモジュールの作成です。リソースモジュールにはすべての標準的なリソースとプログラム専用のリソースが含まれています。リソースモジュールを利用すると，リソースモジュールを交換するだけで数多くの翻訳をサポートするプログラムを作成できます。

リソース DLL ウィザードを使うと，そのプログラムのリソースモジュールを作成できます。リソース DLL ウィザードでは，コンパイルして保存され，かつ現在開いているプロジェクトが必要です。このウィザードは，使用される RC ファイルとプロジェクトのリソース文字列から文字列テーブルを含む RC ファイルを作成し，関連フォームと作成された RES ファイルを含むリソース専用の DLL 用のプロジェクトを生成します。RES ファイルは新しい RC ファイルからコンパイルされます。

サポートしたい各翻訳のリソースモジュールを作成する必要があります。各リソースモジュールには，対象となるロケールに固有のファイル拡張子が必要です。最初の 2 文字が対象となる言語を表し，3 文字目がロケールの国を表します。リソース DLL ウィザードを使用すると，これらは自動的に作成されます。ウィザードを使用しない場合は，以下のコードを使用して対象となる翻訳用のロケールコードを取得します。

```
/* このコールバックは文字列および関連する言語と国をリストボックスに記述する */
BOOL __stdcall EnumLocalesProc(char* lpLocaleString)
{
  AnsiString LocaleName, LanguageName, CountryName;
  LCID lcid;
  lcid = StrToInt("0x" + AnsiString(lpLocaleString));
  LocaleName = GetLocaleStr(lcid, LOCALE_SABBREVLANGNAME, "");
  LanguageName = GetLocaleStr(lcid, LOCALE_SNATIVELANGNAME, "");
  CountryName = GetLocaleStr(lcid, LOCALE_SNATIVECTRYNAME, "");
  if (lstrlen(LocaleName.c_str()) > 0)
    Form1->ListBox1->Items->Add(LocaleName + ":" + LanguageName + "-" + CountryName);
  return TRUE;
}
/* この呼び出しはコールバックをロケールごとに実行する */
  EnumSystemLocales((LOCALE_ENUMPROC)EnumLocalesProc, LCID_SUPPORTED);
```

リソース DLL の使用

アプリケーションを構成する実行形式プログラム，DLL，およびパッケージ（bpl）には，必要なリソースがすべて含まれています。しかし，このリソースをローカライズバージョンに置き換えるのは，実行形式ファイル，DLL，またはパッケージファイルと同じ名前のローカライズリソース DLL をアプリケーションに追加するだけですみます。

アプリケーションを起動すると，ローカルシステムのロケールがチェックされます。使用している .EXE，.DLL，または .BPL ファイルと同じ名前のリソース DLL を検出すると，その拡張子が調べられます。リソース DLL の拡張子がローカルシステムの言語および国と一致すると，アプリケーションは実行形式プログラム，DLL，およびパッケージ内のリソースのかわりにそのリソース DLL 内のリソースを使用します。言語と国の両方が一致するリソース DLL がないと，アプリケーションは言語だけが一致するリソース DLL を使用します。言語に一致するリソース DLL もない場合には，実行形式プログラム，DLL，およびパッケージ内のリソースを使用します。

ローカルシステムのロケールに一致するリソース DLL とは別のリソースモジュールをアプリケーションで使えるようにする場合は，Windows レジストリにロケールオーバーライドエントリを設定できます。HKEY_CURRENT_USER \ Software \ Borland \ Locales キーに，アプリケーションのパスとファイル名を文字列値として追加し，データ値にリソース DLL の拡張子を設定します。アプリケーションは起動時に，この拡張子を持つリソース DLL をまず検索し，次に，システムロケールを検索します。このレジストリエントリを設定することによって，システム上のロケールを変更せずに，アプリケーションのローカライズバージョンをテストできます。

たとえば，インストールプログラムまたはセットアッププログラムで次の手順を使うことにより，C++Builder アプリケーションのロード時に使うロケールを示す，レジストリキーの値を設定できます。

```
void SetLocalOverrides(char* FileName, char* LocaleOverride)
{
  HKEY Key;
  const char* LocaleOverrideKey = "Software \\ Borland \\ Locales";
  if (RegOpenKeyEx(HKEY_CURRENT_USER, LocaleOverrideKey, 0, KEY_ALL_ACCESS, &Key)
        == ERROR_SUCCESS) {
    if (lstrlen(LocaleOverride) == 3)
      RegSetValueEx(Key, FileName, 0, REG_SZ, (const BYTE*)LocaleOverride, 4);
    RegCloseKey(Key);
  }
}
```

Windows API を使ってリソースを取得するには，次のようにグローバル関数 FindResourceHInstance を使って次のように指定します。次に例を示します。

```
LoadString(FindResourceHInstance(HInstance), IDS_AmountDueName, szQuery, sizeof(szQuery));
```

適切なリソースモジュールを提供するだけで，1 つのアプリケーションを，実行するシステムのロケールに自動的に適合させることができます。

リソース DLL の動的な切り替え

リソース DLL をアプリケーションの起動時に検索することに加えて，リソース DLL を実行時に動的に切り替えることができます。この機能をアプリケーションに追加するには，プロジェクトに ReInit ユニットを含める必要があります。ReInit は Examples \ Apps ディレクトリの Richedit サンプルにあります。言語を切り替えるには，LoadResourceModule を呼び出して新しい言語の LCID を渡し，次に ReinitializeForms を呼び出します。

たとえば，以下のコードはインターフェース言語をフランス語に切り替えます。

```
const  FRENCH = (SUBLANG_FRENCH << 10) | LANG_FRENCH;
if (LoadNewResourceModule(FRENCH))
  ReinitializeForms();
```

この技術の利点は，アプリケーションの現在のインスタンスとフォームすべてが使用されることです。レジストリ設定を更新してアプリケーションを再起動したり，またはアプリケーションに必要なリソースをデータベースサーバーにログインして再取得するなどの必要はありません。

リソース DLL を切り替えると，新しい DLL で指定したプロパティがフォームの実行インスタンス内のプロパティに上書きされます。

メモ 実行時にフォームプロパティに加えられた変更は失われます。新しい DLL がロードされると，デフォルト値はリセットされません。ローカライズによる差異は別として，フォームオブジェクトが起動時の状態に再初期化されることを前提としたコードは避けてください。

アプリケーションのローカライズ

アプリケーションを国際化対応した後は，これを配布する特定の海外市場向けに，ローカライズバージョンを作成することができます。

リソースのローカライズ

ユーザーのリソースはフォームファイル（VCL の場合は .dfm，CLX の場合は .xfm）とリソースファイルを含むリソース DLL に分離されました。IDE 内のフォームを開いて関連するプロパティを翻訳することができます。

メモ リソース DLL プロジェクトでは，コンポーネントの追加または削除はできません。プロパティの変更はできますが，実行時エラーが発生する可能性があります。翻訳不要のプロパティは変更しないように注意してください。オブジェクトインスペクタにローカライズ可能なプロパティだけを表示すると，ミスをなくすことができます。不要なプロパティカテゴリを非表示にするには，オブジェクトインスペクタで右クリックし，［表示］メニューを選択します。

RC ファイルを開いて対応する文字列を翻訳できます。プロジェクトマネージャから RC ファイルを開いて文字列テーブルエディタを使用します。

第17章

アプリケーションの配布

C++Builder で作成したアプリケーションが完成し，動作するようになったら，プログラムを配布します。つまり，ほかのユーザー環境でプログラムを動かすことを考えます。異なるコンピュータの上でアプリケーションを正しく動かすようにするには，いくつかの手順が必要となります。どの手順が必要かは，アプリケーションの種類によって異なります。以下の節では，各種のアプリケーションを配布するときの注意点について述べます。

- アプリケーションの配布の基本
- CLX アプリケーションの配布
- データベースアプリケーションの配布
- Web アプリケーションの配布
- さまざまな動作環境を考慮したプログラミング
- ソフトウェアのライセンス

メモ　この章は，Windows で動作するアプリケーションの配布について説明しています。

アプリケーションの配布の基本

アプリケーションには，実行形式ファイル以外に，DLL，パッケージファイル，ヘルパーアプリケーションなどの，いくつかのサポートファイルが必要になる場合があります。また，Windows のレジストリには，サポートファイルの場所の指定から，単純なプログラム設定に至るまで，アプリケーションのエントリが含まれる必要がある場合があります。アプリケーションのファイルをコンピュータにコピーして，必要なレジストリ設定を行うプロセスは，InstallShield Express などのインストールプログラムによって自動化できます。以下のことは，ほとんどの種類のアプリケーションを配布する際に共通する考慮事項です。

- インストールプログラムの使用
- アプリケーションファイルの区別

データベースにアクセスしたり，Web 上で実行したりする C++Builder アプリケーションの場合，通常のアプリケーションのインストール作業以外に，追加の作業が必要です。データベースアプリケー

ションのインストールの詳細については，17-6 ページの「データベースアプリケーションの配布」を参照してください。Web アプリケーションのインストールの詳細については，17-10 ページの「Web アプリケーションの配布」を参照してください。ActiveX コントロールのインストールの詳細については，43-16 ページの「ActiveX コントロールの Web での配布」を参照してください。CORBA アプリケーションの配布に関する詳細は，『VisiBroker Installation and Administration Guide』を参照してください。

インストールプログラムの使用

1 つの実行形式ファイルだけで構成されている単純な C++Builder アプリケーションは，簡単に目的のコンピュータにインストールできます。実行形式ファイルをコンピュータにコピーするだけです。複数のファイルで構成される複雑なアプリケーションの場合には，より広範なインストール手順が必要です。こうした複雑なアプリケーションには，専用のインストールプログラムが必要です。

セットアップツールキットを使用すると，インストールプログラムの作成プロセスが自動化され，コードの作成も不要になる場合が多くあります。セットアップツールキットで作成するインストールプログラムによって，C++Builder アプリケーションのインストールに伴う，さまざまなタスクが実行されます。たとえば，実行形式ファイルとサポートファイルのホストコンピュータへのコピー，Windows のレジストリエントリの作成，および BDE データベースアプリケーション用のボーランドデータベースエンジンのインストール，といったタスクが実行されます。

InstallShield Express は C++Builder にバンドルされているセットアップツールキットです。InstallShield Express は C++Builder およびボーランドデータベースエンジンとの併用が保証されています。InstallShield Express は MSI（Windows Installer）技術をベースにしています。

InstallShield Express は C++Builder のインストール時に自動的にインストールされないため，インストールプログラムの作成に利用するには，手動でインストールしなければなりません。InstallShield Express をインストールするには，C++Builder の CD からインストールプログラムを実行します。InstallShield Express の使い方についての詳細は，InstallShield Express のオンラインヘルプを参照してください。

ほかのセットアップツールキットも利用できますが，BDE データベースアプリケーションを配布する場合は，MSI 技術を利用していて，かつボーランドデータベースエンジンの配布が保証されているツールキットだけを使用してください。

アプリケーションファイルの区別

アプリケーションの配布では，実行形式ファイル以外に，以下のようなファイルを配布する必要があります。

- アプリケーションファイル
- パッケージファイル
- マージモジュール
- ActiveX コントロール

アプリケーションファイル

アプリケーションと一緒に配布するファイルの種類を示します。

表 17.1　アプリケーションファイル

種類	ファイルの拡張子
プログラムファイル	.exe，.dll
パッケージファイル	.bpl，.bpi，.lib
ヘルプファイル	hlp，.cnt，.toc（使用されている場合），その他，開発するアプリケーションがサポートするヘルプファイル
ActiveX ファイル	.ocx（サポート DLL を含む場合もある）
ローカルテーブルファイル	.dbf，.mdx，.dbt，.ndx，.db，.px，.y*，.x*，.mb，.val，.qbe，.gd*

パッケージファイル

アプリケーションで実行時パッケージを使用する場合は，それらのパッケージファイルもアプリケーションとともに配布する必要があります。InstallShield Express では，ファイルのコピーおよび Windows のレジストリ内に必要なエントリを作成することにより，DLL と同様に，パッケージファイルのインストールを処理します。また，マージモジュールを使用すると，InstallShield Express などの MSI ベースのセットアップツールと一緒に実行時パッケージを配布できます。詳細については，次の節を参照してください。

ボーランドが提供する実行時パッケージは，Windows のシステムディレクトリにインストールすることをお勧めします。このディレクトリは共通の場所として機能し，複数のアプリケーションが，ファイルの 1 つのインスタンスにアクセスするようにします。作成したパッケージについては，アプリケーションのインストール先と同じディレクトリにインストールすることをお勧めします。配布する必要があるのは，.bpl ファイルと各言語対応のリソース（たとえば xxx.jpn）です。

メモ　CLX アプリケーションと一緒にパッケージを配布する場合には，vcl60.bpl でなく clx60.bpl を配布する必要があります。

パッケージをほかの開発者に配布する場合は，.bpl ファイルと .bcp ファイルの両方を提供します。

マージモジュール

InstallShield Express 3.0 は，Windows Installer（MSI）技術をベースにしています。このため，C++Builder はいくつかのマージモジュールを同梱しています。マージモジュールは，共有のコード，ファイル，リソース，レジストリエントリ，およびセットアップロジックを単一の複合ファイルとして扱い，標準の方法でこれらをアプリケーションに同梱させます。マージモジュールを使用すると，InstallShield Express などの MSI ベースのセットアップツールと一緒に実行時パッケージを配布できます。

ランタイムライブラリは何らかの方法でグループ化されているため，それぞれ，他の一部のランタイムライブラリと依存関係にあります。このため，あるパッケージをインストールプロジェクトに追加すると，インストールツール側では，依存関係にある他のパッケージを自動的に追加するか，そうした依存関係を報告します。たとえば，VCLInternet マージモジュールをインストールプロジェクトに追加したとします。この場合インストールツールは，VCLDatabase モジュールと StandardVCL モジュールを自動的に追加するか，または，これらのモジュールと依存関係にあることを報告します。

下の表は，各マージモジュールの依存関係を一覧にしたものです。インストールツールによって，依存関係への応答の仕方が異なります。Windows Installer 対応の InstallShied の場合，必要なモジュールを発見すれば自動的に追加します。他のインストールツールの中には，依存関係を報告するだけのものや，必要なモジュールがすべてプロジェクトに入っていないとビルドエラーを生成するものがあります。

表 17.2　マージモジュールと依存関係

マージモジュール	付属 BPL	依存関係
ADO	adortl60.bpl	DatabaseRTL, BaseRTL
BaseClientDataSet	cds60.bpl	DatabaseRTL, BaseRTL, DataSnap, dbExpress
BaseRTL	rtl60.bpl	なし
BaseVCL	vcl60.bpl, vclx60.bpl	BaseRTL
BDEClientDataSet	bdecds60.bpl	BaseClientDataSet, DataBaseRTL, BaseRTL, DataSnap, dbExpress, BDERTL
BDEInternet	inetdbbde60.bpl	Internet, DatabaseRTL, BaseRTL, BDERTL
BDERTL	bdertl60.bpl	DatabaseRTL, BaseRTL
DatabaseRTL	dbrtl60.bpl	BaseRTL
DatabaseVCL	vcldb60.bpl	BaseVCL, DatabaseRTL, BaseRTL
DataSnap	dsnap60.bpl	DatabaseRTL, BaseRTL
DataSnapConnection	dsnapcon60.bpl	DataSnap, DatabaseRTL, BaseRTL
DataSnapCorba	dsnapcrba60.bpl	DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL, BaseVCL
DataSnapEntera	dsnapent60.bpl	DataSnap, DatabaseRTL, BaseRTL, BaseVCL
DBCompatVCL	vcldbx60.bpl	DatabaseVCL, BaseVCL, BaseRTL, DatabaseRTL
dbExpress	dbexpress60.bpl	DatabaseRTL, BaseRTL
dbExpressClientDataSet	dbxcds60.bpl	BaseClientDataSet, DataBaseRTL, BaseRTL, DataSnap, dbExpress
DBXInternet	inetdbxpress60.bpl	Internet, DatabaseRTL, BaseRTL, dbExpress, DatabaseVCL, BaseVCL
DecisionCube	dss60.bpl	TeeChart, BaseVCL, BaseRTL, DatabaseVCL, DatabaseRTL, BDERTL
FastNet	nmfast60.bpl	BaseVCL, BaseRTL
InterbaseVCL	ibxpress60.bpl	BaseClientDataSet, BaseRTL, BaseVCL, DatabaseRTL, DatabaseVCL, DataSnap, dbExpress
Internet	inet60.bpl, inetdb60.bpl	DatabaseRTL, BaseRTL
InternetDirect	indy60.bpl	BaseVCL, BaseRTL
Office2000Components	dcloffice2k60.bpl	DatabaseVCL, BaseVCL, DatabaseRTL, BaseRTL
QuickReport	qrpt60.bpl	BaseVCL, BaseRTL, BDERTL, DatabaseRTL
SampleVCL	vclsmp60.bpl	BaseVCL, BaseRTL
TeeChart	tee60.bpl, teedb60.bpl, teeqr60.bpl, teeui60.bpl	BaseVCL, BaseRTL
VCLIE	vclie60.bpl	BaseVCL, BaseRTL
VisualCLX	visualclx60.bpl	BaseRTL
VisualDBCLX	visualdbclx60.bpl	BaseRTL, DatabaseRTL, VisualCLX
WebDataSnap	webdsnap60.bpl	XMLRTL, Internet, DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL

表 17.2 マージモジュールと依存関係（つづき）

マージモジュール	付属 BPL	依存関係
WebSnap	websnap61.bpl, vcljpg60.bpl	WebDataSnap, XMLRTL, Internet, DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL, BaseVCL
XMLRTL	xmlrtl60.bpl	Internet, DatabaseRTL, BaseRTL

ActiveX コントロール

C++Builder にバンドルされているコンポーネントとして，ActiveX コントロールがあります。コンポーネントラッパーはアプリケーションの実行形式ファイル（または実行時パッケージ）内にリンクしていますが，コンポーネントの .ocx ファイルもアプリケーションとともに配布する必要があります。コンポーネントには以下のものがあります。

- Chart FX，版権元 SoftwareFX Inc.
- VisualSpeller Control，版権元 Visual Components, Inc.
- Formula One（spreadsheet），版権元 Visual Components, Inc.
- First Impression（VtChart），版権元 Visual Components, Inc.
- Graph Custom Control，版権元 Bits Per Second Ltd.

自分で作成した ActiveX コントロールは，使用前に，配布先コンピュータに登録する必要があります。InstallShield Express などのインストールプログラムでは，この登録プロセスが自動化されます。ActiveX コントロールを手動で登録するには，TRegSvr デモアプリケーションまたは Microsoft のユーティリティである REGSERV32.EXE（Windows 9x には含まれません）を使います。

ActiveX コントロールをサポートする DLL も，アプリケーションとともに配布する必要があります。

ヘルパーアプリケーション

ヘルパーアプリケーションは独立したプログラムですが，これがないと，作成した C++Builder アプリケーションの一部または全部が機能しません。ヘルパーアプリケーションは，オペレーティングシステムに付属していることもあれば，ボーランド製品，あるいはサードパーティー製品の場合もあります。ヘルパーアプリケーションの例としては，InterBase のユーティリティプログラム Server Manager があります。

アプリケーションがヘルパープログラムに依存している場合は，できる限り，アプリケーションとともに配布するようにします。ヘルパープログラムの配布は，それぞれのプログラムの再配布ライセンス使用許諾書に従わなければなりません。詳細については，ヘルパープログラムのマニュアルを参照してください。

DLL の場所

1 つのアプリケーションのみが使用する DLL は，そのアプリケーションと同じディレクトリに置くことができます。複数のアプリケーションから使われる DLL は，それらすべてのアプリケーションから参照できるディレクトリに置かなければなりません。こういった共用ファイルは，通常 Windows ディレクトリまたは Windows のシステムディレクトリに置かれます。そうすればアプリケーションは必ず共有ファイルを見つけることができるからです。より良い方法としては，ボーランドデータベースエンジンのインストール方法と同じように，共用 DLL ファイルの専用ディレクトリを作成します。

注意 Windows のシステムディレクトリは Windows 95/98 と Windows NT とで異なります。通常は，Windows 95/98 では System ディレクトリ，NT では System32 ディレクトリが Windows のシステムディレクトリとなります。95/98 にも System32 ディレクトリを持つ環境がありますが，システムディレクトリとしては扱われません。

CLX アプリケーションの配布

Windows で動作する CLX アプリケーションを配布する方法は，一般のアプリケーションの配布と同じです。一般のアプリケーションの配布については，17-1 ページの「アプリケーションの配布の基本」を参照してください。データベース CLX アプリケーションのインストールの詳細については，17-6 ページの「データベースアプリケーションの配布」を参照してください。

CLX ランタイムを含めるには，アプリケーションと一緒に qtintf.dll を配布する必要があります。CLX アプリケーションと一緒にパッケージを配布する場合には，vcl60.bpl でなく clx60.bpl を配布する必要があります。

CLX アプリケーションの作成についての詳細は，第 14 章「クロスプラットフォームアプリケーションの開発」を参照してください。

データベースアプリケーションの配布

データベースにアクセスするアプリケーションの場合，アプリケーションの実行形式ファイルをホストコンピュータにコピーする以外に，インストールに関する特別な考慮事項があります。データベースアクセスは，多くの場合，個別のデータベースエンジンによって処理され，そのファイルは，アプリケーションの実行形式ファイルにリンクできません。データファイルがあらかじめ作成されていないときは，アプリケーションで利用できるようにしなければなりません。多層データベースアプリケーションの場合，アプリケーションを構成するファイルが，通常，複数のコンピュータに配置されているため，インストール時にはさらに特別な処理が必要です。

対応しているデータベーステクノロジー（ADO，BDE，dbExpress，InterBase Express）は種類が異なるため，それぞれ配布要件が異なります。どれを使用するにせよ，データベースアプリケーションが動作するシステム上にクライアント側ソフトをインストールしておく必要があります。加えて，BDE，ADO，dbExpress，InterBase を使用する場合は，データベースのクライアント側ソフトとやり取りするためのドライバが必要です。

dbExpress，BDE，および多層データベースアプリケーションの配布方法についての詳細は，以下の節を参照してください。

- dbExpress データベースアプリケーションの配布
- BDE アプリケーションの配布
- 多層データベースアプリケーションの配布（DataSnap）

クライアントデータセット（TClientDataSet など）またはデータセットプロバイダを使用するデータベースアプリケーションの場合は，midaslib.dcu と crtl.dcu を一緒に配布します（スタンドアロンの実

行形式ファイルを提供するときの静的リンクのため)。また，アプリケーションを（実行形式ファイルおよび必要な DLL とあわせて）パッケージ化する場合は，Midas.dll を含める必要があります。

ADO を使用するデータベースアプリケーションを配布する場合は，アプリケーションが動作するシステムに MDAC バージョン 2.1 以上をインストールしておく必要があります。Windows 2000，Internet Explorer（バージョン 5 以上）などのソフトウェアをインストールすると，MDAC は自動的にインストールされます。加えて，接続先データベースサーバー用のドライバをクライアントにインストールしておく必要があります。必要な配布手順は以上です。

InterBase Express を使用するデータベースアプリケーションを配布する場合は，アプリケーションが動作するシステムに InterBase クライアントをインストールしておきます。InterBase の場合，アクセス可能なディレクトリに gd32.dll と interbase.msg を置きます。必要な配布手順は以上です。InterBase Express コンポーネントが 直接 InterBase Client API と通信するので，ドライバを追加する必要はありません。詳細については，Borland Web サイトの『Embedded Installation Guide』を参照してください。

上に挙げたデータベーステクノロジーだけでなく，サードパーティ製のデータベースエンジンを使って C++Builder アプリケーションでデータベース接続を行うこともできます。再配布の権利，インストール，および環境設定については，データベースエンジンのマニュアルを参照するか，各ベンダーにおたずねください。

dbExpress データベースアプリケーションの配布

dbExpress とは，データベース情報に迅速にアクセスさせる軽量のネイティブドライバ一式です。dbExpress は Linux でも使用できるので，クロスプラットフォーム開発に対応しています。dbExpress コンポーネントの使い方については，第 26 章「単方向データセットの使い方」を参照してください。

dbExpress アプリケーションは，スタンドアロンの実行形式ファイルとして配布するか，または，関連の dbExpress ドライバ DLL が含まれた実行形式ファイルとして配布します。

スタンドアロンの実行形式ファイルで配布するには，dbExpress オブジェクトファイルを実行形式ファイルと静的にリンクしておく必要があります。lib ディレクトリにある以下の DCU ファイルを含めることで，この静的リンクが行われます。

表 17.3　スタンドアロンの実行形式ファイルとして dbExpress アプリケーションを配布する場合

データベースユニット	結合するアプリケーション
dbExpINT	InterBase データベースと接続するアプリケーション
dbExpORA	Oracle データベースと接続するアプリケーション
dbExpDB2	DB2 データベースと接続するアプリケーション
dbExpMYS	MySQL 3.22.x データベースと接続するアプリケーション
dbExpMYSQL	MySQL 3.23.x データベースと接続するアプリケーション
crtl	dbExpress を使用するすべての実行形式ファイルで必要
MidasLib	dbExpress 実行形式ファイルがクライアントデータセット（TClientDataSet など）を使用する場合に必要

メモ　Informix を使用するデータベースアプリケーションの場合，スタンドアロンの実行形式ファイルは配布できません。かわりに，実行形式ファイルと一緒にドライバ DLL の dbexpinf.dll を配布します（下の表を参照)。

スタンドアロンの実行ファイルを配布しない場合，実行形式ファイルと一緒に dbExpress ドライバおよび DataSnap DLL ファイルを配布できます。DLL ファイルと，該当するアプリケーションの一覧を次に示します。

表 17.4　ドライバ DLL と一緒に dbExpress アプリケーションを配布する場合

データベース DLL	配布するアプリケーション
dbexpinf.dll	Informix データベースと接続するアプリケーション
dbexpint.dll	InterBase データベースと接続するアプリケーション
dbexpora.dll	Oracle データベースと接続するアプリケーション
dbexpdb2.dll.	DB2 データベースと接続するアプリケーション
dbexpmys.dll	MySQL 3.22.x データベースと接続するアプリケーション
dbexpmysql.dll	MySQL 3.23.x データベースと接続するアプリケーション
Midas.dll	クライアントデータセットを使用するデータベースアプリケーション

BDE アプリケーションの配布

BDE（ボーランドデータベースエンジン）は，データベースとやり取りするための広範な API を定義しています。BDE は，各種のデータアクセスメカニズムの中でもっとも広範な機能をサポートし，また非常に役立つユーティリティ類が付属しています。Paradox テーブルや dBASE テーブルのデータを操作するには，BDE を使うのがベストです。

アプリケーションのデータベースアクセスは，さまざまなデータベースエンジンによって実行できます。アプリケーションでは，BDE やサードパーティーのデータベースエンジンを利用できます。SQL データベースシステムへのネイティブアクセスを可能にするために，SQL Link が提供されています（バージョン（版）によっては使用できないことがあります）。次に示す節では，アプリケーションのデータベースアクセス要素のインストールについて説明します。

- ボーランドデータベースエンジン
- SQL Link

ボーランドデータベースエンジン

C++Builder の標準データコンポーネントがデータベースにアクセスできるようにするには，ボーランドデータベースエンジン（BDE：Borland Database Engine）がインストールされ，アクセス可能な状態でなければなりません。BDE の再配布に関する権利および制約事項の詳細については BDEDEPLOY ドキュメントを参照してください。

BDE のインストールには，InstallShield Express（またはその他の動作が保証されたインストールプログラム）の使用をお勧めします。InstallShield Express によって，必要なレジストリエントリが作成され，アプリケーションで必要になる任意のエリアスが定義されます。BDE ファイルおよび BDE サブセットの配布に，動作保証されたインストールプログラムを使うことは，以下のような理由のためです。

- BDE または BDE サブセットを不適切にインストールすると，BDE を使用しているほかのアプリケーションに障害の発生する場合がある。そのようなアプリケーションは，ボーランド製品だけでなく，BDE を使用する数多くのサードパーティープログラムも含まれる

- 16 ビット Windows では .INI ファイルが使用されていたが，32 ビット Windows 95/NT 以降 では，BDE 環境設定情報が Windows レジストリに格納される。インストールおよびアンインストールについて，正しいエントリを作成および削除することは，複雑な作業である

アプリケーションが実際に必要とする BDE のコンポーネントだけをインストールすることもできます。たとえば，アプリケーションが Paradox のテーブルだけを使用する場合は，Paradox のテーブルにアクセスするために必要な BDE の部分だけをインストールするだけですみます。これによって，アプリケーションに必要なディスク領域が節約できます。InstallShield Express などの動作保証されたインストールプログラムでは，BDE の部分インストールを実行できます。配布済みアプリケーションでは使用しない場合でも，ほかのプログラムで使用する BDE システムファイルを残しておくように注意が必要です。

SQL Link

SQL Link によって，ボーランドデータベースエンジンを介して，アプリケーションを SQL データベース用のクライアントソフトウェアに接続するドライバが提供されます。SQL Link の再配布に関する具体的な権利および制約事項については DEPLOY ドキュメントを参照してください。ボーランドデータベースエンジン（BDE: Borland Database Engine）の場合と同様に，SQL Link も InstallShield Express（またはその他の動作保証されたインストールプログラム）を使ってインストールしなければなりません。

メモ SQL Link は，BDE をクライアントソフトウェアに接続するものであり，SQL データベース自体には接続しません。使っている SQL データベースシステム用にクライアントソフトウェアをインストールする必要はあります。クライアントソフトウェアのインストールと環境設定についての詳細は，SQL データベースシステムのマニュアルを参照するか，ベンダーにおたずねください。

表 17.5 に，SQL Link が別の SQL データベースシステムに接続するために使用するドライバと設定ファイルの名前を示します。これらのファイルは SQL Link に付属しており，C++Builder のライセンス使用許諾書に従って再配布できます。

表 17.5　SQL データベースクライアントソフトウェアファイル

ベンダー	再配布可能ファイル
Oracle 7	SQLORA32.DLL と SQL_ORA.CNF
Oracle 8	SQLORA8.DLL と SQL_ORA8.CNF
Sybase Db-Lib	SQLSYB32.DLL と SQL_SYB.CNF
Sybase Ct-Lib	SQLSSC32.DLL と SQL_SSC.CNF
Microsoft SQL Server	SQLMSS32.DLL と SQL_MSS.CNF
Informix 7	SQLINF32.DLL と SQL_INF.CNF
Informix 9	SQLINF9.DLL と SQL_INF9.CNF
DB/2 2	SQLDB232.DLL と SQL_DB2.CNF
DB/2 5	SQLDB2V5.DLL と SQL_DB2V5.CNF
InterBase	SQLINT32.DLL と SQL_INT.CNF

InstallShield Express またはその他の動作保証されたインストールプログラムを使って，SQL Link をインストールします。SQL Link のインストールおよび環境設定の詳細については，BDE のメインディレクトリにデフォルトでインストールされるヘルプファイル SQLLNK32.HLP を参照してください。

多層データベースアプリケーションの配布（DataSnap）

DataSnap は，クライアントアプリケーションがアプリケーションサーバー内のプロバイダと接続できるようにします。このようにして，C++Builder アプリケーションに多層データベース機能を提供します。

DataSnap および多層アプリケーションのインストールは，InstallShield Express（または，Borland が保証しているインストールスクリプトユーティリティ）を使って行います。アプリケーションと一緒に再配布するファイルの詳細については，DEPLOY ドキュメント（C++Builder のメインディレクトリにあります）を参照してください。また，REMOTE ドキュメントに，再配布可能な DataSnap ファイルおよび再配布方法に関する説明があります。

Web アプリケーションの配布

一部の C++Builder アプリケーションは，Server-side Extension DLL（ISAPI および Apache），CGI アプリケーション，および ActiveForm といった形式で，World Wide Web 上で実行されるように設計されています。

Web アプリケーションの配布手順は，アプリケーションのファイルが Web サーバーに配布される点を除いて，通常のアプリケーションと同じです。通常のアプリケーションのインストールの詳細については，17-1 ページの「アプリケーションの配布の基本」を参照してください。データベース Web アプリケーションの配布に関する詳細は，17-6 ページの「データベースアプリケーションの配布」を参照してください。

Web アプリケーションの配布に関する特別な注意事項は以下のとおりです。

- BDE データベースアプリケーションの場合，ボーランドデータベースエンジン（または代替データベースエンジン）を，アプリケーションファイルとともに Web サーバーにインストールする
- dbExpress アプリケーションの場合，dbExpress DLL ファイルを同じパスに入れる。同じパスに入っていれば，アプリケーションファイルと一緒に dbExpress ドライバが Web サーバーにインストールされる
- 必要なディレクトリファイルすべてにアプリケーションがアクセスできるよう，該当するディレクトリのセキュリティを設定する
- アプリケーションの入っているディレクトリには，読み出し属性と実行属性を持たせること
- アプリケーションでは，データベースその他のファイルへのアクセスに，ハードコードされたパスを使わないこと
- ActiveX コントロールの場所は，<OBJECT> HTML タグの CODEBASE パラメータによって示される

Apache への配布については，次の節で説明します。

Apache サーバーへの配布

WebBroker は Apache のバージョン 1.3.9 以上をサポートしており，Apache 用の DLL ファイルと CGI アプリケーションを作成できます。

モジュールとアプリケーションは，Apache の httpd.conf ファイル（通常は Apache のインストール場所の \ conf ディレクトリにある）を変更することによって，使用可能にし，また設定の変更を行います。

モジュールを使用可能にする

DLL は，Apache の Modules サブディレクトリに物理的に配置します。

モジュールを使用可能にするには，httpd.conf に対して 2 個所の変更が必要です。

1. LoadModule エントリを追加して，Apache が DLL を見つけてロードできるようにします。

```
LoadModule MyApache_module modules/Project1.dll
```

2. リソースロケータのエントリを追加します（httpd.conf 内で，LoadModule エントリの後ろであればどこでもかまいません）。例を示します。

```
# Sample location specification for a project named project1.
<Location /project1>
   SetHandler project1-handler
</Location>
```

これによって，http://www.somedomain.com/project1 に対するリクエストを，すべて Apache モジュールに渡すことが可能になります。

SetHandler 指令は，リクエストを処理する Web サーバーアプリケーションを指定します。SetHandler の引数は，ContentType グローバル変数の値に設定する必要があります。

CGI アプリケーション

CGI アプリケーションを作成する場合は，プログラムを実行できるように，物理的なディレクトリ（httpd.conf ファイルの Directory 指令で指定するディレクトリ）で ExecCGI オプションおよび SetHandler 節を設定します。これにより，CGI スクリプトを実行可能にします。許可内容を正しく設定するには，Alias 指令で Options ExecCGI および SetHandler を両方とも有効にします。

メモ もう 1 つの方法は，ScriptAlias 指令を（Options ExecCGI なしで）使うことですが，この方法を使うと，CGI アプリケーションが ScriptAlias ディレクトリ内のファイルを読み出せなくなることがあります。

次の httpd.conf の行は，Alias 指令を使ってサーバー上に仮想ディレクトリを作成し，CGI スクリプトの正確な位置を対応付けています。

```
Alias/MyWeb/"c:/httpd/docs/MyWeb/"
```

これによって，\MyWeb\mycgi.exe といったリクエストは，c:\httpd\docs\MyWeb\mycgi.exe というスクリプトを実行することで満たされます。

httpd.conf ファイルで Directory 指令を使って，Options を All または ExecCGI に設定する方法もあります。Options 指令を使うと，特定のディレクトリで使用できるサーバー機能を指定することができます。

Directory 指令の働きは，指定されたディレクトリおよびサブディレクトリに適用される指令のまとまりを囲むことです。Directory 指令の例を示します。

```
<Directory "c:/httpd/docs/MyWeb">
    AllowOverride None
    Options ExecCGI
    Order allow,deny
    Allow from all
    AddHandler cgi-script exe cgi
</Directory>
```

この例では，Options を ExecCGI に設定して，MyWeb ディレクトリ内の CGI スクリプトの実行を許可しています。AddHandler 節によって，exe や cgi などの拡張子を持つファイルが CGI スクリプト（実行形式）であることを Apache に認識させることができます。

メモ Apache は，httpd.conf ファイル内の User 指令で指定されたアカウントの範囲内でサーバー上にてローカルに実行されます。アプリケーションが必要とするリソースに対し，ユーザーが適切な権利を持ってアクセスできるようにしてください。

配布に関する他の情報については，Apache ディストリビューションに付属する Apache の LICENSE ファイルを参照してください。Apache の設定に関する情報は，http://www.apache.org を参照してください。

さまざまな動作環境を考慮したプログラミング

オペレーティングシステム環境の特性が多様であるため，ユーザーの選択事項や環境設定によって変化する要素がいくつかあります。別のコンピュータに配布されたアプリケーションに影響する可能性のある要素には，以下のものがあります。

- 画面解像度と色深度
- フォント
- オペレーティングシステムのバージョン
- ヘルパーアプリケーション
- DLL の場所

画面解像度と色深度

デスクトップのサイズとコンピュータ上で利用可能な色の数はインストールされているハードウェアの種類によって，設定変更できます。これらの属性は，開発用コンピュータの場合と，配布先コンピュータの場合では，異なる可能性があります。

画面環境の変化に対してアプリケーションの外観（ウィンドウ，オブジェクト，およびフォントサイズ）を調整するには，以下の 3 つの方法があります。

- ユーザーが使用できる最小の画素構成（通常は 640 × 480 ピクセル /96 DPI）に合わせて画面デザインを設計する。画面に合わせてオブジェクトの大きさを変更するような特別な作業は行わない。そのため，画素数が多くなる（画面が広くなる）につれて，オブジェクトは視覚的に小さく表示される

- 画素数のことは考慮せずに画面デザインを設計し，実行時に，設計時の解像度と実行時の解像度が比例するように，フォームとすべてのビジュアルコントロールのサイズを画面解像度の違いの比率に合わせて調整する
- 画素数のことは考慮せずに画面デザインを設計し，実行時に，フォームのサイズだけを変更する。フォームにビジュアルコントロールが配置されている場合には，フォーム上のすべてのコントロールを表示させるためには，ユーザーにスクロールしてもらう必要が発生する場合がある

動的なサイズ変更を行わない場合の考慮事項

アプリケーションを構成するフォームおよびビジュアルコントロールに対して，実行時に動的なサイズ変更を行わない場合，最低の解像度で使われることを想定してアプリケーションを設計してください。そのような設計をしなかった場合，画面設計時の環境よりも低い解像度の環境では，フォームの一部が画面に収まる大きさに切り捨てられます。

たとえば，1024 × 768 の解像度の開発環境で，横幅が 700 ピクセルのフォームを作成した場合，640 × 480 の画面解像度を持つコンピュータでは，このフォームの一部は表示されません。

フォームとコントロールを動的にサイズ変更するときの考慮事項

アプリケーションのフォームとビジュアルコントロールが動的にサイズ変更される場合，サイズ変更プロセスのすべての観点を考慮して，アプリケーションがあらゆる画面解像度の下で最適に表示されるようにしておく必要があります。アプリケーションの視覚要素を動的にサイズ変更するときの考慮事項は以下のとおりです。

- ダイアログのような，ほとんど文字だけのフォームを表示するときは，設計時のフォントのポイント数を尊重するようにします。こうすることにより，どの環境でも大体同じ外観を維持できます。これには，TCustomForm の Scaled プロパティを **true** に，AutoScroll プロパティを **false** に設定します。画像を貼り付けているフォームの場合には，コントロールが自動的に拡大縮小されると困る場合があります。イメージコントロールなどでは Stretch プロパティによって，画像そのものを拡大縮小するように指定できますが，アイコンやビットマップなど，画像によっては大きさを変えたくない場合もあります。これを避けるには TCustomForm の Scaled プロパティを **false** にし，自動的なコントロールの位置調整を無効にします。この場合，メモやラベルなど，その他のビジュアルコントロールの位置調整は手動で行わなければなりません。実行時に TScreen::Height または TScreen::Width プロパティを使って，ユーザーのコンピュータの解像度を取得します。設計時の値を実行時の値で割り，2 台のコンピュータの画面解像度の違いの比率を出します。
- アプリケーションのビジュアルコンポーネント（フォームとコントロール）のサイズを変更するには，コンポーネントのサイズとフォーム上の位置を増減します。ビジュアルコントロールのサイズやフォーム上の位置は，設計時と実行時の解像度の違いから計算された比率に従って調整されます。TCustomForm::Scaled プロパティを **true** に設定し，TWinControl::ScaleBy メソッド（クロスプラットフォームアプリケーションの場合は TWidgetControl.ScaleBy）を呼び出すことで，フォーム上のビジュアルなコントロールのサイズと位置が自動的に変更されます。ただし，ScaleBy メソッドでは，フォームの高さと幅は変更されません。フォームの高さと幅については，Height プロパティと Width プロパティを手動で変更する必要があります。
- TWinControl.ScaleBy メソッド（クロスプラットフォームアプリケーションの場合は TWidgetControl::ScaleBy）を使って自動的にコントロールのサイズを変更するかわりに，各ビジュアルコントロールをループ内で参照し，そのサイズと位置を設定することで手動で変更する

こともできます。各コントロールの Height プロパティと Width プロパティ，および Top プロパティと Left プロパティに，画面解像度の違いを表す同じ比率を掛けることで，設計時の配置と相対的なサイズを維持することができます。

- アプリケーションの開発を実行用コンピュータよりも高い画面解像度を持ったコンピュータ上で行うと，ビジュアルコントロールのサイズ調整中にフォントサイズが縮小されるので，設計時のフォントサイズが極端に小さいと，フォントがつぶれて読めなくなる場合があります。たとえば，デフォルトのフォントサイズが 8 だったとします。開発に使用したコンピュータの画面解像度が 1024 × 768 で，そのアプリケーションを解像度が 640 × 480 のコンピュータで使用すると，ビジュアルコントロールのサイズは元の 60% 強（640 / 1024 = 0.625）に縮小されます。したがって，フォントサイズは 8 から 5（8 * 0.625 = 5）に変わります。アプリケーションで表示される文字はつぶれて可読性のないフォントとなってしまいます。
- TLabel や TEdit などの一部のビジュアルコントロールは，コントロールのフォントサイズが変わると，自分自身のサイズを変更します。フォームとコントロールのサイズが動的に変更されると，すでに配布されているアプリケーションに影響を与えます。つまり，画面解像度の差に対応したサイズの変更に加えて，さらにフォントサイズの変更によりコントロールのサイズが変わってしまいます。この副作用をなくすには，コントロールの AutoSize プロパティを **false** に設定してください。
- キャンバスに直接描画するときのように，明示的なピクセル座標は使わないようにします。そのかわりに，開発用コンピュータとユーザーコンピュータの画面解像度の差分比率に比例した比率で座標を変更します。たとえば，アプリケーションで高さ 10 ピクセル，幅 20 ピクセルの矩形を描く場合は，10 と 20 に画面解像度の差分比率をかけます。こうすることによって，矩形は，異なる画面解像度でも視覚的に同じサイズで表示されるようにすることができます。

異なる色深度環境への対応

利用可能な色の種類が同一に環境設定されていないすべての配布先コンピュータに対応するために，もっとも安全な方法は，必要最低限の色のグラフィックを使うことです。特にメニューやボタンのグリフで使われる画像では 16 色に押さえるようにしてください。その他の表示用の画像は，実行環境に合わせ複数の画像を用意するか，またはアプリケーション自身に最低限必要な環境を明記するようにします。

フォント

Windows には，標準で True Type フォントとラスターフォントが付属しています。Linux のディストリビューションによっては，標準のフォントセットが付属しています。複数のコンピュータに配布するアプリケーションを設計するときは，標準フォントセット以外のフォントを持たないコンピュータもあるということを理解しておく必要があります。

アプリケーション内で文字を表示するコンポーネントでは，どの配布先でも利用できるフォントを選ぶ必要があります。

アプリケーション内で標準以外のフォントを使う必要があるときは，そのフォントをアプリケーションとともに配布する必要があります。それには，インストールプログラムかアプリケーション自体によって，そのフォントが配布先コンピュータにインストールされるようにしなければなりません。

サードパーティから提供されるフォントを配布する場合は，そのフォントの再配布が禁止されている場合があるので注意してください。

Windows には，存在しないフォントが指定された場合，適当なフォントを割り当てる機能があります。これにより致命的なエラーを回避でき，とりあえず文字を表示できるようになります。しかし，設計時に意図したデザインを保つことはできません。また，日本語を表示したいのに英語フォントで表示される場合もあります。

オペレーティングシステムのバージョン

アプリケーションからオペレーティングシステムの API 関数を使ったり，オペレーティングシステム領域にアクセスしたりするとき，バージョンの異なるオペレーティングシステムがインストールされているコンピュータでは，関数，操作，または特定の機能が利用できない場合があります。

こういった障害の可能性をユーザーに説明するには，以下の 3 つのオプションがあります。

- アプリケーションのシステム要件で，動作可能なオペレーティングシステムのバージョンを指定する。互換性のあるバージョンのオペレーティングシステムにアプリケーションをインストールして，使用するのは，ユーザーの責任となります。
- アプリケーションのインストール時に，オペレーティングシステムのバージョンを確認する。互換性のないバージョンのオペレーティングシステムが存在する場合は，インストールプロセスを停止するか，インストール作業者にその問題について警告します。
- 実行時にオペレーティングシステムのバージョンを確認してから，バージョン固有の操作を実行する。処理を継続できないバージョンのオペレーティングシステムだったときは，プロセスを終了してユーザーに警告します。または，異なるバージョンのオペレーティングシステムごとに，別のコードを提供します。

メモ 操作の中には Windows 95/98 と Windows NT/2000/XP で呼び方が異なるものもあります。Windows のバージョンを判別するには，Windows API の GetVersionEx を使います。

ソフトウェアのライセンス

C++Builder アプリケーションの関連ファイルを配布する場合，制約事項が該当したり，一切再配布できなかったりするファイルがあります。こうしたファイルの配布に関する規約については，以下の文書で説明しています。

DEPLOY

DEPLOY ドキュメントは，C++Builder アプリケーションに含まれるか，アプリケーションに関連する各種のコンポーネント，ユーティリティ，およびその他の製品の配布について，法的側面をいくつか説明しています。DEPLOY ドキュメントは，C++Builder のメインディレクトリにインストールされています。以下の項目について説明しています。

- .exe ファイル，.dll ファイル，.bpl ファイル

- コンポーネントパッケージと設計時パッケージ
- ボーランドデータベースエンジン（BDE: Borland Database Engine）
- ActiveX コントロール
- サンプルイメージ
- SQL Link

README

README ドキュメントにはコンポーネント，ユーティリティ，またはその他の製品の再配布権に関係のある情報も含まれます。README ドキュメントは，C++Builder のメインディレクトリ にインストールされています。

ライセンス使用許諾書

C++Builder のライセンス使用許諾書および印刷文書によって，C++Builder に関するその他の法的権利および法的義務が保証されます。

サードパーティ製品のドキュメント

サードパーティ製のコンポーネント，ユーティリティ，ヘルパーアプリケーション，データベースエンジン，およびその他の製品の再配布権は，その製品を提供するベンダーが所有しています。C++Builder アプリケーションに付随する他社製品の再配布についての詳細は，配布の前に，製品のドキュメントを参照するか，ベンダーに問い合わせてください。

第II部

データベースアプリケーションの開発

第 II 部「データベースアプリケーションの開発」の各章では，C++Builder データベースアプリケーションの作成に必要な概念とテクニックについて解説しています。

メモ データベースアプリケーションの作成についてのサポートのレベルは，C++Builder の版によって異なります。特に，クライアントデータセットを使用するには Professional 版以上が必要であり，多層データベースアプリケーションの開発には Enterprise 版が必要です。

第18章

データベースアプリケーションの設計

データベースアプリケーションは，ユーザーがデータベースに保存されている情報と対話するためのものです。データベースは，情報を構造化し，異なるアプリケーション間での情報の共有を可能にします。

C++Builder は，リレーショナルデータベースアプリケーションをサポートしています。リレーショナルデータベースは，情報を行（レコード）と列（項目）で構成されるテーブルの形に整理します。これらのテーブルは，リレーショナル計算法と呼ばれる簡単な演算によって操作できます。

データベースアプリケーションの設計にあたっては，データの構造を理解していなければなりません。その構造に基づいて，ユーザーがデータを表示したり新規情報を入力したり既存データを変更したりするためのユーザーインターフェースを設計することができます。

この章では，データベースアプリケーションの設計で一般に考慮すべき点と，ユーザーインターフェースの設計に関して決定すべき事項について述べます。

データベースを使用する

C++Builder には，データベースにアクセスして，それらに含まれる情報を示すための，多くのコンポーネントがあります。それらは，データアクセスのメカニズムに従って，以下のようにまとめられています。

- コンポーネントパレットの［BDE］ページには，Borland Database Engine (BDE) を使用するコンポーネントがあります。BDE は，データベースと対話するための大規模な API を定義しています。データアクセスメカニズムの中で，BDE は，最も広い範囲の機能をサポートしており，ほとんどのサポート用ユーティリティが付属しています。Paradox や dBASE のテーブルを扱う場合には，これを利用するのが最善です。しかしながら，配布の点では，最も複雑なメカニズムでもあります。BDE コンポーネントの使い方については，第 24 章「ボーランドデータベースエンジンの使い方」を参照してください。

- コンポーネントパレットの［ADO］ページには，ActiveX Data Objects（ADO）を使って OLEDB を介してデータベース情報にアクセスするコンポーネントがあります。ADO は，Microsoft による規格です。異なるデータベースサーバーに接続するために利用可能な，数多くの ADO ドライバが存在しています。ADO ベースのコンポーネントを使用すれば，アプリケーションを ADO ベースの環境に統合できます（たとえば，ADO ベースのアプリケーションサーバーを利用できます）。ADO コンポーネントの使い方については，第 25 章「ADO コンポーネントの操作」を参照してください。
- コンポーネントパレットの［dbExpress］ページには，dbExpress を使ってデータベース情報にアクセスするコンポーネントがあります。dbExpress は軽量のドライバのセットで，データベース情報に最も高速にアクセスできます。加えて，dbExpress コンポーネントは，Linux でも利用可能なので，クロスプラットフォーム開発をサポートします。しかしながら，dbExpress データベースコンポーネントは，データ操作関数のうち，最も狭い範囲のものしかサポートしていません。dbExpress コンポーネントの使い方については，第 26 章「単方向データセットの使い方」を参照してください。
- コンポーネントパレットの［InterBase］ページには，独立したエンジンレイヤーを通さずに，InterBase データベースに直接アクセスするコンポーネントがあります。
- コンポーネントパレットの［Data Access］ページには，任意のデータアクセスメカニズムと組み合わせて使用することができるコンポーネントがあります。このページは，TClientDataset を含んでいます。これは，ディスクに格納されているデータを扱えます。または，やはりこのページにある TDataSetProvider コンポーネントを使えば，ほかのグループのコンポーネントを扱えます。クライアントデータセットの使用方法についての詳細は，第 27 章「クライアントデータセットの使い方」を参照してください。TDataSetProvider についての詳細は，第 28 章「プロバイダコンポーネントの使い方」を参照してください。

メモ C++Builder のバージョンによって，BDE，ADO，または dbExpress を使うデータベースサーバーにアクセスするためのドライバは異なっています。

データベースアプリケーションの設計にあたっては，どのコンポーネントのセットを使用するかを決定しなければなりません。データアクセスメカニズムはそれぞれ，サポートする機能の範囲，配布のしやすさ，およびさまざまなデータベースサーバーをサポートするドライバが利用可能かどうかという点で異なっています。

データアクセスメカニズムに加えて，データベースサーバーも選ばなければなりません。データベースの種類は数多くあるので，特定のデータベースサーバーを採用する前に，それぞれの種類の利点と欠点を考慮しておいてください。

どの種類のデータベースにも，情報を格納するテーブルがあります。さらに，ほとんどのサーバーは，以下のような付加的な機能をサポートしています（すべてのサーバーに当てはまるわけではありません）。

- データベースのセキュリティ
- トランザクション
- 参照の整合性，ストアドプロシージャ，トリガー

データベースの種類

リレーショナルデータベースサーバーは，情報を保管する方法や，複数のユーザーが同時に情報にアクセスできるようにする方法の点で，さまざまに異なっています。C++Builder は，2 種類のリレーショナルデータベースサーバーをサポートしています。

- **リモートデータベースサーバー**は，リモートマシン上にあります。ときには，リモートデータベースサーバーのデータが 1 台のマシンではなく，数台のサーバーに分散されていることがあります。それぞれのリモートデータベースサーバーは，情報を保管する方法は異なっていますが，共通したロジックインターフェースをクライアントに提供します。この共通インターフェースは Structured Query Language（SQL）です。SQL を使ってアクセスを行うので，これらは SQL サーバーとも呼ばれます。また，リモートデータベース管理システム（RDBMS）と呼ばれることもあります。SQL を構成する一般的なコマンドに加えて，たいていのリモートデータベースサーバーは，独自の SQL の「方言」をサポートしています。SQL サーバーの例としては，InterBase，Oracle，Sybase，Informix，Microsoft SQL Server，DB2 などがあります。
- **ローカルデータベース**は，ローカルドライブまたはローカルエリアネットワーク上にあります。これらは多くの場合，データにアクセスする独自の API を持っています。何人かのユーザーで共有するときは，ファイルベースのロック機構を使用します。このため，ローカルデータベースは，ファイルベースのデータベースとも呼ばれます。ローカルデータベースには，Paradox，dBASE，FoxPro，Access などがあります。

ローカルデータベースを使用するアプリケーションは，アプリケーションとデータベースが単一のファイルシステムを共用することから，単一層アプリケーションと呼ばれます。リモートデータベースサーバーを使用するアプリケーションは，アプリケーションとデータベースが別個のシステム（層）上で動作するため，**2 層アプリケーション**または多層アプリケーションと呼ばれます。

使用するデータベースにどの種類を選ぶかは，いくつかの要因によって異なります。たとえば，データがすでに既存のデータベースに入っていることもあります。アプリケーションで使用するデータベーステーブルをこれから作成する場合は，次の点を考慮するとよいでしょう。

- テーブルを共用するユーザー数：リモートデータベースサーバーは，複数のユーザーから同時にアクセスされるように設計されています。これらは，トランザクションと呼ばれるメカニズムを通じて，複数ユーザーサポートを提供します。ローカルデータベースの中には，（Local InterBase のように）トランザクションのサポートも提供しているものがありますが，その多くはファイルベースのロックメカニズムしか提供しておらず，（クライアントデータセットファイルのように）マルチユーザーサポートをまったく提供していないものもあります。
- テーブルに保存するデータ量：リモートデータベースサーバーは，ローカルデータベースよりも大量のデータを保存できます。多量のデータを保管することを第一にして設計されているリモートデータベースサーバーもあれば，（更新の速度など）他の基準に基づいて最適化されているものもあります。
- データベースにどの程度のパフォーマンス（処理速度）を要求するか：ローカルデータベースは通常，データベースアプリケーションと同じシステム上に置かれるため，リモートデータベースよりも高速です。リモートデータベースサーバーはそれぞれ，異なった種類の処理のサポートのために最適化されているので，リモートデータベースサーバーを選択する際には，そのパフォーマンスを考慮するとよいでしょう。

- データベース管理のためにどんな種類のサポートが利用可能か：ローカルデータベースは，リモートデータベースサーバーほどサポートを必要としません。通常，別個にインストールするサーバーや高価なサイトライセンスを必要としないため，より安価に運用できます。

データベースのセキュリティ

データベースには，機密情報が保存されることが少なくありません。種々のデータベースがそのような情報を保護するセキュリティ方式を備えています。Paradox や dBASE のような一部のデータベースは，テーブルレベルまたは項目レベルのみのセキュリティを提供しています。保護されたテーブルにユーザーがアクセスするときは，パスワードを入力しなければなりません。ユーザーが認証されたら，アクセス権のある項目（列）に限って表示できます。

たいていの SQL サーバーでは，データベースサーバーを少しでも使用するならばパスワードとユーザー名が必要です。ユーザーがデータベースにログインしたら，そのユーザー名とパスワードによって，どのテーブルを使用できるかが決まります。SQL サーバーにパスワードを入力する方法については，21-4 ページの「サーバーログインの制御」を参照してください。

データベースアプリケーションを設計するときは，データベースサーバーにどんな種類の認証が必要かを考えなければなりません。多くの場合，アプリケーションはデータベースへの明示的なログインをユーザーから隠すように設計されており，ユーザーはアプリケーションにのみログインすればよいようになっています。ユーザーがデータベースパスワードを入力しないですむようにしたい場合，パスワードを必要としないデータベースを使用するか，プログラムによってパスワードとユーザー名がサーバーに入力されるようにしなければなりません。パスワードをプログラムによって入力する場合，アプリケーションからパスワードを読むことによってセキュリティを破られることのないよう注意しなければなりません。

ユーザーにパスワードを入力させたい場合，いつパスワードを要求するかを考えなければなりません。現在はローカルデータベースを使用しているが将来はもっと大きな SQL サーバーにスケールアップするつもりならば，個々のテーブルを開くときではなく，SQL データベースにログインする時点でパスワードを要求するとよいでしょう。

セキュリティ保護されたいくつかのシステムまたはデータベースにログインするために複数のパスワードを必要とするアプリケーションの場合，ユーザーが 1 つのマスターパスワードを入力することによって，保護されたシステムへのパスワードテーブルにアクセスできるようにします。これにより，ユーザーがいくつもパスワードを入力することなく，アプリケーションがプログラムで自動的にパスワードを入力します。

多層アプリケーションでは，まったく異なるセキュリティモデルの使用が適しています。HTTP または COM+ を使って中間層へのアクセスを制御し，データベースサーバーへのログインの詳細はすべて中間層に処理させることができます。

トランザクション

トランザクションはアクションのグループであり，トランザクションがコミット（確定）される前に，そのすべてのアクションがデータベース内の 1 つまたは複数のテーブルに対して正常に実行され

なければなりません。これらのアクションのどれかが失敗すると，すべてのアクションはロールバックされます（元に戻されます）。

トランザクションは，以下の事柄を確実にします。

- 単一のトランザクション内のすべての更新は，コミットされるか，破棄されて前の状態にロールバックされるかのどちらかになります。これは，**原子性**と呼ばれます。
- トランザクションは，状態の不変量を保ちながら，システムの状態を正しく変換することです。これは，**一貫性**と呼ばれます。
- 複数の同時トランザクションが，相互の部分的またはコミットされていない結果を参照することはありません。そのような参照は，アプリケーションの状態に非一貫性を生じさせることがあるからです。これは，**分離性**と呼ばれます。
- レコードへコミットされた更新は，通信障害，プロセス障害，およびサーバーのシステム障害などの障害があっても，そのまま残ります。これは，**耐久性**と呼ばれます。

このように，トランザクションは，1 つのデータベースコマンドまたは一連のコマンドの途中で発生するハードウェア障害に対して保護します。トランザクションによるログ記録を使えば，ディスクのメディア障害の後でも，定常状態を回復できます。トランザクションはまた，SQL サーバーのマルチユーザー並行制御の基本にもなっています。すべてのユーザーがトランザクションだけを通してデータベースを使用しているときには，1 人のユーザーのコマンドが別のユーザーのトランザクションの統一性を乱すことは決してありません。SQL サーバーは，入ってくるトランザクションを管理し，トランザクションは全体として成功または全体として失敗したものとして扱われます。

トランザクションサポートは，ほとんどのローカルデータベースには含まれていませんが，Local InterBase はそれを提供しています。さらに，BDE ドライバは，いくつかのローカルデータベースに対して，制限付きのトランザクションサポートを提供しています。データベーストランザクションサポートは，データベース接続を表わすコンポーネントによって提供されています。データベース接続コンポーネントを使ったトランザクションの管理の詳細は，21-6 ページの「トランザクションの管理」を参照してください。

多層アプリケーションでは，データベース操作以外のアクションを含むトランザクションや複数のデータベースにわたるトランザクションを作成できます。多層アプリケーションにおけるトランザクションの使用の詳細は，29-18 ページの「多層アプリケーションでのトランザクション管理」を参照してください。

参照の整合性，ストアドプロシージャ，トリガー

すべてのリレーショナルデータベースは，アプリケーションがデータを保存および操作するための何らかの機能を共通して持っています。それに加えて，多くの場合，データベースのテーブル間に一貫した関係を持たせるために有効な，データベース特有のその他の機能も備えています。次の機能があります。

- **参照の整合性：**参照の整合性は，テーブル間のマスター / 詳細関係が壊れないように保つメカニズムです。ユーザーがマスターテーブル内の項目を削除しようとしたときに，それが詳細レコードから参照されている場合は，参照の整合性の規則によって削除が阻止されるか，またはそれを参照している詳細レコードも自動的に削除されます。

- **ストアドプロシージャ：**ストアドプロシージャは，いくつかの SQL 文のセットに名前を付けて SQL サーバーに保存したものです。ストアドプロシージャは普通，サーバーに対してデータベース関連の一般的なタスクを実行します。レコードの集合（データセット）を返す場合もあります。
- **トリガー：**トリガーは，特定のコマンドに応答して自動的に実行される SQL 文のセットです。

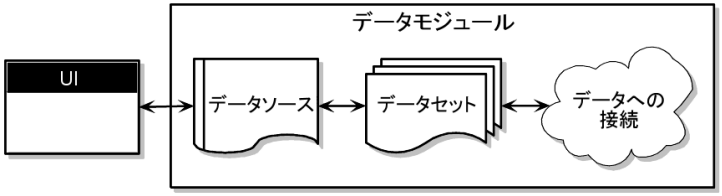
データベースアーキテクチャ

データベースアプリケーションは，ユーザーインターフェース要素，データベース情報を表すコンポーネント（データセット），およびこれらを相互に，またデータベース情報のソースに接続するコンポーネントによって構築されます。これらの部品をどう組み立てるかで，データベースアプリケーションのアーキテクチャが決まります。

一般的な構造

データベースアプリケーション内のコンポーネントを組み合わせる方法は数多くありますが，そのほとんどは，図 18.1 に示した一般的な方式に従っています。

図 18.1　一般的なデータベースアーキテクチャ



ユーザーインターフェースのフォーム

ユーザーインターフェースを，アプリケーションの他の部分からまったく独立したフォームにまとめるのは良い考えです。これにはいくつかの長所があります。ユーザーインターフェースを，データベース情報そのものを表すコンポーネントから独立させれば，より大きな設計上の柔軟性が得られます。データベース情報を管理する方法を変更しても，ユーザーインターフェースを書き直す必要はなく，ユーザーインターフェースを変更しても，アプリケーションのうちのデータベースを扱う部分を変更する必要はありません。加えて，このように独立させれば，複数のアプリケーションが共有する共通の形式を開発することができるので，一貫したユーザーインターフェースを提供できます。適切に設計されたフォームへのリンクをオブジェクトリポジトリに格納することによって，新規プロジェクトごとに最初から開発を始めるのではなく，既存の土台に基づいて開発することが可能になります。フォームを共有することで，アプリケーションインターフェースの社内標準を開発することもできます。ユーザーインターフェースの作成と使用についての詳細は，18-15 ページの「ユーザーインターフェースの設計」を参照してください。

データモジュール

ユーザーインターフェースを独自のフォームとして独立させたなら，データモジュールを使って，データベース情報を表すコンポーネント（データセット），およびこれらのデータセットをアプリ

ケーションの他の部分と接続するコンポーネントを格納することができます。ユーザーインターフェースのフォームと同様に，データモジュールもオブジェクトリポジトリで共有すれば，アプリケーション間で再利用し，共有することができます。

データソース

データモジュールの最初の項目はデータソースです。データソースは，ユーザーインターフェースと，データベースからの情報を表すデータセットとの間の仲介をします。フォーム上の複数のデータベース対応コントロールで 1 つのデータソースを共有できます。この場合，ユーザーがレコードをスクロールすると，各コントロールの表示が同期し，現在のレコードの項目の対応する値が各コントロールに表示されます。

データセット

データベースアプリケーションの中心は，データセットです。このコンポーネントは，基になるデータベースのレコードのセットを表します。レコードとは，単一のデータベーステーブルから取得したデータの場合もあれば，テーブルの一部を成すフィールドやレコードから取得したデータや，複数のテーブルを単一ビューに結合したテーブルから取得した情報の場合もあります。データセットを使用すると，データベースの物理テーブルが再構築されてもアプリケーションロジックは直接影響を受けずにすみます。基になるデータベースを変更した場合，データセットコンポーネントがその中のデータを指定する方法は変更する必要があるかもしれませんが，アプリケーションのそれ以外の部分は変更しなくても機能します。データセットの一般的なプロパティとメソッドについての詳細は，第 22 章「データセットについて」を参照してください。

データ接続

異なる種類のデータセットは，異なったメカニズムを使って，基になるデータベース情報に接続します。これらのメカニズムの違いが，構築するデータベースアプリケーションのアーキテクチャの相違点の大部分を占めます。データに接続する方法としては，以下のような 4 種類の基本的なメカニズムがあります。

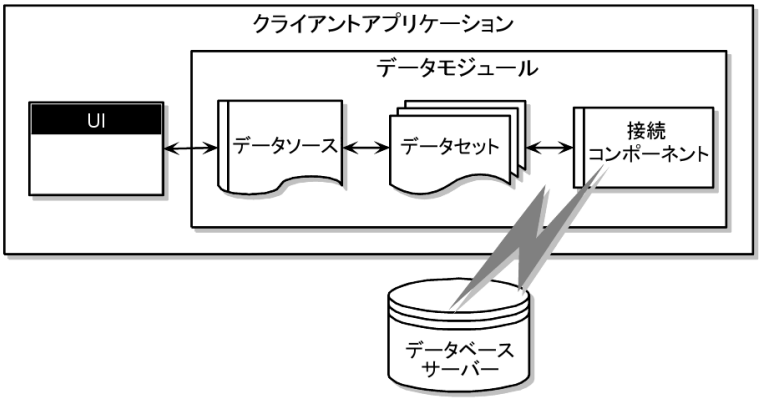
- データベースサーバーへ直接接続する。ほとんどのデータセットは，TCustomConnection の下位オブジェクトを使って，データベースサーバーへの接続を表します。
- ディスク上の専用ファイルを使用する。クライアントデータセットは，ディスク上の専用ファイルを扱う機能をサポートしています。クライアントデータセット自体がファイルの読み書きの方法を知っているので，専用ファイルを扱う場合には，独立した接続コンポーネントは必要ありません。
- ほかのデータセットに接続する。クライアントデータセットは，ほかのデータセットが提供するデータを扱うことができます。TDataSetProvider コンポーネントは，クライアントデータセットと，そのソースデータセットとの間の仲介として働きます。このデータセットプロバイダは，クライアントデータセットと同じデータモジュール内に入れることも，別のマシン上で動作するアプリケーションサーバーの一部とすることもできます。プロバイダがアプリケーションサーバーの一部である場合には，アプリケーションサーバーへの接続を表すために，TcustomConnection の特別な下位オブジェクトも必要となります。
- RDS DataSpace オブジェクトからデータを取得する。ADO データセットは，TRDSConnection コンポーネントを使って，ADO ベースのアプリケーションサーバーを使用して構築された，多層データベースアプリケーションのデータをマーシャルすることができます。

これらのメカニズムは，単一のアプリケーション内で組み合わせることもできます。

データベースサーバーへ直接接続する

最も一般的なデータベースアーキテクチャは，データセットが接続コンポーネントを使用して，データベースサーバへの接続を確立するというものです。その後，データセットは，サーバーからデータを直接取得し，また，サーバーに編集結果を直接登録します。これについては，図 18.2 で示します。

図 18.2　データベースサーバーへ直接接続する



それぞれの種類のデータセットは，単一のデータアクセスメカニズムを表わす，それ独自の接続コンポーネントを使用します。

- データセットが TTable，TQuery，または TStoredProc のような BDE データセットである場合，接続コンポーネントは TDataBase オブジェクト。データセットをデータベースコンポーネントに接続するには，その Database プロパティを設定します。BDE データセットを使用する場合には，データベースコンポーネントを明示的に追加する必要はありません。データセットの DatabaseName プロパティを設定すれば，実行時にデータベースコンポーネントが自動的に作成されます。
- データセットが TADODataSet，TADOTable，TADOQuery，または TADOStoredProc のような ADO データセットである場合，接続コンポーネントは TADOConnection オブジェクト。データセットを ADO 接続コンポーネントに接続するには，その ADOConnection プロパティを設定します。BDE データセットの場合と同様に，接続コンポーネントを明示的に追加する必要はありません。その代わり，データセットの ConnectionString プロパティを設定してください。
- データセットが，TSQLDataSet，TSQLTable，TSQLQuery，または TSQLStoredProc のような dbExpress データセットである場合，接続コンポーネントは TSQLConnection オブジェクト。データセットを SQL 接続コンポーネントに接続するには，その SQLConnection プロパティを設定します。dbExpress データセットを使用する場合，接続コンポーネントは明示的に追加しなければなりません。dbExpress データセットとほかのデータセットの間の別な相違点は，dbExpress データセットが常に読み出し専用，かつ単方向であるということです。これは，レコード間の移動は順方向に 1 つずつしか行えないこと，また，編集をサポートするデータセットメソッドは使用できないことを意味しています。

- データセットが TIBDataSet，TIBTable，TIBQuery，または TIBStoredProc のような InterBase Express データセットである場合，接続コンポーネントは TIBDatabase オブジェクト。データセットを IB データベースコンポーネントに接続するには，その Database プロパティを設定します。dbExpress データセットの場合と同様に，接続コンポーネントは明示的に追加しなければなりません。

上記のコンポーネントに加えて，TBDEClientDataSet，TSQLClientDataSet，または TIBClientDataSet のような特別なクライアントデータセットを，データベース接続コンポーネントとともに使うことができます。これらのクライアントデータセットを使用する場合には，DBConnection プロパティの値として，適切な種類の接続コンポーネントを指定します。

それぞれの種類のデータセットは異なる接続コンポーネントを使用しますが，それらすべては，タスクの多くを同じように実行し，プロパティ，メソッド，およびイベントの多くを共通して持っています。さまざまなデータベース接続コンポーネントに共通する機能については，第 21 章「データベースへの接続」を参照してください。

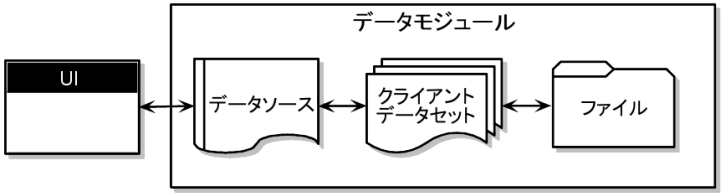
このアーキテクチャは，データベースサーバーがローカルデータベースかリモートデータベースサーバーかに応じて，単一層アプリケーションか 2 層アプリケーションのいずれかになります。データベース情報を操作するロジックは，ユーザーインターフェースを実装しているのと同じアプリケーション内にありますが，データモジュール内に分離されています。

メモ 2 層アプリケーションを作成するために必要な接続コンポーネントやドライバは，C++Builder のすべてのバージョンで利用できるわけではありません。

ディスク上の専用ファイルを使用する

最も単純な形式のデータベースアプリケーションでは，データベースサーバーをまったく使いません。そのかわり，クライアントデータセットが持つ，データをファイルに保存し，ファイルからデータを読み込む機能である MyBase を使用します。このアーキテクチャを図 18.3 に示します。

図 18.3　ファイルベースのデータベースアプリケーション



このファイルベースのアプローチを使用する場合，アプリケーションは，クライアントデータセットの SaveToFile メソッドを使って，変更をディスクに書き込みます。SaveToFile がとるパラメータは 1 つで，テーブルを保存するために作成（または上書き）されるファイルの名前です。以前に SaveToFile メソッドで書き込まれたテーブルを読むには，LoadFromFile メソッドを使います。LoadFromFile が受け取るパラメータも 1 つで，そのテーブルを保存したファイルの名前です。

常に同じファイルから読み込んで同じファイルに保存する場合は，SaveToFile と LoadFromFile メソッドのかわりに FileName プロパティを使用できます。FileName が有効なファイル名に設定されていれば，クライアントデータセットが開かれるときに自動的にそのファイルからデータが読み込まれ，クライアントデータセットが閉じられるときに自動的にそのファイルにデータが保存されます。

この単純なファイルベースのアーキテクチャは，単一層アプリケーションです。データベース情報を操作するロジックは，ユーザーインターフェースを実装しているのと同じアプリケーション内にありますが，データモジュール内に分離されています。

ファイルベースのアプローチには，単純さという利点があります。データベースサーバーをインストール，設定，または配布する必要はありません（ただし，クライアントデータセットには midas.dll が必要です）。サイトライセンスやデータベース管理の必要はありません。

さらに，C++Builder のいくつかのバージョンでは，任意の XML ドキュメントと，クライアントデータセットが使用するデータパケットの間で変換が行えます。したがって，ファイルベースのアプローチは，XML ドキュメントを扱う場合でも，専用のデータセットを扱う場合と同様に用いることができます。XML ドキュメントと，クライアントデータセットデータパケットの間の変換についての詳細は，第 30 章「データベースアプリケーションでの XML の使用」を参照してください。

ファイルベースのアプローチでは，複数ユーザーのサポートは提供しません。データセットは，完全にそのアプリケーション専用になります。データはディスクのファイルに保存され，後で読み込まれますが，複数のユーザーが互いにデータファイルを上書きするのを防ぐ組み込みの保護機能はありません。

ディスク上に格納されたデータを持つクライアントデータセットの使用方法についての詳細は，27-32 ページの「クライアントデータセットでファイルデータを使用する」を参照してください。

ほかのデータセットに接続する

データベースサーバーに接続するために BDE または dbExpress を使用する，特別なクライアントデータセットが存在します。これらの特別なクライアントデータセットは実際には合成コンポーネントであり，そこには，データに内部的にアクセスするもう 1 つのデータセットと，ソースデータセットからのデータをパッケージ化したりデータベースサーバーに更新を適用する内部的なプロバイダが含まれます。これらの合成コンポーネントは，付加的なオーバーヘッドを必要としますが，以下のような利点も持っています。

- クライアントデータセットは，キャッシュされた更新を扱う上での最も堅牢な方法を提供するデフォルトでは，ほかの種類のデータセットは，編集結果をデータベースサーバに直接登録します。更新をローカルにキャッシュし，それらすべてを後ほど単一のトランザクションで適用するデータセットを使用すれば，ネットワークトラフィックを減らすことができます。更新をキャッシュするクライアントデータセットを使用することの利点についての詳細は，27-15 ページの「クライアントデータセットをキャッシュアップデートに使用する」を参照してください。
- データセットが読み出し専用であれば，クライアントデータセットは，編集結果をデータベースサーバーに直接適用できる。dbExpress を使用する場合，これはデータセット内のデータを編集する唯一の方法です（dbExpress を使用する場合，データ内を自由に移動する唯一の方法でもあります）。dbExpress を使用しない場合でも，いくつかのクエリーと，すべてのストアドプロシージャの結果は，読み出し専用です。クライアントデータセットの使用は，そのようなデータを編集可能にする標準の方法を提供します。
- クライアントデータセットは，ディスク上の専用ファイルを直接扱うことができるので，クライアントデータセットの使用とファイルベースのモデルとを組み合わせれば，柔軟な「ブリーフ

ケース」アプリケーションが可能になります。ブリーフケースモデルの詳細については，18-14 ページの「複合のアプローチ」を参照してください。

特別なクライアントデータセットに加えて，汎用のクライアントデータセット（TClientDataSet）もあります。これは内部データセットとデータセットプロバイダを含んでいません。TClientDataSet は組み込みのデータベースアクセスメカニズムをを持っていませんが，外部の別のデータセットに接続して，データを取得し，更新を送ることができます。このアプローチは多少複雑ですが，以下のように，これが望ましい場合もあります。

- ソースデータセットとデータセットプロバイダが外部にあるので，データの取得方法と更新の適用方法が制御しやすくなる。たとえば，プロバイダコンポーネントは，特別なクライアントデータセットを使用してデータにアクセスする場合には利用できないような，多数のイベントを表示します。
- ソースデータセットが外部にあれば，別のデータセットとのマスター / 詳細関係の中にそれをリンクすることができる。外部プロバイダは自動的に，この配置を，ネストされた詳細を持つ単一のデータセットに変換します。ソースデータセットが内部にあると，ネストされた詳細セットをこのような方法で作成することはできません。
- クライアントデータセットの外部データセットへの接続は，容易に多層にスケールアップできるアーキテクチャである。開発プロセスは層の数の増加とともに複雑かつ高価になるので，アプリケーションの開発を，単一層または 2 層アプリケーションから始めたいと思う場合もあるでしょう。後でデータ量，ユーザー数，およびそのデータにアクセスするアプリケーション数が増えてきたら，多層アーキテクチャにスケールアップする必要があります。最終的には多層アーキテクチャを使用することになると思うなら，クライアントデータセットを外部ソースデータセットとともも使用することから始めるのは有益でしょう。こうすれば，アプリケーションが成長してもコードを再利用できるので，中間層にデータアクセスと操作ロジックを移動させるときでも，開発投資を保護することができます。
- TClientDataSet は，任意のソースデータセットにリンクできる。これは，対応する特別のクライアントデータセットのないカスタムデータセット（サードパーティコンポーネント）を使用してもよいことを意味します。C++Builder のバージョンの中には，別のデータセットではなく，XML ドキュメントにクライアントデータセットを接続する，特別のプロバイダコンポーネントを持つものもあります（これは，XML プロバイダがデータセットではなく XML ドキュメントを使用することを除けば，別の（ソース）データセットにクライアントデータセットを接続するのと同じ方法で動作します。XML プロバイダについての詳細は，30-8 ページの「XML ドキュメントをプロバイダのソースとして使う」を参照してください)。

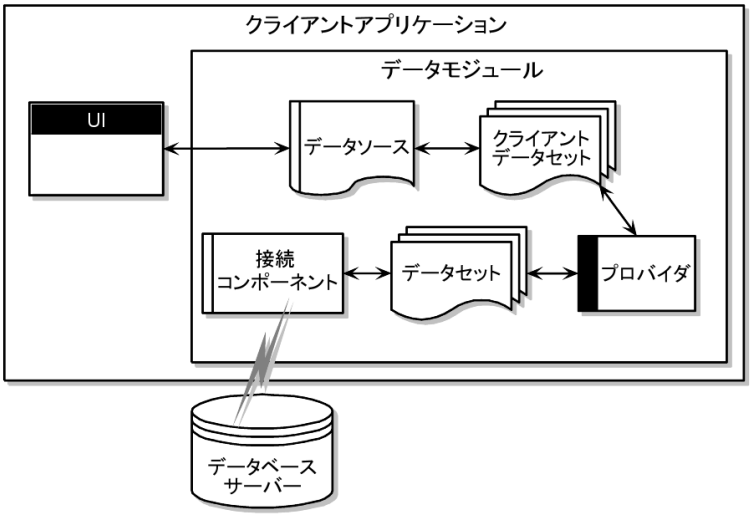
クライアントデータセットを外部データセットに接続するアーキテクチャには，以下の 2 つのバージョンがあります。

- 同じアプリケーションで別のデータセットにクライアントデータセットを接続する
- 多層アーキテクチャを使う

同じアプリケーションで別のデータセットにクライアントデータセットを接続する

プロバイダコンポーネントを使えば，TClientDataSet を別の（ソース）データセットに接続できます。プロバイダは，データベース情報を（クライアントデータセットが使用できる）データパケットの形にパッケージ化し，（クライアントデータセットが作成した）デルタパケットで受信した更新をデータベースサーバーに適用します。このアーキテクチャを図 18.4 に示します。

図 18.4　クライアントデータセットと別のデータセットを組み合わせたアーキテクチャ



このアーキテクチャは，データベースサーバーがローカルデータベースかリモートデータベースサーバーかに応じて，単一層アプリケーションか 2 層アプリケーションのいずれかになります。データベース情報を操作するロジックは，ユーザーインターフェースを実装しているのと同じアプリケーション内にありますが，データモジュール内に分離されています。

クライアントデータセットをプロバイダにリンクするには，その ProviderName プロパティを，プロバイダコンポーネントの名前に設定します。プロバイダは，クライアントデータセットと同じデータモジュールになければなりません。プロバイダをソースデータセットにリンクするには，その DataSet プロパティを設定します。

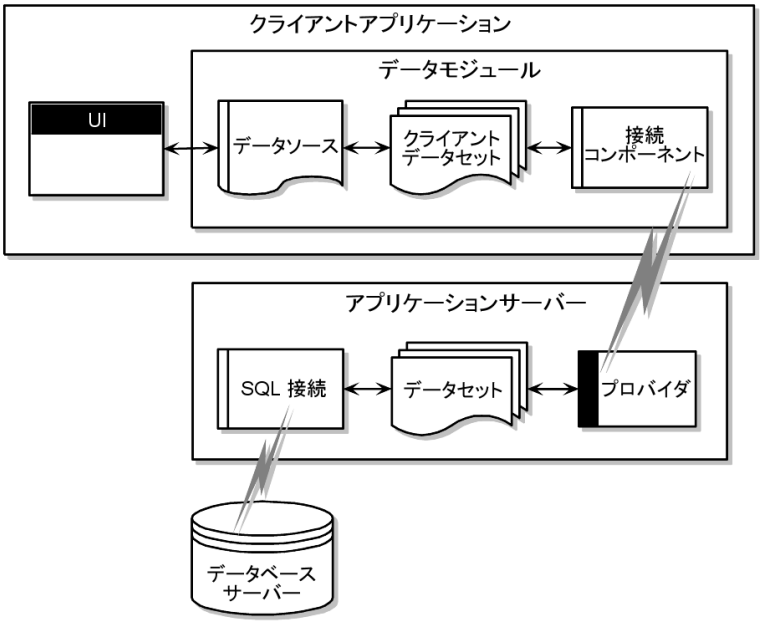
いったんクライアントデータセットをプロバイダにリンクし，プロバイダをソースデータセットにリンクすれば，これらのコンポーネントは自動的に，データベースレコードの取得，表示，および移動に必要な詳細をすべて処理します（ソースデータセットがデータベースに接続されている場合）。編集結果をデータベースに適用するために必要なのは，クライアントデータセットの ApplyUpdates メソッドを呼び出すことだけです。

クライアントデータセットをプロバイダとともに使用する方法の詳細については，27-23 ページの「クライアントデータセットでデータプロバイダを使用する」を参照してください。

多層アーキテクチャを使う

データベース情報に，いくつかのテーブル間の複雑な関係があるときや，クライアントの数が増えるときは，多層アプリケーションの使用が適しています。多層アプリケーションの場合，クライアントアプリケーションとデータベースサーバーの間に中間層があります。このアーキテクチャを図 18.5 に示します。

図 18.5　多層データベースのアーキテクチャ



前の図は，3 層アプリケーションを表しています。データベース情報を操作するロジックは，別のシステム，つまり別の層にあります。中間層には，データベース相互の影響を管理するロジックが集中しており，データ関係の中央制御を実現します。これにより，一貫したデータロジックを保ちながら，異なるクライアントアプリケーションによる同じデータの使用が可能になります。また，多層アプリケーションは，処理の大部分を中間層で負担するので，より小規模なクライアントアプリケーションを可能にします。これらの小さいアプリケーションは，インストール，環境設定，および保守が簡単です。多層アプリケーションは，データ処理をいくつかのシステムに分散させることによってパフォーマンスを向上させることもできます。

多層アーキテクチャは，前のモデルと非常によく似ています。主な相違は，データベースサーバーへの接続を行うソースデータセットと，ソースデータセットとクライアントデータセットとの間で仲介を行うプロバイダが，両方とも，独立したアプリケーションに移っているという点にあります。この独立したアプリケーションは，アプリケーションサーバー（または「リモートデータブローカ」）と呼ばれます。

プロバイダが独立したアプリケーションに移っているため，クライアントデータセットは，単に ProviderName プロパティを設定しただけでは，ソースデータセットに接続することはできません。それに加えて，アプリケーションサーバーの位置を特定し，それと接続するための，ある種の接続コンポーネントを使わなければなりません。

アプリケーションサーバーにクライアントデータセットを接続することができる，いくつかの種類の接続コンポーネントがあります。それらはすべて TCustomRemoteServer の子孫で，主な相違点は使用する通信プロトコルです（TCP/IP，HTTP，DCOM，または SOAP）。クライアントデータセットを接続コンポーネントにリンクするには，RemoteServer プロパティを設定します。

接続コンポーネントは，アプリケーションサーバーへの接続を確立して，ProviderName プロパティで指定されたプロバイダを呼び出すためにクライアントデータセットが使うインターフェースを返します。クライアントデータセットは，アプリケーションサーバーを呼び出すたびに ProviderName の値を渡し，アプリケーションサーバーは，呼び出しをプロバイダに転送します。

クライアントデータセットをアプリケーションサーバーに接続する方法については，第 29 章「多層アプリケーションの作成」を参照してください。

複合のアプローチ

前のトピックでは，データベースアプリケーションを作成する際に使用することができる，いくつかのアーキテクチャについて説明しました。とはいえ，単一のアプリケーションで 2 つ以上のアーキテクチャを組み合わせてはならないわけではありません。実際，ある組み合わせは非常に強力なものとなり得ます。

たとえば，18-9 ページの「ディスク上の専用ファイルを使用する」で説明したディスクベースのアーキテクチャを，18-12 ページの「同じアプリケーションで別のデータセットにクライアントデータセットを接続する」や 18-13 ページの「多層アーキテクチャを使う」で説明した，他のアプローチと組み合わせることができます。これらを組み合わせるのは容易です。モデルはすべて，ユーザーインターフェースに現われるデータを表わすためにクライアントデータセットを使用しているからです。これは，ブリーフケースモデル（または非接続モデルやモバイルコンピューティング）と呼ばれるものです。

ブリーフケースモデルは，次のような状況で役に立ちます。社内のデータベースに顧客名簿データがあって，それを営業部員が出先から使ったり更新したりする場合を考えてみましょう。社内では，営業部員は情報をデータベースからダウンロードします。その後，営業部員は情報をノート型コンピュータに入れて，さまざまな場所に移動します。ときには，既存または新規の顧客サイトでレコードを更新することもあります。営業部員が会社に戻ったときには，データ変更を会社のデータベースにアップロードしてほかの社員も使用できるようします。

社内では，ブリーフケースモデルアプリケーションのクライアントデータセットは，データをプロバイダから取得します。クライアントデータセットはデータベースサーバーに接続されており，プロバイダを通じて，サーバーのデータを取得し，更新をサーバーに送ることができます。プロバイダとの接続を終える前に，クライアントデータセットは，その情報のスナップショットを，ディスク上のファイルに保存します。社外では，クライアントデータセットはデータをファイルから読み込み，変更をファイルに保存します。最後に，会社に戻ったとき，クライアントデータセットはプロバイダに再接続して，更新をデータベースサーバーに適用し，データのスナップショットを最新のものにします。

ユーザーインターフェースの設計

コンポーネントパレットの［Data Controls］ページは，データベースレコード内の項目からのデータを表すデータベース対応コントロールのセットを提供し，ユーザーがそのデータを編集したり変更内容をデータベースに登録したりできるようにします。データベース対応コントロールを使用すると，情報をユーザーから見たりアクセスしたりできるようにデータベースアプリケーションのユーザーインターフェース（UI）を構築できます。データベース対応コントロールについての詳細は，第 19 章「データコントロールの使い方」を参照してください。

以下のように，基本的なデータコントロールに加えて，ほかのエレメントをユーザーインターフェースに追加したい場合もあるでしょう。

- アプリケーションにデータベースに含まれていたデータを分析させたい場合。データを分析するアプリケーションは，データベース中のデータを表示するだけではなく，ユーザーがそのデータのインパクトを把握するのに役立つ形式で，情報を要約します。
- ユーザーインターフェースに表示された情報のハードコピーを提供するレポートを印刷したいと思う場合
- Web ブラウザから表示することができるユーザーインターフェースを作成したい場合。単純な Web ベースのアプリケーションについては，33-17 ページの「レスポンスでのデータベース情報の使い方」で説明されています。加えて，29-30 ページの「Web ベースのクライアントアプリケーションの作成」で説明されているように，Web ベースのアプローチを，多層アーキテクチャと組み合わせることもできます。

データの分析

データベースアプリケーションによっては，データベース情報をユーザーに直接には表示しないことがあります。かわりに，ユーザーがデータから結論を引き出しやすいように，データベースの情報を分析および集計します。

コンポーネントパレットの［Data Controls］ページにある TDBChart コンポーネントを使うと，ユーザーがデータベース情報の重要性を即座に把握できるように，データベース情報をグラフ化して示せます。

加えて，C++Builder のバージョンによってはコンポーネントパレットに［Decision Cube］ページも含まれています。このページには，意思決定支援アプリケーションを構築するときにデータの分析やクロス集計に使う 6 個のコンポーネントがあります。［Decision Cube］ページのコンポーネントの使い方についての詳細は，第 20 章「デシジョンコンポーネントの使い方」を参照してください。

さまざまな分類のしかたによるデータ集計を表示するために独自のコンポーネントを構築したい場合は，保守される集合体をクライアントデータセットと一緒に使用できます。保守される集合体についての詳細は，27-11 ページの「保守される集合体の使用」を参照してください。

レポートの作成

アプリケーション内のデータセットのデータベース情報をユーザーが印刷できるようにしたい場合，コンポーネントパレットの［QReport］ページにあるレポートコンポーネントを使用できます。これらのコンポーネントを使用すると，データベーステーブルの情報を示したり集計するためのバンド付きレポートを視覚的に構築できます。グループヘッダーまたはフッターに集計演算子を追加することにより，グループ化条件に基づいてデータを分析できます。

アプリケーションのレポートを開始するには，［新規作成］ダイアログから［QuickReport］アイコンを選択します。メインメニューから［ファイル｜新規作成｜その他］を選択し，［業務］ページを選択します。［QuickReport ウィザード］アイコンをダブルクリックしてウィザードを起動します。

メモ ［QReport］ページのコンポーネントの利用方法については，C++Builder に付属する QuickReport のサンプルを参照してください。

第19章

データコントロールの使い方

コントロールパレットの［Data Controls］ページは，データベースレコード内の項目からのデータを表すデータベース対応コントロールのセットを提供し，データセットによって許可されている場合には，ユーザーがそのデータを編集したり変更内容をデータベースに登録したりできるようにします。データベースアプリケーションのフォームにデータコントロールを配置すると，情報をユーザーから見たりアクセスしたりできるようにユーザーインターフェース（UI）を構築できます。

ユーザーインターフェースにどのデータ対応コントロールを追加するかは，次のような，いくつかの要素によって決まります。

- 表示するデータの種類。プレーンテキストを表示して編集するコントロール，書式付きテキストを扱うコントロール，グラフィックスを扱うコントロール，マルチメディア要素などの中から選択できます。さまざまな種類の情報を表示するコントロールについては，19-7 ページの「単一レコードを表示する」で説明します。
- 情報を整理する方法。単一のレコードの情報を画面に表示することもできますし，グリッドを使って，複数のレコードの情報をリスト表示することもできます。19-7 ページの「データの整理方法の選択」では，いくつかの方法について説明します。
- そのコントロールにデータを提供するデータセットの種類。基となるデータセットでの制限を反映するようなコントロールを使いたいと思うことでしょう。たとえば，単方向データセットの場合は，一度に 1 つのレコードしか提供しないので，グリッドは使わないはずです。
- ユーザーがデータセットのレコードを操作し，データの追加や編集を行うことを許可するか。許可する場合にはその方法。操作や編集のために，自分独自のコントロールやメカニズムを追加したい場合もあり，データナビゲータのような組み込みのコントロールを用いたい場合もあるでしょう。データナビゲータの使い方については，19-28 ページの「レコード間の移動と操作」.を参照してください。

メモ デシジョンサポート用のもっと複雑なデータベース対応コントロールについては，第 20 章「デシジョンコンポーネントの使い方」で説明しています。

インターフェースにどのデータ対応コントロールを追加した場合でも，いくつかの共通した機能があります。これらについては後述します。

データコントロールに共通した機能の使い方

以下の操作は，ほとんどのデータコントロールに共通です。

- データセットへのデータコントロールの関連付け
- データの編集と更新
- データ表示の無効化と有効化
- データ表示の更新
- マウス，キーボード，タイマーのイベントの有効化

データコントロールは，データセット内の現在のレコードに関連付けられたデータ項目の表示と編集のために使われます。表 19.1 にコンポーネントパレットの［Data Controls］ページにあるデータコントロールの概要を示します。

表 19.1　データコントロール

データコントロール	説明
TDBGrid	データソースの情報を表計算ワークシートのようなグリッドに表示する。グリッドの列は，基礎となるテーブルまたは問い合わせのデータセットの列に対応している。グリッドの行はレコードに対応している
TDBNavigator	データセットのデータレコード間の移動，レコードの更新，登録，削除，編集の取り消し，データ表示の更新用に設定可能なボタンセットを提供する
TDBText	項目のデータをラベルとして表示する
TDBEdit	項目のデータを編集ボックスに表示する
TDBMemo	メモ型項目または BLOB 項目のデータを複数行の編集ボックスに表示する
TDBImage	データ項目のビットマップ，アイコン，メタファイルをグラフィックボックスに表示する
TDBListBox	現在のデータレコードの項目を更新するために選択可能な項目名のリストを表示する
TDBComboBox	項目を更新するために選択可能な項目名のリストを表示する。また，標準のデータベース対応編集ボックスと同じようにテキストの直接入力も可能にする
TDBCheckBox	論理型項目の値を示すチェックボックスを表示する
TDBRadioGroup	項目に対して同時に複数は選択できないオプションセットを表示する
TDBLookupListBox	項目の値に基づいてほかのデータセットを参照し，項目名のリストを表示する
TDBLookupComboBox	項目の値に基づいてほかのデータセットを参照し，項目名のリストを表示する。また，標準のデータベース対応編集ボックスと同じようにテキストの直接入力も可能にする
TDBCtrlGrid	設定可能な繰り返しのデータベース対応コントロールセットをグリッド内に表示する
TDBRichEdit	メモ型項目のデータを，書式付きで複数行の編集ボックスに表示する

データコントロールは設計時にもデータベース対応しています。設計時にデータコントロールをアクティブなデータセットと関連付けると，そのコントロールには即座に実際のデータが表示されます。項目エディタを使うと，アプリケーションをコンパイルして実行しなくても，設計時にデータセットをスクロールしてアプリケーションがデータを正しく表示することを確認できます。項目エディタについての詳細は，23-3 ページの「持続的項目の作成」を参照してください。

実行時には，データコントロールはデータを表示し，アプリケーションとそのコントロールとデータセットのすべてで許可されていれば，ユーザーはそのコントロールを通してデータを編集できます。

データセットへのデータコントロールの関連付け

データコントロールはデータソースを使ってデータベースに接続します。データソースコンポーネント（TDataSource）は，データの入ったデータセットとコントロールとの仲介の役目を果たします。データの表示や操作を行うには，すべてのデータベース対応コントロールをデータソースコンポーネントに関連付けなければなりません。同様に，フォーム上のデータベース対応コントロールの中でデータの表示や操作を行うには，すべてのデータセットをデータソースコンポーネントに関連付けなければなりません。

メモ データソースコンポーネントは，ネストしていないデータセットをマスター / 詳細関係でリンクする場合にも必要となります。

データコントロールをデータセットと関連付ける手順は次のとおりです。

1. データセットをデータモジュールまたはフォームに配置し，そのプロパティを適切に設定します。
2. データソースを同じデータモジュールまたはフォームに配置します。オブジェクトインスペクタを使って，その DataSet プロパティを，手順 1 で配置したデータセットに設定します。
3. コンポーネントパレットの［Data Access］ページにあるデータセットコンポーネントとデータソースコンポーネントをフォームに配置し関連付けを行います。
4. オブジェクトインスペクタを使って，コントロールの DataSource プロパティを，手順 2 で配置したデータソースコンポーネントに設定します。
5. コントロールの DataField プロパティを表示項目の名前に設定するか，DataField プロパティのドロップダウンリストの項目名を選択します。この手順は，TDBGrid，TDBCtrlGrid，および TDBNavigator では不要です。これらのコントロールはデータセット内の利用可能なすべての項目にアクセスするからです。
6. コントロールにデータを表示するには，データセットの Active プロパティを **true** に設定します。

実行時に関連付けられたデータセットを変更する

前の例では，データソースは設計時に DataSet プロパティを設定することにより，データセットと関連付けられました。実行時には，必要に応じてデータソースコンポーネント用のデータセットを切り替えられます。たとえば次の例は，CustSource データソースコンポーネント用のデータセットを，Customers および Orders という名前のデータセットコンポーネントの間で切り替えます。

```
if (CustSource->DataSet == Customers)
  CustSource->DataSet = Orders;
else
  CustSource->DataSet = Customers;
```

DataSet プロパティを別のフォーム上のデータセットに対して設定し，2 つのフォーム上のデータコントロールの同期をとることもできます。例を示します。

```
void __fastcall TForm2::FormCreate(TObject *Sender)
{
  DataSource1->DataSet = Form1->Table1;
}
```

データソースの有効化と無効化

データソースには，データセットに接続するかどうかを決める Enabled プロパティがあります。Enabled が **true** の場合，そのデータソースはデータセットに接続されます。

Enabled を **false** に設定すれば，そのデータソースを一時的にデータセットから切断できます。Enabled が **false** の場合，そのデータソースコンポーネントに結び付いたすべてのデータコントロールは空白になり，Enabled が **true** に設定されるまで非アクティブになります。ただし，データセットへのアクセスは，データセットコンポーネントの DisableControls メソッドと EnableControls メソッドを使って制御することをお勧めします。これらのメソッドは結び付けられたすべてのデータソースに影響を与えるからです。

データソースに仲介された変更に応答する

データソースは，データコントロールとそのデータセットの間のリンクを提供するので，両者の間のすべての通信を仲介します。通常，データベース対応コントロールは，データセットでの変更に自動的に応答します。しかし，ユーザーインターフェースでデータベース対応ではないコントロールを使っている場合には，データソースコンポーネントのイベントを使って，手動で同様の応答を行わせることができます。

OnDataChange イベントは，項目を編集したり，カーソルが新しいレコードに移ったりするなど，レコードのデータが変化するたびに発生します。このイベントは，どんな変更の場合でも発生するので，コントロールに，データセットの現在の項目値が確実に反映されるようにする上で役に立ちます。通常，OnDataChange イベントハンドラで，項目データを表示する，データベース対応でないコントロールの値を更新します。

OnUpdateData イベントは，現在のレコードのデータを登録しようとするたびに発生します。たとえば，Post が呼び出された後，データが実際に基になるデータベースサーバーかローカルキャッシュに登録される前に，OnUpdateData イベントが発生します。

OnStateChange イベントは，データセットの状態が変わるたびに発生します。このイベントが発生したなら，データセットの State プロパティを調べれば，データセットの現在の状態がわかります。

たとえば，次の OnStateChange イベントハンドラは，現在の状態に基づいてボタンやメニュー項目の使用可能と使用不可を切り替えます。

```
void __fastcall TForm1::DataSource1StateChange(TObject *Sender)
{
  CustTableActivateBtn->Enabled = (CustTable->State == dsInactive);
  CustTableEditBtn->Enabled = (CustTable->State == dsBrowse);
  CustTableCancelBtn->Enabled = (CustTable->State == dsInsert ||
                                 CustTable->State == dsEdit ||
                                 CustTable->State == dsSetKey);
  ...
}
```

メモ データセットの状態についての詳細は，22-3 ページの「データセットの状態の決定」を参照してください。

データの編集と更新

ナビゲータ以外のデータコントロールはすべて，データベース項目からのデータを表示します。また，対象となるデータセットで許されていれば，データコントロールを使ってデータを編集して更新することもできます。

メモ 単方向データセットでは，ユーザーがデータの編集や更新を行うことは，一切許可されません。

ユーザーの入力に対するコントロールでの編集の有効化

データセットのデータを編集できるようにするには，そのデータセットを dsEdit 状態にしなければなりません。データソースの AutoEdit プロパティが **true**（デフォルト）であれば，ユーザーがそのデータを編集しようとすると，データコントロールがタスクを制御して，データセットを dsEdit モードにします。

AutoEdit が **false** の場合には，データセットを編集モードにするための別のメカニズムを用意しなければなりません。そのようなメカニズムの 1 つとして，TDBNavigator コントロールを［編集］ボタンと組み合わせるというものがあります。ユーザーはそれによってデータセットを明示的に編集モードにすることができます。TDBNavigator についての詳細は，19-28 ページの「レコード間の移動と操作」を参照してください。または，データセットを編集モードにしたいときにデータセットの Edit メソッドを呼び出すコードを書くこともできます。

コントロールのデータの編集

データコントロールがその編集結果を関連するデータセットに登録できるのは，データセットの CanModify プロパティが **true** になっている場合だけです。単方向データセットの場合には，CanModify は常に **false** になります。クライアントデータセットには ReadOnly プロパティがあり，これを使えば，CanModify を **true** にするかどうかを指定できます。

メモ クライアントデータセットがデータを更新できるかどうかは，基になるデータベーステーブルが更新を許可しているかどうかで決まります。

データセットの CanModify プロパティが **true** になっている場合でも，コントロールが更新をデータベーステーブルに登録できるようにするためには，データセットをコントロールに接続しているデータソースの Enabled プロパティも **true** にしなければなりません。データソースの Enabled プロパティは，コントロールがそのデータセットの項目値を表示できるかどうか，またユーザーがその値を編集し登録できるかどうかを決めます。Enabled が **true**（デフォルト）の場合，コントロールは項目値を表示できます。

最後の点として，コントロールに表示されているデータをユーザーが編集できるかどうかを制御することも可能です。データコントロールの ReadOnly プロパティは，そのコントロールによって表示されたデータをユーザーが編集できるかどうかを決めます。**false** の場合（デフォルト），ユーザーはデータを編集できます。当然ながら，データセットの CanModify を **false** にした場合には，コントロールの ReadOnly プロパティが **true** になっていることを確認しておいてください。そうしておかないと，ユーザーは，基のデータベーステーブルのデータを変更できるものと誤って解釈するでしょう。

TDBGrid を除くすべてのデータコントロールでは，項目を変更した場合，【Tab】でコントロールから離れるときに基のデータセットに変更がコピーされます。【Tab】で項目から移動する前に【Esc】を押すと，変更が破棄され，項目の値は変更前の値に戻ります。

TDBGrid では，別のレコードに移動したときだけ変更が登録されます。別のレコードに移動する前に項目のレコードで【Esc】を押すと，レコードに対するすべての変更を取り消せます。

レコードを登録するときに，データセットに関連付けられたすべてのデータベース対応コントロールの状態の変化が調べられます。変更のあるデータを含む項目を更新するときに問題が発生すると，例外が生成されて，レコードへの変更は行われません。

メモ アプリケーションが更新をキャッシュする場合（たとえば、クライアントデータセットを使用する場合）には，変更はすべて内部キャッシュに登録されます。これらの変更は，データセットの ApplyUpdates メソッドを呼び出すまでは，基のデータベーステーブルには適用されません。

データ表示の無効化と有効化

アプリケーションがデータセットの繰り返し処理または検索をするときは，現在のレコードが変更されるたびにデータベース対応コントロールに表示されている値が更新されないように設定を一時的に変えるとよいでしょう。値が更新されないようにすると，繰り返し処理や検索の速度が上がり，目障りな画面のちらつきが起こりません。

DisableControls は，データセットにリンクされたすべてのデータベース対応コントロールの表示を無効にするデータセットのメソッドです。繰り返し処理や検索が終わったら，すぐにアプリケーションはデータセットの EnableControls メソッドを呼び出して，コントロールの表示を再び有効にしなければなりません。

通常は，繰り返し処理の前にコントロールを使用不可にします。繰り返し処理自体は，処理中に例外が生成されてもコントロールを再び使用可能にできるように，**try...__finally** 文の内部で発生させなければなりません。**__finally** 節は，EnableControls を呼び出していなければなりません。次のコードは，このようにして DisableControls と EnableControls を使う方法を示しています。

```
CustTable->DisableControls();
try
{
  // データセットのすべてのレコードを巡回する
  for (CustTable->First(); !CustTable->Eof; CustTable->Next())
  {
  // ここで各レコードを処理する
  ...
  }
}
__finally
{
  CustTable->EnableControls();
}
```

データ表示の更新

データセットの Refresh メソッドは，ローカルバッファの内容を消去して，開いているデータセット用にデータを取得し直します。アプリケーションで使っているデータにほかの複数のアプリケーションが同時にアクセスしてきたため，基礎となるデータが変更されていると考えられる場合に，この Refresh メソッドを使うとデータベース対応コントロールの表示を更新できます。キャッシュアップ

デートを使用している場合には，データセットを更新する前に，データセットが現在キャッシュしているアップデートを適用しなければなりません。

更新により，予期しない結果になることがあります。たとえば，ほかのアプリケーションで削除されたレコードをユーザーが表示している場合，アプリケーションが Refresh を呼び出すと，ただちにそのレコードが消去されます。初めにデータを取得した後で Refresh を呼び出す前に，ほかのユーザーがレコードを変更した場合も，データの表示状態が変化することがあります。

マウス，キーボード，タイマーのイベントの有効化

データコントロールの Enabled プロパティは，そのデータコントロールがマウス，キーボード，タイマーのイベントに応答して，データソースに情報を渡すかどうかを指定します。Enabled プロパティのデフォルトの設定は **true** です。

マウス，キーボード，タイマーのイベントがデータコントロールに届かないようにするには，データコントロールの Enabled プロパティを **false** に設定します。Enabled が **false** であれば，コントロールをそのデータセットに接続しているデータソースはデータコントロールから情報を受け取りません。データコントロールはデータを表示し続けますが，このデータコントロールに表示されるテキストは淡色表示になります。

データの整理方法の選択

データベースアプリケーションのユーザーインターフェースを構築するときには，情報の表示や情報を操作するコントロールを整理する方法に関して，いくつかの選択肢があります。

最初に下すべき決定の 1 つとして，一度に単一のレコードを表示するか，それとも複数のレコードを表示するか，という点があります。

加えて，レコードを操作するコントロールを追加したいことがあります。TDBNavigator コントロールは，必要となる機能の多くを組み込みでサポートしています。

単一レコードを表示する

多くのアプリケーションでは，一度に 1 レコードの情報を表示したいだけです。たとえば，受注登録アプリケーションで，現在ほかにどんな注文が記録されているか示さずに 1 つの注文の情報だけを表示することがあります。この情報は普通，注文データセットの中の 1 レコード分の情報です。

単一レコードを表示するアプリケーションは，一般に，すべてのデータベース情報がどれも同じものについてなので（前の例では同じ注文内容）読みやすくわかりやすいものです。ユーザーインターフェースのデータベース対応コントロールは，データベースレコードの 1 項目を表します。コンポーネントパレットの［Data Controls］ページでは，さまざまな種類の項目を表す幅広いコントロールの中から選択することができます。これらのコントロールは通常，コンポーネントパレット上の他のコントロールのデータベース対応バージョンになっています。たとえば TDBEdit コントロールは，テキスト文字列の表示と編集に使用する標準の TEdit コントロールのデータベース対応バージョンです。

どのコントロールを使うかは，項目のデータの種類（テキスト，書式付きテキスト，グラフィック，論理情報など）によって決まります。

データのラベル表示

TDBText は，コンポーネントパレットの［Standard］ページにある TLabel コンポーネントに似た読み出し専用のコントロールです。TDBText コントロールは，ユーザーがほかのコントロールに入力を行うためのフォームに，表示専用データを提供する場合に役立ちます。たとえば，顧客リストテーブルの項目に基づいて作成されたフォームを考えます。ユーザーがフォームに都道府県，市区町村，番地の情報を入力すると，動的参照によって自動的に別のテーブルから郵便番号項目を取得するとします。ここで，TDBText コンポーネントを郵便番号テーブルに結合すると，ユーザーが入力した住所に該当する郵便番号項目を表示できます。

TDBText は，データセットの現在のレコード内の指定された項目から，表示するテキストを取得します。TDBText はデータセットからテキストを取得するので，表示されるテキストは動的なテキストです。つまり，ユーザーがデータベーステーブル内を移動するにつれてテキストは変化します。したがって，TLabel とは異なり，設計時に TDBText の表示テキストを指定することはできません。

メモ フォームに TDBText コンポーネントを配置する場合は，その AutoSize プロパティを **true** に設定しておき（デフォルト），表示されるデータの幅に合わせてコントロールのサイズが自動的に変更されるようにします。AutoSize を **false** に設定した場合，コントロールが小さすぎると，データの一部が表示されなくなります。

編集ボックスでの項目の表示と編集

TDBEdit は，編集ボックスコンポーネントのデータベース対応バージョンです。TDBEdit は，リンク先のデータ項目の現在値を表示します。この値は，標準の編集ボックスの操作方法で編集できます。

たとえば CustomersSource が，CustomersTable という開いている TClientDataSet にリンクされたアクティブな TDataSource コンポーネントであるとします。ここで，TDBEdit コンポーネントをフォームに配置して，そのプロパティを以下のように設定します。

- DataSource：CustomersSource
- DataField：CustNo

データベース対応の編集ボックスコンポーネントは，設計時にも実行時にも，CustomersTable データセットの CustNo 列にある現在の行の値をただちに表示します。

メモコントロールでのテキストの表示と編集

TDBMemo は，標準の TMemo コンポーネントと似たデータベース対応コンポーネントですが，非常に長いテキストデータを表示できます。TDBMemo では複数行テキストが表示され，ユーザーによる複数行テキストの入力も可能になります。TDBMemo コントロールを使うと，長いテキスト項目や，BLOB 項目に含まれているテキストデータを表示できます。

デフォルトでは，TDBMemo ではユーザーがメモテキストを編集できます。編集できないようにするには，メモコントロールの ReadOnly プロパティを **true** に設定します。タブを表示してメモを入力できるようにするには，WantTabs プロパティを **true** に設定します。ユーザーがデータベースのメモに入力できる文字数を制限するには，MaxLength プロパティを使います。MaxLength のデフォルト値は 0 で，文字数についてはオペレーティングシステムが課す制限以外の制限がないことを意味します。

データベースのメモの外観とテキストの入力方法には，いくつかのプロパティが影響します。メモにスクロールバーを追加するには，ScrollBars プロパティを使います。ワードラップを抑止するには，WordWrap プロパティを **false** に設定します。Alignment プロパティは，コントロール内でのテキストの揃え方を決定します。選択肢は taLeftJustify（デフォルト），taCenter，および taRightJustify です。テキストのフォントを変更するには，Font プロパティを使います。

実行時にユーザーは，データベースのメモコントロールとの間でテキストの切り取り，コピー，貼り付けができます。これらの操作ができるプログラムを書くには，CutToClipboard，CopyToClipboard，および PasteFromClipboard メソッドを使います。

TDBMemo は大量のデータを表示できるため，実行時の表示に時間がかかることがあります。データレコードのスクロールにかかる時間を減らすため，TDBMemo には，アクセスされたデータを自動的に表示するかどうかを制御する AutoDisplay プロパティがあります。AutoDisplay を **false** に設定すると，TDBMemo は実際のデータではなく項目名を表示します。実際のデータを表示するには，コントロール内をダブルクリックします。

書式付きテキスト編集メモコントロールでのテキストの表示と編集

TDBRichEdit は標準の TRichEdit コンポーネントに似たデータベース対応コンポーネントで，BLOB（バイナリラージオブジェクト）項目に格納された書式付きテキストを表示できます。TDBRichEdit は，書式付きの複数行テキストを表示し，ユーザーによる書式付きの複数行テキストの入力も許可します。

メモ TDBRichEdit は書式付きテキストを入力して操作するためのプロパティとメソッドを提供しますが，書式を変更するためのツールやメニューなどは提供されません。アプリケーション自身で書式付きテキストのためのユーザーインターフェースを実装しなければなりません。

デフォルトでは，TDBRichEdit ではユーザーがメモテキストを編集できます。編集を防止するには，書式付きテキスト編集コントロールの ReadOnly プロパティを **true** に設定します。タブを表示してメモを入力できるようにするには，WantTabs プロパティを **true** に設定します。ユーザーがデータベースのメモに入力できる文字数を制限するには，MaxLength プロパティを使います。MaxLength のデフォルト値は 0 で，文字数についてはオペレーティングシステムが課す制限以外の制限がないことを意味します。

TDBRichEdit は大量のデータを表示できるため，実行時の表示に時間がかかることがあります。データレコードのスクロールにかかる時間を減らすため，TDBRichEdit には，アクセスされたデータを自動的に表示するかどうかを制御する AutoDisplay プロパティがあります。AutoDisplay を **false** に設定すると，TDBRichEdit は実際のデータではなく項目名を表示します。実際のデータを表示するには，コントロール内をダブルクリックします。

イメージコントロールでのグラフィック型項目の表示と編集

TDBImage は，BLOB 項目に含まれているビットマップグラフィックを表示するデータベース対応コンポーネントです。

デフォルトでは，TDBImage は，ユーザーが CutToClipboard，CopyToClipboard，および PasteFromClipboard メソッドを使ってクリップボードとの間で切り取りと貼り付けを行ってグラフィックイメージを編集することを可能にします。かわりに，イベントハンドラに結び付けられた独自の編集メソッドをこのコントロールに与えることもできます。

デフォルトでは，イメージコントロールはそのサイズに収まる分のグラフィックを表示します。はみ出す部分は表示しません。イメージコントロールのサイズを変更したときにその内部に収まるようグラフィックのサイズを調整するには，Stretch プロパティを **true** に設定します。

TDBImage は大量のデータを表示できるため，実行時の表示に時間がかかることがあります。データレコードのスクロールにかかる時間を減らすため，TDBImage には，アクセスされたデータを自動的に表示するかどうかを制御する AutoDisplay プロパティがあります。AutoDisplay を **false** に設定すると，TDBImage は実際のデータではなく項目名を表示します。実際のデータを表示するには，コントロール内をダブルクリックします。

リストボックスとコンボボックスでのデータの表示と編集

実行時に一連のデフォルトのデータ値を選択肢としてユーザーに提供する 4 つのデータコントロールがあります。これらは，標準のリストボックスコントロールとコンボボックスコントロールのデータベース対応バージョンです。

- TDBListBox は，スクロール可能な項目のリストを表示します。ユーザーはこのリストから項目を選択して，データ項目に入力できます。データベース対応リストボックスは，現在のレコードの項目にデフォルト値を表示しリストの対応する値を強調表示します。現在の行の項目値がリストにない場合，リストボックスで強調表示される値はありません。ユーザーがリストの項目を選択すると，対応するデータセットの項目値が変更されます。
- TDBComboBox コントロールは，データベース対応編集コントロールとドロップダウンリストの両方の機能を兼ね備えています。実行時に TDBComboBox コントロールが表示するドロップダウンリストで，ユーザーは定義済みの一連の値から選択できます。ユーザーはまったく別の値を入力することもできます。
- TDBLookupListBox は，表示項目名のリストを他のデータセットから参照することを除けば，TDBListBox と同様の動作をします。
- TDBLookupComboBox, は，表示項目名のリストを他のデータセットから参照することを除けば，TDBComboBox と同様の動作をします。

メモ 実行時にユーザーは，インクリメンタルサーチによってリストボックスの項目名を検索できます。このコントロールにフォーカスがあるときに，たとえば「ROB」と入力すると，「ROB」という文字列で始まる最初の項目名がリストボックスで選択されます。続けて「E」と入力すると，「Robert Johnson」のように，「ROBE」で始まる最初の項目名が選択されます。検索では，大文字と小文字は区別されません。キー入力とキー入力の間が 2 秒以上空いた場合，または【BS】と【Esc】を押すと，現在の検索文字列が取り消されます（ただし選択された項目はそのまま残ります）。

TDBListBox および TDBComboBox を使う

TDBListBox や TDBComboBox を使う場合には，設計時に文字列リストエディタを使って，表示する項目名のリストを作成します。文字列リストエディタを開くには，オブジェクトインスペクタで Items プロパティの省略記号ボタンをクリックします。それから，リストに表示する項目を入力します。実行時には，Items プロパティのメソッドを使用してその文字列リストを操作します。

TDBListBox や TDBComboBox コントロールを，DataField プロパティによって項目とリンクしている場合には，項目値がリスト内で選択状態になって表示されます。現在の値がリスト内になければ，ど

の項目も選択状態にはなりません。しかし，TDBComboBox は項目の現在の値を，それが Items リストに表示されているかどうかにはかかわりなく，編集ボックスに表示します。

TDBListBox の場合，Height プロパティは，リストボックスに一度に表示される項目名の数を決めます。IntegralHeight プロパティは，最後の項目名の表示方法を決めます。IntegralHeight が **false** の場合（デフォルト），リストボックスの下端は ItemHeight プロパティによって決まるので，一番下の項目名が完全には表示されないことがあります。IntegralHeight が **true** の場合，リストボックスに表示されている一番下の項目名まで完全に表示されます。

TDBComboBox の場合，Style プロパティは，ユーザーとコントロールとの対話の方法を決めます。デフォルトでは Style は csDropDown で，ユーザーがキーボードを使った値の入力や，ドロップダウンリストの項目名の選択ができることを表します。実行時に Items リストがどのように表示されるかは，以下のプロパティによって決まります。

- Style：コンポーネントの表示スタイルを決める
  - csDropDown（デフォルト）：ユーザーがテキストを入力できる編集ボックスのあるドロップダウンリストを表示する。すべての項目名は同じ高さの文字列
  - csSimple：編集コントロールと，常に表示されている固定長の項目のリストが組み合わせられる。Style を csSimple に設定するときは，リストが表示されるように Height プロパティを大きくする必要がある
  - csDropDownList：ドロップダウンリストと編集ボックスを表示するが，ユーザーは実行時にドロップダウンリストにない値の入力や変更ができない
  - csOwnerDrawFixed，csOwnerDrawVariable：項目名リストに，文字列以外の値（ビットマップなど）や，各項目ごとに異なるフォントを使った文字列を表示できる
- DropDownCount：リストに表示される項目名の最大数を決める。Items の項目名の数が DropDownCount の値より大きい場合は，ユーザーがリストをスクロールできる。Items の項目名の数が DropDownCount の値より小さい場合，リストはすべての項目名が収まるだけのサイズになる
- ItemHeight：スタイルが csOwnerDrawFixed の場合に各項目名の高さを決める
- Sorted：**true** ならば Items リストをソートして表示する

参照リストボックスとコンボボックスでのデータの表示と編集

参照リストボックスと参照コンボボックス（TDBLookupListBox および TDBLookupComboBox）では，有効な項目値を設定するための選択肢からなる一定のリストがユーザーに提供されます。ユーザーがリストの項目を選択すると，対応するデータセットの項目値が変更されます。

たとえば，OrdersTable に結合された項目を持つ注文フォームを考えます。OrdersTable には顧客 ID に対応した CustNo 項目はありますが，ほかの顧客情報はありません。これに対して，CustomersTable には顧客 ID に対応した CustNo 項目だけでなく，顧客の会社名や住所などの追加情報もあります。事務員が請求書を作成するときに，注文フォームで顧客 ID ではなく会社名によって顧客を選択できれば便利です。CustomersTable のすべての会社名を表示する TDBLookupListBox を使うと，ユーザーはリストの会社名を選択して，注文フォームの CustNo を適切な値に設定できます。

これらの参照コントロールは，表示項目名のリストを，2 つのソースのいずれかから取得します。

- **データセットに定義された参照項目：**
  参照項目を使ってリストボックスの項目名を指定するには，そのコントロールにリンクされているデータセットで参照項目を定義しておかなければなりません（この手順については，23-8 ページの「参照項目の定義」で述べます）。リストボックスの項目名に対する参照項目を指定する手順は次のとおりです。

  1. リストボックスの DataSource プロパティを，参照項目を持つデータセットを示すデータソースに設定します。
  2. DataField プロパティに対し，使いたい参照項目をドロップダウンリストから選択します。

  参照コントロールに関連付けられたテーブルをアクティブにすると，そのコントロールは指定されたデータ項目が参照項目であると認識し，参照項目の適切な値を表示します。

- **二次データソース，データ項目，キー：**
  データセットに参照項目を定義していない場合，二次データソース，二次データソースで検索する項目値，リストの項目名として返される項目値を使って，同じような関係を確立できます。リストボックスの項目名に対する二次データソースを指定する手順は次のとおりです。

  1. リストボックスの DataSource プロパティを元となるデータソースに設定します。
  2. DataField プロパティに対し，使いたい参照項目をドロップダウンリストから選択します。ここには参照項目は選択できません。
  3. リストボックスの ListSource プロパティを，参照項目を持つデータセットのデータソースに設定します。
  4. KeyField プロパティに対し，参照キーとして使う項目をドロップダウンリストから選択します。ドロップダウンリストには，手順の 3 で指定したデータソースに関連付けられているデータセットの項目が表示されます。インデックスの付いた項目を選択する必要はありませんが，インデックス付きの項目を選択すると，参照の処理効率が上がります。
  5. ListField プロパティに対し，参照の結果として返される値を含む項目をドロップダウンリストから選択します。ドロップダウンリストには，手順 3 で指定したデータソースに関連付けられているデータセットの項目が表示されます。

  参照コントロールに関連付けられたテーブルをアクティブにすると，そのコントロールはリストの項目名が二次ソースから取得されていると認識し，その二次ソースの適切な値を表示します。

TDBLookupListBox コントロールに一度に表示される項目名の数を指定するには，RowCount プロパティを使います。リストボックスの高さは，この行数にちょうど合うように調節される

TDBLookupComboBox のドロップダウンリストに表示される項目名の数を指定するには，かわりに，DropDownRows プロパティを使います。

メモ また，データグリッド内の列が参照コンボボックスとして機能するように設定することもできます。その手順については，19-20 ページの「参照リスト列を定義する」を参照してください。

チェックボックスを使った論理型項目値の処理

TDBCheckBox はデータベース対応のチェックボックスコントロールです。TDBCheckBox を使うと，データセット内の論理型項目の値を設定できます。たとえば，顧客あての請求書フォームにチェック

ボックスコントロールを配置して，チェックマークを付けると顧客を非課税扱いにし，チェックマークをはずすと顧客を非課税扱いにしないように定義できます。

データベース対応チェックボックスコントロールはそのチェックボックスのチェック状態と非チェック状態の管理を，現在の項目値を ValueChecked プロパティおよび ValueUnchecked プロパティの内容と比較して行います。項目値が ValueChecked プロパティと一致した場合，コントロールにチェックマークが付きます。そうでなく，項目が ValueUnchecked プロパティと一致した場合は，コントロールのチェックマークがはずれます。

メモ ValueChecked と ValueUnchecked を同じ値にすることはできません。

ユーザーが別のレコードに移動したときに，コントロールにチェックマークが付いている場合，ValueChecked プロパティには，コントロールがデータベースに登録する値を設定します。デフォルトではこの値は「true」に設定されていますが，必要であれば任意の文字型項目値に変更できます。ValueChecked の値としては，要素がセミコロンで区切られているリストも指定できます。いずれかの要素が現在のレコードにある項目の内容に一致する場合，チェックボックスにチェックマークが付きます。たとえば，次のように ValueChecked の文字列を指定します。

```
DBCheckBox1->ValueChecked = "True;Yes;On";
```

現在のレコードの項目の値が「True」，「Yes」，または「On」の場合，チェックボックスにチェックマークが付きます。項目と ValueChecked の文字列との比較では，大文字と小文字は区別されません。ユーザーが複数の ValueChecked 文字列のあるボックスにチェックマークを付けた場合は，最初の文字列がデータベースに登録される値になります。

ユーザーが別のレコードに移動したときに，コントロールにチェックマークが付いていない場合，ValueUnchecked プロパティにはコントロールがデータベースに登録する値を設定します。デフォルトではこの値は「false」に設定されていますが，必要であれば任意の文字型項目値に変更できます。ValueUnchecked の値としては，要素がセミコロンで区切られているリストも指定できます。いずれかの要素が現在のレコードにある項目の内容に一致する場合，チェックボックスのチェックマークがはずされます。

データベース対応チェックボックスは，現在のレコードにある項目の内容が ValueChecked プロパティまたは ValueUnchecked プロパティに指定されたどの値とも一致しない場合は，AllowGrayed プロパティが **true** であれば使用不可になります。

チェックボックスが関連付けられている項目が論理型項目の場合は，項目の内容が True であれば常にチェックボックスにチェックマークが付き，項目の内容が False であればチェックマークがはずされます。この場合，ValueChecked プロパティと ValueUnchecked プロパティに指定された文字列は論理型項目に影響を与えません。

ラジオコントロールを使った項目値の制限

TDBRadioGroup はラジオグループコントロールのデータベース対応バージョンです。
TDBRadioGroup を使うと，取り得る値が限られているデータ項目に対して，ラジオボタンコントロールで値を設定できます。ラジオグループの各ボタンは，項目が取り得る値のそれぞれに対応しています。ユーザーは，目的のラジオボタンを選択すれば，データ項目の値を設定できます。

Items プロパティは，グループ内に表示されるラジオボタンを決定します。Items は文字列リストです。Items の文字列ごとに，1 つのラジオボタンが表示されます。各文字列は，ラジオボタンのラベルとして，対応するラジオボタンの右側に表示されます。

ラジオグループに関連付けられている項目の現在値が Items プロパティの文字列のいずれかと一致する場合，そのラジオボタンが選択されます。たとえば，「Red」，「Yellow」，「Blue」という 3 つの文字列をこの順番で Items に指定した場合，現在のレコードの項目の値が「Blue」であれば，グループの 3 番目のボタンが選択された状態になります。

メモ 項目が Items のどの文字列とも一致しない場合でも，項目が Values プロパティの文字列と一致すれば，ラジオボタンが選択されることがあります。現在のレコードの項目が Items と Values のどの文字列にも一致しない場合は，ラジオボタンは選択されません。

Values プロパティには，ユーザーがラジオボタンを選択してレコードを登録したときにデータセットに返される文字列のリストを，オプションで指定できます。文字列は順番にボタンに関連付けられます。最初の文字列は最初のボタンに，2 番目の文字列は 2 番目のボタンに，という具合に関連付けられます。たとえば，Items に「Red」，「Yellow」，「Blue」をこの順番で指定し，Values に「Magenta」，「Yellow」，「Cyan」をこの順番で指定したとします。ユーザーが「Red」というラベルのボタンを選択すると，「Magenta」がデータベースに登録されます。

Values に文字列を指定していない場合は，選択されたラジオボタンの Item の文字列がレコードの登録時にデータベースに返されます。

複数のレコードを表示する

同じフォーム上にいくつものレコードを表示したいことがあります。たとえば，請求書作成アプリケーションで，1 人の顧客からのすべての注文を同じフォームに表示させる場合などです。

複数のレコードを表示するには，グリッドコントロールを使用します。グリッドコントロールを使用すると，複数のデータ項目をマルチレコードで表示できるので，アプリケーションのユーザーインターフェースはさらに強力で効果的なものになります。これについては 19-15 ページの「TDBGrid を使ったデータの表示と編集」および 19-26 ページの「ほかのデータベース対応コントロールが入ったグリッドの作成」で述べます。

メモ 単方向データセットを使っている場合，複数のレコードを表示することはできません。

1 つのレコードの各項目と複数のレコードを表すグリッドとを合わせて表示するユーザーインターフェースを設計したい場合もあります。これら 2 つのアプローチを組み合わせるモデルは 2 つあります。

- マスター / 詳細フォーム：マスターテーブルと詳細テーブルの両方の情報を表すには，単一項目を表示するコントロールとグリッドコントロールの両方を使用します。たとえば，1 人の顧客についての情報を表示するときに，その顧客からの注文を表示するグリッドを一緒に使用します。マスター詳細フォームの基礎となるテーブルのリンクについての詳細は，22-33 ページの「マスター / 詳細関係の作成」および 22-45 ページの「マスター / 詳細関係をパラメータを使用して確立する」を参照してください。
- ドリルダウンフォーム：複数のレコードを表示するフォームに，現在のレコードについてのみ詳細情報を表示する単一項目コントロールを含めることができます。このアプローチは特に，レ

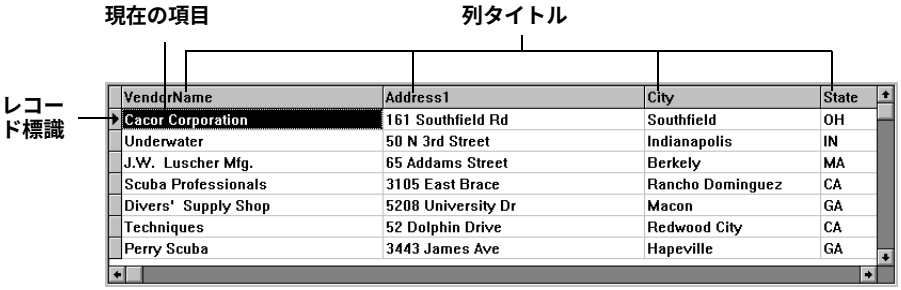
コードに長いメモやグラフィック情報が含まれているときに便利です。ユーザーがグリッド内のレコードをスクロールすると，それにつれてメモまたはグラフィックは現在のレコードの値に更新されます。これのセットアップは簡単です。グリッドとメモまたはイメージのデータソースが共通であれば，2 つの表示は自動的に同期します。

ヒント これら 2 つのアプローチを 1 つのフォームで併用するのはお勧めできません。通常，そのようなフォーム内のデータ関係はユーザーにとってわかりにくく，混乱をまねきます。

TDBGrid を使ったデータの表示と編集

TDBGrid コントロールは，データセットのレコードを表形式のグリッドで表示したり編集するのに使います。

図 19.1 TDBGrid コントロール



グリッドコントロールに表示されるレコードの外観には，次の 3 つの要因が影響します。

- 列エディタを使ってグリッド用に定義された持続的列オブジェクトがあるかどうか。持続的列オブジェクトを使うと，グリッドやデータの外観を柔軟に設定できる。持続的項目の使用方法については，19-16 ページの「カスタマイズされたグリッドの作成」を参照してください。
- グリッドに表示されるデータセットの持続的項目コンポーネントを作成するかどうか。項目エディタを使った持続的項目コンポーネントの作成についての詳細は，第 23 章「項目コンポーネントの操作」を参照
- ADT および配列項目を表示するグリッドについての，データセットの ObjectView プロパティの設定。19-21 ページの「ADT 項目および配列項目を表示する」を参照してください。

グリッドコントロールは，それ自体が TDBGridColumns オブジェクトのラッパーである Columns プロパティを持っています。TDBGridColumns は，グリッドコントロール内のすべての列を表す TColumn オブジェクトのコレクションです。カラムエディタを使って設計時に列属性を設定することも，グリッドの Columns プロパティを使って実行時に TDBGridColumns のプロパティ，イベント，メソッドにアクセスすることもできます。

グリッドコントロールのデフォルト状態での使い方

グリッドの Columns プロパティの State プロパティは，そのグリッドに対する持続的列オブジェクトが存在するかどうかを示します。Columns->State は実行時のみのプロパティで，グリッドに対して自動的に設定されます。デフォルトの状態は csDefault で，そのグリッドについて持続的列オブジェクトが存在しないことを意味します。その場合，グリッドへのデータの表示は，グリッドのデータセット内の項目のプロパティによって決まります。持続的項目コンポーネントがない場合には，表示特性のデフォルトセットによって決まります。

グリッドの Columns->State プロパティが csDefault になっている場合，グリッド列はデータセットの表示可能な項目に基づいて動的に生成され，グリッド内の列の順序はデータセット内の項目の順序と一致します。グリッド内の列はそれぞれ 1 つの項目コンポーネントと関連付けられます。項目コンポーネントのプロパティが変化すると，すぐにグリッドに反映されます。

動的に生成された列を持つグリッドコントロールを使うと，実行時に選択した任意のテーブルの内容を表示したり編集するのに便利です。グリッドの構造が設定されていないので，動的に変化してさまざまなデータセットを表示できます。動的に生成された列を持つ単一グリッドは即座に Paradox テーブルを表示でき，グリッドの DataSource プロパティが変化したりデータソースそのものの DataSet プロパティが変化した場合には表示を切り替えて SQL の問い合わせ結果を表示できます。

設計時または実行時に動的な列の外観を変更できますが，実際に変更されるのは列に表示されている項目コンポーネントの対応するプロパティです。動的な列のプロパティが存在するのは，列が単一のデータセット内の特定の項目と関連付けられている間だけです。たとえば，ある列の Width プロパティを変更すると，その列に関連付けられている項目の DisplayWidth プロパティも変わります。Font のように項目プロパティに基づいていない列プロパティに対する変更は，その列が残っている限り存続します。

グリッドのデータセットが動的な項目コンポーネントで構成されている場合，その項目はデータセットが閉じるたびに破棄されます。項目コンポーネントが破棄されると，それに関連付けられている動的な列もすべて破棄されます。グリッドのデータセットが持続的項目コンポーネントで構成されている場合は，たとえデータセットが閉じてもその項目コンポーネントは存続します。そのため，その項目に関連付けられている列も，データセットが閉じてもそのプロパティを保持できます。

メモ 実行時にグリッドの Columns->State プロパティを csDefault に変更すると，グリッドのすべての列オブジェクトが（持続的な列も含めて）削除され，グリッドのデータセットの表示される項目に基づいて動的な列が再構築されます。

カスタマイズされたグリッドの作成

カスタマイズされたグリッドは，列の外観や列データの表示方法を記述する持続的列オブジェクトが定義されたコントロールです。グリッドをカスタマイズすると，複数のグリッドを使って同一のデータセットの表示（列順序，項目選択，列の色やフォントなど）をいろいろと変更することができます。さらに，グリッドが使う項目やデータセットの項目順を変えずに，ユーザーが実行時にグリッドの外観を変更できるようになります。

カスタマイズされたグリッドは，設計時に構造のわかっているデータセットと一緒に使うのがもっとも望ましい方法です。カスタマイズされたグリッドは，設計時に設定した項目名がデータセット内にあることを前提としているので，実行時に選択された任意のテーブルの参照に使うのはあまり適切ではありません。

持続的な列について

グリッドに作成した持続的列オブジェクトは，グリッドのデータセットの基になる項目とはゆるやかに関連付けられているだけです。持続的な列のプロパティのデフォルト値は，値がその列プロパティに割り当てられるまでデフォルトのソース（関連付けられた項目やグリッド自体）の中から動的に取得されます。列のプロパティは，値を割り当てられるまでは，デフォルトのソースが変わるたびに値が変化します。いったん列プロパティに値を割り当てると，デフォルトのソースが変わっても値は変わらなくなります。

たとえば，列タイトルキャプションのデフォルトソースは，関連付けられている項目の DisplayLabel プロパティです。DisplayLabel プロパティを変更すると，すぐにその変更が列タイトルに反映されます。列タイトルのキャプションに文字列を割り当てると，そのタイトルキャプションは関連付けられている項目の DisplayLabel プロパティから独立します。それ以降はその項目の DisplayLabel プロパティを変更しても列タイトルは変わりません。

持続的な列は，関連付けられている項目コンポーネントから独立して存在しています。実際，持続的な列は項目オブジェクトとまったく関連付けられていなくてもかまいません。持続的な列の FieldName プロパティが空白の場合，あるいは項目名がグリッドの現在のデータセットの中にある項目名と一致しない場合，その列の Field プロパティは NULL になり，空白のセルからなる列が描かれます。セルのデフォルトの描画メソッドをオーバーライドすれば，空白のセル内に独自のカスタム情報を表示することができます。たとえば，空白の列を使って，集合を集計するレコードのグループの最後のレコードに集合値を表示することができます。別な利用法として，レコードデータのある側面を図示するビットマップや棒グラフなどを表示することもできます。

複数の持続的な列をデータセットの同一項目に関連付けることができます。たとえば，横に広いグリッドの左右の端に部品番号の項目を表示すれば，グリッドをスクロールしなくても部品番号を簡単に見つけられるようになります。

メモ 持続的な列は，データセットの項目と関連付ける必要がないため，また複数の列で同一の項目を参照できるため，カスタマイズされたグリッドの FieldCount プロパティはグリッドの列数と等しいかそれ以下になります。また，カスタマイズされたグリッド内で現在選択されている列が項目と関連付けられていない場合，グリッドの SelectedField プロパティは NULL になり，SelectedIndex プロパティは -1 になります。

持続的な列は，グリッドのセルを別のデータセットや持続的選択リストからの参照値のコンボボックスドロップダウンリストとして表示したり，現在のセルと関連した特殊なデータビューアやダイアログボックスをクリックで起動する省略記号のボタン（[...]）として表示するように，構成することができます。

持続的な列を作成する

設計時にグリッドの外観をカスタマイズするには，列エディタを呼び出してグリッドの持続的列オブジェクトを作成します。実行時には，持続的列オブジェクトを持つグリッドの State プロパティは自動的に csCustomized に設定されます。

グリッドコントロールの持続的な列を作成する手順は次のとおりです。

1. フォーム内のグリッドコンポーネントを選択します。
2. オブジェクトインスペクタでグリッドの Columns プロパティをダブルクリックしてカラムエディタを呼び出します。

［カラム］リストボックスには，選択したグリッド用に定義されている持続的な列が表示されます。初めてカラムエディタを呼び出したときには，グリッドがデフォルト状態で動的な列しか含まれないため，このリストは空です。

一度にデータセットのすべての項目について持続的な列を作成することも，個々に持続的な列を作成することもできます。すべての項目について持続的な列を作成する手順は次のとおりです。

1. グリッドを右クリックしてコンテキストメニューを呼び出し，［すべての項目の追加］を選択します。グリッドがまだデータソースに関連付けられていない場合，［すべての項目の追加］は使用不可になっています。［すべての項目の追加］を選択する前に，グリッドをアクティブなデータセットを持っているデータソースと関連付けてください。
2. グリッドに持続的な列がすでにある場合は，既存の列を削除するか列の集合に追加するかを尋ねるダイアログボックスが表示されます。［はい］を選択すると，既存の持続的な列の情報は削除され，現在のデータセットのすべての項目がデータセット内での項目名順に挿入されます。［いいえ］を選択すると，既存の持続的な列の情報が保持され，新しい列の情報がデータセットの追加項目に基づいてデータセットに追加されます。
3. ［閉じる］をクリックして持続的な列をグリッドに適用し，ダイアログボックスを閉じます。

持続的な列を個々に作成する手順は次のとおりです。

1. カラムエディタの［追加］ボタンを選択します。リストボックス内の新規の列が選択されます。新しい列には，番号とデフォルトの名前（たとえば 0 - TColumn）が付けられます。
2. 項目をこの新しい列と関連付けるには，オブジェクトインスペクタで FieldName プロパティを設定します。
3. 新しい列のタイトルを設定するには，オブジェクトインスペクタで Title プロパティの［+］をクリックして，Caption プロパティを設定します。
4. カラムエディタを終了して持続的な列をグリッドに適用し，ダイアログボックスを閉じます。

実行時には，Columns::State プロパティに csCustomized を割り当てることにより，持続的列に切り替えることができます。グリッドの既存の列は破棄され，グリッドのデータセットの各項目について持続的な列が新しく構築されます。また，列リストの Add メソッドを呼び出せば，持続的な列を追加することもできます。

```
DBGrid1->Columns->Add();
```

持続的な列を削除する

グリッドの持続的な列の削除は，表示したくない項目を消去するのに役立ちます。グリッドから持続的な列を削除する手順は次のとおりです。

1. グリッドをダブルクリックしてカラムエディタを呼び出します。
2. ［カラム］リストボックスで，削除する項目を選択します。
3. ［削除］をクリックします（コンテキストメニューまたは【Del】キーを使っても列を削除できます）。

メモ グリッドからすべての列を削除すると，Columns->State プロパティが csDefault 状態に戻り，データセットの各項目について動的な列を自動的に構築します。

列オブジェクトを次のように解放するだけで，持続的な列を実行時に削除できます。

```
delete DBGrid1->Columns->Items[5];
```

持続的な列を並べ替える

列が列エディタに表示される順序は，列がグリッドに表示される順序と同じです。列の順序を変更するには，［カラム］リストボックス内で列をドラッグアンドドロップします。

列の順序を変更する手順は次のとおりです。

1. ［カラム］リストボックスで列を選択します。
2. 選択した列をリストボックス内の新しい位置までドラッグします。

実行時も列の順序を変更できます。実行時に変更するには，列タイトルをクリックして新しい位置にドラッグします。

メモ 項目エディタで持続的な項目の順序を変更すると，デフォルトグリッドの列の順序は変わりますが，カスタムグリッドの列の順序は変わりません。

重要 動的な列と動的な項目の両方を持つグリッドの列の順序は，設計時には変更できません。変更された項目や列の順序を記録する持続的なものが何もないからです。

列の DragMode プロパティが dmManual に設定されていれば，ユーザーが実行時にマウスでその列をドラッグしてグリッド内での位置を変更することができます。csDefault 状態の State プロパティを使ってグリッドの列の順序を変更しても，グリッドの基になるデータセットの項目コンポーネントの順序が変わります。物理テーブルの項目の順序は変わりません。実行時にユーザーが列の順序を変更できないようにするには，グリッドの DragMode プロパティを dmAutomatic に設定します。

実行時には，列が移動された後には，グリッドの OnColumnMoved イベントが発生します。

設計時に列プロパティを設定する

列プロパティはその列のセルにデータを表示する方法を決めます。列プロパティのほとんどは，グリッドや関連項目コンポーネントなど，デフォルトソースと呼ばれる，別のコンポーネントと関連付けられているプロパティからデフォルト値を取得します。

列のプロパティを設定するには，カラムエディタで列を選択し，そのプロパティをオブジェクトインスペクタで設定します。次の表に，設定できる主要な列プロパティを示します。

表 19.2 列のプロパティ

プロパティ	目的
Alignment	列の項目データを左揃え，右揃え，または中央揃えにする。デフォルトソース：TField::Alignment
ButtonStyle	cbsAuto（デフォルト）の場合，関連付けられている項目が参照項目であったり，列の PickList プロパティにデータが入っている場合にドロップダウンリストを表示する。 cbsEllipsis の場合，セルの右に省略記号ボタン（[...]）を表示する。このボタンをクリックするとグリッドの OnEditButtonClick イベントが発生する。 cbsNone の場合，列は通常の編集コントロールを使って列のデータを編集する。
Color	列のセルの背景色を指定する。デフォルトソース：TDBGrid::Color（テキストの前景色については，Font プロパティを参照）
DropDownRows	ドロップダウンリストに表示するテキストの行数。デフォルト：7
Expanded	列が展開されているかどうかを指定する ADT または配列項目を表す列にのみ適用される
FieldName	この列と関連付けられている項目名を指定する。空白のままでもよい
ReadOnly	**true** の場合，ユーザーは列のデータを編集できない **false**（デフォルト）の場合，ユーザーは列のデータを編集できる
Width	画面ピクセル数で列幅を指定する。デフォルトソース：TField::DisplayWidth
Font	列にテキストを表示するために使うフォントの種類，サイズ，色を指定する。デフォルトソース：TDBGrid::Font
PickList	列のドロップダウンリストに表示する値のリストを含む
Title	選択した列のタイトルのプロパティを設定する

次の表に，Title プロパティで指定できるオプションについてまとめます。

表 19.3 展開した TColumn の Title プロパティ

プロパティ	目的
Alignment	列タイトルのキャプションテキストを左揃え（デフォルト），右揃え，または中央揃えにする
Caption	列タイトルに表示するテキストを指定する。デフォルトソース：TField::DisplayLabel
Color	列のタイトルセルを描くのに使う背景色を指定する。デフォルトソース：TDBGrid::FixedColor
Font	列タイトルにテキストを描くのに使うフォントの種類，サイズ，色を指定する。デフォルトソース：TDBGrid::TitleFont

参照リスト列を定義する

参照コンボボックスコントロールに似た，値のドロップダウンリストを表示する列を作成することができます。コンボボックスのように動作する列を指定するには，列の ButtonStyle プロパティを cbsAuto に設定します。リストに値を入れておくと，実行時にその列のセルが編集モードになったときに，グリッドには自動的にコンボボックスのようなドロップダウンボタンが表示されます。

リストにユーザーが選択する値を表示するには以下の 2 通りの方法があります。

- 値を参照テーブルから取得します。ある列が別の参照テーブルの値をドロップダウンリストで表示できるようにするには，データセットの参照項目を定義しなければなりません。参照項目の作成方法については，23-8 ページの「参照項目の定義」を参照してください。参照項目を定義したら，列の FieldName をその参照項目名に設定します。このドロップダウンリストには，参照項目で定義された参照値が自動的に表示されます。

- 値のリストを設計時に明示的に指定します。設計時にリスト値を入力するには，オブジェクトインスペクタで列の PickList プロパティをダブルクリックします。これにより，文字列リストエディタが開くので，列の選択リストに表示する値を入力できます。

デフォルトでは，ドロップダウンリストは 7 つの値を表示します。このリストの長さは，DropDownRows プロパティを設定すれば，変更できます。

メモ 明示的な選択リストを持つ列を通常の状態に戻すには，文字列リストエディタを使って選択リストからすべてのテキストを削除します。

ボタンを列に入れる

列の中では，通常のセルエディタの右に省略記号ボタン（[...]）を表示できます。〔Ctrl〕+〔Enter〕を押すかマウスをクリックすると，グリッドの OnEditButtonClick イベントが発生します。省略記号ボタンを使うと，列のデータの詳細表示ができるフォームを呼び出せます。たとえば，請求書の概要を表示するテーブルで，請求総計列に省略記号ボタンを設定してその請求書の項目や税計算方法などを表示するフォームを呼び出すようにすることができます。グラフィック項目では，省略記号ボタンを使ってイメージを表示するフォームを呼び出せます。

列に省略記号ボタンを作成する手順は次のとおりです。

1. ［カラム］リストボックスで列を選択します。
2. ButtonStyle を cbsEllipsis に設定します。
3. OnEditButtonClick イベントハンドラを記述します。

列をデフォルト値に戻す

実行時には，列の AssignedValues プロパティで，列プロパティが明示的に割り当てられているかどうかを確認できます。明示的に定義されたのではない値は，関連付けられた項目に基づいて動的に定義されたものか，またはグリッドのデフォルトです。

1 つまたは複数の列に対して加えたプロパティの変更は元に戻せます。カラムエディタで戻す対象の列を選択し（複数の選択も可），コンテキストメニューから［デフォルトに戻す］を選択します。これを選択すると，設定されたプロパティの値が破棄され，列のプロパティがその基になっている項目コンポーネントからのプロパティに戻されます。

実行時には，列の RestoreDefaults メソッドを呼び出せば，個々の列のプロパティをすべてデフォルトに戻すことができます。また，列リストの RestoreDefaults メソッドを呼び出せば，グリッド内のすべての列のプロパティをデフォルトに戻すことができます。

```
DBGrid1->Columns->RestoreDefaults();
```

ADT 項目および配列項目を表示する

ときには，グリッドのデータセットの項目が，テキスト，グラフィック，数値などの，単純な値を表していないことがあります。データベースサーバーの中には，ADT 項目や配列項目など，より単純なデータタイプから合成された項目を許可しているものもあります。

グリッドで合成項目を表示するには，2 つの方法があります。

- 合成項目タイプを「完全に展開」して，項目を構成している単純なタイプのそれぞれを，データセット内の個別のフィールドに表示します。合成項目を完全に展開すると，その各部分は個別の項目として表示されます。その共通のソースは，各項目名の前に，基のデータベーステーブルの共通の親項目の名前が来るという点にのみ，反映されます。

  合成項目を完全に展開して表示するには，データセットの ObjectView プロパティを **false** に設定します。データセットは，合成項目を個別の項目のセットとして保管します。グリッドは，各部分に個別の列を割り当てることによって，このことを反映します。

- 合成項目は，実際には単一の項目であるので，それを反映して単一の列内に表示することもできます。合成項目を単一の列内に表示した場合には，その項目のタイトルバーの中にある矢印をクリックするか，その列の Expanded プロパティを設定すれば，列を展開したり折りたたんだりすることができます。

  - 列が展開されると，親項目のタイトルバーの下に，子項目がそれぞれ個別の下位列の中にタイトルバー付きで表示されます。つまり，グリッドのタイトルバーが高さ方向に広がって，上の行には合成項目の名前，下の行には個別の部分の下位区分が表示されます。複合ではない項目は，広がったタイトルバーにそのまま表示されます。構成部分がまた合成項目であった場合（たとえば，詳細テーブルの中に詳細テーブルがネストしているような場合）には，タイトルバーはそれに応じてさらに広がります。
  - 項目が折りたたまれるとその項目の列は 1 列だけになり，その列にすべての子項目を収めた編集不可能なカンマで区切られた文字列が表示されます。

  合成項目を展開と折りたたみが可能な列の中に表示するには，データセットの ObjectView プロパティを **true** に設定します。データセットは合成項目を，ネストした下位項目のセットを含む単一の項目コンポーネントとして保管します。グリッドは，展開と折りたたみが可能な列として表示することによって，このことを反映します。

図 19.2 は，ADT 項目と配列項目が 1 つずつ表示されたグリッドを示しています。このデータセットの ObjectView プロパティは **false** に設定されているので，子項目はそれぞれ個別の列に表示されています。

図 19.2　ObjectView が false に設定された TDBGrid コントロール

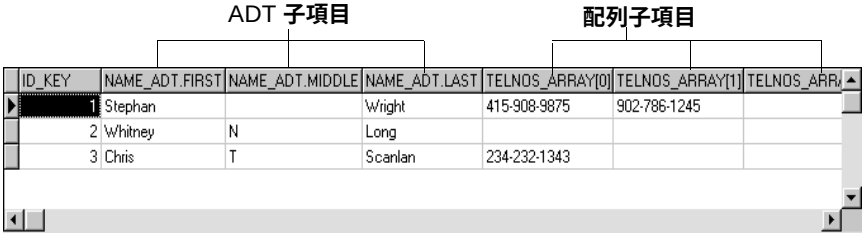


図 19.3 と 19.4 は，ADT 項目と配列項目が 1 つずつ表示されたグリッドを示しています。図 19.3 では項目が折りたたまれています。この状態では項目を編集することはできません。図 19.4 では項目が展開されています。項目を展開したり折りたたんだりするには，項目のタイトルバーの中にある矢印をクリックします。

図 19.3 Expanded が false に設定された TDBGrid コントロール

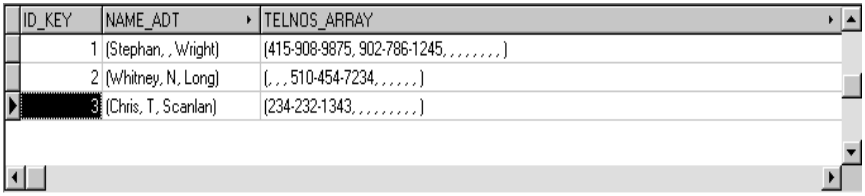
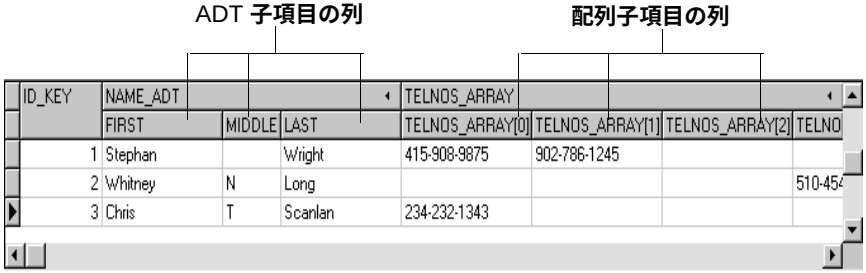


図 19.4 Expanded が true に設定された TDBGrid コントロール



次の表に，TDBGrid 内での ADT 項目と配列項目の表示方法に影響するプロパティを示します。

表 19.4 合成項目の表示方法に影響するプロパティ

プロパティ	オブジェクト型	目的
Expandable	TColumn	列を展開して個別の編集可能な列の中に子項目を表示できるかどうかを示す（読み出し専用）
Expanded	TColumn	列が展開されているかどうかを指定する
MaxTitleRows	TDBGrid	グリッド内に表示できるタイトル行の最大数を指定する
ObjectView	TDataSet	項目が完全に展開されて表示されるか，それとも項目を展開したり折りたたんだりできるオブジェクトモードで表示されるかを指定する
ParentColumn	TColumn	子項目の列を所有している TColumn オブジェクトを参照する

メモ ADT 項目と配列項目に加えて，データセットが，他のデータセット（データセット項目）や，他のデータセット（参照）項目内のレコードを参照している項目を含んでいる場合があります。データベース対応グリッドは，そのような項目をそれぞれ「(DataSet)」または「(Reference)」として表示します。実行時には，右側に省略記号ボタンが表示されます。省略記号をクリックすると，項目の内容を表示するグリッドを持つ新しいフォームが表示されます。データセット項目の場合，このグリッドは項目の値であるデータセットを表示します。参照項目の場合，このグリッドには他のデータセットのレコードを表示する単一の行が含まれます。

グリッドオプションの設定

設計時に，グリッドの Options プロパティを使って，グリッドの実行時の基本的な動作と外観を制御できます。設計時に初めてグリッドコンポーネントをフォームに配置すると，オブジェクトインスペクタでは Options プロパティが［+］（正符号）付きで表示されます。これは，Options プロパティを展開して，個別に設定できる一連の論理型プロパティを表示できることを示しています。これらの論

理型プロパティを表示して設定するには，［+］をクリックします。すると，オプションのリストが Options プロパティの下のオブジェクトインスペクタに表示されます。［+］は［-］（負符号）に変化します。［-］をクリックすれば，プロパティのリストを折りたたむことができます。

次の表に，設定可能な Options プロパティと，各プロパティが実行時にグリッドに与える効果を示します。

表 19.5　展開した TDBGrid の Options プロパティ

オプション	目的
dgEditing	**true**（デフォルト）の場合，グリッド内のレコードを編集，挿入，削除できる **false** の場合は，グリッド内のレコードを編集，挿入，削除できない
dgAlwaysShowEditor	**true** の場合，項目は選択されると自動的に編集状態になる **false**（デフォルト）の場合，項目は選択されても自動的に編集状態にはならない
dgTitles	**true**（デフォルト）の場合，グリッドの上部に項目名が表示される **false** の場合，項目名は表示されない
dgIndicator	**true**（デフォルト）の場合，標識列がグリッドの左に表示され，現在のレコードの標識（グリッド左にある矢印）がアクティブになって現在のレコードを示す。挿入時には矢印がアスタリスクになる。編集時には矢印が I 形になる **false** の場合，標識列はオフになる
dgColumnResize	**true**（デフォルト）の場合，タイトル領域の列ルーラーをドラッグして列のサイズ変更ができる。サイズ変更をすると基になる TField コンポーネントに対応して決められていた幅も変更される **false** の場合，グリッド内の列はサイズ変更できない
dgColLines	**true**（デフォルト）の場合，列の間に垂直分割線が表示される **false** の場合，列の間に垂直分割線は表示されない
dgRowLines	**true**（デフォルト）の場合，レコード間に水平分割線が表示される **false** の場合，レコード間に水平分割線は表示されない
dgTabs	**true**（デフォルト）の場合，レコードの項目間をタブ移動できる **false** の場合，タブ移動するとグリッドコントロールの外に出る
dgRowSelect	**true** の場合，選択バーがグリッドの幅全体に広がる **false**（デフォルト）の場合，レコード内の項目を選択するとその項目だけが選択される
dgAlwaysShowSelection	**true**（デフォルト）の場合，グリッドの選択バーは別のコントロールにフォーカスがあっても常に表示される **false** の場合，グリッドの選択バーはグリッドにフォーカスがあるときにだけ表示される
dgConfirmDelete	**true**（デフォルト）の場合，レコード削除（〔Ctrl〕+〔Del〕）の確認を行う **false** の場合，確認なしでレコードを削除する
dgCancelOnExit	**true**（デフォルト）の場合，グリッドからフォーカスがはずれたときに未完了の挿入はキャンセルされる。このオプションにより，不完全なレコードや空のレコードが生じるのを防止できる **false** の場合，未完了の挿入が許可される
dgMultiSelect	**true** の場合，ユーザーは〔Ctrl〕+〔Shift〕または〔Shift〕+ 矢印キーを使ってグリッド内の連続していない複数の行を選択できる **false**（デフォルト）の場合，ユーザーは複数の行を選択できない

グリッド内の編集

以下の条件（デフォルト）を満たす場合，実行時にグリッドを使って既存のデータを変更したり新しいレコードを入力することができます。

- データセットの CanModify プロパティが **true** である
- グリッドの ReadOnly プロパティが **false** である

グリッド内でレコードを編集する場合，各項目に対して行った変更はレコードバッファに記録されます。ただし，ユーザーがグリッド内の別のレコードに移動するまでポストされません。たとえフォーカスがフォーム内の別のコントロールに移っても，そのデータセットに対するカーソルが別のレコードに移動しない限り，グリッドは変更を登録しません。レコードを登録するときに，関連付けられたすべてのデータベース対応コンポーネントの状態の変化が調べられます。変更のあるデータを含む項目を更新するときに問題が発生すると，グリッドは例外を生成し，レコードへの変更は行われません。

メモ アプリケーションが更新をキャッシュしている場合，レコードの変更を登録しても，変更は，内部キャッシュに追加されるだけです。変更は，アプリケーションが更新を適用するまで，基のデータベーステーブルには登録されません。

別のレコードに移動する前にいずれかの項目で【Esc】を押せば，レコードの編集内容をすべて取り消せます。

グリッドの描画の制御

グリッドコントロールの描画方法を制御する最初のレベルは，列のプロパティを設定することです。グリッドは，自動的に列のフォント，色，位置合わせの各プロパティを使って列のセルを描きます。データ項目に入るテキストは，その列に関連付けられた項目コンポーネントの DisplayFormat プロパティか EditFormat プロパティを使って描かれます。

グリッドの OnDrawColumnCell イベントのコードを使って，デフォルトのグリッドの表示論理を拡張できます。グリッドの DefaultDrawing プロパティが **true** の場合，イベントハンドラ OnDrawColumnCell が呼び出される前に通常のすべての描画が実行されます。デフォルトの表示以外にもコードで描画できます。これは主に，空白の持続的な列を定義してその列のセルに特殊なグラフィックを描きたい場合に使います。

グリッドの描画方法全体を置換したい場合は，グリッドの DefaultDrawing を false に設定し，グリッドの OnDrawColumnCell イベントに描画コードを入れます。特定の列や項目データ型に限って描画方法を置換したい場合は，イベントハンドラ OnDrawColumnCell の中で DefaultDrawColumnCell を呼び出して，選択した列ではグリッドが通常の描画コードを使うようにします。たとえば論理項目型の描画方法だけを変更したい場合などには，この方法を使うと手間が省けます。

実行時のユーザーの操作への応答

グリッド内での特定の動作に反応するイベントハンドラを記述すれば，実行時のグリッドの動作を変更できます。一般に，グリッドは一度に多数の項目とレコードを表示するので，個々の列に加えられた変更に細かく対応する必要があります。たとえば，ユーザーが特定の列に入ったり出たりするたびに，フォーム上の別の場所にあるボタンをアクティブにしたり非アクティブにしたい場合があります。

次の表にオブジェクトインスペクタで使用可能なグリッドイベントを示します。

表 19.6 グリッドコントロールのイベント

イベント	目的
OnCellClick	ユーザーがグリッド内のセルをクリックしたときに発生する
OnColEnter	ユーザーがグリッドの列に入ったときに発生する
OnColExit	ユーザーがグリッドの列から出たときに発生する
OnColumnMoved	ユーザーが列を新しい位置に移動したときに発生する
OnDblClick	ユーザーがグリッド内でダブルクリックしたときに発生する
OnDragDrop	ユーザーがグリッド内でドラッグアンドドロップしたときに発生する
OnDragOver	ユーザーがグリッドを越えてドラッグしたときに発生する
OnDrawColumnCell	アプリケーションが個々のセルを描画する必要があるときに発生する
OnDrawDataCell	（旧式）State が csDefault になっている場合に，アプリケーションが個々のセルを描画する必要があるときに発生する
OnEditButtonClick	ユーザーが省略記号ボタンをクリックしたときに発生する
OnEndDrag	ユーザーがグリッド内でドラッグをやめたときに発生する
OnEnter	フォーカスがグリッドに移ったときに発生する
OnExit	グリッドがフォーカスを失ったときに発生する
OnKeyDown	グリッド上でユーザーがいずれかのキー（またはキーの組み合わせ）を押したときに発生する
OnKeyPress	グリッド上でユーザーがどれか 1 つの英数字キーを押したときに発生する
OnKeyUp	グリッド上でユーザーがキーを放したときに発生する
OnStartDrag	ユーザーがグリッド上でドラッグを開始したときに発生する
OnTitleClick	ユーザーが列のタイトルをクリックしたときに発生する

これらのイベントには多くの用途があります。たとえば，OnDblClick イベント用にハンドラを記述して，ユーザーが列に入力する値を選択できるリストをポップアップさせることができます。このようなハンドラは SelectedField プロパティを使って現在の行と列を指定します。

ほかのデータベース対応コントロールが入ったグリッドの作成

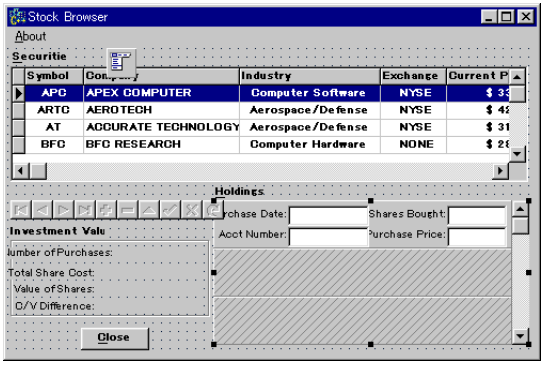
TDBCtrlGrid コントロールは複数のレコードの複数の項目を表形式のグリッドに表示します。グリッドの個々のセルは単一の行からの複数の項目を表示します。データベースコントロールグリッドを使う手順は次のとおりです。

1. データベースコントロールグリッドをフォーム上に入れます。
2. グリッドの DataSource プロパティをデータソースの名前に設定します。
3. グリッドの設計セルの中に個々のデータコントロールを入れます。グリッドの設計セルはグリッドの一番上か左のセルで，ほかのコントロールを配置できる唯一のセルです。

4. 各データコントロールの DataField プロパティを項目の名前に設定します。これらのデータコントロールのデータソースは，すでにデータベースコントロールグリッドのデータソースに設定されています。
5. セル内でコントロールを希望どおりに並べ替えます。

データベースコントロールグリッドを含むアプリケーションをコンパイルして実行すると，実行時に設計セル内に設定したデータコントロールの配置がグリッドの各セルに複製されます。個々のセルはデータセット内の異なるレコードを表示します。

図 19.5　設計時の TDBCtrlGrid



次の表に，設計時に設定できるデータベースコントロールグリッドに特有なプロパティの一部を示します。

表 19.7　データベースコントロールグリッドのプロパティ（抜粋）

プロパティ	目的
AllowDelete	**true**（デフォルト）の場合，レコードの削除を許す **false** の場合，レコードの削除を許さない
AllowInsert	**true**（デフォルト）の場合，レコードの挿入を許す **false** の場合，レコードの挿入を許さない
ColCount	グリッド内の列数を設定する。デフォルト = 1
Orientation	goVertical（デフォルト）の場合，レコードを上から下へと表示する goHorizontal の場合，レコードを左から右へと表示する
PanelHeight	個々のパネルの高さを設定する。デフォルト = 72
PanelWidth	個々のパネルの幅を設定する。デフォルト = 200
RowCount	表示するパネル数を設定する。デフォルト = 3
ShowFocus	**true**（デフォルト）の場合，実行時に現在のレコードのパネルの周囲にフォーカスの四角形を表示する **false** の場合，フォーカスの四角形を表示しない

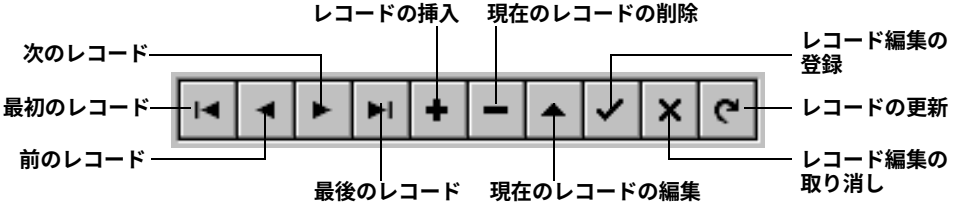
データベースコントロールグリッドのプロパティとメソッドについての詳細は，オンラインヘルプの『VCL リファレンス』を参照してください。

レコード間の移動と操作

TDBNavigator を使うと，ユーザーはデータセットのレコード間の移動とレコードの操作を簡単に制御できます。ナビゲータは一連のボタンで構成されます。ナビゲータのボタンを使うと，ユーザーは一度に 1 件ずつレコードを前後にスクロールできます。また，最初のレコードへの移動，最後のレコードへの移動，新規レコードの挿入，既存のレコードの更新，データ変更の登録，データ変更の取り消し，レコードの削除，レコード表示の更新などもできます。

図 19.6 に，設計時にフォームへ配置したときにデフォルトで表示されるナビゲータを示します。ナビゲータの各ボタンは，データセットのレコード間の移動やレコードの編集，削除，挿入，登録をユーザーが行えるようにするものです。ナビゲータの VisibleButtons プロパティを使うと，このようなボタンの一部を動的に表示したり非表示にしたりできます。

図 19.6　TDBNavigator **コントロールのボタン**



次の表にナビゲータのボタンを示します。

表 19.8　TDBNavigator **のボタン**

ボタン	目的
最初のレコード	データセットの First メソッドを呼び出し，最初のレコードを現在のレコードに設定する
前のレコード	データセットの Prior メソッドを呼び出し，前のレコードを現在のレコードに設定する
次のレコード	データセットの Next メソッドを呼び出し，次のレコードを現在のレコードに設定する
最後のレコード	データセットの Last メソッドを呼び出し，最後のレコードを現在のレコードに設定する
レコードの挿入	データセットの Insert メソッドを呼び出し，現在のレコードの直前に新規レコードを挿入し，データセットを挿入状態にする
現在のレコードの削除	現在のレコードを削除する。ConfirmDelete プロパティが **true** の場合，削除する前に確認のメッセージが表示される
現在のレコードの編集	現在のレコードを変更できるように，データセットを編集状態にする
レコード編集の登録	現在のレコードの変更内容をデータベースに書き込む
レコード編集の取り消し	現在のレコードの編集内容を取り消し，データセットを参照状態に戻す
レコードの更新	データコントロールの表示バッファをクリアし，物理テーブルや問い合わせによってそのバッファを更新する。基礎となるデータが別のアプリケーションで変更された可能性がある場合に役立つ

表示するナビゲータボタンの選択

設計時に初めてフォームに TDBNavigator を配置したときは，そのすべてのボタンが表示されます。VisibleButtons プロパティを使うと，フォームで使わないボタンを非表示にできます。たとえば，単方向データセットを扱う場合には，意味があるのは，最初のレコード，次のレコード，更新の 3 つのボタンだけです。編集を行わずに参照だけを行うフォームでは，編集，挿入，削除，登録，および取り消しのボタンを非表示にできます。

設計時のナビゲータボタンの消去と表示

VisibleButtons プロパティは，オブジェクトインスペクタで +（正符号）付きで表示されます。これは，VisibleButtons プロパティを展開して，ナビゲータの各ボタンの論理値を表示できることを表します。これらの論理値を表示して設定するには［+］をクリックします。オブジェクトインスペクタの VisibleButtons プロパティの下に，ボタンのリストが表示されます。ここで，各ボタンの表示と非表示を切り替えられます。+ は -（負符号）に変化します。クリックすれば，プロパティのリストを折りたたむことができます。

ボタンが表示されるかどうかは，ボタンの値の論理状態で示されます。値が **true** の場合，ボタンは TDBNavigator に表示されます。**false** の場合，設計時にも実行時にも，ボタンはナビゲータから削除されます。

メモ あるボタンの値を false に設定すると，そのボタンはフォームの TDBNavigator から削除され，残りのボタンがこのコントロールの幅全体に広がります。コントロールのハンドルをドラッグすれば，ボタンのサイズを変更できます。

実行時のナビゲータボタンの消去と表示

ユーザーの操作やアプリケーションの状態に応じて，実行時に特定のナビゲータボタンを消去することも表示することもできます。たとえば，単一のナビゲータを使って 2 種類のデータセットでレコード間の移動をする場合を考えます。一方のデータセットはユーザーのレコード編集が可能で，もう一方のデータセットは読み出し専用とした場合，挿入，削除，登録，編集，取り消し，更新の各ナビゲータボタンは，読み出し専用のデータセットでは消去し，レコード編集が可能なデータセットでは表示するようにします。

たとえば，OrdersTable では，挿入，削除，編集，登録，取り消し，更新のナビゲータボタンを消去して編集ができないようにし，CustomersTable では編集ができるようにすることが考えられます。ナビゲータにどのボタンを表示するかは，VisibleButtons プロパティによって決まります。次にイベントハンドラの記述例を示します。

```
void __fastcall TForm1::CustomerCompanyEnter(TObject *Sender)
{
  if (Sender == (TObject *)CustomerCompany)
  {
    DBNavigatorAll->DataSource = CustomerCompany->DataSource;
    DBNavigatorAll->VisibleButtons =
        TButtonSet() << nbFirst << nbPrior << nbNext << nbLast;
  }
  else
  {
    DBNavigatorAll->DataSource = OrderNum->DataSource;
```

```
    DBNavigatorAll->VisibleButtons = TButtonSet() << nbInsert << nbDelete << nbEdit
                                   << nbPost << nbCancel << nbRefresh;
  }
}
```

ヘルプヒントの表示

実行時にナビゲータの各ボタンについてのヘルプヒントを表示するには，ナビゲータの ShowHint プロパティを true に設定します。ShowHint が true の場合，マウスカーソルがナビゲータのボタンの上にくると，ヘルプヒントが表示されます。ShowHint のデフォルト値は false です。

Hints プロパティは，各ボタンのヘルプヒントのテキストを制御します。デフォルトでは，Hints は空の文字列リストです。Hints が空の場合，各ナビゲータボタンでデフォルトのヘルプテキストが表示されます。ナビゲータのボタンのヘルプヒントをカスタマイズするには，文字列リストエディタを使って，Hints プロパティで各ボタンに対する別のヒントテキストを入力します。入力した文字列は，ナビゲータコントロールが提供するデフォルトのヒントに優先します。

複数のデータセットに単一のナビゲータを使う

ほかのデータベース対応コントロールの場合と同じように，ナビゲータの DataSource プロパティは，そのコントロールをデータセットにリンクするデータソースを指定します。実行時にナビゲータの DataSource プロパティを変更すると，単一のナビゲータを使って，複数のデータセットでレコード間の移動とレコードの操作ができます。

フォームに 2 つの編集コントロールがあり，CustomersSource と OrdersSource の各データソースによってそれぞれ CustomersTable と OrdersTable のデータセットにリンクされているとします。CustomersSource に接続されている編集コントロールにユーザーが移動すると，ナビゲータも CustomersSource を使わなければなりません。OrdersSource に接続されている編集コントロールにユーザーが移動すると，ナビゲータも OrdersSource に切り替えなければなりません。編集コントロールの一方に対して OnEnter イベントハンドラを記述して，もう一方の編集コントロールとそのイベントを共有できます。例を示します。

```
void __fastcall TForm1::CustomerCompanyEnter(TObject *Sender)
{
  if (Sender == (TObject *)CustomerCompany)
    DBNavigatorAll->DataSource = CustomerCompany->DataSource;
  else
    DBNavigatorAll->DataSource = OrderNum->DataSource;
}
```

第20章

デシジョンコンポーネントの使い方

デシジョン（意思決定支援）コンポーネントを使うと，クロス集計の表やグラフを簡単に作成できます。これらの表やグラフを使えば，何通りもの観点からデータの表示や集計ができます。データのクロス集計についての詳細は，20-2 ページの「クロス集計について」を参照してください。

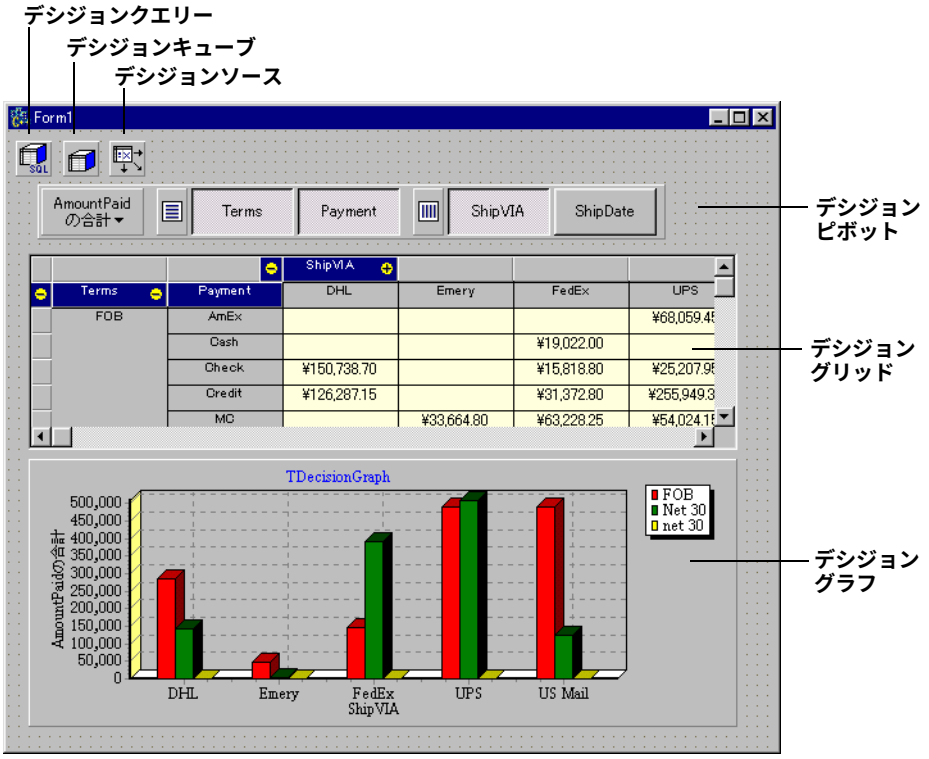
概要

デシジョンコンポーネントはコンポーネントパレットの［Decision Cube］ページに表示されます。

- デシジョンキューブ TDecisionCube は多次元のデータストア
- デシジョンソース TDecisionSource はデシジョングリッドまたはデシジョングラフの現在のピボット状態を定義する
- デシジョンクエリー TDecisionQuery はデシジョンキューブ内のデータを定義するために使用される TQuery の 特殊な形式
- デシジョンピボット TDecisionPivot により，ボタンを押してデシジョンキューブの次元やフィールドを開閉できる
- デシジョングリッド TDecisionGrid は 1 次元および複数次元のデータを表形式で表示する
- デシジョングラフ TDecisionGraph は，デシジョングリッドの項目を動的グラフとして表示する。このグラフはデータの次元が変更されるとそれに応じて変化する

図 20.1 は，設計時にすべてのデシジョンコンポーネントがフォーム上にどのように配置されているかを示しています。

図 20.1　設計時のデシジョンコンポーネント



クロス集計について

クロス集計は，データの部分集合を互いの関係や傾向がよくわかるように表す方法です。クロス集計では表の項目が次元に相当し，項目値は同一次元内のカテゴリと集計を定義します。

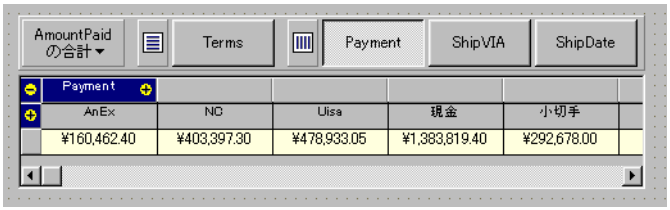
デシジョンコンポーネントを使うとフォーム内にクロス集計を設定できます。TDecisionGrid はデータを表形式で示し，TDecisionGraph はグラフ形式で示します。TDecisionPivot のボタンを押すと次元の表示 / 非表示，列や行間の次元の移動が簡単にできます。

クロス集計は単次元でも多次元でも可能です。

単次元クロス集計

単次元クロス集計は，1 つの次元のカテゴリ別集計行（または列）を示します。たとえば，Payment を列次元，Amount Paid を集計カテゴリとすると，図 20.2 のクロス集計は，それぞれの方法での支払い額を示しています。

図 20.2　単次元クロス集計

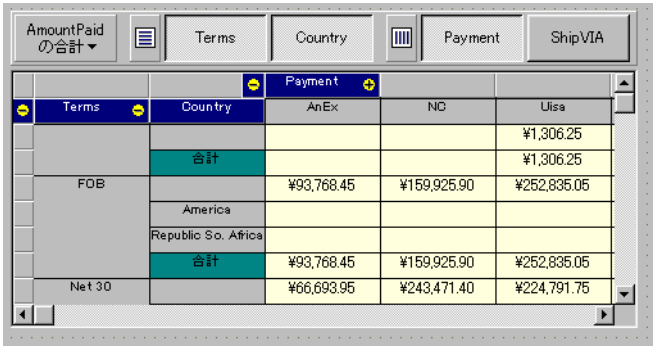


多次元クロス集計

多次元クロス集計は行または列に複数の次元を使います。たとえば，2 次元クロス集計では，各国の支払い方法に従って支払い額を表示できます。

3 次元クロス集計は，図 20.3 に示すように，各国の支払い方法と支払い条件に従って支払い額を表示できます。

図 20.3　3 次元クロス集計



デシジョンコンポーネントの使い方

20-1 ページに示すデシジョンコンポーネントは，多次元データを表やグラフとして表すのに使えます。データセットごとに複数のグリッドやグラフを結合できます。TDecisionPivot の複数のインスタンスを使うと，実行時にさまざまな観点からデータを表示できます。

多次元データの表やグラフを含むフォームを作成する手順は次のとおりです。

1. フォームを作成します。
2. 以下のコンポーネントをフォームに追加し，オブジェクトインスペクタを使って以下のとおりに結合します。
   - データセット。通常は，TDecisionQuery（詳細は 20-6 ページの「デシジョンクエリーエディタを使ったデシジョンデータセットの作成」を参照）または TQuery を使用する
   - デシジョンキューブ TDecisionCube。DataSet プロパティにデータセット名を設定してそのデータセットに結合する

- デシジョンソース TDecisionSource。DecisionCube プロパティにデシジョンキューブの名前を設定してそのデシジョンキューブに結合する

3. デシジョンピボット TDecisionPivot を追加し，オブジェクトインスペクタで DecisionSource プロパティに適切なデシジョンソース名を設定して，そのデシジョンソースに結合します。デシジョンピボットは省略できますが使うと便利です。たとえばフォーム開発者やエンドユーザーはデシジョンピボットのボタンを押すだけで，デシジョングリッドやデシジョングラフに表示する次元を変更できます。

   デフォルトの方向（水平方向）では，デシジョンピボットの左側のボタンがデシジョングリッドの左端の項目（行）を制御し，右側のボタンがデシジョングリッドの上端の項目（列）を制御します。

   デシジョンピボットのボタンの表示位置を指定するには，その GroupLayout プロパティを xtVertical，xtLeftTop，xtHorizontal（デフォルト）のいずれかに設定します。デシジョンピボットのプロパティについては，20-9 ページの「デシジョンピボットの使い方」を参照してください。

4. 1 つまたは複数のデシジョングリッドとデシジョングラフを追加し，デシジョンソースに結合します。詳細は，20-10 ページの「デシジョングリッドの作成と使い方」および 20-12 ページの「デシジョングラフの作成と使い方」を参照してください。

5. デシジョンクエリーエディタを使うか TDecisionQuery（または TQuery）の SQL プロパティを使って，グリッドまたはグラフに表示するテーブル，項目，集計を指定します。SQL SELECT 句の最後の項目は集計項目でなければなりません。SELECT 句のほかの項目は GROUP BY 項目でなければなりません。手順については，20-6 ページの「デシジョンクエリーエディタを使ったデシジョンデータセットの作成」を参照してください。

6. デシジョンクエリー（またはかわりのデータセットコンポーネント）の Active プロパティを true に設定します。

7. デシジョングリッドとデシジョングラフを使って，さまざまな次元のデータを表形式とグラフ形式で表示します。手順とアドバイスについては，20-10 ページの「デシジョングリッドの使い方」および 20-13 ページの「デシジョングラフの使い方」を参照してください。

フォーム上のすべてのデシジョンサポートコンポーネントについては，20-2 ページの 図 20.1 を参照してください。

デシジョンコンポーネントでのデータセットの使い方

デシジョンキューブ TDecisionCube だけがデータセットと直接結合できます。TDecisionCube では、SQL 文によりグループ化または集計された受け入れ可能なデータが与えられることを仮定しています。GROUP BY 句には SELECT 句と同じ非集計項目が同じ順序で入っていなければならず，集計項目は識別されている必要があります。

デシジョンクエリーコンポーネント TDecisionQuery は，TQuery の特殊な形式です。TDecisionQuery を使うと，デシジョンキューブ TDecisionCube にデータを割り当てるのに使う次元（行と列）と集計値の設定をより簡単に定義できます。通常の TQuery やほかの BDE 対応のデータセットも TDecisionCube 用のデータセットとして使えますが，その場合は設計者がデータセットと TDecisionCube を自分で適切に設定しなければなりません。

デシジョンキューブを正しく操作するには，データセット内のすべての設計項目を次元または集計にしなければなりません。集計は累積値（合計または件数など）であり，次元値の組み合わせごとの合計を表します。設定をできるだけ簡単にするには，データセットで合計には「Sum...」，件数には「Count...」という名前を付けます。

デシジョンキューブは，そのセルが累積である集計の値に対してのみ正しくピボット，小計，およびドリルインできます（SUM と COUNT は累積で，AVERAGE，MAX，および MIN は累積ではない）。ピボットクロス集計は，累積集合子だけを含むグリッドでのみ表示されます。非累積集合子を使用している場合，ピボット，ドリル，または小計を行わない静的なデシジョングリッドを使います。

平均は SUM を COUNT で除算して計算できるので，項目の SUM と COUNT の値がデータセットに含まれている場合，ピボット平均は自動的に追加されます。AVERAGE 文を使って計算される平均よりも，むしろこの種類の平均を使用してください。

また，平均は COUNT(*) を使っても計算できます。COUNT(*) を使って平均を計算するには，問い合わせに「COUNT(*) COUNTALL」選択肢を含めます。平均を計算するために COUNT(*) を使う場合，すべての項目について 1 つの集合子を使えます。集計される項目が空白値を含まない場合，またはすべての項目に対して COUNT 集合子を利用できない場合だけ，COUNT(*) を使います。

TQuery または TTable を使ったデシジョンデータセットの作成

通常の TQuery コンポーネントをデシジョンデータセットとして使う場合は，必ず SELECT 句と同じ項目が同じ順序で入っている GROUP BY 句を指定するように SQL 文を自分で設定しなければなりません。

SQL の例を次に示します。

```
SELECT ORDERS."Terms", ORDERS."ShipVIA",
  ORDERS."PaymentMethod", SUM( ORDERS."AmountPaid" )
FROM "ORDERS.DB" ORDERS
GROUP BY ORDERS."Terms", ORDERS."ShipVIA", ORDERS."PaymentMethod"
```

SELECT 項目の順序は GROUP BY 項目の順序と一致していなければなりません。

TTable を使う場合には，問い合わせのうちどれがグループ化項目でどれが集計かをデシジョンキューブに指定しなければなりません。これを行うには，デシジョンキューブの DimensionMap で各項目に対して［タイプ］を指定します。各項目が次元かまたは集計かを指定する必要があります。集計の場合，集計の種類を指定する必要があります。ピボット平均は SUM/COUNT 計算に依存するため，デシジョンキューブに SUM と COUNT 集合子のペアと一致させるために基本となる項目の名前も指定しなければなりません。

デシジョンクエリーエディタを使ったデシジョンデータセットの作成

デシジョンコンポーネントが使うすべてのデータは，SQL 問い合わせが最も容易に生成する一定の形式のデータを受け入れるデシジョンキューブに渡されます。詳細は，20-4 ページの「デシジョンコンポーネントでのデータセットの使い方」を参照してください。

TTable と TQuery のどちらもデシジョンデータセットとして使えますが，TDecisionQuery を使う方が簡単です。このコンポーネントと一緒に提供されるデシジョンクエリーエディタを使うと，デシジョンキューブに表示するテーブル，項目，集計を指定できるほか，SQL の SELECT 句と GROUP BY 句を正しく設定できます。

デシジョンクエリーエディタを使う手順は次のとおりです。

1. フォームのデシジョンクエリーコンポーネントを選択します。次に右クリックし，［デシジョン SQL の設定］を選択します。［デシジョン SQL の設定］ダイアログボックスが表示されます。
2. 使いたいデータベースを選択します。
3. 単一テーブルの問い合わせの場合は［テーブル］コンボボックスから選択します。

   複数テーブルの結合を含む複雑な問い合わせの場合は，［クエリービルダ］ボタンをクリックして SQL ビルダを表示するか，SQL タブページの編集ボックスに SQL 文を入力してください。
4. ［デシジョン SQL の設定］ダイアログボックスに戻ります。
5. ［デシジョン SQL の設定］ダイアログボックスの［選択可能な項目の一覧］リストボックスの項目を選択し，該当する右矢印ボタンをクリックしてその項目を［次元］または［集計］に割り当てます。［集計］リストに項目を追加するときに，表示されるメニューから使用する集計の種類（合計，件数，平均）を選択します。
6. デフォルトの場合，デシジョンクエリーの SQL プロパティで定義されるすべての項目と集計は［次元］リストボックスと［集計］リストボックスに表示されます。次元または集計を削除するには，リストでその項目を選択してリストの横にある左矢印をクリックするか，その項目をダブルクリックします。元に戻すには［選択可能な項目の一覧］リストボックスでその項目を選択し，該当する右矢印をクリックします。

デシジョンキューブの内容を定義したら，DimensionMap プロパティと TDecisionPivot のボタンを使って次元表示を詳細に操作できます。詳細は，「デシジョンキューブの使い方」20-8 ページの「デシジョンソースの使い方」，20-9 ページの「デシジョンピボットの使い方」を参照してください。

メモ デシジョンクエリーエディタを使う場合，問い合わせは最初に ANSI-92 SQL 構文で処理され，次にサーバー固有の構文に（必要ならば）変換されます。デシジョンクエリーエディタは ANSI 標準 SQL のみ読み込んで表示します。変換された文は自動的にサーバー固有の構文に変換され，TDecisionQuery の SQL プロパティへ代入されます。問い合わせを変更するには，SQL プロパティではなくデシジョンクエリーエディタで ANSI-92 標準構文の方を編集します。

デシジョンキューブの使い方

デシジョンキューブコンポーネント TDecisionCube はデータセットからデータを検索する多次元のデータストアで，通常は TDecisionQuery または TQuery 経由で入力される特殊な構造の SQL 文です。データは，もう一度問い合わせを実行しなくても簡単にピボット（データの構成および集計方法）の変更が行える形式で格納されます。

デシジョンキューブのプロパティとイベント

TDecisionCube の DimensionMap プロパティは，どの次元や集計を表示するかを制御するだけでなく，日付範囲を設定し，デシジョンキューブがサポートできる次元の最大数を指定します。設計時にデータを表示するかどうかも指定できます。名前，値（カテゴリ別），小計，データなどを表示できます。ただし設計時にデータを表示すると，データソースによっては時間がかなりかかることがあります。

オブジェクトインスペクタの DimensionMap の横にある省略記号をクリックすると，［デシジョンキューブの設定］ダイアログボックスが表示されます。このダイアログボックスのページとコントロールを使うと，DimensionMap プロパティを設定できます。

デシジョンキューブのキャッシュが再構築されるたびに，OnRefresh イベントが起動します。このとき開発者は新しい次元マップにアクセスしてそのマップを変更することにより，メモリを解放したり，集計や次元の最大数を変更したりできます。ユーザーがデシジョンキューブエディタにアクセスする場合にも OnRefresh は役立ちます。このとき，ユーザーの変更にアプリケーションコードが応答できます。

デシジョンキューブエディタの使い方

デシジョンキューブエディタを使うと，デシジョンキューブの DimensionMap プロパティを設定できます。［デシジョンキューブの設定］ダイアログボックスは，直前の節で説明したようにオブジェクトインスペクタから表示することも，設計時にフォーム上でデシジョンキューブを右クリックしてから［デシジョンキューブの設定］を選択して表示することもできます。

［デシジョンキューブの設定］ダイアログボックスには以下の 2 つのタブがあります。

- ［次元の設定］は使用可能な次元のアクティブ / 非アクティブの設定，次元の名前と形式の変更，次元の固定状態化，表示する日付範囲の設定などに使う
- ［メモリ管理］は一度にアクティブにできる次元と集計の最大数を設定するのに使う。またメモリの使用状況に関する情報を表示，および設計時に表示する名前とデータを決定するのにも使う

次元設定を表示して変更する

次元の設定を表示するには，デシジョンキューブエディタを表示し，［次元の設定］タブをクリックします。次に［選択可能な項目］リストで次元または集計を選択します。その情報がエディタの右端にあるボックス内に表示されます。

- デシジョンピボット，デシジョングリッド，デシジョングラフに表示する次元または集計の名前を変更するには，［表示名］編集ボックスに新しい名前を入力する

- 選択した項目が次元か集計のどちらであるかを決定するには，［タイプ］編集ボックスのテキストを読む。データセットが TTable コンポーネントの場合，［タイプ］編集ボックスを使って，選択した項目が次元か集計かを指定できる
- 選択した次元または集計をアクティブまたは非アクティブにするには，［有効化の種類］ドロップダウンリストボックスの設定を変更する（［有効］，［必要に応じて］，［停止］）。次元を非アクティブにするか［必要に応じて］に設定するとメモリを節約できる
- 次元または集計の形式を変更するには，［書式］編集ボックスに形式文字列を入力する
- 次元または集計を年単位，四半期単位，月単位で表示するには，［グループ化］ドロップダウンリストボックスの設定を変更する。［グループ化］リストボックスで［固定値］を選択すると，選択した次元または集計を「固定した」ままの状態にできる。このように設定すると，次元の値の数が多いときにメモリを節約できる。詳細については，20-19 ページの「デシジョンコンポーネントとメモリ制御」を参照してください。
- 範囲の開始値または「固定した」次元の値を決めるには，まず［グループ化］ドロップダウンで適切なグループ化を選択し，次に［初期値］ドロップダウンリストに範囲の開始値かドリルダウンのための固定値を入力する

使用可能な次元と集計の最大数を設定する

選択したデシジョンキューブに結合したデシジョンピボット，デシジョングリッド，デシジョングラフで使用可能な次元と集計の最大数を指定するには，デシジョンキューブエディタを表示し，［メモリ管理］タブをクリックします。必要であれば編集コントロールを使って現在の設定を調整します。これらの設定を使うと，デシジョンキューブに必要なメモリの量を制御できます。詳細については，20-19 ページの「デシジョンコンポーネントとメモリ制御」を参照してください。

設計時オプションの表示と変更

設計時に表示する情報を決めるには，デシジョンキューブエディタを表示して［メモリ管理］タブをクリックします。次に，設計時オプションから表示したい情報を選択します。設計時にデータや項目名を表示すると，データを取得するための時間が必要になり，パフォーマンスが低下する場合があります。

デシジョンソースの使い方

デシジョンソースコンポーネント TDecisionSource はデシジョングリッドまたはデシジョングラフの現在のピボット状態を定義します。2 つのオブジェクトが同じデシジョンソースを使う場合，ピボット状態も同じです。

プロパティとイベント

デシジョンソースの外観と動作を制御する特殊なプロパティとイベントには以下のものがあります。

- TDecisionSource の ControlType プロパティは，デシジョンピボットのボタンがチェックボックス（一度に複数の選択が可能）またはラジオボタン（一度に 1 つしか選択できない）のどちらの機能を持つかを示す

- TDecisionSource の SparseCols プロパティと SparseRows プロパティは空の列または行を表示するかどうか示す。**true** ならば，空の行または列も含めて表示される
- TDecisionSource のイベントは以下のとおり
  - OnLayoutChange は，データの再編成を伴うピボット移動または固定化をユーザーが実行すると発生する
  - OnNewDimensions は，データが完全に変更されると（集計項目または次元項目が変更される場合など）発生する
  - OnSummaryChange は現在の集計が変化すると発生する
  - OnStateChange は，デシジョンキューブがアクティブまたは非アクティブになると発生する
  - OnBeforePivot は，変更内容がコミットされたがユーザーインターフェースにまだ反映されていないと発生する。開発者はアプリケーションユーザーに直前の動作結果を表示する前に，容量やピボット状態などを変更できる
  - OnAfterPivot はピボット状態が変化すると発生する。開発者はこのときに情報を取得できる

デシジョンピボットの使い方

デシジョンピボットコンポーネント TDecisionPivot により，ボタンを押すことで，デシジョンキューブの次元や項目を開閉できるユーザーインターフェースを提供できます。TDecisionPivot ボタンを押して行または列を開くと，対応する次元が TDecisionGrid コンポーネントまたは TDecisionGraph コンポーネントに表示されます。次元を閉じると，その詳細なデータは表示されずにほかの次元の合計の中に折りたたまれます。次元は「固定」状態にもできます。この場合，次元項目の特定の値の集計だけが表示されます。

デシジョンピボットを使うと，デシジョングリッドやデシジョングラフに表示する次元の再編成もできます。再編成は，ボタンを行領域または列領域までドラッグするか，同じ領域内でボタンの順序を入れ替えるだけでできます。

設計時のデシジョンピボットについては，図 20.1，20.2，20.3 を参照してください。

デシジョンピボットのプロパティ

デシジョンピボットの外観と動作を制御する特殊なプロパティには以下のものがあります。

- TDecisionPivot で最初に表示されるプロパティは，全体の動作と外観を定義する。TDecisionPivot の ButtonAutoSize を false に設定すると，コンポーネントのサイズを調整するときボタンの大きさが固定される
- TDecisionPivot の Groups プロパティはどの次元ボタンを表示するかを定義する。行，列，集計の選択ボタングループはどのような組み合わせでも表示できる。これらのグループをもっと自由に配置するには，フォーム上で行だけ入っている TDecisionPivot をある位置に割り当て，次に列だけ入っている TDecisionPivot を別の位置に割り当てる

- 通常の場合，TDecisionPivot は TDecisionGrid の上に追加される。デフォルトの方向（水平方向）では，TDecisionPivot の左側のボタンが TDecisionGrid の左端の項目（行）を制御し，右側のボタンが TDecisionGrid の上端の項目（列）を制御する
- TdecisionPivot のボタンの表示位置を指定するには，その GroupLayout プロパティを xtVertical, xtLeftTop，xtHorizontal（デフォルト設定については上の段落を参照）のいずれかに設定する

デシジョングリッドの作成と使い方

デシジョングリッドコンポーネント TDecisionGrid はクロス集計したデータを表形式で表示します。これらのテーブルはクロス集計と呼ばれ，20-2 ページに説明があります。20-2 ページの 図 20.1 に設計時のフォーム上のデシジョングリッドを示しています。

デシジョングリッドの作成

クロス集計データが入った 1 つまたは複数の表を含むフォームを作成する手順は次のとおりです。

1. 20-3 ページの「デシジョンコンポーネントの使い方」の 1 ～ 3 の手順に従います。
2. 1 つまたは複数のデシジョングリッドコンポーネント（TDecisionGrid）を追加し，デシジョンソース TDecisionSource に結合します。この場合，オブジェクトインスペクタで DecisionSource プロパティに適切なデシジョンソースコンポーネントを設定します。
3. 「デシジョンコンポーネントの使い方」の手順 5 ～ 7 を実行します。

デシジョングリッドの表示内容の説明とその使用法については，20-10 ページの「デシジョングリッドの使い方」を参照してください。

グラフをフォームに追加するには，20-12 ページの「デシジョングラフの作成」の手順に従います。

デシジョングリッドの使い方

デシジョングリッドコンポーネント TDecisionGrid は，デシジョンソース（TDecisionSource）に結合されたデシジョンキューブ（TDecisionCube）のデータを表示します。

デフォルトでは，データセットで定義されたグループ化命令に応じて，グリッドの左端と上端のどちらか一方あるいは両方に次元項目が表示されます。各項目の下にはデータ値ごとにカテゴリが 1 つずつ表示されます。以下の操作ができます。

- 次元を開く / 閉じる
- 行と列の再編成（ピボット）
- 詳細を表示するための固定化
- 次元選択の制限（軸ごとに 1 つの次元）

デシジョングリッドの特殊なプロパティとイベントについての詳細は，20-11 ページの「デシジョングリッドのプロパティ」を参照してください。

デシジョングリッドの項目を開く / 閉じる

次元項目または集計項目にプラス記号（+）が表示されている場合，その右にある 1 つまたは複数の項目は閉じています（隠れています）。プラス記号をクリックすれば，右にある項目やカテゴリを開くことができます。マイナス記号（-）が表示されている場合，その項目は完全に開いています（展開されています）。マイナス記号をクリックすると，項目が閉じます。このアウトライン機能は使用不可にできます。詳細は 20-11 ページの「デシジョングリッドのプロパティ」を参照してください。

デシジョングリッドの行と列を再編成する

行見出しや列見出しは，同じ軸内の新しい位置へでもほかの軸へでもドラッグできます。これにより，データのグループ化の変更に合わせて，グリッドを再編成し，新しい観点でデータを表示できます。このピボット機能は使用不可にできます。詳細は 20-11 ページの「デシジョングリッドのプロパティ」を参照してください。

デシジョンピボットを入れておけば，そのボタンを押してドラッグするだけで表示を再編成できます。詳細は 20-9 ページの「デシジョンピボットの使い方」を参照してください。

デシジョングリッドで次元を固定して詳細を表示する

ある次元を固定して詳細を表示できます。

たとえば，ほかの次元がその下に折りたたまれている次元のカテゴリラベル（行見出し）を右クリックすると，特定のカテゴリに固定してそのカテゴリのデータだけを表示することができます。次元を固定すると，1 つのカテゴリ値のレコードだけが表示されるため，グリッドに表示されるその次元のカテゴリラベルは表示されなくなります。フォームにデシジョンピボットを追加すると，カテゴリ値が表示されるので，必要であればほかの値を変更できます。

1 つの次元に固定する方法は次のとおりです。

- カテゴリラベルを右クリックし，［この値でドリルイン］を選択する
- ピボットボタンを右クリックし，［ドリルイン］を選択する

次元全体をアクティブに戻すには，次の方法があります。

- 対応するピボットボタンを右クリックするか，デシジョングリッドの左上隅を右クリックしてから次元を選択する

デシジョングリッドの次元選択を制限する

デシジョングリッドの軸ごとに複数の次元を選択できるようにするかどうか指定するには，デシジョンソースの ControlType プロパティを変更します。詳細については，20-8 ページの「デシジョンソースの使い方」を参照してください。

デシジョングリッドのプロパティ

デシジョングリッドコンポーネント TDecisionGrid は，TDecisionSource に結合された TDecisionCube コンポーネントのデータを表示します。デフォルトでは，データがグリッド内に表示され，カテゴリ項目がグリッドの左端と上端に表示されます。

デシジョングリッドの外観と動作を制御する特殊なプロパティには以下のものがあります。

- TDecisionGrid には次元ごとに固有のプロパティがある。これらのプロパティを設定するには，オブジェクトインスペクタで Dimensions を選択してから次元を選択する。オブジェクトインスペクタにそのプロパティが表示される。Alignment はその次元のカテゴリラベルの位置合わせを定義する。Caption を使うとデフォルトの次元名を変更できる。Color はカテゴリラベルの色を定義する。FieldName はアクティブな次元の名前を表示する。Format はそのデータ型の標準形式を保持できる。Subtotals はその次元の小計を表示するかどうかを指定する。集計項目では，これらと同じプロパティを使って，グリッドの集計領域に表示するデータの外観を変更する。次元プロパティの設定がすべて済んだら，フォームのコンポーネントをクリックするか，オブジェクトインスペクタの上端にあるドロップダウンリストボックスのコンポーネントを選択する
- TDecisionGrid の Options プロパティを使うと，グリッド線の表示の制御（cgGridLines = **true**），アウトライン機能（+ 記号と - 記号を使って次元を折りたたんだり展開したりする機能）の設定（cgOutliner = **true**），ドラッグアンドドロップピボットの設定（cgPivotable = **true**）ができる
- TDecisionGrid の OnDecisionDrawCell イベントを使うと，描画時に各セルの外観を変更できる。このイベントは参照パラメータとして現在のセルの String，Font，Color を渡す。負の値を特殊な色で表すなどの効果が得られるように，これらのパラメータは変更できる。TCustomGrid が渡す Drawstate のほかに，TDecisionDrawState も渡される。これはどの種類のセルを描画するか指定する。セルに関する情報は，関数 Cells，CellValueArray，CellDrawState を使って取り出せる
- TDecisionGrid の OnDecisionExamineCell イベントを使うと，右クリックイベントをデータセルに関連付けることができる。このイベントは，特定のデータセルに関する情報（詳細記録など）をプログラムが表示できるようにする。ユーザーがデータセルを右クリックすると，データ値の作成に使ったすべての情報（現在アクティブな集計値など）や集計値の作成に使ったすべての次元値の ValueArray がイベントに提供される

デシジョングラフの作成と使い方

デシジョングラフコンポーネント TDecisionGraph は，クロス集計データをグラフ形式で表示します。デシジョングラフは 1 つの集計値（Sum，Count，Avg など）を単次元または多次元のグラフ形式で表示します。クロス集計の詳細は 20-2 ページを参照してください。設計時のデシジョングラフについては，20-2 ページの 図 20.1 ，20-14 ページの 図 20.4 を参照してください。

デシジョングラフの作成

1 つまたは複数のデシジョングラフを含むフォームを作成する手順は次のとおりです。

1. 20-3 ページの「デシジョンコンポーネントの使い方」の 1 ～ 3 の手順に従います。
2. 1 つまたは複数のデシジョングラフコンポーネント（TDecisionGraph）を追加し，デシジョンソース TDecisionSource に結合します。この場合，オブジェクトインスペクタで DecisionSource プロパティに適切なデシジョンソースコンポーネントを設定します。
3. 「デシジョンコンポーネントの使い方」の手順 5 ～ 7 を実行します。
4. 最後にグラフを右クリックし，［チャートの編集］を選択してグラフ系列の外観を変更します。グラフ次元ごとにテンプレートプロパティを設定してから，個々の系列プロパティを設定してデ

フォルトを変更できます。詳細は，20-15 ページの「デシジョングラフのカスタマイズ」を参照してください。

デシジョングラフの表示内容の説明とその使用法については，「デシジョングラフの使い方」を参照してください。

フォームにデシジョングリッドまたはクロス集計テーブルを追加するには，20-10 ページの「デシジョングリッドの作成と使い方」の指示に従います。

デシジョングラフの使い方

デシジョングラフコンポーネント TDecisionGraph は，デシジョンソース（TDecisionSource）の項目を動的グラフとして表示します。このグラフはデシジョンピボット（TdecisionPivot）を使ったデータ次元の開閉，ドラッグアンドドロップ，再編成に応じて変化します。

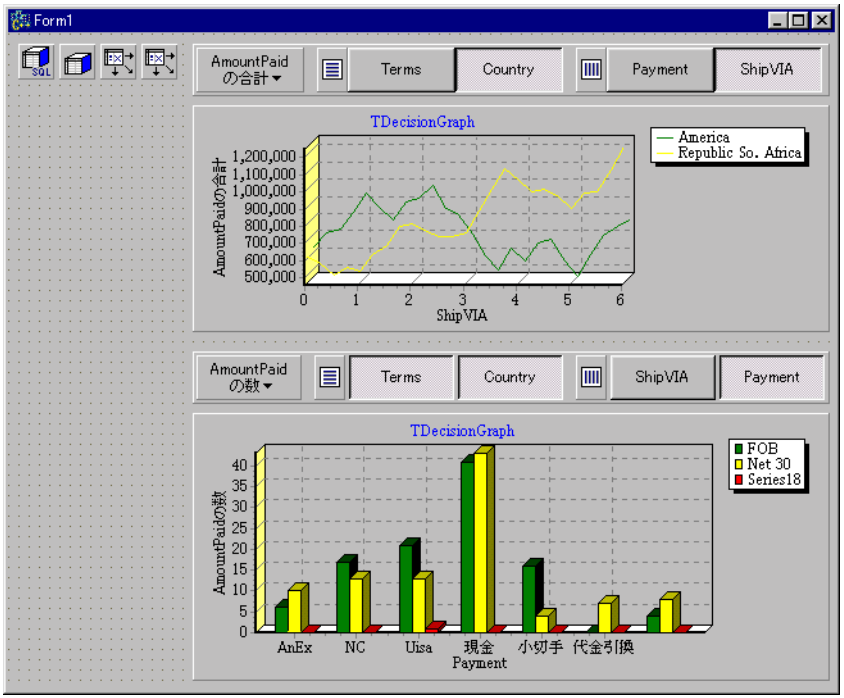
グラフ化するデータは，特殊な形式のデータセット（TDecisionQuery など）から取り出します。デシジョンコンポーネントがこのデータを処理して配置する方法については，20-1 ページを参照してください。

デフォルトでは，最初の行次元は X 軸として表示され，最初の列次元は Y 軸として表示されます。

デシジョングリッドではクロス集計データを表形式で表示しますが，このデシジョングリッドのかわりとして，またはデシジョングリッドと一緒にデシジョングラフを使えます。同じデシジョンソースに結合されたデシジョングリッドとデシジョングラフは同じデータ次元を表します。同じ次元についてさまざまな集計データを表示するには，複数のデシジョングラフを同じデシジョンソースに結合します。さまざまな次元を表示するには，デシジョングラフを別々のデシジョンソースに結合します。

たとえば，図 20.4 では，最初のデシジョンピボットとデシジョングラフは最初のデシジョンソースに結合され，2 番目のデシジョンピボットとデシジョングラフは 2 番目のデシジョンソースに結合されています。このようにしてグラフごとに異なる次元を表示できます。

図 20.4　別々のデシジョンソースに結合されたデシジョングラフ



デシジョングラフの表示内容の説明とその使用法については，次の節を参照してください。「デシジョングラフの表示」

デシジョングラフの作成方法については，前節「デシジョングラフの作成」を参照してください。

デシジョングラフのプロパティとデシジョングラフの概観と動作の変更方法については，20-15 ページの「デシジョングラフのカスタマイズ」を参照してください。

デシジョングラフの表示

デフォルトのデシジョングラフでは，最初のアクティブな列項目の値（X 軸方向）に対して最初のアクティブな行項目のカテゴリ別の集計値（Y 軸方向）がグラフ化されます。グラフ化したカテゴリは，それぞれ独立した系列として表示されます。

次元を 1 つだけ選択した場合，たとえば 1 つの TDecisionPivot ボタンだけをクリックした場合などは，1 つの系列だけがグラフ化されます。

デシジョンピボットを使う場合，そのボタンを押してどのデシジョンキューブ項目（次元）をグラフ化するか指定できます。グラフ軸を変更するには，デシジョンピボットの次元ボタンを分割スペースの一方の端から反対側の端までドラッグします。単次元グラフですべてのボタンが分割スペースの一方の端にある場合，［行］アイコンまたは［列］アイコンをドロップ先として使って分割スペースの反対側にボタンを追加し，多次元グラフにできます。

1 回に 1 つの列と 1 つの行だけをアクティブにするには，TDecisionSource の ControlType プロパティを xtRadio に設定します。こうすると，デシジョンキューブ軸ごとに一度にアクティブにできる項目は 1 つだけとなり，デシジョンピボットの機能がグラフの動作に対応します。xtRadioEx は，xtRadio と同じ働きをしますが，すべての行またはすべての列の次元が閉じている状態にはできません。

デシジョングリッドとデシジョングラフの両方を同じ TDecisionSource に接続した場合，TDecisionGrid の動作の方が柔軟なため，それに合わせて ControlType の設定を xtCheck に戻す方がよいでしょう。

デシジョングラフのカスタマイズ

デシジョングラフコンポーネント TDecisionGraph は，デシジョンソース（TDecisionSource）の項目を動的グラフとして表示します。このグラフはデシジョンピボット（TdecisionPivot）を使ったデータ次元の開閉，ドラッグアンドドロップ，再編成に応じて変化します。デシジョングラフではその種類，色，折れ線グラフのマーカの種類などの各種のプロパティを変更できます。

グラフをカスタマイズする手順は次のとおりです。

1. グラフを右クリックし，［チャートの編集］を選択します。［チャートの編集］ダイアログボックスが表示されます。
2. ［チャートの編集］ダイアログボックスの［チャート］ページを使って可視系列のリストを表示し，同じ系列で複数の定義を利用できるときに使う系列定義を選択し，テンプレートまたは系列に合わせてグラフの種類を変更し，すべてのグラフプロパティを設定します。

   ［チャート］ページの［系列］リストには，デシジョンキューブのすべての次元（Template: が前に付いています）と現在の可視カテゴリが表示されます。カテゴリ（系列）はそれぞれ独立したオブジェクトです。以下の操作ができます。

   - 既存のデシジョングラフ系列から派生した系列の追加または削除。派生した系列は既存の系列に注釈を与えたり，ほかの系列で計算した値を表すことができる
   - デフォルトのグラフの種類の変更と，テンプレートや系列のタイトルの変更

   ［チャート］ページのその他のタブについては，オンラインヘルプトピック「［チャート］ページ（［DecisionGraph の編集］ダイアログボックス）」を参照してください。
3. ［系列］ページを使って次元テンプレートを設定し，グラフ系列ごとにプロパティをカスタマイズします。

デフォルトでは，すべての系列が棒グラフとして表示され，最大 16 色が割り当てられます。テンプレートの種類やプロパティを編集すれば新しいデフォルトを作成できます。デシジョンソースをさまざまな状態にピボットする場合，テンプレートを使って新しい状態ごとに動的に系列を作成できます。テンプレートの詳細は 20-16 ページの「デシジョングラフテンプレートのデフォルトを設定する」を参照してください。

系列ごとのカスタマイズについては，20-16 ページの「デシジョングラフ系列をカスタマイズする」を参照してください。

［系列］ページの各タブについては，オンラインヘルプのトピック「［系列］ページ（［DecisionGraph の編集］ダイアログボックス）」を参照してください。

デシジョングラフテンプレートのデフォルトを設定する

デシジョングラフは，デシジョンキューブの 2 つの次元の値を表示します。1 つはグラフ軸として表示され，もう 1 つは系列セットの作成に使います。後者の次元のテンプレートはこれらの系列のデフォルトプロパティ（系列が棒グラフ，折れ線グラフ，面グラフなどのどれかなど）を提供します。ユーザーがある状態から別の状態にピボットすると，テンプレートで指定された系列の種類やその他のデフォルトを使って，その次元に必要な系列がすべて作成されます。

アクティブな次元が 1 つだけの状態にユーザーがピボットすると，そのたびに別のテンプレートが提供されます。単次元状態は円グラフで表されることがよくあります。この場合は別のテンプレートが提供されます。

以下の操作ができます。

- グラフのデフォルトの種類の変更
- グラフテンプレートのその他のプロパティの変更
- グラフのすべてのプロパティの表示と設定

デシジョングラフのデフォルトの種類の変更

グラフのデフォルトの種類の変更

1. ［DecisionGraph の編集］ダイアログボックスの［チャート］ページにある［系列］リストでテンプレートを選択します。
2. ［変更］ボタンをクリックします。
3. 新しい種類を選択し，［TeeChart ギャラリー］ダイアログボックスを閉じます。

デシジョングラフテンプレートのその他のプロパティを変更する

デシジョングラフテンプレートのその他のプロパティ（色）を変更する

1. グラフエディタで［系列］ページを選択します。
2. ［系列］ページの上端にあるドロップダウンリストでテンプレートを選択します。
3. 該当するプロパティタブを選択し，設定を選択します。

デシジョングラフのすべてのプロパティを表示する

デシジョングラフの種類と系列以外のプロパティを表示し設定する手順は次のとおりです。

1. ［DecisionGraph の編集］ダイアログボックスの上端で［チャート］ページを選択します。
2. 該当するプロパティタブを選択し，設定を選択します。

デシジョングラフ系列をカスタマイズする

テンプレートはデシジョンキューブの次元ごとにいくつものデフォルト設定（グラフの種類，系列の表示方法など）を提供します。それ以外のデフォルト（系列の色など）は TDecisionGraph によって定義されます。必要であれば系列ごとにデフォルトを変更できます。

テンプレートは，必要になるとカテゴリの系列を作成し，不要になると破棄するようにしたい場合に使います。特定のカテゴリ値用にカスタム系列を設定することもできます。これには，グラフをピボットして，カスタマイズしたいカテゴリの系列が表示されるようにします。その系列がグラフに表示されたら，グラフエディタを使って以下の操作ができます。

- グラフの種類の変更
- 系列のその他のプロパティの変更
- カスタマイズした特定のグラフ系列の保存

系列テンプレートの定義とグラフのすべてのデフォルトの設定については，20-16 ページの「デシジョングラフテンプレートのデフォルトを設定する」を参照してください。

系列グラフの種類を変更する

デフォルトでは，各系列はその次元のテンプレートによって定義された同一のグラフの種類に設定されています。すべての系列を同じ種類のグラフに変更するには，テンプレートの種類を変更します。詳細は 20-16 ページの「デシジョングラフのデフォルトの種類の変更」を参照してください。

1 つの系列のグラフの種類を変更する手順は次のとおりです。

1. グラフエディタの［チャート］ページにある［系列］リストで系列を選択します。
2. ［変更］ボタンをクリックします。
3. 新しい種類を選択し，［TeeChart ギャラリー］ダイアログボックスを閉じます。
4. ［保存］チェックボックスにチェックマークを付けます。

デシジョングラフ系列のその他のプロパティを変更する

デシジョングラフ系列のその他のプロパティ（色など）を変更する手順は次のとおりです。

1. グラフエディタで［系列］ページを選択します。
2. ［系列］ページの上端にあるドロップダウンリストで系列を選択します。
3. 該当するプロパティタブを選択し，設定を選択します。
4. ［保存］チェックボックスにチェックマークを付けます。

デシジョングラフ系列の設定を保存する

デフォルトでは，設計時にはテンプレートの設定だけが保存されます。グラフエディタのダイアログで特定の系列の［保存］ボックスにチェックマークを付けると，その系列の変更内容だけが保存されます。

系列を保存するとメモリが多量に必要になります。したがって，保存の必要がない場合は［保存］ボックスのチェックマークははずします。

実行時のデシジョンコンポーネント

ユーザーは実行時に，可視デシジョンコンポーネントのクリック，右クリック，ドラッグにより多様な操作ができます。これらの操作についてはこの章の始めの方で説明しましたが，要約すると以下のようになります。

実行時のデシジョンピボット

ユーザーが実行できる操作は以下のとおりです。

- デシジョンピボットの左端にある集計ボタンをクリックして，使用可能な集計のリストを表示する。このリストを使うと，デシジョングリッドやデシジョングラフに表示する集計データを変更できる
- 次元ボタンを右クリックし，以下のどちらかを選択する
  - ［移動］（データを行領域と列領域の間で移動する）
  - ［ドリルイン］（対象を固定して詳細データを表示する）
- ［ドリルイン］コマンドの実行後に次元ボタンをクリックし，以下のいずれかを選択する
  - ［軸として開く］（その次元の最上位に戻る）
  - ［すべての値］（デシジョングリッドの集計だけを表示するか，集計とそれ以外のすべての値を表示するか切り替える）
  - 詳細値を表示するために固定するカテゴリ（その次元で使用可能なカテゴリのリストから選択する）
- 次元ボタンをクリックして，その次元を開閉する
- 次元ボタンを行領域と列領域の間でドラッグアンドドロップする。同じ領域の既存ボタンの横にドロップしたり，行アイコンまたは列アイコンにドロップすることもできる

実行時のデシジョングリッド

ユーザーが実行できる操作は以下のとおりです。

- デシジョングリッド内を右クリックし，以下のいずれかを選択する
  - 個々のデータグループ, ある次元のすべての値, グリッド全体の小計の設定 / 解除を切り替える
  - 20-7 ページに記載されているデシジョンキューブエディタを表示する
  - 次元と集計の開閉を切り替える
- 行見出しと列見出し内で［+］と［-］をクリックして次元を開閉する
- 次元を行と列の間でドラッグアンドドロップする

実行時のデシジョングラフ

ユーザーはグラフのグリッド領域を上下左右にドラッグして，画面に表示されていないカテゴリや値をスクロールできます。

デシジョンコンポーネントとメモリ制御

次元または集計をデシジョンキューブに読み込むと，メモリを消費します。集計を新たに追加すると，メモリの使用量は比例的に増加します。つまり，デシジョンキューブでは集計が 1 つだけのときと比べ，集計が 2 つになるとメモリの消費量は 2 倍になり，集計が 3 つになるとメモリの消費量は 3 倍になるという具合に増加していきます。次元に対するメモリの消費量はもっと急激に増加していきます。10 個の値を持つ次元を追加するとメモリの消費量は 10 倍になり，100 個の値を持つ次元を追加するとメモリの消費量は 100 倍になります。このように次元をデシジョンキューブに追加すると，メモリの使用量に大きな影響を及ぼすため，処理効率が急激に低下することがあります。値の数が多い次元を追加した場合，この影響は顕著です。

デシジョンコンポーネントには，メモリの使用方法や使用条件を制御するのに役立つ設定がいくつも用意されています。ここで取り上げたプロパティと制御方法についての詳細は，オンラインヘルプで TDecisionCube を検索してください。

次元，集計，セルの最大数の設定

デシジョンキューブの MaxDimensions プロパティと MaxSummaries プロパティを CubeDim->ActiveFlag プロパティと一緒に使うと，一度に読み込みできる次元や集計の数を制御できます。デシジョンキューブエディタの［メモリ管理］ページで値の最大数を設定すると，一度に何個の次元または集計をメモリに読み込めるかを全体的に制御できます。

次元や集計の数を制限すると，デシジョンキューブに必要なメモリ量をおおまかに制限できます。ただし，値の数が多い次元と値の数が少ない次元が区別されるわけではありません。キューブ内のセル数も制限すると，デシジョンキューブで必要なメモリの絶対量をより厳密に制御できます。セルの最大数は，デシジョンキューブエディタの［メモリ管理］ページで設定します。

次元状態の設定

ActiveFlag プロパティは，どの次元をロードするかを制御します。このプロパティを設定するには，デシジョンキューブエディタの［次元の設定］タブにある Activity Type コントロールを使います。このコントロールを Active に設定すると，次元が無条件で読み込まれ，常にメモリ領域を占めます。この状態の次元の数は，必ず MaxDimensions より小さくし，Active に設定する集計数は，MaxSummaries よりも小さくしなければなりません。次元と集計は，常時使用可能にすべき場合にのみ Active に設定します。Active に設定すると，使用可能なメモリの管理を行うキューブの機能は制限されます。

ActiveFlag を AsNeeded に設定すると，MaxDimensions，MaxSummaries，または MaxCells の制限を超えないで読み込めるときだけ，次元または集計が読み込まれます。デシジョンキューブは MaxCells，MaxDimensions，MaxSummaries によって指定された制限内に保つために，［必要に応じて］（AsNeeded）に設定された次元または集計をメモリに入れたりメモリから出したりします。したがって，次元または集計はそのときに使われていないと，メモリに読み込まれていないことがあります。使用頻度が低い次元を［必要に応じて］に設定すれば，読み込みやピボットの処理効率が上がります。ただし，現在読み込まれていない次元へのアクセス時間は遅くなります。

ページングした次元の使い方

デシジョンキューブエディタの［次元の設定］タブで［グループ化］を［固定値］に設定したときに，［初期値］がヌル以外の場合，次元は「ページングされている」，つまり「固定されたままの状態である」とみなされます。この場合，一度にアクセスできるのはその次元の 1 つの値のデータだけですが，プログラムで記述すれば一連の値にシーケンシャルにアクセスできます。このような次元はピボットしたり開いたりできません。

値の数が膨大な次元のデータを読み込むと多量のメモリが必要になります。このような次元はページングすれば，一度に 1 つの値の集計情報を表示できます。通常はこのように表示した方が情報は読みとりやすくなり，メモリの消費量の管理も容易になります。

第21章

データベースへの接続

ほとんどのデータセットコンポーネントは，データベースサーバーに直接接続することができます。いったん接続すると，データセットはサーバーと自動的に通信します。データセットを開くと，サーバーからデータが自動的にデータセットに取り込まれ，ユーザーがレコードを登録すると，レコードはサーバーに送り返されて適用されます。単一の接続コンポーネントを複数のデータセットで共有することもできますし，データセットごとに別の接続を使うこともできます。

それぞれの種類のデータセットは，単一のデータアクセスメカニズムを扱うように設計された，それ独自の接続コンポーネントを使用して，データベースサーバーに接続します。次の表に，これらのデータアクセスメカニズムと，関連する接続コンポーネントの一覧を示します。

表 21.1　データベース接続コンポーネント

データアクセスメカニズム	接続コンポーネント
BDE（ボーランドデータベースエンジン）	TDatabase
ActiveX Data Objects（ADO）	TADOConnection
dbExpress	TSQLConnection
InterBase Express	TIBDatabase

メモ　これらの各メカニズムの長所と短所については，18-1 ページの「データベースを使用する」を参照してください。

接続コンポーネントは，データベース接続を確立するのに必要なすべての情報を提供します。この情報は，接続コンポーネントの種類ごとに異なっています。以下を参照してください。

- BDE ベースの接続についての説明は，24-14 ページの「データベースの識別」を参照してください。
- ADO ベースの接続についての説明は，25-3 ページの「TADOConnection の使用によるデータストアへの接続」を参照してください。
- dbExpress 接続についての説明は，26-3 ページの「TSQLConnection の設定」を参照してください。
- InterBase Express 接続についての説明は，オンラインヘルプで TIBDatabase を参照してください。

それぞれの種類のデータセットは異なる接続コンポーネントを使用しますが，それらはすべて TCustomConnection の子孫です。それらすべては，タスクの多くを同じように実行し，プロパティ，メソッド，およびイベントの多くを共通して持っています。この章では，これら共通するタスクの多くについて説明します。

暗黙の接続を使う

どのデータアクセスメカニズムを使用している場合でも，明示的に接続コンポーネントを作成し，それを使用して，データベースサーバーへの接続と通信を管理することができます。さらに，BDE 対応および ADO ベースのデータセットの場合には，データセットのプロパティによってデータベース接続を記述し，データセットに暗黙の接続を生成させるオプションがあります。BDE 対応のデータセットでは，DatabaseName プロパティを使って暗黙の接続を指定します。ADO ベースのデータセットでは，ConnectionString プロパティを使います。

暗黙の接続を使用する場合，接続コンポーネントを明示的に作成する必要はありません。これにより，アプリケーション開発を簡素化できます。また，デフォルト接続の指定によって，広範な状況に対応できます。ただし，多数のユーザーがいてデータベース接続の条件が異なる複雑なミッションクリティカルなクライアント / サーバーアプリケーションについては，独自の接続コンポーネントを作成して，それぞれのデータベース接続をアプリケーションのニーズに合わせて調節しなければなりません。明示的な接続コンポーネントを使えば，きめ細かな制御を行えます。たとえば，以下のタスクを実行するには，接続コンポーネントにアクセスする必要があります。

- データベースサーバーへのログインサポートのカスタマイズ（暗黙の接続では，ユーザー名およびパスワードの入力を要求する，デフォルトのログインダイアログが表示されます）
- トランザクションの制御とトランザクション排他レベルの指定
- データセットを使用せずに，サーバー上で SQL コマンドを実行すること
- 同じデータベースに接続されているすべての開いたデータセット上でアクションを実行すること

加えて，すべて同じサーバーを使用する複数のデータセットがある場合，使用するサーバーの指定を 1 か所だけで行えばよいので，接続コンポーネントの使用ははより容易になります。そうすれば，後からサーバーを変更する場合でも，接続コンポーネントだけを更新すればよく，複数のデータセットコンポーネントを更新する必要はありません。

接続の制御

データベースサーバーへの接続を確立する前に，アプリケーションはサーバーに関する主な情報をいくつか提供しなければなりません。それぞれの種類の接続コンポーネントは，サーバーを識別するための異なったプロパティのセットを表示します。とはいえ，一般に，それらはすべて，必要とするサーバーに名前を付け，どのように接続を形成するかを制御する接続パラメータのセットを用意するための方法を提供するものです。接続パラメータはサーバーごとに異なります。ユーザー名とパスワード，BLOB 項目の最大サイズ，SQL ロールなどの情報が含まれます。

いったん，希望するサーバーと接続パラメータを識別したなら，接続コンポーネントを使用して，明示的に接続を開いたり閉じたりすることができます。接続コンポーネントは，接続を開いたり閉じたりする際にイベントを生成するので，それらを使用して，データベース接続の変化に対しアプリケーションがどう応答するかをカスタマイズすることができます。

データベースサーバーへの接続

接続コンポーネントを使ってデータベースサーバーに接続するには以下の 2 通りの方法があります。

- Open メソッドを呼び出す
- Connected プロパティを **true** に設定する

Open メソッドを呼び出すと，Connected が **true** に設定されます。

メモ 接続コンポーネントがサーバーに接続されていないときに，アプリケーションが，それに関連付けられたデータセットのいずれかを開こうとすると，データセットは自動的に接続コンポーネントの Open メソッドを呼び出します。

Connected を **true** に設定すると，接続コンポーネントはまず BeforeConnect イベントを生成します。このイベントは初期化処理に利用できます。たとえば，このイベントを使って，接続パラメータを変更することができます。

サーバーログインがどのように制御される設定になっているかにもよりますが，BeforeConnect イベントが終了した時点で，接続コンポーネントはデフォルトのログインダイアログボックスを表示します。それから，ユーザー名とパスワードがドライバに渡され，接続が開きます。

接続が開くと，接続コンポーネントは AfterConnect イベントを生成します。このイベントは，接続を開く際に必要な処理を実行するのに利用できます。

メモ いくつかの接続コンポーネントは，接続を確立するときにも，同様に追加のイベントを生成します。

いったん確立された接続は，その接続を使用するアクティブなデータセットが少なくとも 1 つある限りは維持されます。アクティブなデータセットがなくなった場合は，接続コンポーネントは接続を切断します。いくつかの接続コンポーネントには，それを使用するデータセットがすべて閉じても接続を開いたままにするための KeepConnection プロパティがあります。KeepConnection が **true** なら，接続は維持されます。リモートデータベースサーバーへ接続する場合，またはデータセットを頻繁に開いたり閉じたりするアプリケーションの場合は，KeepConnection を true に設定すると，ネットワークのトラフィックが減りアプリケーションの動作が速くなります。KeepConnection が **false** なら，データベースを使っているアクティブなデータセットがなくなると，接続は切断されます。そのデータベースを使うデータセットを後になって再度開く場合は，接続を再度確立して初期化しなければなりません。

データベースサーバーからの切断

接続コンポーネントを使ってサーバーから切断するには以下の 2 通りの方法があります。

- Connected プロパティを false に設定する
- Close メソッドを呼び出す

Close を呼び出すと，Connected は **false** に設定されます。

Connected が **false** に設定されると，接続コンポーネントは BeforeDisconnect イベントを生成します。このイベントは接続を閉じる前のクリーンアップ処理に利用できます。たとえば，このイベントを使って，開いているすべてのデータセットに関する情報を，データセットが閉じられる前にキャッシュできます。

BeforeConnect イベントが終了すると，接続コンポーネントは開いているすべてのデータセットを閉じ，サーバーからの接続を切断します。

最後に接続コンポーネントは AfterDisconnect イベントを生成します。このイベントは，ユーザーインターフェース上の接続ボタンを有効にするなど，接続状態が変化したことに対応する処理に利用できます。

メモ Close を呼び出したり，Connected を **false** に設定したりすると，接続コンポーネントの KeepConnection が **true** の場合でも，データベースサーバーから切断されます。

サーバーログインの制御

ほとんどのリモートデータベースサーバーには，権限のないアクセスを禁止するセキュリティ機能があります。通常，サーバーは，データベースへのアクセスを許可する前にユーザー名とパスワードのログインを要求してきます。

サーバーがログインを要求する場合，設計時には，データベースへの接続を最初に試みたときに標準の［データベースへのログイン］ダイアログボックスが表示され，ユーザー名とパスワードの入力が求められます。

実行時には，以下の方法でサーバーのログイン要求を処理できます。

- デフォルトのログインダイアログとプロセスを使ってログインを処理させる。これがデフォルトのアプローチです。接続コンポーネントの LoginPrompt プロパティを **true**（デフォルト）に設定し，その接続コンポーネントを宣言しているユニットに DBLogDlg.hpp を含めます。サーバーがユーザー名とパスワードを要求すると，標準の［データベースへのログイン］ダイアログボックスが表示されます。
- ログインを試みる前にログイン情報を提供する。それぞれの種類の接続コンポーネントは，ユーザー名とパスワードの指定のために，以下のような異なるメカニズムを使用します。
  - BDE，dbExpress および InterBase Express データセットの場合には，Params プロパティによってユーザー名とパスワード接続パラメータにアクセスできる（BDE データセットの場合，パラメータ値は BDE エリアスにも関連付けられる。一方，dbExpress データセットの場合，それらは接続名にも関連付けられる）
  - ADO データセットの場合，ユーザー名およびパスワードは ConnectionString プロパティに含めることができる（またはパラメータとして Open メソッドに提供できる）

  サーバーが要求する前にユーザー名とパスワードを指定する場合は，デフォルトログインダイアログが表示されないようにするため，LoginPrompt を **false** に設定しておいてください。たとえば，

次のコードはユーザー名とパスワードを BeforeConnect イベントハンドラ内の SQL 接続コンポーネント上で設定し，現在の接続名と関連付けられている暗号化されたパスワードを復号します。

```
void __fastcall TForm1::SQLConnectionBeforeConnect(TObject *Sender)
{
  if (SQLConnection1->LoginPrompt == false)
  {
    SQLConnection1->Params->Values["User_Name"] = "SYSDBA";
    SQLConnection1->Params->Values["Password"] =
        Decrypt(SQLConnection1->Params->Values["Password"]);
  }
}
```

開発時にユーザー名とパスワードを設定したり，コード内でハードコードされた文字列を使用すると，アプリケーションの実行ファイルに値が埋め込まれることに注意してください。これらは簡単に見ることができるので，サーバーセキュリティ上の問題を引き起こすことがあります。

- ログインイベント用の独自のカスタム処理を作成する。ユーザー名とパスワードが必要になると，接続コンポーネントはイベントを生成します。
  - TDatabase，TSQLConnection，および TIBDatabase の場合，これは OnLogin イベント。イベントハンドラは，文字列リスト内に 2 つのパラメータ，接続コンポーネントおよびユーザー名とパスワードパラメータのローカルコピーを持っています。(TSQLConnection も同様に，データベースパラメータを含んでいます)。このイベントを発生させるためには，LoginPrompt プロパティをに **true** に設定しなければなりません。LoginPrompt の値が **false** のときに OnLogin イベントにハンドラを割り当てると，データベースにログインできなくなります。デフォルトのダイアログが表示されず，OnLogin イベントハンドラは決して実行されないからです。
  - TADOConnection の場合，イベントは OnWillConnect イベント。イベントハンドラは 5 つのパラメータ，接続コンポーネントおよび接続に影響を及ぼす値を返す 4 つのパラメータを持っています（ユーザー名とパスワードのための 2 つを含みます)。このイベントは，LoginPrompt の値にはかかわりなく常に発生します。

  ログインパラメータを設定するイベント用のイベントハンドラを書いてください。次の例は，グローバル変数（UserName）とユーザー名に与えられたパスワードを返すメソッド（PasswordSearch）から，USER NAME と PASSWORD パラメータに対する値を得るものです。

```
void __fastcall TForm1::Database1Login(TDatabase *Database, TStrings *LoginParams)
{
  LoginParams->Values["USER NAME"] = UserName;
  LoginParams->Values["PASSWORD"] = PasswordSearch(UserName);
}
```

ログインパラメータを提供するほかのメソッドの場合と同様，OnLogin または OnWillConnect イベントハンドラを作成するときは，パスワードをアプリケーションのコードにハードコードすることは避けてください。この情報は暗号化しておくか，安全なデータベースからアプリケーションで検索するか，またはユーザーに入力してもらう必要があります。

トランザクションの管理

トランザクションはアクションのグループであり，トランザクションがコミット（確定）される前に，そのすべてのアクションがデータベース内の 1 つまたは複数のテーブルに対して正常に実行されなければなりません。グループ内のアクションが 1 つでも失敗すると，すべてのアクションはロールバックされ（取り消され）ます。トランザクションを使用すると，トランザクションを構成するアクションのうちの 1 つが完了した時点で問題が起きてもデータベースの整合性が失われることはありません。

たとえば，バンキングアプリケーションでの口座から口座への資金振り替えは，トランザクションによる保護が適している操作です。もしも，一方の口座の残高を減らしたあとでもう一方の口座の残高を増やすときにエラーが発生した場合，それでもデータベース全体の総額が正しく保たれるようにするには，トランザクションをロールバックします。

SQL コマンドをデータベースに直接送信してトランザクションを管理する処理はいつでも実行可能です。ほとんどのデータベースは，独自のトランザクション管理モデルを提供しています（ただし，トランザクションをまったくサポートしないものもあります）。管理モデルをサポートするサーバーでは，独自のトランザクション管理を直接コーディングして，スキーマキャッシングなど，特定のデータベースサーバーの高度なトランザクション管理機能を利用できます。

高度なトランザクション管理が不要な場合は，接続コンポーネントが提供している一連のメソッドとプロパティを使うことによって，SQL コマンドを明示的に送信しなくてもトランザクションを管理できます。このプロパティやメソッドを利用すると，トランザクションをサポートするデータベースサーバーである限り，アプリケーションをその種類ごとにカスタマイズしなくても済むという利点があります（BDE も，ローカルテーブル用に制限付きのトランザクションサポートを提供していますが，サーバートランザクションサポートは提供していません。BDE を使用しない場合，トランザクションをサポートしていないデータベースでトランザクションを始めようとすると，接続コンポーネントは例外を発生します）。

注意 データセットプロバイダコンポーネントが更新を適用すると，それは暗黙的に，更新のためのトランザクションを生成します。明示的に開始するトランザクションが，プロバイダによって生成されたトランザクションと競合しないように注意してください。

トランザクションの開始

トランザクションを開始すると，それ以降にデータベースを読み書きするすべての文は，トランザクションが明示的に終了された場合，または（オーバーラップするトランザクションなら）別のトランザクションが開始された場合を除いて，開始されたトランザクションのコンテキスト内にあるものと見なされます。それぞれの文はグループの一部とみなされます。変更がデータベースに正常にコミットされなければ，変更はすべて元に戻されます。

トランザクションの進行中は，データベーステーブルのデータの表示は，トランザクション排他レベルによって指定されます。トランザクション排他レベルについては，21-9 ページの「トランザクションの排他レベルの指定」を参照してください。

TADOConnection の場合，トランザクションを開始するには，BeginTrans メソッドを呼び出します。

```
Level = ADOConnection1->BeginTrans();
```

BeginTrans は，開始したトランザクションのネストのレベルを返します。ネストしたトランザクションとは，別の，親トランザクションの内に入れ子になっているもののことです。サーバーがトランザクションを開始すると，ADO 接続は OnBeginTransComplete イベントを受け取ります。

TDatabase の場合，かわりに StartTransaction メソッドを使用します。TDataBase は，ネストした，またはオーバーラップしたトランザクションをサポートしていません。別のトランザクションが進行中のときに TDatabase コンポーネントの StartTransaction メソッドを呼び出すと，例外が発生します。StartTransaction の呼び出しを避けるには，InTransaction プロパティをチェックします。

```
if (!Database1->InTransaction)
  Database1->StartTransaction();
```

TSQLConnection は StartTransaction メソッドも使用しますが，それはより制御性の高いものです。つまり，StartTransaction は，トランザクション記述子というパラメータを持っています。この記述子を使って，複数の同時トランザクションを管理し，トランザクション排他レベルをトランザクションごとに指定することができます（トランザクションレベルについての詳細は，21-9 ページの「トランザクションの排他レベルの指定」を参照してください）。複数の同時トランザクションを管理するには，トランザクション記述子の TransactionID 項目をユニークな値に設定します。TransactionID には，値がユニーク（進行中の他のトランザクションと競合しない）であれば，任意の値を設定できます。サーバーによっては，TSQLConnection で開始したトランザクションを（ADO を使用する場合のように）ネストしたりオーバーラップしたりすることができます。

```
TTransactionDesc TD;
TD.TransactionID = 1;
TD.IsolationLevel = xilREADCOMMITTED;
SQLConnection1->StartTransaction(TD);
```

オーバーラップされたトランザクションの場合，デフォルトでは，2 番目のトランザクションが開始されると，最初のトランザクションは非アクティブになります。ただし，最初のトランザクションのコミットやロールバックは後で実行できます。InterBase データベースで TSQLConnection を使っている場合，TransactionLevel プロパティを設定すると，それぞれのデータセットと特定のアクティブなトランザクションをアプリケーション内で関連付けることができます。つまり 2 番目のトランザクションを開始した後でも，データセットを目的のトランザクションに関連付けることによって，2 つのトランザクションの処理を同時に継続できます。

メモ TADOConnection とは異なり，TSQLConnection および TDatabase は，トランザクションが開始してもイベントを受け取りません。

InterBase Express では，接続コンポーネントを使用してトランザクションを開始するかわりに，別個のトランザクションコンポーネントを使用すると，TSQLConnection よりもさらに高い制御性が得られます。ただし，デフォルトのトランザクションを開始するには，TIBDatabase を使用します。

```
if (!IBDatabase1->DefaultTransaction->InTransaction)
  IBDatabase1->DefaultTransaction->StartTransaction();
```

2 つの別個のトランザクションコンポーネントを使用すると，トランザクションをオーバラップさせることができます。それぞれのトランザクションコンポーネントには，トランザクションを設定するためのパラメータのセットがあります。これらにより，トランザクション排他レベル，およびトランザクションのほかのプロパティを指定することができます。

トランザクションの終了

理想的には，トランザクションは必要最小限の間だけ存続すべきです。トランザクションがアクティブな時間が長くなるほど，そのデータベースに同時にアクセスするユーザーが増え，トランザクションの存続期間中に並行して開始および終了する同時トランザクションが多くなり，変更をコミットしようとするときに別のトランザクションと競合する可能性が高くなります。

成功したトランザクションの終了

トランザクションの構成要素であるすべてのアクションが正常に実行されたら，トランザクションをコミットしてデータベースの変更を確定できます。TDatabase の場合，Commit メソッドを使用してトランザクションをコミットします。

```
MyOracleConnection->Commit();
```

TSQLConnection の場合も，Commit メソッドを使用できますが，StartTransaction メソッドに与えたトランザクション記述子を指定して，どのトランザクションをコミットするかを明示しなければなりません。

```
MyOracleConnection->Commit(TD);
```

TIBDatabase の場合，Commit メソッドを使用してトランザクションオブジェクトをコミットします。

```
IBDatabase1->DefaultTransaction->Commit();
```

TADOConnection の場合，CommitTrans メソッドを使用してトランザクションをコミットします。

```
ADOConnection1->CommitTrans();
```

メモ ネストされた（子）トランザクションをコミットすることは可能です。これは，親トランザクションがロールバックされた場合に，後で変更をロールバックするためです。

トランザクションが正常にコミットされると，ADO 接続コンポーネントは OnCommitTransComplete イベントを受け取ります。ほかの接続コンポーネントは，同様のイベントを受け取りません。

現在のトランザクションをコミットする呼び出しは通常，**try...catch** 文の中で試行します。このようにして，トランザクションが正常にコミットできなければ，**catch** ブロックを使ってエラーを処理し，操作を再試行するかトランザクションをロールバックします。

失敗したトランザクションの終了

トランザクションの一部で変更を行うときや，トランザクションをコミットしようとしてエラーが発生したら，そのトランザクション内で行われたすべての変更を破棄します。これらの変更を破棄することを，トランザクションをロールバックすると言います。

TDatabase の場合，Rollback メソッドを呼び出してトランザクションをロールバックします。

```
MyOracleConnection->Rollback();
```

TSQLConnection の場合も，Rollback メソッドを使用できますが，StartTransaction メソッドに与えたトランザクション記述子を指定して，どのトランザクションをロールバックするかを明示しなければなりません。

```
MyOracleConnection->Rollback(TD);
```

TIBDatabase の場合，Rollback メソッドを呼び出してトランザクションオブジェクトをロールバックします。

```
IBDatabase1->DefaultTransaction->Rollback();
```

TADOConnection の場合，RollbackTrans メソッドを呼び出してトランザクションをロールバックします。

```
ADOConnection1->RollbackTrans();
```

トランザクションが正常にロールバックされると，ADO 接続コンポーネントは OnRollbackTransComplete イベントを受け取ります。ほかの接続コンポーネントは，同様のイベントを受け取りません。

現在のトランザクションをロールバックする呼び出しは，通常，以下の場合に使います。

- 例外処理コード（データベースのエラーを回復できないとき）
- ボタンまたはメニューのイベントコード（ユーザーが［キャンセル］ボタンをクリックした場合など）

トランザクションの排他レベルの指定

トランザクション排他レベルは，同一のテーブルを操作する同時トランザクション間の対話方式を指定するものです。これは特に，1 つのトランザクションが他のトランザクションによるテーブルへの変更にどれだけ遭遇するかに影響します。

サーバーの種類ごとに，可能なトランザクション排他レベルのサポートは異なります。可能なトランザクション排他レベルは，以下の 3 通りです。

- DirtyRead：あるトランザクションによって実行された変更のすべてを，その変更がコミットされていない時点でも使用中のトランザクションから読み出せます。コミットされていない変更は確定したものではなく，いつでもロールバックされる可能性があります。この値では最低限の排他処理しか提供されないため，Oracle，Sybase，MS-SQL，InterBase などの多くのデータベースサーバーで使用できません。
- ReadCommitted：ほかのトランザクションによってコミットされた変更だけが読み出せます。この設定の場合，ロールバックの可能性があるコミットされていない変更を，使用中のトランザクションが読み出す危険はありませんが，読み出しているときに他のトランザクションがコミットされると，整合性が取れていないデータベースのビューを受け取ってしまう可能性は残ります。このレベルは，BDE が管理するローカルトランザクションを除く，すべてのトランザクションで利用可能です。
- RepeatableRead：使用中のトランザクションが読み出したデータベースのデータに，整合性が取れていることが保証されます。使用中のトランザクションは，データを 1 度だけ読み出せます。以後，ほかの同時トランザクションによってデータが変更されコミットされたとしても，使用中のトランザクションで読み出すことはできません。この排他レベルでは，トランザクションがいったんレコードを読み出すと，そのレコードのビューは変化しないことが保証されます。これは最も高い排他レベルです。このレベルは，Sybase や MS-SQL など，いくつかのサーバーでは利用できず，BDE が管理するローカルトランザクションでも利用できません。

さらに，TSQLConnection では，データベース固有のカスタム排他レベルを指定できます。カスタム排他レベルは，dbExpress ドライバによって定義されます。詳細については，ドライバのマニュアルを参照してください。

メモ 排他レベルの実装状況についての詳細は，使用するサーバーのドキュメントを参照してください。

TDatabase と TADOConnection では，TransIsolation プロパティを設定することにより，トランザクション排他レベルを指定できます。TransIsolation をデータベースサーバーがサポートしていない値に設定すると，（利用可能であれば）その次に高い排他レベルになります。より高い利用可能なレベルがない場合には，トランザクションを開始しようとすると，接続コンポーネントで例外が発生します。

TSQLConnection を使用する場合，トランザクション排他レベルはトランザクション記述子の IsolationLevel 項目によって制御されます。

InterBase Express を使用する場合，トランザクション排他レベルはトランザクションパラメータによって制御されます。

サーバーにコマンドを送信する

TIBDatabase 以外のデータベース接続コンポーネントでは，Execute メソッドを呼び出すことにより，関連するサーバー上で SQL 文を実行できます。文が SELECT 文の場合，Execute はカーソルを返すことができますが，この使用方法はお勧めしません。データを返す文を実行する上での望ましい方法は，データセットを使用することです。

Execute メソッドは，どんなレコードも返さない単純な SQL 文を実行するのに非常に便利です。そのような文には，DDL（データ定義言語）文が含まれます。これは，CREATE INDEX，ALTER TABLE，DROP DOMAIN のような，データベースのメタデータの操作と作成を行います。データ操作言語（DML）SQL 文の中にも，結果セットを返さないものがあります。データに対するアクションを起こすが結果セットを返さない DML 文は，INSERT，DELETE，および UPDATE です。

Execute メソッドの構文は，以下のように，接続の種類ごとに異なります。

- TDatabase：Execute は 4 つのパラメータを取ります。実行する単一の SQL 文を指定する AnsiString，その文のパラメータ値を指定する TParams オブジェクト，次回の呼び出しのために文をキャッシュするかどうかを指定する論理値，返される BDE カーソルへのポインタです（これにはヌルを渡すことが推奨されます）。
- TADOConnection：Execute には 2 つのバージョンがあります。第 1 の構文はパラメータを 2 つとります。1 つは SQL 文を指定する WideString，もう 1 つは文を非同期に実行するか，およびレコードを返すかどうかを制御するオプションのセットを指定します。この第 1 の構文は，返されるレコードのためのインターフェースを返します。第 2 の構文は，SQL 文を指定する WideString，文の実行時に影響を受けるレコードの数を返す 2 番目のパラメータ，文が非同期に実行されるかどうかといったオプションを指定する 3 番目のパラメータを取ります。どちらの構文も，SQL 文のパラメータ渡しについては想定していないことに注意してください。
- TSQLConnection：Execute が取るパラメータは 3 つあります。実行する SQL 文を 1 つだけ指定する AnsiString，SQL 文のパラメータ値を指定する TParams オブジェクト，そしてレコード NULL を返すために作成される TCustomSQLDataSet を受け取るポインタです。

メモ Execute を使って実行できる SQL 文は，一度に 1 つだけです。SQL スクリプトユーティリティのように複数の SQL 文を 1 回の Execute 呼び出しで実行することはできません。複数の文を実行するには，Execute の呼び出しを繰り返します。

パラメータのない文を実行するのは比較的簡単です。たとえば次のコードは，TSQLConnection コンポーネント上で，パラメータなしで CREATE TABLE 文（DDL）を実行します。

```
void __fastcall TDataForm::CreateTableButtonClick(TObject *Sender)
{
  SQLConnection1->Connected = true;
  AnsiString SQLstmt = "CREATE TABLE NewCusts " +
    "( " +
    " CustNo INTEGER, " +
    " Company CHAR(40), " +
    " State CHAR(2), " +
    " PRIMARY KEY (CustNo) " +
    ")";
  SQLConnection1->Execute(SQLstmt, NULL, NULL);
}
```

パラメータを使うには，TParams オブジェクトを作成しなければなりません。それぞれのパラメータ値に対して TParams::CreateParam メソッドを使って TParam オブジェクトを追加します。次に，TParam のプロパティを使ってパラメータを記述し，その値を設定します。

以上の処理を，TDatabase を使って INSERT 文を実行する次の例で説明します。この INSERT 文には，StateParam という名前の 1 つのパラメータがあります。そのパラメータに「CA」という値を設定するために，TParams オブジェクト（stmtParams という名前）が作成されます。

```
void __fastcall TForm1::INSERT_WithParamsButtonClick(TObject *Sender)
{
  AnsiString SQLstmt;
  TParams *stmtParams = new TParams;
  try
  {
    Database1->Connected = true;
    stmtParams->CreateParam(ftString, "StateParam", ptInput);
    stmtParams->ParamByName("StateParam")->AsString = "CA";
    SQLstmt = "INSERT INTO 'Custom.db' ";
    SQLstmt += "(CustNo, Company, State) ";
    SQLstmt += "VALUES (7777, 'Robin Dabank Consulting', :StateParam)";
    Database1->Execute(SQLstmt, stmtParams, false, NULL);
  }
  __finally
  {
    delete stmtParams;
  }
}
```

パラメータ付き SQL 文に対して，TParam オブジェクトを作成せずパラメータの値を指定しなかった場合は，SQL 文の実行時にエラーが発生することがあります（エラーが発生するかどうかは，データベースのバックエンドに何を使用しているかによります）。TParam オブジェクトが作成されても，それに対応するパラメータが SQL 文内になければ，アプリケーションが TParam を使用しようとしたときに例外が発生します。

関連付けられたデータセットを操作する

すべてのデータベース接続コンポーネントは，アクティブなすべてのデータセットのリストを保持しており，データベースへの接続に使います。たとえば，接続コンポーネントはこのリストを，データベースへの接続を閉じる際に，すべてのデータセットを閉じるために使います。

ユーザーがこのリストを使って，特定の接続コンポーネントで特定のデータベースに接続しているすべてのデータセットに対して，アクションを実行することもできます。

サーバーから切断せずにデータセットを閉じる

ユーザーが接続を閉じると，接続コンポーネントは自動的にすべてのデータセットを閉じます。しかし，データベースサーバーとの接続は切断せずに，すべてのデータセットを閉じたいこともあります。

サーバーから切断せずに開いているすべてのデータセットを閉じるには，CloseDataSets メソッドを使います。

TADOConnection と TIBDatabase の場合，CloseDataSets を呼び出せば，接続は開いたままになります。TDatabase と TSQLConnection の場合は，さらに KeepConnection プロパティを true に設定しなければなりません。

関連付けられたデータセットを繰り返し処理する

接続コンポーネントを使用するすべてのデータセットに対して，それを閉じる処理以外の処理を実行するには，DataSets と DataSetCount プロパティを使います。DataSets は，接続コンポーネントにリンクされている，すべてのデータセットの配列です。TADOConnection 以外のすべての接続コンポーネントの場合，このリストはアクティブなデータセットだけを含んでいます。TADOConnection のリストは，非アクティブなデータセットも含んでいます。DataSetCount は，この配列中のデータセットの数です。

メモ 更新をキャッシュするために（汎用の TClientDataSet クライアントデータセットではなく）特別なクライアントデータセットを使用する場合，DataSets プロパティには，クライアントデータセット自体ではなく，クライアントデータセットに所有されている内部データセットが含まれます。

DataSets とともに DataSetCount を使うと，現在アクティブなすべてのデータセットをコード中で順に処理できます。たとえば次のコードは，すべてのアクティブなデータセットについて順番に処理し，データセットのデータを使用するコントロールがあれば使用不可にします。

```
for (int i = 0; i < MyDBConnection->DataSetCount; i++)
  MyDBConnection->DataSets[i]->DisableControls();
```

メモ TADOConnection は，データセットとともに，コマンドオブジェクトもサポートしています。これらは，Commands および CommandCount プロパティを使用すれば，データセットと同じ方法で繰り返し処理を行えます。

メタデータの取得

すべてのデータベース接続コンポーネントは，データベースサーバー上のメタデータのリストを検索することができます。ただし，検索できるメタデータの種類は異なります。メタデータを検索するメソッドは，サーバーで利用可能なさまざまなエンティティの名前を文字列リストに入れます。この情報は，たとえば，実行時にユーザーが動的にテーブルを選択できるようにするために使用できます。

TADOConnection コンポーネントを使えば，ADO データストアで利用可能なテーブルとストアドプロシージャに関するメタデータを検索することができます。この情報は，たとえば，実行時にユーザーが動的にテーブルまたはストアドプロシージャを選択できるようにするために使用できます。

使用可能なテーブルのリスト取得

GetTableNames メソッドは，テーブル名のリストを既存の文字列リストオブジェクトにコピーします。これは，たとえば，ユーザーが開くテーブルを選択できるように，リストボックスにテーブル名を設定するために使用することができます。次のコードは，リストボックスにデータベース上のすべてのテーブル名を設定します。

```
MyDBConnection->GetTableNames(ListBox1->Items, false);
```

GetTableNames には 2 つのパラメータがあります。テーブルネームを入れる文字列リスト，および，リストがシステムテーブルと通常のテーブルのどちらを含むかを示す論理値です。ただし，すべてのサーバーがシステムテーブルにメタデータを保存しているわけではありません。システムテーブルを要求したときに空白のリストが返されることもあります。

メモ ほとんどのデータベース接続コンポーネントの場合，第 2 のパラメータが **false** であれば，GetTableNames はすべての利用可能な非システムテーブルのリストを返します。ただし，TSQLConnection の場合は，システムテーブルの名前以外のものを取得するとき，リストにどの種類のものを追加するかをさらに細かく制御できます。TSQLConnection を使用する場合，リストに追加される名前の種類は，TableScope プロパティはによって制御できます。TableScope は，リストが通常のテーブル，システムテーブル，シノニム，ビューのうちのどれを含むかを示します。

テーブル内の項目名のリスト化

GetFieldNames は，指定されたテーブル内のすべての項目名を既存の文字列リストに入れます。GetFieldNames は 2 つのパラメータを取ります。リストに項目名を入れるテーブルの名前，および，項目名を入れる既存の文字列リストです。

```
MyDBConnection->GetTableNames("Employee", ListBox1->Items);
```

使用可能なストアドプロシージャのリスト取得

データベース内のすべてのストアドプロシージャのリストを取得するには，GetProcedureNames メソッドを使用します。このメソッドは 1 つのパラメータを取ります。既存の文字列リストです。

```
MyDBConnection->GetProcedureNames(ListBox1->Items);
```

メモ GetProcedureNames は，TADOConnection と TSQLConnection でのみ使用可能です。

使用可能なインデックスのリスト化

特定のテーブルで定義されたすべてのインデックスのリストを取得するには，GetIndexNames メソッドを使用します。このメソッドは 2 つのパラメータを取ります。インデックスを含むテーブルと，既存の文字列リストです。

```
MyDBConnection1->GetIndexNames("Employee", ListBox1->Items);
```

メモ GetIndexNames は，TSQLConnection でのみ使用可能です。ただし，ほとんどのテーブルタイプのデータセットには，同様のメソッドがあります。

ストアドプロシージャのパラメータのリスト化

特定のストアドプロシージャで定義されたすべてのパラメータのリストを取得するには，GetProcedureParams メソッドを使用します。GetProcedureParams は，パラメータの記述構造体へのポインタを TList オブジェクトに入れます。各構造体には，指定されたストアドプロシージャのパラメータ（名前，インデックス，パラメータの種類，項目型など）が記載されています。

GetProcedureParams は 2 つのパラメータを取ります。ストアドプロシージャの名前と，既存の TList オブジェクトです。

```
MyDBConnection1->GetIndexNames("GetInterestRate", List1);
```

リストに追加されたパラメータ記述を使い慣れた TParams オブジェクトに変換するには，グローバルな LoadParamListItems 手続きを使います。GetProcedureParams は各構造体の領域を動的に割り当てるため，アプリケーションは情報の処理後に領域を解放しなければなりません。これは，グローバルな FreeProcParams ルーチンを使って行えます。

メモ GetProcedureParams は，TSQLConnection でのみ使用可能です。

第22章

データセットについて

データへアクセスする場合，データセットとそのグループを基本とします。アプリケーションはどのデータベースにアクセスするときもデータセットを使います。データセットオブジェクトは，論理テーブルに整理されたデータベースの一連のレコードを表します。これらのレコードは，単一のデータベーステーブルからのレコードである場合もあれば，問い合わせやストアドプロシージャの実行結果を表わす場合もあります。

データベースアプリケーションでは，TDataSet から派生したデータセットオブジェクトを使います。これらのオブジェクトはこのクラスからデータ項目，プロパティ，イベント，メソッドを継承しています。この章では，データベースアプリケーションで使うデータセットオブジェクトが継承する TDataSet の機能について説明します。データセットオブジェクトを使うには，この共通の機能を理解している必要があります。

TDataSet は仮想データセットです。つまり，そのプロパティやメソッドの多くは仮想または純粋仮想として宣言されます。仮想メソッドとは，そのメソッドの実装が下位オブジェクトによってオーバーライドできる（通常そうされる）関数または手続き宣言です。純粋仮想メソッドは，実際の実装がない関数または手続きの宣言です。この宣言は，すべての下位データセットオブジェクトに実装しなければならないが，実装方法はそれらのオブジェクトごとに異なるメソッド（およびそのパラメータと戻り型）を記述するプロトタイプです。

TDataSet には純粋仮想メソッドがあるので，アプリケーションで直接使うと実行時エラーになります。かわりに，組み込みの TDataSet の下位オブジェクトのインスタンスを作成してアプリケーションで使うか，TDataSet またはその下位オブジェクトから独自ののデータセットオブジェクトを派生させてからそのすべての**純粋仮想**メソッドの実装を記述します。

TDataSet はすべてのデータセットオブジェクトに共通する機能をいくつも定義しています。たとえば，TDataSet は，すべてのデータセットの基本的構造として，1 つまたは複数のデータベーステーブル内の実際の列に対応する TField コンポーネントの配列や，アプリケーションが提供する参照項目または計算項目などを定義します。TField コンポーネントについては，第 23 章「項目コンポーネントの操作」を参照してください。

この章では，TDataSet が共通に持つデータベース機能の使用法について説明します。TDataSet には，これらの機能を利用するためのメソッドが実装されていますが，TDataSet の下位オブジェクトでは，このメソッドが必ずしも実装されているとは限らないことに注意してください。特に，単方向データセットでは，限られたサブセットしか実装していません。

TDataSet の下位オブジェクトの使い方

TDataSet には，直接の下位オブジェクトがいくつかあり，そのそれぞれが異なるデータアクセスメカニズムに対応しています。これらの下位オブジェクトを直接扱うことはありません。かわりに，それぞれの下位オブジェクトは，特定のデータアクセスメカニズムを使用するためのプロパティとメソッドを持っています。これらのプロパティとメソッドは，異なる種類のサーバーデータに適応した，下位クラスによって公開されます。TDataSet の直接の下位オブジェクトには，以下のものがあります。

- TBDEDataSet：データベースサーバーとの通信に BDE（ボーランドデータベースエンジン）を使用します。TBDEDataSet の下位オブジェクトは，TTable，TQuery，TStoredProc，および TNestedTable です。BDE 対応データセットの独自機能については，第 24 章「ボーランドデータベースエンジンの使い方」で説明されています。
- TCustomADODataSet：OLEDB データストアとの通信に ActiveX Data Objects（ADO）を使用します。TCustomADODataSet の下位オブジェクトは，TADODataSet，TADOTable，TADOQuery，および TADOStoredProc です。ADO ベースのデータセットの独自機能については，第 25 章「ADO コンポーネントの操作」で説明されています。
- TCustomSQLDataSet：データベースサーバーとの通信に dbExpress を使用します。TCustomSQLDataSet の下位オブジェクトは，TSQLDataSet，TSQLTable，TSQLQuery，および TSQLStoredProc です。dbExpress データセットの独自機能については，第 26 章「単方向データセットの使い方」で説明されています。
- TIBCustomDataSet：InterBase データベースサーバーと直接通信します。TIBCustomDataSet の下位オブジェクトは，TIBDataSet，TIBTable，TIBQuery，および TIBStoredProc です。
- TCustomClientDataSet：別のデータセットコンポーネントからのデータ，またはディスク上の専用ファイルからのデータを表わします。TCustomClientDataSet の下位オブジェクトは，外部（ソース）データセットに接続することができる TClientDataSet と，特定のデータアクセスメカニズム（TBDEClientDataSet，TSQLClientDataSet，および TIBClientDataSet）のための特別なクライアントデータセットです。後者は，内部ソースデータセットを使用します。クライアントデータセットの独自機能については，第 27 章「クライアントデータセットの使い方」で説明されています。

これら TDataSet の下位オブジェクトが採用しているさまざまなデータアクセスメカニズムの長所と短所については，18-1 ページの「データベースを使用する」で説明されています。

組み込みのデータセットのほかにも，たとえば，表計算プログラムなど，データベースサーバー以外のプロセスからデータを提供することなどを目的として，TDataSet の下に独自の下位オブジェクトを作成することができます。カスタムデータセットを書くと，任意の方法を選択してデータを管理する自由度が高まりますが，ユーザーインターフェースの構築には VCL データコントロールを使用できます。カスタムコンポーネントの作成についての詳細は，第 45 章「コンポーネント作成の概要」を参照してください。

TDataSet の下位オブジェクトはそれぞれ独自のプロパティとメソッドを持っていますが，下位クラスが導入するプロパティとメソッドの中には，別のデータアクセスメカニズムを使用する他の下位クラスが導入しているのと同じものもあります。たとえば，「テーブル」コンポーネント（TTable，TADOTable，TSQLTable，および TIBTable）は，互いに似ています。TDataSet の下位クラスの共通点については，22-22 ページの「データセットの種類」を参照してください。

データセットの状態の決定

データセットの状態，またはモードによって，そのデータに対して何ができるかが決まります。たとえばデータセットが閉じた場合，その状態は dsInactive になり，そのデータに対して何もできないことが示されます。実行時にデータセットの State 読み出し専用プロパティを調べれば，データセットの現在の状態がわかります。次の表に State プロパティの値とその意味を示します。

表 22.1　データセットの State プロパティの値

値	状態	説明
dsInactive	非アクティブ状態	データセットは閉じている。データは使えない
dsBrowse	参照状態	データセットは開いている。データの取得はできるが，変更はできない。これは開いているデータセットのデフォルト状態である
dsEdit	Edit	データセットは開いている。現在の行を変更できる（単方向データセットでは未サポート）
dsInsert	Insert	データセットは開いている。新しい行が挿入された（単方向データセットでは未サポート）
dsSetKey	SetKey	データセットは開いている。範囲の設定および範囲の設定に使われるキーの値と GotoKey の操作ができる（すべてのデータセットでサポートされているわけではない）
dsCalcFields	CalcFields	データセットは開いている。OnCalcFields イベントが進行中であることを示す。計算項目以外の項目の変更を防止する
dsCurValue	CurValue	データセットは開いている。キャッシュされた更新の適用時のエラーに応答するイベントハンドラのために，項目の CurValue プロパティを取得中であることを示す
dsNewValue	NewValue	データセットは開いている。キャッシュされた更新の適用時のエラーに応答するイベントハンドラのために，項目の NewValue プロパティを取得中であることを示す
dsOldValue	OldValue	データセットは開いている。キャッシュされた更新の適用時のエラーに応答するイベントハンドラのために，項目の OldValue プロパティを取得中であることを示す
dsFilter	Filter	データセットは開いている。フィルタが動作中であることを示す。取得できるデータは限られており，データの変更はできない（単方向データセットでは未サポート）
dsBlockRead	ブロック読み出し	データセットは開いている。現在のレコードが変更されたときにも，データベース対応コントロールは更新されず，イベントは発生しない
dsInternalCalc	内部計算	データセットは開いている。レコードに保存されている計算値に対して OnCalcFields イベントが進行中である（クライアントデータセット専用）
dsOpening	開いている	DataSet を開いている途中だが，完了していない。この状態は，非同期取得のためデータセットを開く場合に生じる

通常，アプリケーションはデータセットの状態をチェックして，特定のタスクをいつ実行するべきかを決定します。たとえば，更新を登録する必要があるかどうかを確認するために，dsEdit か dsInsert 状態をチェックする場合があります。

メモ データセットが状態を変える場合は，必ずそのデータセットに関連したデータソースコンポーネントの OnStateChange イベントが呼び出されます。データソースコンポーネントと OnStateChange についての詳細は，19-4 ページの「データソースに仲介された変更に応答する」を参照してください。

データセットを開く / 閉じる

データセットのデータを読み出したり書き込んだりするには，まずデータセットをアプリケーションで開かなければなりません。データセットを開く方法は 2 つあります。

- 設計時にオブジェクトインスペクタを使うか，実行時にコードによって，データセットの Active プロパティを **true** に設定する

```
CustTable->Active = true;
```

- 実行時にデータセットの Open メソッドを呼び出す

```
CustQuery->Open();
```

データセットを開くと，データセットは最初に BeforeOpen イベントを受け取り，データを取り込んでカーソルを開き，最後に AfterOpen イベントを受け取ります。

新しく開かれたデータセットは参照モードになっています。これは，アプリケーションがデータを読めること，また，その中を移動できることを意味しています。

データセットの閉じ方は 2 つあります。

- 設計時にオブジェクトインスペクタを使うか，実行時にコードによって，データセットの Active プロパティを **false** に設定する

```
CustQuery->Active = false;
```

- 実行時にデータセットの Close メソッドを呼び出す

```
CustTable->Close();
```

データセットを開くときにデータセットが BeforeOpen と AfterOpen イベントを受け取るのと同様に，閉じるときには BeforeClose と AfterClose イベントを受け取ります。これらのイベントのハンドラでデータセットの Close メソッドに応答できます。これらのイベントは，たとえば，データセットを閉じる前に，保留中の変更を登録するか，またはそれらを破棄するよう，ユーザーに促すために用いることができます。ここで述べたイベントハンドラのコード例を次に示します。

```
void __fastcall TForm1::VerifyBeforeClose(TDataSet *DataSet)
{
  if (DataSet->State == dsEdit || DataSet->State == dsInsert)
  {
    TMsgDlgButtons btns;
    btns << mbYes << mbNo;
    if (MessageDlg("Post changes before closing?", mtConfirmation, btns, 0) == mrYes)
      DataSet->Post();
    else
      DataSet->Cancel();
```

```
    }
  }
```

メモ TTable コンポーネントの TableName などのプロパティを変更したい場合は，そのデータセットを閉じる必要があります。データセットを再び開いた時点で，新規のプロパティ値が有効になります。

データセットの操作

アクティブなデータセットにはデータセットの現在の行を指すカーソル，つまりポインタがあります。データセットの現在行の項目値は，TDBEdit，TDBLabel，TDBMemo などのフォーム上の単一項目のデータベース対応コントロールに表示されます。データセットが編集をサポートしている場合は，現在のレコードの値を編集，挿入，削除のメソッドを使って操作できます。

カーソルを別の行に移動すれば現在の行を変更できます。表に別のレコードへの移動のためにアプリケーションのコードで使えるメソッドを示します。

表 22.2　データセット内移動メソッド

メソッド	カーソルの移動先
First	データセットの最初の行に移動する
Last	データセットの最後の行に移動する（単方向データセットでは使用不可）
Next	データセットの次の行に移動する
Prior	データセット中の前の行に移動する（単方向データセットでは使用不可）
MoveBy	指定された行だけデータセット内を前後に移動する

データベース対応のビジュアルコンポーネント TDBNavigator は，以上のメソッドをボタンとしてカプセル化し，ユーザーが実行時にそのボタンをクリックしてレコード間を移動できるようにしています。ナビゲータコンポーネントについては，19-28 ページの「レコード間の移動と操作」を参照してください。

これらのメソッド（または検索基準に基づいて移動するほかのメソッド）のうちの 1 つを使用して現在のレコードを変更する場合，データセットは常に 2 つのイベントを受け取ります。BeforeScroll（現在のレコードを離れる前）および AfterScroll（新しいレコードに到着した後）です。これらのイベントは，ユーザーインターフェースを更新するために使用することができます（たとえば，現在のレコードについての情報を示すステータスバーの更新）。

TDataSet にはまた，データセットのレコードを繰り返し処理するときに役立つ情報を提供するものとして，以下の 2 つの論理値プロパティがあります。

表 22.3　データセット内移動プロパティ

プロパティ	説明
Bof（ファイルの先頭）	**true**：カーソルはデータセットの最初の行にある
	false：カーソルはデータセットの最初の行にない
Eof（ファイルの終わり）	**true**：カーソルはデータセットの最後の行にある
	false：カーソルはデータセットの最初の行にない

First メソッドと Last メソッド

First メソッドはカーソルをデータセットの最初の行に移動し，Bof プロパティを **true** にします。カーソルがすでにデータセットの最初の行にある場合は，First メソッドを指定しても何も起こりません。

たとえば次のコードは，CustTable の最初のレコードに移動するためのコードです。

```
CustTable->First();
```

Last メソッドはデータセットの最後の行にカーソルを移動し，Eof プロパティを **true** に設定します。カーソルがすでにデータセットの最後の行にある場合は，Last メソッドを呼び出しても何も起こりません。

次のコードは，CustTable の最後のレコードに移動するためのコードです。

```
CustTable->Last();
```

メモ 単方向データセットに対して Last メソッドを実行すると例外が発生します。

ヒント ユーザーに介入されずにデータセットの最初か最後の行に移動するようなプログラムになっている場合でも，ユーザーが TDBNavigator コンポーネントを使ってレコード間を移動できるようにすることが可能です。ナビゲータコンポーネントがアクティブで表示されているときには，アクティブなデータセットの最初と最後の行に移動するためにユーザーが使用できる 2 つのボタンがあります。これらのボタンの OnClick イベントはデータセットの First メソッドと Last メソッドをそれぞれ呼び出します。ナビゲータコンポーネントの効果的な使い方については，19-28 ページの「レコード間の移動と操作」を参照してください。

Next メソッドと Prior メソッド

Next メソッドは，データセットが空でなければデータセットのカーソルを 1 行先に進め，Bof プロパティを **false** に設定します。Next メソッドを呼び出したときにカーソルがすでにデータセットの最後の行にある場合は，何も起こりません。

たとえば次のコードは，CustTable の次のレコードに移動するためのコードです。

```
CustTable->Next();
```

Prior メソッドは，データセットが空でなければデータセットのカーソルを 1 行前に戻し，Eof プロパティを false に設定します。Prior メソッドを呼び出したときにカーソルがすでにデータセットの最初の行にある場合は，何も起こりません。

たとえば次のコードは，CustTable の前のレコードに移動するためのコードです。

```
CustTable->Prior();
```

メモ 単方向データセットに対して Prior メソッドを実行すると例外が発生します。

MoveBy メソッド

MoveBy メソッドを使用すると，データセットのカーソルを前後に移動させる行数を指定できます。移動は MoveBy が呼び出されたときの現在のレコードを基準にして相対的に行われます。また，MoveBy メソッドはデータセットの Bof および Eof プロパティを状況に応じて適切に設定します。

このメソッドには，移動するレコード数を指定する整数のパラメータがあります。正の整数は先へ進む移動を示し，負の整数は後へ戻る移動を示します。

メモ 単方向データセットに対して負の引数を指定して MoveBy メソッドを実行すると例外が発生します。

MoveBy メソッドは移動した行数を返します。データセットの最初や最後を超えるような移動を試みた場合，MoveBy が返す行数は移動するよう指定した行数と異なります。これは MoveBy がデータセットの最初や最後のレコードに到達するとそこで止まるからです。

次のコードは，CustTable で 2 レコード前に戻るためのコードです。

```
CustTable->MoveBy(-2);
```

メモ マルチユーザーデータベース環境下でアプリケーションが MoveBy メソッドを使うときは，データセットは流動的であることを忘れないでください。複数のユーザーが同時にデータベースにアクセスして，そのデータを変更すると，ある時点で 5 レコード先にあったレコードが，わずかの間に 4 レコード先になったり，6 レコードあるいはそれ以上になったりします。

Eof プロパティと Bof プロパティ

読み出し専用の実行時のプロパティとして，Eof（ファイルの終わり）と Bof（ファイルの始まり）という 2 つのプロパティがあります。この 2 つのプロパティはデータセット内のすべてのレコードを繰り返し処理するときなどに便利です。

Eof

Eof プロパティが **true** のとき，カーソルはデータセットの最後の行にあります。Eof が **true** に設定されるのは，アプリケーションが以下の動作をしたときです。

- 空のデータセットを開いた
- データセットの Last メソッドを呼び出した
- データセットの Next メソッドを呼び出したが，データセットの最後の行にカーソルがあったために Next メソッドが失敗した
- 空の範囲またはデータセットで SetRange を呼び出した

上記の場合以外では，Eof プロパティは必ず **false** になります。したがって，上記の条件が満たされていない場合，およびプロパティを直接テストしていない場合は，Eof は **false** であると想定する必要があります。

Eof プロパティは，データセットのすべてのレコードの繰り返し処理を制御するためにループ条件でよくテストします。レコードの入ったデータセットを開いたり First メソッドを呼び出すと，Eof は **false** になります。1 レコードずつデータセット全体の繰り返し処理をするには，Next を呼び出してレコードを 1 つずつ進め，Eof が **true** になるとループが終了するようにループを作成します。カーソルがすでに最後のレコードにある場合は，Next を呼び出すまで Eof は **false** のままです。

次のコードは，CustTable というデータセットのレコード処理ループのコード記述方法の 1 つを示しています。

```
CustTable->DisableControls();
try
```

```
{
  for (CustTable->First(); !CustTable->Eof; CustTable->Next())
  {
    // ここで各レコードを処理する
    ...
  }
}
__finally
{
  CustTable->EnableControls();
}
```

ヒント 上記の例では，データセットに結び付けられたデータベース対応ビジュアルコントロールの使用可 / 不可を切り替える方法も示しています。ビジュアルコントロールを使用不可にしてあれば，レコードが移動したり値が変わってもコントロールの内容をアプリケーションが更新しないで済むため，処理が高速化されます。すべての処理が完了したら，コントロールを再び使用可能にすることで，最新の現在行の値に更新させることを忘れないでください。ビジュアルコントロールは **try...__finally** 文の **__finally** 節で使用可能にしている点に注意してください。この位置で使用可能にしておくと，たとえ処理中に例外が発生し，繰り返し処理が未完了で終わってもコントロールが使用不可のままになることを回避できます。

Bof

Bof プロパティが **true** のとき，カーソルはデータセットの最初の行にあります。Bof が **true** に設定されるのは，アプリケーションが以下の動作をしたときです。

- データセットを開いた
- データセットの First メソッドを呼び出した
- データセットの Prior メソッドを呼び出したが，データセットの最初のレコードにカーソルがあったために Prior メソッドが失敗した
- 空の範囲またはデータセットで SetRange を呼び出した

上記の場合以外では，Bof プロパティは必ず **false** になります。したがって，上記の条件が満たされていない場合，およびプロパティを直接テストしていない場合は，Bof は **false** であると想定する必要があります。

Eof と同様に Bof もループ条件に入ってデータセットのレコードの繰り返し処理を制御できます。次のコードは，CustTable というデータセットのレコード処理ループのコード記述方法の 1 つを示しています。

```
CustTable->DisableControls(); // 画面描画を止めることで処理速度を上げ，ちらつきをさける
try
{
  while (!CustTable->Bof) // Bof が true になるまで繰り返す
  {
    // ここで各レコードを処理する
    ...
    CustTable->Prior();
    // 成功した場合には Bof は false になる。最初のレコードに対し Prior を呼ぶと失敗し，
    // Bof は true となる
  }
}
```

```
__finally
{
  CustTable->EnableControls();
}
```

レコードにマークを付けて戻る

データセット内で別のレコードに移動したり，レコード数を指定してレコード間を移動する以外に，データセットの特定の位置にマークを付けて，必要なときにすぐにその位置に戻れるようにしておくと便利です。TDataSet にはブックマーク機能があり，Bookmark プロパティと 5 つのブックマークメソッドがあります。

TDataSet には，仮想ブックマークメソッドが用意されています。TDataSet から派生したデータセットオブジェクトでブックマークメソッドが呼び出された場合には値が返されますが，この値は現在の位置を反映していない単なるデフォルトの値です。TDataSet の下位オブジェクトでのブックマークのサポートは，種類ごとに異なります。dbExpress データセットはいずれも，ブックマーク機能をサポートしていません。ADO データセットは，ブックマークをサポートする場合もあります。これは，基になっているデータベーステーブルに応じて決まります。BDE データセット，InterBase Express データセット，およびクライアントデータセットは，常にブックマークをサポートします。

Bookmark プロパティ

Bookmark プロパティは，アプリケーション内のいくつものブックマークのうちどれが現在のものかを示します。Bookmark は現在のブックマークを特定する文字列です。別のブックマークを追加すると，その追加したブックマークが現在のブックマークになります。

GetBookmark メソッド

ブックマークを作るには，アプリケーションで TBookmark 型の変数を宣言してから，GetBookmark を呼び出してその変数に記憶域を割り当て，さらにその変数の値にデータセットの特定の位置を設定しなければなりません。TBookmark 型は（**void** *）のポインタです。

GotoBookmark および BookmarkValid メソッド

GotoBookmark にブックマークを渡すと，ブックマークで指定した位置にデータセットのカーソルが移動します。GotoBookmark を呼び出す前に，BookmarkValid を呼び出して，ブックマークが正しくレコードを指しているかどうか確認できます。指定されたブックマークがレコードを指していれば BookmarkValid は **true** を返します。

CompareBookmarks メソッド

移動先のブックマークが別の（または現在の）ブックマークと異なっているかどうか確認するため，CompareBookmarks を呼び出すこともできます。2 つのブックマークが同じレコードを指している場合（または両方ともヌルである場合），CompareBookmarks は 0 を返します。

FreeBookmark メソッド

FreeBookmark は，不要になったブックマークに割り当てられているメモリを解放します。FreeBookmark は，既存のブックマークを再利用する前にも呼び出す必要があります。

ブックマークの例

次のコードはブックマークの使い方の一例を示しています。

```
void DoSomething (const TTable *Tbl)
{
  TBookmark Bookmark = Tbl->GetBookmark(); // メモリを割り当て，値を代入する
  Tbl->DisableControls(); // データコントロールのレコード表示をオフにする
  try
  {
    for (Tbl->First(); !Tbl->Eof; Tbl->Next()) // テーブルの各レコードを繰り返し処理する
    {
      // 処理を実行する
      ...
    }
  }
  __finally
  {
    Tbl->GotoBookmark(Bookmark);
    Tbl->EnableControls(); // データコントロールのレコード表示をオンにする
    Tbl->FreeBookmark(Bookmark); // ブックマークのメモリ割り当てを解除する
  }
}
```

レコードの繰り返し処理に入る前に，途中経過を表示させないようにして速度を上げるためにコントロールは使用不可にされます。レコードの繰り返し処理中にエラーが発生すると，ループ処理が未完了であったとしても，**__finally** 節で，コントロールは常に使用可能にされ，ブックマークは常に解除されます。

データセットの検索

データセットが単方向でない場合は，Locate メソッドと Lookup メソッドを使ってデータセットを検索できます。この 2 つのメソッドは任意のデータセットのどの列でも検索ができます。

メモ TDataSet の下位オブジェクトの中には，インデックスに基づいてレコードを検索を行うためのメソッドを持つものもあります。これらのメソッドについては，22-27 ページの「インデックスを使ってレコードを検索する方法」を参照してください。

Locate メソッド

Locate メソッドは指定された検索条件に一致する最初の行にカーソルを移動します。もっとも簡単な形式としては，検索対象の列名，比較の基準になる項目値を Locate に渡し，さらに大文字小文字の区別なしの検索なのか，部分キー照合を使えるかなどを指定するオプションフラグを渡します。（部分キー照合とは，検索文字列として項目値の接頭文字列のみを必要とする検索です）。たとえば次のコードは，Company の列の値が「Professional Divers，Ltd.」であるような CustTable の最初の行にカーソルを移動します。

```
TLocateOptions SearchOptions;
SearchOptions.Clear();
SearchOptions << loPartialKey;
bool LocateSuccess = CustTable->Locate("Company", "Professional Divers, Ltd.",
        SearchOptions);
```

Locate メソッドでは一致すると，一致した最初のレコードが現在のレコードになります。一致するレコードが見つかった場合，Locate は **true** を返します。見つからなかった場合，Locate は **false** を返します。検索に失敗した場合，現在のレコードは変わりません。

Locate は，複数の列で複数の値を検索する場合に最も力を発揮します。検索する値は Variant 型であるため，検索条件としてさまざまなデータ型を指定できます。検索文字列に複数の列を指定するには，その文字列の各項目をセミコロンで区切ります。

検索値は Variant 型であるため，複数の値を渡す場合は，たとえば Lookup メソッドからの戻り値などの Variant 配列を引数として指定するか，VarArrayOf 関数を使ってコード内で Variant 配列を作成しなければなりません。次のコードは，複数の検索値と部分キー照合を使って複数の列で検索するコードの例です。

```
TLocateOptions Opts;
Opts.Clear();
Opts << loPartialKey;
Variant locvalues[2];
locvalues[0] = Variant("Sight Diver");
locvalues[1] = Variant("P");
CustTable->Locate("Company;Contact", VarArrayOf(locvalues, 1), Opts);
```

Locate は可能な限り速い方法を使って一致するレコードの位置を突き止めます。検索する列がインデックス付きで，そのインデックスが指定の検索オプションと互換性があれば，Locate はそのインデックスを使います。

Lookup メソッド

Lookup メソッドは指定された検索条件に一致する最初の行を探します。一致する行が見つかると，そのデータセットに関連した計算項目と参照項目が必ず再計算され，一致した行から 1 つまたは複数の項目が返されます。Lookup はカーソルを一致した行には移動せず，そこから値を返すだけです。

もっとも簡単な形式としては，検索対象の列名，比較の基準になる項目値，返すべき 1 つまたは複数の項目を Lookup に渡します。たとえば次のコードは，Company 列の値が「Professional Divers，Ltd.」であるような CustTable の最初の行を検索し，会社名，担当者，会社の電話番号を返します。

```
Variant LookupResults = CustTable->Lookup("Company", "Professional Divers, Ltd",
    "Company;Contact;Phone");
```

Lookup メソッドは，検索の結果最初に一致したレコードから，指定された項目の値を返します。戻り値は Variant 型です。複数の値を返すように指定すると，Lookup は Variant 配列を返します。一致するレコードがない場合，Lookup はヌルの Variant を返します。Variant 配列についての詳細は，オンラインヘルプを参照してください。

Lookup は，複数の列で複数の値を検索する場合に最も力を発揮します。複数の列や結果項目を持つ文字列を指定するには，その文字列の各項目をセミコロンで区切ります。

検索値は Variant 型であるため，複数の値を渡す場合は，たとえば Lookup メソッドからの戻り値などの Variant 配列を引数として指定するか，VarArrayOf 関数を使ってコード内で Variant 配列を作成しなければなりません。次のコードは，複数の列で Lookup の検索をするコードの例です。

```
Variant LookupResults;
Variant locvalues[2];
```

```
Variant v;
locvalues[0] = Variant("Sight Diver");
locvalues[0] = Variant("Kato Paphos");
LookupResults = CustTable->Lookup("Company;City", VarArrayOf(locvalues, 1),
                  "Company;Addr1;Addr2;State;Zip");
// ここで結果を（どこかで作成しておいた）グローバル文字列リストに入れる
pFieldValues->Clear();
for (int i = 0; i < 5; i++) // Lookup は要求された 5 項目を呼び出す
{
  v = LookupResults.GetElement(i);
  if (v.IsNull())
    pFieldValues->Add("");
  else
    pFieldValues->Add(v);
}
```

Locate と同様に，Lookup は可能な限り速い方法を使って一致するレコードの位置を突き止めます。検索する列がインデックス付きであれば，Lookup はそのインデックスを使います。

フィルタを使って編集する

ほとんどの場合，データセットのレコードのうち，アプリケーションの操作対象になるものはほんの一部です。たとえば，顧客データベースの中のカリフォルニアを拠点とする会社のレコードだけを取り出したり表示したい場合や，特定の項目値セットを持つレコードを見つけたい場合などです。このような場合，フィルタを使うと，アプリケーションがデータセット内の特定のレコードだけにアクセスするようになります。

単方向データセットの場合，データセット内のレコードを制限するには，データセット内のレコードを限定する問い合わせを使うしかありません。TDataSet の他の下位オブジェクト場合，取得済みのデータに対してそのサブセットを定義できます。アプリケーションがアクセスするレコードをデータセット内の特定のレコードだけに限定するには，フィルタを使うことができます。

フィルタは条件を指定し，その条件に一致するレコードだけが表示されます。フィルタ条件は，データセットの Filter プロパティで指定することも，OnFilterRecord イベントハンドラに記述することもできます。フィルタ条件は，指定した数のデータセット項目にインデックスが付いているかどうかにかかわらず，これらの項目の値に基づいて設定されます。たとえば，カリフォルニアを拠点とする会社のレコードだけを表示するには，State 項目の値が「CA」であるレコードを探す単純なフィルタを使います。

メモ フィルタはデータセットで取り出したすべてのレコードに対して適用されます。大量のデータをフィルタにかける場合は，問い合わせを使ってレコードの取り出しを制限するか，インデックス付きテーブル上で範囲指定をする方がフィルタを使うよりも効果的なことがあります。

フィルタの設定と解除

データセットでフィルタを設定する手順は次のとおりです。

1. フィルタを作成します。
2. 必要に応じて，文字列に基づくフィルタテストのオプションを設定します。

3. Filtered プロパティを **true** に設定します。

フィルタを設定すると，フィルタ条件に一致するレコードだけをアプリケーションで使えます。フィルタは一時的な条件にすぎません。フィルタを解除するには，Filtered プロパティを **false** に設定します。

フィルタの作成

データセットのフィルタを作成するには，以下の 2 通りの方法があります。

- Filter プロパティで単純なフィルタ条件を指定する。Filter プロパティは実行時にフィルタを作成して適用する場合に特に便利である
- OnFilterRecord イベントハンドラを記述する。単純なフィルタ条件も複雑なフィルタ条件も作成できる。OnFilterRecord では設計時にフィルタ条件を指定する。Filter プロパティでは 1 つの文字列でフィルタ論理を指定しなければならないが，OnFilterRecord イベントハンドラの場合は論理を分岐またはループさせて複雑な多重フィルタ条件を作成できる

Filter プロパティを使ってフィルタを作成する主な利点は，アプリケーションが動的に，たとえばユーザーの入力に応じて，フィルタの作成，変更，適用ができることです。欠点は，フィルタ条件を 1 つのテキスト文字列で表さなければならないことです。また，分岐やループの構造を利用できず，データセットにまだ存在していない値との比較やテストもできません。

OnFilterRecord イベントの利点は，複雑で調整可能なフィルタを作成でき，分岐やループの構造を使う複数行にわたるコードを使えることです。データセットの値をそのデータセットにない値（編集ボックス内のテキストなど）と比較してテストすることもできます。OnFilterRecord イベントを使う場合の欠点は，フィルタを設計時に設定するのでユーザーの入力に応じて変更できないことです。ただし，複数のフィルタハンドラを作成して，一般的なアプリケーション条件に応じてフィルタハンドラを切り替えることができます。

以降の節では，Filter プロパティと OnFilterRecord イベントハンドラを使ってフィルタを作成する方法について説明します。

Filter プロパティの設定

Filter プロパティを使ってフィルタを作成するには，このプロパティの値として，フィルタのテスト条件を示す文字列を設定します。たとえば次の文は，データセットの State 項目をテストして，カリフォルニア州を表す値が入っているかどうか調べるフィルタを作成します。

```
Dataset1->Filter = "State = 'CA'";
```

ユーザーが入力したテキストに基づいて Filter の値を指定することもできます。たとえば次の文は，編集ボックスのテキストを Filter に代入します。

```
Dataset1->Filter = Edit1->Text;
```

ハードコードされたテキストと，ユーザーが入力したデータの両方に基づいて文字列を作成することもできます。

```
Dataset1->Filter = AnsiString("State = '") + Edit1->Text + "'";
```

フィルタ中に明示的に設定しない限り，空白の項目値は現れません。

```
Dataset1->Filter = "State <> 'CA' or State = BLANK";
```

メモ Filter の値を指定した後にフィルタをデータセットに適用するには，Filtered プロパティを **true** に設定します。

フィルタで次の比較演算子と論理演算子を使うと，項目値どうしの論理演算のほか，項目値をリテラルまたは定数と比較できます。

表 22.4 フィルタに使用できる比較演算子と論理演算子

演算子	説明
<	より小さい
>	より大きい
>=	以上
<=	以下
=	等しい
<>	等しくない
AND	2 つの文がどちらも **true** かどうか調べる
NOT	次の文が **true** でないか調べる
OR	2 つの文のどちらか一方または両方が **true** か調べる
+	数値の加算，文字列の連結，または日付 / 時刻値に数値を加算する（一部のドライバでのみ使用可能）
-	数値の減算，日付の減算，または日付から数値を減算する（一部のドライバでのみ使用可能）
*	2 つの数値を乗算する（一部のドライバでのみ使用可能）
/	2 つの数値で除算を行う（一部のドライバでのみ使用可能）
*	部分的な比較のためのワイルドカード（FilterOptions には foPartialCompare が含まれていなければならない）

これらの演算子を組み合わせて使うと，高度なフィルタを作成できます。たとえば次の文は，2 つのテスト条件を満足していることを確認してから，レコードの表示を認めます。

```
(Custno > 1400) AND (Custno < 1500);
```

メモ フィルタがオンのとき，ユーザーがレコードを編集することによって，そのレコードがフィルタのテスト条件に一致しなくなることがあります。そのレコードを次にデータセットから取り出したときには，それが消えているように見えるかもしれません。その場合，そのフィルタ条件に一致した次のレコードが現在のレコードになります。

イベントハンドラ OnFilterRecord の記述方法

レコードを取り出すたびにデータセットが生成する OnFilterRecord イベントを使用してレコードのフィルタ処理を行うコードを書くことができます。このイベントハンドラは，あるレコードをアプリケーションで表示すべきかどうかを決めるテストを実行します。

レコードがフィルタ条件に一致しているかどうかを示すには，OnFilterRecord フィルタハンドラの Accept パラメータを **true**（レコードの採用）または **false**（レコードの除外）に設定しなければなりません。たとえば次のようなフィルタは，State 項目が「CA」に設定されているレコードだけを表示します。

```
void __fastcall TForm1::Table1FilterRecord(TDataSet *DataSet; bool &Accept)
{
  Accept = DataSet->FieldByName["State"]->AsString == "CA";
}
```

フィルタを設定すると，レコードを取り出すたびに OnFilterRecord イベントが発生します。イベントハンドラはそのレコードを調べて，フィルタ条件に一致するレコードだけを表示します。
OnFilterRecord イベントはデータセット内のすべてのレコードに対して発生するので，処理効率に影響しないように，このイベントハンドラはできるだけ簡潔に記述しておかなければなりません。

実行時にフィルタのイベントハンドラを切り替える

OnFilterRecord イベントハンドラは必要なだけ記述し，実行時に切り替えることができます。たとえば次の文は，NewYorkFilter という OnFilterRecord イベントハンドラに切り替えます。

```
DataSet1->OnFilterRecord = NewYorkFilter;
Refresh();
```

フィルタオプションの設定

FilterOptions プロパティを使うと，フィルタで文字列ベースの項目を比較するときに部分的な比較に基づいてレコードを受け付けるかどうか，また，大文字と小文字を区別して文字列を比較するかどうかを指定できます。FilterOptions は集合プロパティです。空集合にすることも（デフォルト），以下の値のどちらか一方または両方を含むこともできます。

表 22.5　FilterOptions の値

値	説明
foCaseInsensitive	文字列を比較するときに大文字と小文字を区別しない
foNoPartialCompare	部分的一致は受け付けない（たとえば，* で終わる文字の比較結果は不一致）

たとえば次の文は，State 項目で値を比較するときに大文字と小文字を区別しないフィルタを設定します。

```
TFilterOptions FilterOptions;
FilterOptions->Clear();
FilterOptions << foCaseInsensitive;
Table1->FilterOptions = FilterOptions;
Table1->Filter = "State = 'CA'";
```

フィルタが設定されたデータセット内のレコード間の移動

フィルタが設定されたデータセット内のレコード間を移動するには，以下の 4 つのデータセットメソッドが使えます。表にこれらのメソッドとその用途を示します。

表 22.6　フィルタが設定されたデータセット内の移動メソッド

メソッド	目的
FindFirst	現在のフィルタ条件に一致する最初のレコードに移動する。最初に一致するレコードの検索は，フィルタが設定されていないデータセットの最初のレコードから始まる
FindLast	現在のフィルタ条件に一致する最後のレコードに移動する
FindNext	フィルタが設定されたデータセットの現在のレコードから次のレコードに移動する
FindPrior	フィルタが設定されたデータセットの現在のレコードから 1 つ前のレコードに移動する

たとえば次の文は，データセットで最初にフィルタ処理されたレコードを探します。

```
DataSet1->FindFirst();
```

アプリケーションで Filter プロパティを設定するか，OnFilterRecord イベントハンドラを作成すると，フィルタが現在設定されているかどうかに関係なく，これらのメソッドは指定されたレコードにカーソルを移動します。フィルタが設定されていないときにこのメソッドを呼び出すと，次のような処理が行われます。

- 一時的にフィルタが設定される
- 一致するレコードが見つかればそのレコードにカーソルが移動する
- フィルタが解除される

メモ フィルタが使用不可になっていて Filter プロパティを設定したり OnFilterRecord イベントハンドラを作成しなければ，これらのメソッドは First()，Last()，Next()，および Prior() メソッドと同じように動作します。

移動用フィルタメソッドはどれも，一致するレコードが見つかるとそのレコードにカーソルを移動し，そのレコードを現在のレコードにしてから **true** を返します。一致するレコードが見つからないと，カーソルの位置はそのままで変わらず，これらのメソッドは **false** を返します。これらの呼び出しをラップするために Found プロパティのステータスをチェックし，Found が **true** のときだけアクションを起こすことができます。たとえば，データセットで一致する最後のレコードにカーソルがすでにあるときに FindNext を呼び出すと，このメソッドは **false** を返し，現在のレコードは変化しません。

データの変更

CanModify 読み出し専用プロパティが **true** の場合，データの挿入，更新，削除には以下のデータセットメソッドを使います。CanModify は，データセットが単方向である場合，データセットの基になるデータベースが読み出し特権と書き込み特権を許可していない場合，またはほかの何らかの要因が妨げている場合を除いて **true** です（妨げとなる要因としては，あるデータセット上の ReadOnly プロパティ，または TQuery コンポーネント上の RequestLive プロパティがあります）。

表 22.7　データの挿入，更新，および削除のためのデータセットのメソッド

メソッド	説明
Edit	データセットが dsEdit や dsInsert の状態でない場合は，dsEdit 状態にする
Append	保留状態のデータを登録し，現在のレコードをデータセットの最後に移動し，データセットを dsInsert 状態にする
Insert	保留状態のデータを登録し，データセットを dsInsert 状態にする
Post	新規のレコードや変更されたレコードをデータベースに登録しようと試みる。正常終了の場合，データセットは dsBrowse 状態になる。異常終了の場合，データセットの状態は変わらない
Cancel	現在の動作を取り消して，データセットを dsBrowse 状態にする
Delete	現在のレコードを削除して，データセットを dsBrowse 状態にする

レコードの編集

アプリケーションでレコードを変更するには，その前にデータセットを dsEdit モードにしなければなりません。データセットの CanModify 読み出し専用プロパティが **true** であれば，コードで Edit メソッドを使ってデータセットを dsEdit モードにできます。

データセットが dsEdit モードに移行すると，最初に BeforeEdit イベントを受け取ります。編集モードへ正常に移行すると，データセットは AfterEdit イベントを受け取ります。通常，これらのイベントは，データセットの現在状態を示すユーザーインターフェースを更新するために使用します。なんらかの理由でデータセットが編集モードに入れないと，OnEditError イベントが発生します。その場合，問題をユーザーに通知したり，データセットが編集モードに入るのを妨げた状況を修正することができます。

アプリケーションのフォーム上では，以下の条件がすべて満たされると，データベース対応コントロールのいくつかはデータセットを自動的に dsEdit 状態にできます。

- コントロールの ReadOnly プロパティがデフォルトの **false** である
- コントロールのデータソースの AutoEdit プロパティが **true** である
- データセットの CanModify プロパティが **true** である

メモ データセットが dsEdit 状態になっていても，アプリケーションユーザーに適切な SQL アクセス権がない場合には，SQL ベースのデータベースでレコードの編集ができないこともあります。

いったんデータセットが dsEdit モードになると，ユーザーはフォーム上のデータベース対応コントロールに表示される現在のレコードの項目値を変更できます。グリッド中の別のレコードに移動するなどの現在のレコードを変える操作をユーザーが実行した場合，編集が有効になっているデータベース対応コントロールは自動的に Post を呼び出します。

フォーム上にナビゲータコンポーネントがある場合，ユーザーはそのナビゲータの編集の取り消しボタンをクリックして編集を取り消せます。編集を取り消すとデータセットは dsBrowse 状態に戻ります。

コードでは，適切なメソッドを呼び出すことによって，編集を書き込むか取り消すかのどちらかを選ばなければなりません。Post の呼び出しでは変更を書き込みます。Cancel の呼び出しでは取り消します。コードで Edit と Post は，よく一緒に使います。例を示します。

```
Table1->Edit();
Table1->FieldValues["CustNo"] = 1234;
Table1->Post();
```

上の例で，コードの最初の行はデータセットを dsEdit モードにします。次の行は，現在のレコードの CusNo 項目に「1234」という数字を代入します。最後の行は，変更されたレコードを書き込んで登録します。更新をキャッシュしていなければ，登録はデータベースに変更を書き込みます。更新をキャッシュしていれば，変更は一時バッファに書き込まれて，データセットの ApplyUpdates メソッドが呼び出されるまで，バッファの中にとどまります。

新規レコードの追加

アプリケーションが新規レコードを追加するには，その前にデータセットが dsInsert モードになっていなければなりません。データセットの CanModify 読み出し専用プロパティが **true** であれば，コードの中で Insert メソッドや Append メソッドを使ってデータセットを dsInsert モードにできます。

データセットが dsInsert モードに移行すると，最初に BeforeInsert イベントを受け取ります。挿入モードへ正常に移行すると，データセットはまず OnNewRecord イベント，それから AfterInsert イベントを受け取ります。これらのイベントは，たとえば，新しく挿入されたレコードに初期値を設定するために使用することができます。

```
void __fastcall TForm1::OrdersTableNewRecord(TDataSet *DataSet)
{
  DataSet->FieldByName("OrderDate")->AsDateTime = Date();
}
```

アプリケーションのフォーム上では，以下の条件が両方満たされると，データベース対応のグリッドコントロールとナビゲータコントロールがデータセットを dsInsert 状態にできます。

- コントロールの ReadOnly プロパティがデフォルトの **false** である
- データセットの CanModify プロパティが **true** である

メモ データセットが dsInsert 状態になっていても，アプリケーションユーザーに適切な SQL アクセス権がない場合には，SQL ベースのデータベースでレコードの追加ができないこともあります。

いったんデータセットが dsInsert モードになると，ユーザーまたはアプリケーションは新規レコードの項目に値を入力できます。グリッドコントロールとナビゲータコントロールを除いて，ユーザーには Insert と Append は同じに見えます。Insert メソッドを呼び出すと，現在のレコードだった場所の上のグリッドに空行が表示されます。Append メソッドを呼び出すと，グリッドはデータセットの最後のレコードにスクロールし，グリッドの最後に空行が表示されます。そして，そのデータセットに関連したナビゲータコンポーネントでは次のレコードボタンと最後のレコードボタンが淡色表示になります。

グリッド中の別のレコードに移動するなどの現在のレコードを変更する操作をユーザーが実行した場合，挿入が有効になっているデータベース対応コントロールは自動的に Post を呼び出します。これ以外の場合は，コードで Post を呼び出さなければなりません。

Post メソッドは新規レコードをデータベースに書き込みます。または，更新をキャッシュしている場合には，Post はレコードをメモリ内のキャッシュに書き込みます。キャッシュに格納された挿入および追加の内容をデータベースに書き込むには，データセットの ApplyUpdates メソッドを呼び出します。

レコードを挿入する

Insert メソッドは現在のレコードの前に空のレコードを新しく開き，ユーザーまたはアプリケーションのコードが項目値を入力できるよう，空のレコードを現在のレコードにします。

アプリケーションが Post メソッドを呼び出した場合（キャッシュアップデートを使用するときには ApplyUpdates メソッドを呼び出した場合），挿入されたレコードは以下の 3 通りの方法のいずれかでデータベースに書き込まれます。

- インデックス付きの Paradox テーブルおよび dBASE テーブルの場合，レコードはデータセットのそのインデックスに基づいた場所に挿入される

- インデックスの付いていない Paradox テーブルおよび dBASE テーブルの場合，データセットの現在の位置にレコードが挿入される
- SQL データベースの場合，物理的な挿入場所はシステムによって異なる。テーブルがインデックス付きであれば，新規レコード情報と一緒にインデックスも更新される

レコードの追加

Append メソッドはデータセットの最後に空のレコードを新しく開き，ユーザーまたはアプリケーションのコードが項目値を入力できるよう，空のレコードを現在のレコードにします。

アプリケーションが Post メソッドを呼び出した場合（キャッシュアップデートを使用するときには ApplyUpdates メソッドを呼び出した場合），追加されたレコードは以下の 3 通りの方法のいずれかでデータベースに書き込まれます。

- インデックス付きの Paradox テーブルおよび dBASE テーブルの場合，レコードはデータセットのそのインデックスに基づいた場所に挿入される
- インデックスの付いていない Paradox テーブルおよび dBASE テーブルの場合，レコードはデータセットの最後に追加される
- SQL データベースの場合，物理的な追加場所はシステムによって異なる。テーブルがインデックス付きであれば，新規レコード情報とともにインデックスも更新されます。

レコードの削除

アクティブなデータセットの現在のレコードを削除するには，Delete メソッドを使います。Delete メソッドが呼び出された場合

- データセットは BeforeDelete イベントを受け取る
- データセットは現在のレコードの削除を試みる
- データセットは dsBrowse 状態に戻る
- データセットは AfterDelete イベントを受け取る

BeforeDelete イベントハンドラでの削除を防ぎたい場合には，グローバルな Abort 手続きを呼び出してください。

```
void __fastcall TForm1::TableBeforeDelete (TDataSet *Dataset)
{
  if (MessageBox(0, "Delete this record?", "CONFIRM", MB_YESNO) != IDYES)
    Abort();
}
```

Delete が失敗すると，OnDeleteError イベントが生成されます。OnDeleteError イベントハンドラがこの問題を修正できない場合，データセットは dsEdit 状態のままです。Delete は成功すると，データセットを dsBrowse 状態に戻します。削除されたレコードの次のレコードが，現在のレコードになります。

更新をキャッシュしている場合，この削除レコードは，ApplyUpdates を呼び出すまでは，基となるデータベーステーブルから削除されません。

データセットに関連付けられたナビゲータコンポーネントがある場合，ユーザーはそのナビゲータのレコード削除ボタンをクリックして現在のレコードを削除できます。コードで現在のレコードを削除するには，明示的に Delete を呼び出さなければなりません。

データの登録

レコードの編集を終えたなら，Post メソッドを呼び出して変更を書き出さなければなりません。Post メソッドは，データセットの状態，および更新をキャッシュしているかどうかに応じて，異なった動作を行います。

- 更新をキャッシュしておらず，データセットが dsEdit か dsInsert 状態であれば，Post は現在のレコードをデータベースに書きこみ，データセットを dsBrowse 状態に戻す
- 更新をキャッシュしており，データセットが dsEdit か dsInsert 状態であれば，Post は現在のレコードを内部キャッシュに書きこみ，データセットを dsBrowse 状態に戻す。編集結果は，ApplyUpdates を呼び出すまではデータベースに書き込まれない
- データセットが dsSetKey 状態であれば，Post はデータセットを dsBrowse 状態に戻す

データセットの初期状態にはかかわりなく，Post は現在の変更を書き込む前と後に，BeforePost と AfterPost イベントを生成します。これらのイベントは，ユーザーインターフェースを更新するため，または，Abort 手続きを呼び出して，データセットが変更を登録するのを防ぐために使用することができます。Post の呼び出しが失敗すると，データセットは OnPostError イベントを受け取ります。その場合，問題をユーザーに通知したり，修正を試みたりすることができます。

登録は明示的に実行することも，別の手続きの一部として暗黙的に実行することもできます。アプリケーションが現在のレコードから移動すると，Post メソッドが暗黙的に呼び出されます。First, Next, Prior, および Last メソッドを呼び出すと，テーブルが dsEdit または dsInsert モードであれば，Post が実行されます。Append と Insert メソッドも，未登録データがあれば暗黙的に登録します。

注意 Close メソッドは，Post を暗黙的に呼び出すことはしません。保留状態の編集内容を登録するには，BeforeClose イベントを明示的に使ってください。

変更の取り消し

アプリケーションが直接または間接に Post メソッドを呼び出していなければ，現在のレコードに加えられた変更はいつでも取り消せます。たとえば，データセットが dsEdit モードになっていてユーザーが 1 つまたは複数の項目のデータを変更したとします。このときアプリケーションはデータセットの Cancel メソッドを呼び出すことでレコードを元の値に戻せます。Cancel を呼び出すとデータセットは必ず dsBrowse 状態に戻ります。

アプリケーションが Cancel を呼び出したとき，データセットが dsEdit か dsInsert モードだったなら，現在のレコードが元の値に戻される前と後に，BeforeCancel と AfterCancel イベントを受け取ります。

ユーザーが編集，挿入，追加の各操作を取り消せるようにするには，フォーム上でデータセットに関連したナビゲータコンポーネントに編集の取り消しボタンを組み入れます。あるいは，取り消し用のボタンをフォーに置いてコードを記述することもできます。

レコード全体の変更

フォーム上では，グリッドとナビゲータを除くすべてのデータベース対応コントロールはレコード中の 1 つの項目にアクセスします。

ただし，データセットの基になるデータベーステーブルの構造が安定していて変わらないならば，レコード構造全体を処理する以下のメソッドをプログラムから使えます。次の表にレコードの個々の項目ではなくレコード単位で処理するメソッドを示します。

表 22.8　レコード全体を操作するメソッド

メソッド	説明
AppendRecord ([値の配列])	指定された列値を持つレコードをテーブルの最後に追加する。Append に似ている。暗黙的に Post を呼び出す
InsertRecord ([値の配列])	テーブルの現在のカーソル位置の直前に，指定された値をレコードとして挿入する。Insert と似ている。暗黙的に Post を呼び出す
SetFields ([値の配列])	対応する項目の値を設定する。TFields に値を代入することに似ている。アプリケーションは明示的に Post を呼び出さなければならない

これらのメソッドは，引数として値の配列をとります。配列の各値は，基になるデータセットの列に対応します。配列を作成するには ARRAYOFCONST マクロを使います。値に指定できるのは，リテラル，変数，ヌルです。引数の値の数がデータセットの列数より少ない場合，残りの値はヌルとみなされます。

インデックスのないデータセットでは，AppendRecord はデータセットの末尾にレコードを追加し，InsertRecord は現在のカーソル位置の後ろにレコードを挿入します。インデックス付きのデータセットでは，どちらのメソッドも，インデックスに基づいてテーブル内の適正な位置にレコードを配置します。いずれの場合も，これらのメソッドによって，カーソルはそのレコード位置に移動します。

SetFields メソッドはパラメータ配列で指定された値をデータセットの項目に代入します。SetFields を使うには，アプリケーションが最初に Edit を呼び出してデータセットを dsEdit モードにしなければなりません。変更内容を現在のレコードに適用するには Post を実行しなければなりません。

既存のレコードのすべての項目ではなく，一部の項目を変更するために SetFields メソッドを使う場合は，変更したくない項目についてヌル値を渡すことができます。指定した値の数がレコードのすべての項目数より少ない場合，SetFields はヌル値を足りない分に代入します。

たとえば，データベースに Name，Capital，Continent，Area，Population という列を持つ COUNTRY というテーブルがあるとします。CountryTable という TTable コンポーネントが COUNTRY テーブルにリンクされていれば，次の文によってレコードが COUNTRY テーブルに挿入されます。

```
CountryTable->InsertRecord(ARRAYOFCONST(("Japan", "Tokyo", "Asia")));
```

この文では Area と Population の値を指定していないため，その 2 つの項目にはヌルが挿入されます。

テーブルは Name にインデックスが付いているので，次の文は「Japan」をアルファベット順に照合した結果に基づいてレコードを挿入します。

レコードを更新するには，たとえば次のようなコードを使えます。

```
TLocateOptions SearchOptions;
SearchOptions->Clear();
SearchOptions << loCaseInsensitive;
if (CountryTable->Locate("Name", "Japan", SearchOptions))
{
  CountryTable->Edit();
  CountryTable->SetFields(ARRAYOFCONST(((void *)NULL, (void *)NULL, (void *)NULL,
        344567, 164700000)));
```

```
    CountryTable->Post();
  }
```

このコードは Area と Population の項目に値を代入し，データベースに登録します。3 つのヌルポインタは，最初の 3 列の現在の内容を保持するためのプレースホルダとして機能します。

注意 SetFields メソッドでヌルポインタを使って一部の項目値をそのままにしておく場合，必ずヌルを **void** * にキャストしてください。ヌルをキャストしないでパラメータとして使うと，項目は空白値に設定されます。

項目を計算する

項目エディタを使えば，データセットに計算項目を定義できます。データセットに計算項目が含まれている場合には，それらの項目の値を計算するコードを OnCalcFields イベントハンドラ内に記述してください。項目エディタを使って計算項目を定義する方法についての詳細は，23-7 ページの「計算項目の定義」を参照してください。

AutoCalcFields プロパティによって，OnCalcFields がいつ呼び出されるかが決まります。AutoCalcFields が **true** であれば，次の時点で OnCalcFields が呼び出されます。

- データセットを開いたとき
- データセットが編集モードに入ったとき
- レコードをデータベースから取り出したとき
- フォーカスがあるビジュアルコンポーネントから別のビジュアルコンポーネントへ移動したり，データベース対応グリッドコントロールのある列から別の列へ移動したとき

AutoCalcFields プロパティが **false** の場合，レコードの個々の項目の編集中（上の 4 番目の条件）は，OnCalcFields は呼び出されません。

注意 OnCalcFields イベントは頻繁に呼び出されるため，コードを短くする必要があります。また，AutoCalcFields が **true** の場合は OnCalcFields は再帰的に呼び出されるため，データセット（またはマスター / 詳細関係の一部分の場合はリンクされたデータセット）を変更する操作は実行しないようにします。たとえば，OnCalcFields が Post を実行して AutoCalcFields が **true** ならば，OnCalcFields が再び呼び出されて別の Post が実行され，これがずっと続きます。

OnCalcFields を実行すると，データセットは dsCalcFields モードに入ります。このモードでは，イベントハンドラ自身が変更する計算項目を除いて，レコードに対する変更や追加はできません。目的とする計算項目以外の変更ができないのは，OnCalcFields が，ほかの項目の値を使って計算項目の値を決定するからです。もし，計算項目を処理している間に関係する項目が変更できたとすると，得られる結果は信頼できないものになるでしょう。OnCalcFields イベントが終了すると，データセットは dsBrowse 状態に戻ります。

データセットの種類

22-2 ページの「TDataSet の下位オブジェクトの使い方」では，TDataSet の下位オブジェクトを，データにアクセスするために使う方式によって分類しています。TDataSet の下位オブジェクトを分

類する上で役立つもう 1 つの方法は，それらが表わすサーバーデータの種類を考慮に入れることです。この点からすると，データセットは基本的に 3 種類に分類できます。

- **テーブルタイプのデータセット**：データベースサーバーの単一のテーブルを表わし，その行と列全体を含みます。テーブルタイプのデータセットには，TTable，TADOTable，TSQLTable，および TIBTable が含まれます。

  テーブルタイプのデータセットでは，サーバー上で定義されたインデックスを利用できます。データベーステーブルとデータセットは 1 対 1 に対応しているので，データベーステーブル用に定義されたサーバーインデックスを使用することができます。インデックスを使えば，アプリケーションで，テーブルのレコードのソート，スピードサーチおよびルックアップを行うことが可能になります。また，マスター / 詳細関係の基礎ともなります。いくつかのテーブルタイプのデータセットでは，データセットとデータベーステーブルの 1 対 1 関係を利用して，データベーステーブルの作成や削除のような，テーブルレベルの操作を可能にしています。

- **問い合わせタイプのデータセット**：単一の SQL コマンド，つまり問い合わせを表わします。問い合わせは，コマンド（典型的には SELECT 文）の実行の結果セットを表わすことができます。また，レコードを返さないコマンドを実行することもできます（UPDATE 文など）。問いあわせタイプのデータセットには，TQuery，TADOQuery，TSQLQuery，および TIBQuery が含まれます。

  問い合わせタイプのデータセットを効果的に使用するには，SQL とサーバーの SQL 実装の詳細（SQL-92 標準に対する機能拡張や制約など）をよく知っていなければなりません。SQL を扱うのが初めての方は，SQL を詳細に解説した市販の解説書を参照してください。「Understanding the New SQL: A Complete Guide」（Jim Melton，Alan R.Simpson 共著，Morgan Kaufmann Publishers）は好著の 1 つです。

- **ストアドプロシージャタイプのデータセット**：データベースサーバー上のストアドプロシージャを表します。ストアドプロシージャタイプのデータセットには，TStoredProc，TADOStoredProc，TSQLStoredProc，および TIBStoredProc が含まれます。

  ストアドプロシージャは，使用されるデータベースシステム固有のプロシージャ / トリガー言語で書かれた自己包含型のプログラムです。通常は，頻繁に繰り返されるデータベース関連のタスクを行うもので，多数のレコード上での操作や，集合関数や算術関数を使う操作でとくに有効です。一般にストアドプロシージャを使用すると，次の点でデータベースアプリケーションの処理効率が改善されます。

  - サーバーのより大きな処理能力と速度を利用する
  - 処理をサーバーに移動することにより，ネットワークトラフィックを減少させる

  ストアドプロシージャには，データを返すものと返さないものがあります。データを返すストアドプロシージャにも，1 つのカーソルを返すもの（SELECT 問い合わせの結果に類似），複数のカーソルを返すもの（事実上複数のデータセットを返す），出力パラメータでデータを返すものがあります。これは，一部にはサーバーの違いによるものです。ストアドプロシージャでデータを返せないサーバーや，出力パラメータのみが使用できるサーバーがあります。ストアドプロシージャをまったくサポートしないサーバーもあります。実際に何が利用できるかは，サーバーのドキュメントを参照してください。

メモ 通常は，問い合わせタイプのデータセットを使用して，ストアドプロシージャを実行することができます。ほとんどのサーバーが，ストアドプロシージャを扱う SQL の拡張を提供しているからです。

ただし，そのために使う構文はサーバーごとに異なります。ストアドプロシージャタイプのデータセットのかわりに問い合わせタイプのデータセットを使う場合には，サーバーのドキュメントを参照して，構文を確認してください。

TDataSet には，これら 3 つのカテゴリのいずれかにきちんと収まるデータセットのほかに，複数のカテゴリにまたがる下位オブジェクトもあります。

- TADODataSet と TSQLDataSet には CommandType プロパティがあり，テーブル，問い合わせ，ストアドプロシージャのどれを表すかを指定することができます。TADODataSet は，テーブルタイプのデータセットのようにインデックスを指定できますが，プロパティとメソッドの名前は問い合わせタイプのデータセットに似ています。
- TClientDataSet は，別のデータセットからのデータを表します。そのため，テーブル，問い合わせ，ストアドプロシージャのいずれをも表すことができます。TClientDataSet の動作は，テーブルタイプのデータセットに最も似ています。インデックスをサポートしているからです。しかし，問い合わせおよびストアドプロシージャの一部の機能も持っています。パラメータの管理と，結果セットを取得せずに実行する機能です。
- ほかのいくつかのクライアントデータセット（TBDEClientDataSet および TSQLClientDataSet）には CommandType プロパティがあり，テーブル，問い合わせ，ストアドプロシージャのどれを表すかを指定することができます。プロパティとメソッド名は TClientDataSet のものに似ています。その中には，パラメータのサポート，インデックス，および結果セットを取得せずに実行する機能が含まれます。
- TIBDataSet は問い合わせとストアドプロシージャの両方を表わすことができます。実際のところ，同時に複数の問い合わせとストアドプロシージャを表わし，それぞれに対して別個のプロパティを持つことができます。

テーブルタイプのデータセットの使い方

テーブルタイプのデータセットを使う手順は次のとおりです。

1. 適切なデータセットコンポーネントをデータモジュール上かフォーム上に配置して，Name プロパティにアプリケーションに適したユニークな値を設定します。
2. 使用するテーブルを所持しているデータベースサーバーを設定します。設定の方法は，テーブルタイプのデータセットごとに異なっています。しかし通常は，データベース接続コンポーネントを指定します。
   - TTable の場合，DatabaseName プロパティを使用し，TDatabase コンポーネントまたは BDE エリアスを指定する
   - TADOTable の場合，Connection プロパティを使用し，TADOConnection コンポーネントを指定する
   - TSQLTable の場合，SQLConnection プロパティを使用し，TSQLConnection コンポーネントを指定する
   - TIBTable の場合，Database プロパティを使用し，TIBConnection コンポーネントを指定する

   データベース接続コンポーネントの使い方については，第 21 章「データベースへの接続」を参照してください。

3. TableName プロパティをデータベースのテーブル名に設定します。データベース接続コンポーネントがすでに指定されている場合は，ドロップダウンリストからテーブルを選択できます。
4. データソースコンポーネントをデータモジュールまたはフォームに入れ，その DataSet プロパティをデータセットの名前に設定します。データソースコンポーネントは，表示用にデータセットからデータベース対応コンポーネントに結果セットを渡すのに使います。

テーブルタイプのデータセットの利点

テーブルタイプのデータセットを使用する主要な利点は，インデックスが利用できることです。インデックスを使うと，アプリケーションで以下のことが可能になります。

- データセット内のレコードのソート
- レコードをすばやく見つけること
- 表示されるレコードの制限
- マスター / 詳細関係の確立

さらに，テーブルタイプのデータセットとデータベーステーブルの 1 対 1 関係により，これらの多くを，以下の目的で使用することができます。

- テーブルに対する読み書きの制御
- テーブルの作成と削除
- テーブルを空にする
- テーブルの同期

インデックスを持つレコードのソート

インデックスはテーブルのレコードの表示順を決めます。典型的には，レコードは一次インデックス，つまりデフォルトのインデックスに基づいて昇順で表示されます。このデフォルトの動作には，アプリケーションの介入は必要ありません。ただし，別のソート順を希望する場合は，以下の指定が必要です。

- 代替インデックス
- ソートの基準列のリスト（SQL ベース以外のサーバーでは使用不可）

インデックスを使えば，テーブルのデータを別な順序で表示できます。SQL ベースのテーブルでは，このソート順は，インデックスを使用し，テーブルのレコードを取得する問い合わせで ORDER BY 句を生成することにより実現されます。（Paradox や dBASE などの）ほかのテーブルでは，データアクセスメカニズムがインデックスを使用して，希望する順でレコードを表示します。

インデックス情報の取得

アプリケーションは，すべてのテーブルタイプのデータセットから，サーバー定義のインデックスに関する情報を取得するとができます。データセットで利用可能なインデックスのリストを取得するには，GetIndexNames メソッドを呼び出します。GetIndexNames は，有効なインデックス名の一覧を返します。たとえば次のコードは，リストボックスに，CustomersTable データセット用に定義されたすべてのインデックスの名前を設定します。

```
CustomersTable->GetIndexNames(ListBox1->Items);
```

メモ Paradox テーブルの場合，一次インデックスに名前は付かないので，GetIndexNames では一次インデックスは返されません。しかし，IndexName プロパティに空白の文字列を設定すれば，Paradox テーブル上のインデックスを一次インデックスに戻すことができます。

現在のインデックスの項目に関する情報を得るには，以下のプロパティを使用します。

- インデックスの列数を決める IndexFieldCount プロパティ
- インデックスを構成する列の項目コンポーネントのリストを調べる IndexFields プロパティ

次のコードは，IndexFieldCount プロパティと IndexFields プロパティを使ってアプリケーション内で列名リストを繰り返し処理する方法を示しています。

```
AnsiString ListOfIndexFields[20];
for (int i = 0; i < CustomersTable->IndexFieldCount; i++)
  ListOfIndexFields[i] = CustomersTable->IndexFields[i]->FieldName;
```

メモ IndexFieldCount は式インデックスで開いている dBASE テーブルでは無効です。

IndexName を使ったインデックスの指定

インデックスをアクティブにするには，IndexName プロパティを使います。アクティブになると，インデックスはデータセットのレコードの表示順を決めます（また，マスター / 詳細リンク，インデックスベースの検索，インデックスベースのフィルタ処理のための基礎として使用することもできます）。

インデックスをアクティブにするには，IndexName プロパティでインデックス名を設定します。データベースシステムによっては，一次インデックスの名前がない場合があります。その場合は，IndexName プロパティに空の文字列を設定します。

設計時には，IndexName プロパティの省略記号ボタンをクリックして，表示されるインデックスリストからインデックスを選ぶことができます。実行時に IndexName プロパティを設定するには，AnsiString（リテラルまたは変数）を使います。GetIndexNames メソッドを呼び出せば，データセットで利用可能なインデックスのリストを取得できます。

次のコードは，CustomersTable のインデックスを CustDescending に設定します。

```
CustomersTable->IndexName = "CustDescending";
```

IndexFieldNames を使ったインデックスの作成

希望するソート順を実現する定義済みのインデックスがない場合には，IndexFieldNames プロパティを使用して，疑似インデックスを作成することができます。

メモ IndexName と IndexFieldNames を同時に使用することはできません。一方のプロパティを設定すると，もう一方のプロパティの値がクリアされます。

IndexFieldNames の値は AnsiString です。ソート順を指定するには，使われる順に各列名をセミコロンで区切って指定します。ソートは昇順によるものだけです。ソートの大文字と小文字を区別するかどうかは，サーバーの機能によって決まります。詳細はサーバーのドキュメントを参照してください。

次のコードは PhoneTable のソート順を LastName に基づいて設定し，続いて FirstName に基づいて設定します。

```
PhoneTable->IndexFieldNames = "LastName;FirstName";
```

メモ Paradox や dBASE のテーブルで IndexFieldNames を使う場合，データセットは指定された列を使うインデックスを見つけようと試みます。そのようなインデックスを見つけられない場合は，例外を生成します。

インデックスを使ってレコードを検索する方法

TDataSet の Locate メソッドと Lookup メソッドを使えばすべてのデータセットで検索ができます。しかし，一部のテーブルタイプのデータセットでは，明示的にインデックスを使用すれば，Locate メソッドおよび Lookup メソッドが提供する検索の処理効率が向上します。

ADO データセットはすべて Seek メソッドをサポートしています。これは，現在のインデックス内の項目に対する項目値のセット基づいて，レコードに移動します。Seek を使えば，一致する最初または最後のレコードを基準にして，カーソルをどこに移動するかを指定できます。

TTable およびすべての種類のクライアントデータセットは，同様のインデックスベースの検索をサポートしていますが，関連するメソッドの組合せを使用します。次の表に，インデックスベースの検索をサポートするために TTable とクライアントデータセットが提供している 6 つの関連するメソッドについて要約します。

表 22.9 インデックスベースの検索メソッド

メソッド	目的
EditKey	検索キーバッファの現在の内容を保持してデータセットを dsSetKey 状態にし，検索の実行前にアプリケーションで既存の検索条件を変更できるようにします。
FindKey	SetKey メソッドと GotoKey メソッドを 1 つのメソッドに結合します。
FindNearest	SetKey メソッドと GotoKey メソッドを 1 つのメソッドに結合します。
GotoKey	検索条件と完全に一致する最初のレコードをデータセット内から検索して，見つかったレコードにカーソルを移動します。
GotoNearest	部分キー値に基づいて文字列項目で値がもっとも近いレコードを検索して，カーソルをそのレコードに移動します。
SetKey	検索キーバッファをクリアしてテーブルを dsSetKey 状態にし，検索実行前にアプリケーションで新しい検索条件を指定できるようにします。

GotoKey と FindKey は論理関数で，検索に成功すると一致したレコードにカーソルを移動して **true** を返します。検索に失敗するとカーソルは移動せず，**false** を返します。

GotoNearest と FindNearest は，完全に一致した最初のレコードにカーソルを移動します。一致したレコードが見つからないときは，指定された検索条件よりも大きい値で最初のレコードにカーソルを移動します。

Goto メソッドによる検索の実行

Goto メソッドを使用して検索を実行する手順は次のとおりです。

1. 検索に使用するインデックスを指定します。これはデータセット中のレコードをソートするのと同じインデックスです（22-25 ページの「インデックスを持つレコードのソート」を参照してください）。インデックスを指定するには，IndexName か IndexFieldNames プロパティを使用します。
2. データセットを開きます。
3. SetKey を呼び出して，データセットを dsSetKey 状態にします。

4. 検索値を Fields プロパティに指定します。Fields は TFields オブジェクトで，項目コンポーネントがインデックス付けされているリストを格納しています。その項目コンポーネントへは，対応する列の順番を指定してアクセスできます。データセットの最初の列番号は 0 です。
5. GotoKey または GotoNearest で検索して，一致した最初のレコードに移動します。

たとえばボタンの OnClick イベントに結び付けられた次のコードでは，GotoKey メソッドを使用して，インデックスの最初の項目値と編集ボックスの文字が完全に一致する最初のレコードに移動します。

```
void __fastcall TSearchDemo::SearchExactClick(TObject *Sender)
{
  ClientDataSet1->SetKey();
  ClientDataSet1->Fields->Fields[0]->AsString = Edit1->Text;
  if (!ClientDataSet1->GotoKey())
    ShowMessage("Record not found");
}
```

GotoNearest も同様です。GotoNearest は部分項目値にもっとも近いレコードを検索します。これは文字列項目にだけ使用できます。例を示します。

```
Table1->SetKey();
Table1->Fields->Fields[0]->AsString = "Sm";
Table1->GotoNearest();
```

最初のインデックス項目の 2 文字が「Sm」であるレコードが存在する場合，カーソルはそのレコードに移動します。そのようなレコードが見つからない場合，カーソルの位置は変わらず，GotoNearest は **false** を返します。

Find メソッドによる検索の実行

Find メソッドは Goto メソッドと同じではありませんが，両方とも dsSetKey 状態で検索するキー項目値を指定するために，データセットを明示する必要はありません。Find メソッドを使って検索を実行する手順は次のとおりです。

1. 検索に使用するインデックスを指定します。これはデータセット中のレコードをソートするのと同じインデックスです（22-25 ページの「インデックスを持つレコードのソート」を参照してください）。インデックスを指定するには，IndexName か IndexFieldNames プロパティを使用します。
2. データセットを開きます。
3. FindKey または FindNearest で検索して，一致した最初のレコードまたは値がもっとも近いレコードに移動します。どちらのメソッドも，カンマで区切られた項目値のリストをパラメータとして取ります。各項目値は，テーブル内のインデックスの付いた列に対応しています。

メモ FindNearest は，文字列項目にだけ使用できます。

検索成功後の現在レコードを指定する

デフォルトでは，検索に成功すると，検索条件に一致した最初のレコードにカーソルが移動します。KeyExclusive プロパティを **true** に設定すれば，一致した最初のレコードの次のレコードにカーソルを移動できます。

KeyExclusive のデフォルトは **false** で，検索が成功するとカーソルは一致した最初のレコードに移動します。

部分キーでの検索

データセットに 2 つ以上のキー列がある場合，そのキーの一部だけから値を検索したければ，KeyFieldCount を検索の対象とする列の数に設定します。たとえばクライアントデータセットの現在のインデックスに 3 列のキーがあり，最初の列だけで値を検索する場合は，KeyFieldCount を 1 に設定します。

複数列のキーを持つテーブルタイプのデータセットの場合，値の検索は第 1 列目から始まる連続した列でだけできます。たとえばキーが 3 列の場合，最初の列，最初の列と 2 番目の列，最初の列と 2 番目の列と 3 番目の列の組で値の検索はできますが，最初の列と 3 番目の列という組み合わせでは検索できません。

検索の繰り返しと拡張

SetKey や FindKey を呼び出すたびに，メソッドは Fields プロパティの前回の値をクリアします。前回設定した項目を使用して検索を繰り返したり，検索に使用する項目を追加したい場合は，SetKey や FindKey のかわりに EditKey を呼び出します。

たとえば，CityIndex インデックスの City 項目に基づいて，すでに Employee テーブルの検索を実行した場合を考えてみます。そして，CityIndex インデックスには，City 項目と Company 項目が存在するとします。指定した市区町村の指定した会社名の値を持つレコードを検索するには，次のコードを利用します。

```
Employee->KeyFieldCount = 2;
Employee->EditKey();
Employee->FieldValues["Company"] = Variant(Edit2->Text);
Employee->GotoNearest();
```

範囲でレコードを制限する

フィルタを使うとデータセットのデータの一部だけを一時的に表示して編集できます（22-12 ページの「フィルタを使って編集する」を参照してください）。いくつかのテーブルタイプのデータセットは，使用可能なレコードのサブセット（範囲と呼ぶ）にアクセスする別の方法をサポートしています。

範囲は TTable とクライアントデータセットにだけ適用できます。フィルタと範囲は似ている点もありますが，使用方法が異なります。次のトピックでは，フィルタと範囲の違いを示した後，範囲の使い方を説明します。

範囲とフィルタの違いについて

範囲もフィルタも，一部のレコードだけ表示する点では同じですが，その方法が異なります。範囲では，指定した 2 つの境界値の間にあるすべてのインデックス付きレコードが取り出されます。たとえば，名字にインデックスが付いた社員データベースで，名字が「Jones」より大きく「Smith」より小さいすべての社員を表示するのに範囲を適用できます。範囲はインデックスに依存するため，範囲の定義に使用できるインデックスを現在のインデックスに設定しなければなりません。レコードのソートのためにインデックスを指定する場合と同様に，IndexName または IndexFieldNames プロパティを使用して範囲を設定するインデックスを割り当てることができます。

フィルタでは，インデックスが付いているかどうかに関係なく，指定した条件を満たす連続または非連続のレコードが取り出されます。たとえば，社員データベースにフィルタを使用すると，カリフォ

ルニア在住で勤続 5 年以上のすべての社員を表示できます。フィルタはインデックスが付いていればそれを使いますが，フィルタはインデックスには依存しません。アプリケーションがデータセット内をスクロールする場合，フィルタは各レコードに適用されます。

一般に，範囲よりフィルタの方が柔軟な使い方ができます。しかしデータセットが大きくて，アプリケーションが必要とするレコードが連続的にインデックス付けされたグループにブロック化されている場合は，範囲の方が効率的です。非常に大きなデータセットでは，表示や編集のために，問い合わせタイプのデータセットの WHERE 句を使ってデータを選択するとさらに効果的です。問い合わせを指定する方法の詳細については，22-40 ページの「問い合わせタイプのデータセットの使い方」を参照してください。

範囲の指定

範囲を指定するには，2 つの相互に排他的な方法があります。

- SetRangeStart および SetRangeEnd を使用して，開始値と終了値を別々に指定する
- SetRange を使用して，開始値と終了値を一度に指定する

範囲の開始値の設定

SetRangeStart 手続きを呼び出してデータセットを dsSetKey 状態にし，範囲の開始値のリストを作成します。SetRangeStart を呼び出すと，それ以降 Fields プロパティに代入される値は，範囲の適用時に使われるインデックスの開始値とみなされます。指定された項目は，現在のインデックスに適合している必要があります。

たとえば，CUSTOMER テーブルにリンクした，「Customers」という名前の TSQLClientDataSet コンポーネントを使用する場合を考えます。さらに，Customers データセットの各項目に静的な項目コンポーネントを作成したとします。CUSTOMER は，その最初の列（CustNo）に基づいてインデックス付けされています。アプリケーションのフォームには StartVal と EndVal という 2 つの編集コンポーネントがあり，範囲の開始値と終了値を指定するのに使われます。次のコードが，範囲の作成と適用に利用できます。

```
Customers->SetRangeStart();
Customers->FieldValues["CustNo"] = StrToInt(StartVal->Text);
Customers->SetRangeEnd();
if (!EndVal->Text.IsEmpty())
  Customers->FieldValues["CustNo"] = StrToInt(EndVal->Text);
Customers->ApplyRange();
```

このコードは Fields に値を代入する前に，EndVal に入力された文字がヌルでないか調べます。StartVal に入力された文字がヌルの場合は，すべての値はヌルよりも大きいため，データセット先頭からの全レコードが範囲に含まれます。逆に EndVal に入力された文字がヌルの場合は，ヌルより小さいものはないため，レコードはまったく範囲に含まれません。

複数列インデックスでは，インデックスのすべてまたは一部の項目に開始値を指定できます。インデックスで使われている項目に値を指定しないときは，範囲の適用時にヌル値が設定されます。インデックスにない項目に値を設定しようとすると，データセットは例外を生成します。

ヒント データセットの先頭で範囲を開始するには，SetRangeStart の呼び出しを省略します。

範囲の開始値の設定を完了するには SetRangeEnd を呼び出すか，あるいは範囲を適用するか取り消します。範囲の適用と取り消しについて詳しくは，22-33 ページの「範囲の適用または取り消し」を参照してください。

範囲の終了値の設定

SetRangeEnd 手続きを呼び出してデータセットを dsSetKey 状態にし，範囲の終了値のリストを作成します。SetRangeEnd を呼び出すと，それ以降 Fields プロパティに代入された値は，範囲の適用時に使われるインデックスの終了値であるとみなされます。指定された項目は，現在のインデックスに適合している必要があります。

注意 データセットの最終レコードまでを範囲にする場合でも，範囲の終了値は必ず指定してください。範囲の終了値を指定しないと，C++Builder では終了値はヌル値であるとみなされます。終了値がヌルの範囲は，常に空です。

FieldByName メソッドを使用すると，終了値を簡単に代入できます。例を示します。

```
Contacts->SetRangeStart();
Contacts->FieldByName("LastName")->Value = Edit1->Text;
Contacts->SetRangeEnd();
Contacts->FieldByName("LastName")->Value = Edit2->Text;
Contacts->ApplyRange();
```

範囲の開始値を指定する場合と同様に，インデックスにない項目に値を設定しようとすると，データセットは例外を生成します。

範囲の終了値の設定を完了するには，範囲を適用するか取り消します。範囲の適用と取り消しについて詳しくは，22-33 ページの「範囲の適用または取り消し」を参照してください。

範囲の開始値と終了値の設定

SetRangeStart と SetRangeEnd を別々に呼び出して範囲境界を設定するかわりに，SetRange 手続きを呼び出せば，データセットを dsSetKey 状態にして，範囲の開始値と終了値を一度に設定できます。

SetRange は開始値群と終了値群の 2 つの定数配列パラメータを取ります。たとえば，次の文は 2 列インデックスに基づいて範囲を設定しています。

```
TVarRec StartVals[2];
TVarRec EndVals[2];
StartVals[0] = Edit1->Text;
StartVals[1] = Edit2->Text;
EndVals[0] = Edit3->Text;
EndVals[1] = Edit4->Text;
Table1->SetRange(StartVals, 1, EndVals, 1);
```

複数列インデックスでは，インデックスのすべてまたは一部の項目に開始値と終了値を指定できます。インデックスで使われている項目に値を指定しないときは，範囲の適用時にヌル値が設定されます。インデックスの最初の項目の値を省略し，それ以外の項目に値を指定するには，省略する項目にヌル値を渡します。

データセットの最終レコードまでを範囲にする場合でも，範囲の終了値は必ず指定してください。範囲の終了値を指定しないと，データセットでは終了値はヌル値とみなされます。終了値がヌルの範囲は，開始範囲値が終了範囲値以上になるため常に空です。

部分キーに基づく範囲の指定

キーが 1 つまたは複数の文字列項目で構成されている場合，SetRange メソッドは部分キーをサポートします。たとえば LastName 列と FirstName 列に基づくインデックスがある場合，次の範囲指定は有効です。

```
Contacts->SetRangeStart();
Contacts->FieldValues["LastName"] = "Smith";
Contacts->SetRangeEnd();
Contacts->FieldValues["LastName"] = "Zzzzzz";
Contacts->ApplyRange();
```

このコードは，LastName が「Smith」以上であるすべてのレコードを範囲に含みます。値は，次のように指定することもできます。

```
Contacts->FieldValues["LastName"] = "Sm";
```

この文は，LastName が「Sm」以上であるレコードを含みます。

境界値と等しいレコードを包含するか除外するかを指定する

デフォルトでは，範囲には指定の開始値以上で指定の終了値以下のすべてのレコードが含まれます。KeyExclusive プロパティを使用すれば，開始値や終了値に等しいレコードが含まれるかどうかを指定できます。デフォルトでは，KeyExclusive は **false** です。

クライアントデータセットの KeyExclusive プロパティを **true** に設定すれば，範囲境界値に等しいレコードを除外できます。例を示します。

```
Contacts->SetRangeStart();
Contacts->KeyExclusive = true;
Contacts->FieldValues["LastName"] = "Smith";
Contacts->SetRangeEnd();
Contacts->FieldValues["LastName"] = "Tyler";
Contacts->ApplyRange();
```

このコードは，LastName が「Smith」より大きいか等しい，または「Tyler」より小さいすべてのレコードを範囲に含みます。

範囲の変更

範囲の既存の境界条件を変更する機能が 2 つあります。範囲の開始値を変更する EditRangeStart と，範囲の終了値を変更する EditRangeEnd です。

範囲の編集と適用を行う手順は次のとおりです。

1. データセットを dsSetKey 状態にし，範囲の開始インデックス値を変更する
2. 範囲の終了インデックス値を変更する
3. 範囲をデータセットに適用する

範囲の開始値か終了値のどちらかを変更することも，両方の境界条件を変更することもできます。データセットに現在適用されている範囲の境界条件を変更する場合，新たな変更が適用されるのは ApplyRange をもう一度呼び出したときです。

範囲の開始値の変更

EditRangeStart 手続きを呼び出してデータセットを dsSetKey 状態にし，範囲の開始値の現在のリストを変更します。EditRangeStart を呼び出すと，それ以降に Fields プロパティに代入された値は，範囲の適用時に使われる現在のインデックス値を上書きします。

ヒント 最初に部分キーに基づいて開始範囲を作成すると，EditRangeStart を使用して範囲の開始値を拡張できます。部分キーに基づく範囲についての詳細は，22-32 ページの「部分キーに基づく範囲の指定」を参照してください。

範囲の終了値の変更

EditRangeEnd 手続きを呼び出してデータセットを dsSetKey 状態にし，範囲の終了値のリストを作成します。EditRangeEnd を呼び出すと，それ以降 Fields プロパティに代入された値は，範囲の適用時に使われるインデックスの終了値であるとみなされます。

範囲の適用または取り消し

SetRangeStart か EditRangeStart を呼び出して範囲を指定したり，SetRangeEnd か EditRangeEnd を呼び出して範囲の終了値を指定する場合，データセットは dsSetKey 状態になります。データセットは，範囲を適用するか取り消すまでその状態です。

範囲の適用

範囲を指定する場合，設定した境界条件は範囲を適用するまで有効になりません。範囲を有効にするには ApplyRange 手続きを呼び出します。ApplyRange では，指定されたデータセット部分のデータに対し，ユーザーによる表示とアクセスはただちに制限されます。

範囲の取り消し

CancelRange メソッドは，範囲の適用を終了し，データセット全体にアクセスできるよう復元します。範囲を取り消してデータセットの全レコードへのアクセスが復元されても，後でその範囲をもう一度適用できるように，範囲の境界条件はそのまま使用できます。新しい範囲境界を指定するか既存の境界を変更するまで，範囲境界は保存されます。たとえば，次のコードは有効です。

```
...
MyTable->CancelRange();
...
// 後で同じ範囲をもう一度使う。SetRangeStart などを呼び出す必要はない
MyTable->ApplyRange();
...
```

マスター / 詳細関係の作成

テーブルタイプのデータセットは，マスター / 詳細関係にリンクできます。マスター / 詳細関係を設定すると，一方のデータセット（詳細）の全レコードが常にもう一方のデータセット（マスター）の現在の 1 レコードに対応するように 2 つのデータセットがリンクされます。

テーブルタイプのデータセットは，2 つの別の方法でマスター / 詳細関係をサポートしています。

- すべてのテーブルタイプのデータセットは，カーソルのリンクによって，別のデータセットの詳細としての役割を果たします。この手順は，以下の「テーブルを別のデータセットの詳細にする」で説明します。

- TTable，TSQLTable，およびすべてのクライアントデータセットは，ネストされた詳細テーブルを使用するマスター / 詳細関係中で，マスターとしての役割を果たすことができます。この手順については，22-35 ページの「ネストされた詳細テーブルの使用」で述べます。

各アプローチにはそれぞれ，独自のメリットがあります。カーソルのリンクでは，マスターテーブルがどの種類のデータセットでも，マスター / 詳細関係を作成できます。ネストした詳細では，詳細テーブルの役割を果たすデータセットの種類は限られますが，データを表示する方法については，より多くのオプションがあります。マスターがクライアントデータセットの場合，ネストした詳細は，キャッシュされた更新を適用する上で，より堅牢なメカニズムを提供します。

テーブルを別のデータセットの詳細にする

テーブルタイプのデータセットの MasterSource プロパティと MasterFields プロパティを使用すると，2 つのデータセット間に 1 対多の関係を確立できます。

MasterSource プロパティは，マスターテーブル用のデータの取得元であるデータソースを指定するために使います。このデータソースは，任意の種類のデータセットにリンクすることができます。たとえば，このプロパティで問い合わせのデータソースを指定すると，クライアントデータセットを問い合わせの詳細としてリンクすることができます。その結果，クライアントデータセットは，問い合わせで生じたイベントを追跡できます。

データセットは，現在のインデックスに基づいてマスターテーブルにリンクされます。詳細データセットから追跡するマスターデータセット内の項目を指定する前に，まず詳細データセット内のインデックスを指定します。指定では IndexName プロパティか IndexFieldNames プロパティを使用できます。

使用するインデックスを指定したら MasterFields プロパティを使用して，詳細テーブル内のインデックス項目に対応するマスターデータセット内の列を指定します。複数の列名に基づいてデータセットをリンクするには，項目名をセミコロンで区切ります。

```
Parts->MasterFields = "OrderNo;ItemNo";
```

2 つのデータセット間のリンクを作成するには，リンク項目デザイナが使用できます。リンク項目デザイナを使用するには，MasterSource とインデックスを割り当ててから，オブジェクトインスペクタ内の MasterFields プロパティをダブルクリックします。

ユーザーが顧客レコードをスクロールして現在の顧客の注文すべてを表示できる単純なフォームを作成する手順は次のとおりです。マスターテーブルは CustomersTable テーブルで，詳細テーブルは OrdersTable テーブルです。この例では，BDE ベースの TTable コンポーネントを使用します。しかし，同じ方法を使用して，どのテーブルタイプのデータセットでもリンクすることができます。

1. 2 つの TTable コンポーネントと 2 つの TDataSource コンポーネントをデータモジュールに入れます。
2. 最初の TTable コンポーネントのプロパティを以下のように設定します。
   - DatabaseName：BCDEMOS
   - TableName：CUSTOMER
   - Name：CustomersTable
3. 2 つ目の TTable コンポーネントのプロパティを以下のように設定します。
   - DatabaseName：BCDEMOS

- TableName：ORDERS
- Name：OrdersTable

4. 最初の TDataSource コンポーネントのプロパティを以下のように設定します。
   - Name：CustSource
   - DataSet：CustomersTable
5. 2 番目の TDataSource コンポーネントのプロパティを以下のように設定します。
   - Name：OrdersSource
   - DataSet：OrdersTable
6. 新しくフォームを作り 2 つの TDBGrid コンポーネントを入れます。
7. ［ファイル｜ユニットヘッダーファイルの追加］を選択して，フォームがデータモジュールを使用するように指定します。
8. 最初のグリッドコンポーネントの DataSource プロパティを「DataModule2.CustSource」に設定し，2 番目のグリッドコンポーネントの DataSource プロパティを「DataModule2.OrdersSource」に設定します。
9. OrdersTable の MasterSource プロパティを「CustSource」に設定します。これによって，CUSTOMER テーブル（マスターテーブル）が ORDERS テーブル（詳細テーブル）にリンクされます。
10. オブジェクトインスペクタで MasterFields プロパティの値ボックスをダブルクリックしてリンク項目デザイナを呼び出し，以下のようにプロパティを設定します。
    - ［選択可能なインデックス］フィールドで［CustNo］を選択して，CustNo 項目で 2 つのテーブルをリンクする
    - ［詳細項目］と［マスター項目］の両方の項目リストで，CustNo を選択します。
    - ［追加］ボタンをクリックしてこの結合条件を追加します。［結合した項目］リストに，「CustNo -> CustNo」と表示される
    - 設定が終わったら［OK］を選択して，リンク項目デザイナを終了します。
11. CustomersTable と OrdersTable の Active プロパティを **true** に設定して，フォームのグリッドにデータを表示します。
12. アプリケーションをコンパイルして実行します。

アプリケーションを実行すると，2 つのテーブルがリンクしていることがわかるでしょう。CUSTOMER テーブルを表示しているグリッドは，CUSTOMER テーブルの現在位置のデータに関連付けられたレコードだけを表示します。

ネストされた詳細テーブルの使用

ネストされたテーブルは，別の（マスター）データセット中の単一のデータセット項目の値である，詳細データセットです。サーバーデータを表わすデータセットについては，ネストされた詳細データセットはサーバー上のデータセット項目のためにのみ使用することができます。TClientDataSet コンポーネントはサーバーデータを表わしませんが，ネストされた詳細を含んでいるデータセットを作成するか，マスター / 詳細関係のマスターテーブルにリンクされたプロバイダからデータを受け取れば，データセット項目を含むことができます。

メモ TClientDataSet の場合，マスターテーブルと詳細テーブルの更新内容をデータベースサーバーに適用するには，ネストされた詳細セットが必要です。

ネストされた詳細セットを使うには，マスターデータセットの ObjectView プロパティが **true** でなければなりません。テーブルタイプのデータセットにネストされた詳細データセットが含まれている場合，TDBGrid がネストされた詳細セットのポップアップウィンドウ表示をサポートします。この機能については，23-24 ページの「データセット項目の表示」を参照してください。

または，詳細セット用に別のデータセットコンポーネントを使用することにより，これらのデータセットをデータベース対応コントロールで表示したり，編集できます。設計時に，項目エディタを使用して，（マスター）データセット内の項目用に，持続的項目を作成します。マスターデータセットを右クリックして，［項目エディタ］を選択します。表示された項目エディタを右クリックして［項目の追加］を選択することにより，新規の持続的項目をデータセットに追加します。新規項目の型を DataSet 項目と定義します。項目エディタで，詳細テーブルの構造を定義します。さらに，マスターデータセットの中で使用される他の項目のために，持続的項目を追加しなければなりません。

詳細テーブル用のデータセットコンポーネントは，マスターテーブルによって許可された種類の，下位データセットオブジェクトです。TTable コンポーネントは，TnestedDataSet コンポーネントを，ネストされたデータセットとしてのみ許可します。TSQLTable コンポーネントは，ほかの TSQLTable コンポーネントを許可します。TClientDataset コンポーネントは，ほかのクライアントデータセットを許可します。コンポーネントパレットから適切な種類のデータセットを選び，フォームかデータモジュールに追加します。この詳細クライアントデータセットの DataSetField プロパティを，マスターデータセット内の持続的 DataSet 項目に設定します。最後に，データソースコンポーネントをフォームまたはデータモジュールに入れ，その DataSet プロパティを詳細データセットに設定します。データベース対応のコントロールは，詳細セット中のデータにアクセスするためにこのデータソースを使用することができます。

テーブルに対する読み書きの制御

デフォルトでは，テーブルタイプのデータセットを開く場合，基礎になるデータベーステーブルの読み書き特権が要求されます。そのデータベーステーブルの特性に基づき，要求された書き込み特権が付与されないこともあります（リモートサーバー上の SQL テーブルに対して書き込みアクセスを要求したときに，テーブルのアクセスをサーバーが読み出しに制限している場合など）。

メモ このことは，TClientDataSet には当てはまりません。これ自体が，データセットプロバイダがデータパケットとともに供給する情報からのデータをユーザーが編集することができるかどうかを決めるからです。また，TSQLTable にも当てはまりません。これは単方向データセットであり，そのため常に読み出し専用だからです。

テーブルを開いたなら，CanModify プロパティをチェックして，ユーザーがテーブル中のデータを編集するのを基となるデータベース（またはデータセットプロバイダ）が許可しているかどうかを確認できます。CanModify が **false** の場合，そのアプリケーションはデータベースに書き込みができません。CanModify が **true** の場合，アプリケーションはテーブルの ReadOnly プロパティが **false** に設定されているデータベースに書き込めます。

ReadOnly はデータの表示と編集をユーザーに許可するかどうかを指定します。ReadOnly が **false**（デフォルト）の場合，ユーザーはデータの表示と編集が両方ともできます。ユーザーがデータの表示だけしかできないようにするには，テーブルを開く前に ReadOnly を **true** に設定します。

メモ ReadOnly は，TSQLTable（常に読み出し専用）以外の，すべてのテーブルタイプのデータセットに実装されています。

テーブルの作成と削除

テーブルタイプのデータセットの中には，設計時でも実行時でも，基になるテーブルの作成と削除が行えるものがあります。通常は，データベースの管理者がデータベーステーブルの作成と削除を行います。しかし，アプリケーションが使用するデータベーステーブルの作成と破棄が行えれば，アプリケーションの開発やテストを行う上で便利です。

テーブルの作成

TTable と TIBTable では，SQL を使わずに，基になるデータベーステーブルを作成できます。同様に，TClientDataSet では，データセットプロバイダを操作していないときでも，データセットを作成できます。TTable と TClientDataSet を使う場合には，設計時でも実行時でも，テーブルを作成できます。TIBTable の場合，テーブルを作成できるのは実行時のみです。

テーブルを作成する前に，テーブルの構造を指定するプロパティを設定しなければなりません。特に，以下のものを指定しなければなりません。

- 新しいテーブルをホストするデータベース。TTable の場合，DatabaseName プロパティを使用してデータベースを指定します。TIBTable の場合，TIBDatabase コンポーネントを使用しなければなりません。これは，Database プロパティに割り当てられています（クライアントデータセットはデータベースを使用しません）。
- データベースの種類（TTable のみ）。TableType プロパティを希望するテーブルの種類に設定します。Paradox，dBASE，または ASCII テーブルの場合は，それぞれ TableType を ttParadox，ttDBase，または ttASCII に設定します。それ以外の種類のテーブルについては，TableType を ttDefault に設定します。
- 作成するテーブルの名前。TTable と TIBTable には，新しいテーブルの名前を指定するための，TableName プロパティがあります。クライアントデータセットはテーブル名を使用しませんが，新しいテーブルを保存する前に，FileName プロパティを指定してください。既存のテーブルの名前と重複するテーブルを作成した場合，既存のテーブルとそのすべてのデータは新しく作成するテーブルで上書きされます。上書きされたテーブルとそのデータは復元できません。既存のテーブルの上書きを避けるには，実行時に Exists プロパティをチェックします。Exists は，TTable と TIBTable でのみ使用可能です。
- 新しいテーブルの項目。これには 2 つの方法があります。
  - FieldDefs プロパティに項目定義を追加する。設計時には，オブジェクトインスペクタで FieldDefs プロパティをダブルクリックすれば，コレクションエディタを起動できます。項目定義のプロパティを追加，削除，または変更するには，コレクションエディタを使います。実行時には，既存の項目定義をクリアし，AddFieldDef メソッドを使ってそれぞれの新しい項目

定義を追加します。新しい各項目定義について，TFieldDef オブジェクトのプロパティを設定し，項目の属性を指定します。

- 持続的項目コンポーネントを使う。設計時には，データセットをダブルクリックすれば，項目エディタを起動できます。項目エディタで，右クリックして［項目の新規作成］コマンドを選択します。項目の基本プロパティを記述します。項目が作成された後は，オブジェクトインスペクタから項目エディタで項目を選択することにより，その項目のプロパティを変更できます。

- 新しいテーブルのインデックス（省略可能）設計時には，オブジェクトインスペクタで IndexDefs プロパティをダブルクリックすれば，コレクションエディタを起動できます。インデックス定義のプロパティを追加，削除，または変更するには，コレクションエディタを使います。実行時には，既存のインデックス定義をクリアし，AddIndexDef メソッドを使ってそれぞれの新しいインデックス定義を追加します。新しい各インデックス定義について，TIndexDef オブジェクトのプロパティを設定し，インデックスの属性を指定します。

メモ 項目定義オブジェクトのかわりに持続的項目コンポーネントを使用している場合には，新しいテーブルのためのインデックスを定義することはできません。

設計時にテーブルを作成するには，データセットを右クリックして，［テーブルの作成］（TTable）または［データセットの作成］（TClientDataSet）を選択します。このコマンドは，必要な情報をすべて指定するまではコンテキストメニューに表示されません。

実行時にテーブルを作成するには，CreateTable メソッド（TTable および TIBTable）または CreateDataSet メソッド（TClientDataSet）を呼び出します。

メモ 設計時に定義をセットアップしておき，実行時に CreateTable（または CreateDataSet）メソッドを呼び出して , テーブルを作成することができます。しかし，そうするためには，実行時に指定された定義を，データセットコンポーネントとともに保存する必要があることを示さなければなりません（デフォルトでは，項目とインデックスの定義は，実行時に動的に生成されます）。定義をデータセットとともに保存するよう指定するには，StoreDefs プロパティを **true** に設定します。

ヒント TTable を使用する場合には，設計時に，既存テーブルの項目定義とインデックス定義を読み込むことができます。DatabaseName プロパティと TableName プロパティは，既存のテーブルを指定するように設定します。テーブルコンポーネントを右クリックして［テーブル定義の更新］を選択します。これによって，FieldDefs プロパティと IndexDefs プロパティの値が既存テーブルの項目とインデックスを記述するように設定されます。次に，DatabaseName と TableName をリセットして作成したいテーブルを指定し，既存テーブルの名前を変更するプロンプト表示はキャンセルします。

メモ Oracle8 テーブルを作成するときは，オブジェクト項目（ADT 項目，配列項目，およびデータセット項目）は作成できません。

次のコードは実行時に新しいテーブルを作成し，それを BCDEMOS エリアスに関連付けます。新しいテーブルを作成する前に，指定されたテーブル名が既存のテーブル名と重複していないことを確認します。

```
TTable *NewTable = new TTable(Form1);
NewTable->Active = false;
NewTable->DatabaseName = "BCDEMOS";
NewTable->TableName = Edit1->Text;
NewTable->TableType = ttDefault;
NewTable->FieldDefs->Clear();
TFieldDef *NewField = NewTable->FieldDefs->AddFieldDef(); // 最初の項目を定義する
```

```
NewField->DataType = ftInteger;
NewField->Name = Edit2->Text;
NewField = NewTable->FieldDefs->AddFieldDef(); // 2 番目の項目を定義する
NewField->DataType = ftString;
NewField->Size = StrToInt(Edit3->Text);
NewField->Name = Edit4->Text;
NewTable->IndexDefs->Clear();
TIndexDef *NewIndex = NewTable->IndexDefs->AddIndexDef(); // インデックスを追加する
NewIndex->Name = "PrimaryIndex";
NewIndex->Fields = Edit2->Text;
NewIndex->Options << ixPrimary << ixUnique;
// このテーブルがすでに存在しているかどうか調べる
bool CreateIt = (!NewTable->Exists);
if (!CreateIt)
   if (Application->MessageBox((AnsiString("Overwrite table ") + Edit1->Text +
                                AnsiString("?")).c_str(),
                                "Table Exists", MB_YESNO) == IDYES)
    CreateIt = true;
if (CreateIt)
  NewTable->CreateTable(); // テーブルを作成する
```

テーブルの削除

TTable と TIBTable では，SQL を使わずに，テーブルを基になるデータベーステーブルから削除できます。実行時にテーブルを削除するには，データセットの DeleteTable メソッドを呼び出します。たとえば次の文は，データセットの基礎になるテーブルを削除します。

```
CustomersTable->DeleteTable();
```

注意 DeleteTable でテーブルを削除した場合，テーブルとその中のすべてのデータは元に戻せません。

TTable を使用している場合，設計時にテーブルを削除するには，テーブルコンポーネントを右クリックして，コンテキストメニューから［テーブルの削除］を選択することもできます。［テーブルの削除］が表示されるのは，テーブルコンポーネントが既存のデータベーステーブルを表している場合だけです（DatabaseName プロパティと TableName プロパティで既存のテーブルを指定します）。

テーブルを空にする

多くのテーブルタイプデータセットでは，テーブル中のデータの列をすべて削除するための単一のメソッドを提供しています。

- TTable と TIBTable の場合，実行時に EmptyTable メソッドを呼び出せば，すべてのレコードを削除できる

  ```
  PhoneTable->EmptyTable();
  ```

- TADOTable の場合，DeleteRecords メソッドを使用できる

  ```
  PhoneTable->DeleteRecords(arAll);
  ```

- TSQLTable の場合も，DeleteRecords メソッドを使用できる。ただし，TSQLTable の DeleteRecords はパラメータを取らないことに注意してください。

  ```
  PhoneTable->DeleteRecords();
  ```

- クライアントデータセットの場合，EmptyDataSet メソッドを使用できる

  ```
  PhoneTable->EmptyDataSet();
  ```

メモ SQL サーバーの場合，これらのメソッドが成功するのは，そのテーブルに対して DELETE 特権を持っている場合だけです。

注意 データセットを空にすると，削除したデータを回復することはできません。

テーブルの同期

複数のデータセットが同一のデータベーステーブルを表しているものの，データソースコンポーネントを共有していない場合，各データセットはデータに独自のビューを持ち，独自の現在のレコードを持ちます。ユーザーが各データセットを使ってレコードにアクセスした場合，そのコンポーネントの現在のレコードはそれぞれ異なります。

データセットがすべて TTable のインスタンス，すべて TIBTable のインスタンス，またはすべてクライアントデータセットである場合，GotoCurrent メソッドを呼び出すことにより，これらのデータセットのそれぞれの現在のレコードを強制的に同じにすることができます。GotoCurrent は，それ自身のデータセットの現在のレコードを，一致するデータセットの現在のレコードと同じにします。たとえば次のコードは，CustomerTableOne の現在のレコードを CustomerTableTwo の現在のレコードと同じにします。

```
CustomerTableOne->GotoCurrent(CustomerTableTwo);
```

ヒント アプリケーションでこのようなデータセットの同期が必要な場合は，それらのデータセットをデータモジュールに入れ，そのデータモジュールのヘッダーを，テーブルにアクセスする各ユニットの中に含めます。

別々のフォームにあるデータセットを同期させる必要がある場合は，一方のフォームのヘッダーファイルをもう一方のフォームのソースユニットに含め，少なくとも一方のデータセット名をそのフォーム名と関連付けなければなりません。例を示します。

```
CustomerTableOne->GotoCurrent(Form2->CustomerTableTwo);
```

問い合わせタイプのデータセットの使い方

問い合わせタイプのデータセットを使う手順は次のとおりです。

1. 適切なデータセットコンポーネントをデータモジュール上かフォーム上に配置して，Name プロパティにアプリケーションに適したユニークな値を設定します。
2. 問い合わせを行うデータベースサーバーを設定します。設定の方法は，問い合わせタイプのデータセットごとに異なっています。しかし通常は，データベース接続コンポーネントを指定します。
   - TQuery の場合，DatabaseName プロパティを使用し，TDatabase コンポーネントまたは BDE エリアスを指定する
   - TADOQuery の場合，Connection プロパティを使用し，TADOConnection コンポーネントを指定する
   - TSQLQuery の場合，SQLConnection プロパティを使用し，TSQLConnection コンポーネントを指定する
   - TIBQuery の場合，Database プロパティを使用し，TIBConnection コンポーネントを指定する

データベース接続コンポーネントの使い方については，第 21 章「データベースへの接続」を参照してください。

3. データセットの SQL プロパティに SQL 文を指定します。必要であれば SQL 文のパラメータを指定します。詳細については，22-41 ページの「問い合わせの指定」と 22-43 ページの「問い合わせでパラメータを使用する」を参照してください。
4. 問い合わせデータがビジュアルデータコントロールで使用される場合は，データソースコンポーネントをフォームやデータモジュールに追加して，DataSet プロパティに問い合わせタイプのデータセットを設定します。データソースコンポーネントは，問い合わせの結果（結果セット）を，データベース対応コンポーネントに転送し，表示できるようにします。データベース対応コンポーネントの DataSource プロパティと DataField プロパティを使用してデータソースに接続します。
5. 問い合わせコンポーネントをアクティブにします。結果セットを返す問い合わせの場合は，Active プロパティか Open メソッドを使います。テーブルに対して操作を実行するだけで結果を返さない問い合わせを実行する場合は，実行時に ExecSQL メソッドを使います。複数回問い合わせを実行する予定であれば，Prepare を呼び出して，データアクセス層を初期化し，パラメータ値を問い合わせに結合するとよいでしょう。ストアドプロシージャの準備については，22-46 ページの「問い合わせの準備」を参照してください。

問い合わせの指定

真の問い合わせタイプのデータセットの場合，データセットが実行する SQL 文を指定するために SQL プロパティを使用します。TADODataSet，TSQLDataSet，およびクライアントデータセットといったデータセットの場合，同じことを行うには CommandText プロパティを使用します。

レコードを返す問い合わせのほとんどは SELECT コマンドです。通常この種の問い合わせでは，選択する項目，その項目のソースとなるテーブル，選択するレコードを制限する条件，結果のデータセットの順序を設定します。例を示します。

```
SELECT CustNo, OrderNo, SaleDate
FROM Orders
WHERE CustNo = 1225
ORDER BY SaleDate
```

レコードを返さない問い合わせとしては，SELECT 文以外のデータ定義言語（DDL）文またはデータ操作言語（DML）文を使用する文があります（たとえば，INSERT，DELETE，UPDATE，CREATE INDEX，ALTER TABLE コマンドはレコードを返しません）。コマンドに使用される言語はサーバー固有ですが，通常は SQL 言語の SQL-92 標準に準拠しています。

実行する SQL コマンドは，使用しているサーバーに受け入れられる必要があります。データセットは SQL を評価せず，実行もしません。単方向データセットは，単にコマンドをサーバーに渡すだけです。ほとんどの場合，SQL コマンドは 1 つの完全な SQL 文でなければなりません。ただし，SQL 文は必要に応じて複雑にすることができます（たとえば，WHERE 節付きの SELECT 文では，AND や OR などの論理演算子をネストして複数使うことができます）。一部のサーバーでは複数の文を許可する「バッチ」構文をサポートしています。つまり，このような構文を提供しているサーバーでは，問い合わせを指定する際に複数の文を入力できます。

SQL 文は，パラメータなしで使うこともパラメータ付きで使うこともできます。パラメータを使う問い合わせはパラメータ付き問い合わせとも呼ばれます。パラメータ付き問い合わせを使う場合，パラメータに現在代入されている実際の値を問い合わせに挿入してから，問い合わせを実行します。パラメータ付き問い合わせは，SQL 文を変えることなく実行時にユーザーのデータ表示やデータアクセスを変更できるのでたいへん柔軟性に富んでいます。パラメータ付き問い合わせについての詳細は，22-43 ページの「問い合わせでパラメータを使用する」を参照してください。

TSQLDataSet を使用した問い合わせの指定

真の問い合わせタイプのデータセット（TQuery，TADOQuery，TSQLQuery，または TIBQuery）を使用する場合は，問い合わせを SQL プロパティに割り当てます。SQL プロパティは TStrings オブジェクトです。TStrings オブジェクト内の各文字列は，問い合わせではそれぞれ別の行になります。複数行を使用しても，サーバー上での問い合わせの実行には影響はありませんが，文を論理的な単位に分割すれば，問い合わせの修正やデバッグが容易になります。

```
MyQuery->Close();
MyQuery->SQL->Clear();
MyQuery->SQL->Add("SELECT CustNo, OrderNO, SaleDate");
MyQuery->SQL->Add("FROM Orders");
MyQuery->SQL->Add("ORDER BY SaleDate");
MyQuery->Open();
```

次のコードは，既存の SQL 文の中の 1 行だけを修正する場合を示しています。この場合，SQL 文の 3 行目にすでに ORDER BY 節が存在しています。この ORDER BY 節は，SQL プロパティを介してインデックス 2 を使って参照されています。

```
MyQuery->SQL->Strings[2] = "ORDER BY OrderNO";
```

メモ SQL プロパティの指定や修正を行う場合，データセットは閉じていなければなりません。

設計時には，文字列リストエディタを使用して問い合わせを指定します。オブジェクトインスペクタ内の SQL プロパティで省略記号ボタンをクリックすると，文字列リストエディタが表示されます。

メモ C++Builder のバージョンによっては，TQuery を使っている場合，SQL ビルダでもデータベースのテーブルや項目の表示形式に基づいて問い合わせを作成できます。SQL ビルダを使うには，問い合わせコンポーネントを選択し，それを右クリックしてコンテキストメニューを表示し，［SQL ビルダ］を選択します。SQL ビルダの使い方についての詳細は，SQL ビルダのオンラインヘルプを参照してください。

SQL プロパティは TStrings オブジェクトなので，TStrings::LoadFromFile メソッドを呼び出して，ファイルの問い合わせのテキストをロードすることができます。

```
MyQuery->SQL->LoadFromFile("custquery.sql");
```

SQL プロパティの Assign メソッドを使って，文字列リストオブジェクトの内容を SQL プロパティにコピーすることもできます。Assign メソッドは，新しい文をコピーする前に，SQL プロパティの現在の内容を自動的にクリアします。

```
MyQuery->SQL->Assign(Memo1->Lines);
```

CommandText プロパティを使用した問い合わせの指定

TADODataSet，TSQLDataSet，またはクライアントデータセットを使用する場合，問い合わせ文のテキストを CommandText プロパティに入力します。

```
MyQuery->CommandText = "SELECT CustName, Address FROM Customer";
```

設計時には，問い合わせを直接オブジェクトインスペクタに入力したり，SQL データセットとデータベースとの接続がアクティブな場合は，CommandText プロパティで省略記号ボタンをクリックして，コマンドテキストエディタを表示できます。コマンドテキストエディタは，問い合わせの作成に便利なように，利用可能なテーブルとテーブル内の項目を表示します。

問い合わせでパラメータを使用する

パラメータ付きの SQL 文にはパラメータつまり変数が含まれており，その値は設計時または実行時に変えることができます。パラメータは，SQL 文の中に現れるデータ値（比較用の WHERE 節で使われるものなど）の代わりになることができます。通常，パラメータは文に渡すデータ値の代わりとして使います。たとえば，次の INSERT 文では，挿入する値がパラメータで渡されています。

```
INSERT INTO Country (Name, Capital, Population)
VALUES (:Name, :Capital, :Population)
```

この SQL 文では，:Name，:Capital，:Population はアプリケーションで実行時に文へ渡す実際の値の代わりのプレースホルダです。パラメータの名前が，コロンで始まることに注目してください。コロンはリテラル値とパラメータ名を区別するために必要です。また，問い合わせ内に疑問符（?）を追加することにより，名前なしパラメータを入力できます。名前なしパラメータはユニークな名前を持たないため，位置で識別されます。

問い合わせテキストのパラメータに値を指定しないと，データセットは問い合わせを実行できません。TQuery，TIBQuery，TSQLQuery，およびクライアントデータセットは，これらの値を格納するために，Params プロパティを使用します。TADOQuery はかわりに Parameters プロパティを使用します。Params（または Parameters）は，パラメータオブジェクト（TParam または TParameter）のコレクションであり，各オブジェクトが 1 つのパラメータを表します。問い合わせのテキストを指定すると，データセットはパラメータオブジェクトのセットを生成し，（データセットのタイプに応じて）問い合わせから導き出せるプロパティを初期化します。

メモ ParamCheck プロパティを **false** に設定すると，問い合わせテキストを変更しても自動的にパラメータオブジェクトを生成しないようにできます。これは，問い合わせ自身のパラメータでない DDL 文の一部としてパラメータを含んでいるデータ定義言語（DDL）文で役立ちます。たとえば，ストアドプロシージャを作成する DDL 文が，ストアドプロシージャの一部であるパラメータを定義する場合があります。ParamCheck を **false** に設定すれば，これらのパラメータが問い合わせのパラメータと混同されるのを防ぐことができます。

パラメータ値は初めて SQL 文を実行する前に SQL 文に結合する必要があります。問い合わせを実行する前に明示的に Prepare メソッドを呼び出さない場合，問い合わせコンポーネントが自動的に行います。

ヒント プログラミング時に，パラメータには列の実際の名前に対応した変数名を指定することをお勧めします。たとえば，列名が「Number」ならば，対応したパラメータは「:Number」とします。一致した名前を使用することは，別のデータセットからパラメータ値を取得するためデータセットがデータソースを使用している場合，特に重要です。この手順については，22-45 ページの「マスター / 詳細関係をパラメータを使用して確立する」で述べます。

設計時にパラメータを与える

設計時には，パラメータコレクションエディタを使用してパラメータ値を指定できます。パラメータコレクションエディタを表示するには，オブジェクトインスペクタで Params または Parameters プロパティの省略記号ボタンをクリックします。SQL 文がパラメータを含んでいない場合は，コレクションエディタにオブジェクトは表示されません。

メモ パラメータコレクションエディタは，ほかのコレクションプロパティで表示されるコレクションエディタと同じです。エディタはほかのプロパティと共通なので，コンテキストメニューを右クリックすると［追加］コマンドと［削除］コマンドがあります。ただし，問い合わせパラメータに対しては使用できません。パラメータの追加と削除は，SQL 文の中でしか行えません。

各パラメータごとに，パラメータコレクションエディタで値を選択します。そして，オブジェクトインスペクタを使ってプロパティを変更します。

Params プロパティ（TParam オブジェクト）を使用する場合，以下の点を調べて，必要であれば修正します。

- DataType プロパティは，パラメータ値のデータ型を示す。いくつかのデータセットについては，この値が正確に初期化される場合があります。データセットが種類を導き出せない場合，DataType は ftUnknown となり，開発者はパラメータ値の種類を示すように変更しなければなりません。

  DataType プロパティは，パラメータ値のデータ型を示す。一般に，これらのデータ型はサーバーのデータ型と一致します。特定の論理型からサーバーデータ型へのマッピングについては，そのデータアクセスメカニズム（BDE，dbExpress，InterBase）のマニュアルを参照してください。

- ParamType プロパティは，選択されたパラメータの種類を示す。問い合わせでは入力パラメータしか指定できないため，このプロパティは常に ptInput になります。ParamType の値が ptUnknown である場合には，ptInput に変更してください。
- Value プロパティは，選択されたパラメータの値を指定する。実行時にアプリケーションでパラメータに値を供給する場合は，Value を空のままにできます。

Parameters プロパティ（TParameter オブジェクト）を使用する場合，以下の点を検査し，修正したいと思うでしょう。

- DataType プロパティは，パラメータ値のデータ型を示す。いくつかのデータ型については，以下のような，補足的な情報を提供しなければなりません。
  - NumericScale プロパティは，数値パラメータの小数点以下の桁数を示す
  - Precision プロパティは，数値パラメータの全桁数を示す
  - Size プロパティは，文字列パラメータ内の文字数を示す
- ParamType プロパティは，選択されたパラメータの種類を示す。問い合わせでは入力パラメータしか指定できないため，このプロパティは常に ptInput になります。
- Attributes プロパティは，パラメータが受け取る値の型を制御する。Attributes は，psSigned，psNullable，および psLong の組み合わせになり得ます。
- Value プロパティは，選択されたパラメータの値を指定する。実行時にアプリケーションでパラメータに値を供給する場合は，Value を空のままにできます。

実行時にパラメータ値を与える

実行時にパラメータを作成するには，次のリソースを使用できます。

- ParamByName メソッド（パラメータ名に基づいてパラメータ値を代入する場合。TADOQuery では使用不可）
- Params::Items または Parameters::Items プロパティ（SQL 文の中でのパラメータの順序に基づいてパラメータ値を代入する場合）
- Params::ParamValues または Parameters::ParamValues プロパティ（各パラメータセットの名前に基づいて，単一のコマンド行でパラメータに値を代入する）。

次のコードは，ParamByName を使用して編集ボックス内のテキストを :Capital パラメータに代入します。

```
SQLQuery1->ParamByName("Capital")->AsString = Edit1->Text;
```

同じコードを Params プロパティとインデックス 0 を使って書き直すと，次のようになります（:Capital パラメータが SQL 文の先頭パラメータである場合）。

```
SQLQuery1->Params->Items[0]->AsString = Edit1->Text;
```

次のコードは，Params::ParamValues プロパティを使って 3 つのパラメータを一度に設定します。

```
Query1->Params->ParamValues["Name;Capital;Continent"] =
  VarArrayOf(OPENARRAY(Variant, (Edit1->Text, Edit2->Text, Edit3->Text)));
```

ParamValues は，値をキャストする必要を避けるため Variants を使用します。

マスター / 詳細関係をパラメータを使用して確立する

詳細セットが問い合わせタイプのデータセットの場合，マスター / 詳細関係を設定するには，パラメータを使用する問い合わせを指定する必要があります。このパラメータは，マスターデータセットの現在の項目値を参照します。マスターデータセットの現在の項目値は実行時に動的に変化するため，マスターレコードが変わったら詳細セットのパラメータを再バインドしなければなりません。これを実行するコードはイベントハンドラを使って書くこともできますが，TIBQuery を除く問い合わせタイプのデータセットには，DataSource プロパティを使用した簡単なメカニズムがあります。

パラメータ付き問い合わせのパラメータ値を設計時または実行時に設定しなかった場合，問い合わせタイプのデータセットはその値を DataSource プロパティに基づいて与えようとします。DataSource には，バインドされていないパラメータ名に一致する項目名を検索する別のデータセットを設定します。この検索データセットは，任意の種類のデータセットです。検索データセットは，それを使用する詳細データセットよりも前に作成して，データを入力しておく必要があります。検索データセットで一致したものが見つかると，詳細データセットはそのデータソースが指す現在のレコードの項目値とパラメータ値をバインドします。

その処理方法を説明するため，顧客テーブルと注文テーブルという 2 つのテーブルを考えてみましょう。注文テーブルには，すべての顧客について，顧客が行った注文のセットが入っています。顧客テーブルには，ユニークな顧客 ID を指定する ID 項目があります。注文テーブルには，注文した顧客の ID を指定する CustID 項目があります。

最初の手順は，顧客データセットを設定することです。

1. アプリケーションにテーブルタイプのデータセットを追加して，Customer テーブルにそれを結合します。
2. CustomerSource という名前の TDataSource コンポーネントを追加します。DataSet プロパティを，手順 1 で追加したデータセットに設定します。このデータソースは顧客データセットを表します。
3. 問い合わせタイプのデータセットを追加して，その SQL プロパティを以下のように設定します。

```
SELECT CustID, OrderNo, SaleDate
FROM Orders
WHERE CustID = :ID
```

   パラメータ名は，マスター（顧客）テーブルの項目名と同じ

4. 詳細データセットの DataSource プロパティを CustomerSource に設定します。このプロパティを設定すると，詳細データセットはリンクされた問い合わせになります。

実行時には詳細データセットの SQL 文内の :ID パラメータには値が割り当てられていないので，データセットは CustomersSource によりデータセット内で識別された列でパラメータと名前が一致するものを検索します。CustomersSource はそのデータをマスターデータセットから取得し，マスターデータセットはそのデータを顧客テーブルから取得します。顧客テーブルには［ID］という列があり，マスターデータセットの現在レコードの ID 項目の値は，詳細データセットの SQL 文の ID パラメータに割り当てられます。データセットは，マスター / 詳細関係でリンクされています。顧客データセットの現在レコードが変化すると，詳細データセットの SELECT 文は現在の顧客 ID に基づいてすべての注文を取得します。

問い合わせの準備

問い合わせの準備は，問い合わせの実行前に行いますが省略することもできます。問い合わせが準備されると，SQL 文とそのパラメータがある場合，解析し，リソースを割り当て，効率を上げるためにデータアクセス層とデータベースサーバーに送られます。いくつかのデータセットでは，問い合わせの準備する際に，データセットが追加のセットアップ処理を実行する場合があります。このように準備をすると，特に更新可能な問い合わせの処理速度が高まり，アプリケーションが高速になります。

アプリケーションは Prepared プロパティを **true** に設定して問い合わせを準備できます。実行前に問い合わせを準備しないと，Open または ExecSQL を呼び出すごとにデータセットが問い合わせを自動的に準備します。データセットは自動的に問い合わせを準備しますが，最初に開く前に明示的にデータセットを準備することにより処理効率を改善できます。

```
CustQuery->Prepared = true;
```

データセットを明示的に準備する場合，文を実行するために割り当てられたリソースは，Prepared を **false** にするまで解放されません。

パラメータを追加する場合など，実行前に必ずデータセットの準備ができているようにするには，Prepared プロパティを **false** に設定します。

メモ 問い合わせの SQL プロパティのテキストを変更すると，データセットは問い合わせを自動的に閉じて準備を解除します。

結果セットを返さない問い合わせを実行する

SELECT 問い合わせなど，問い合わせがレコードのセットを返す場合には，データセットからレコードを取り出すのと同じ方法で問い合わせを実行できます。つまり，Active を **true** に設定するか，Open メソッドを呼び出します。

しかし，多くの SQL コマンドはレコードを返しません。そのようなコマンドには，SELECT 文以外のデータ定義言語（DDL）文またはデータ操作言語（DML）文を使用する文があります（たとえば，INSERT，DELETE，UPDATE，CREATE INDEX，ALTER TABLE コマンドはレコードを返しません）。

問い合わせタイプのデータセットの場合，結果セットを返さない問い合わせは，ExecSQL を呼び出すことによって実行できます。

```
CustomerQuery->ExecSQL(); // 結果セットを返さない
```

ヒント 問い合わせを複数回実行する場合は，Prepared プロパティを **true** に設定したほうがよいでしょう。

問い合わせがレコードを返さなくても，それが影響を及ぼしたレコードの数を知りたいと思うかもしれません（たとえば，DELETE 問い合わせによって削除されたレコードの数）。RowsAffected プロパティは，ExecSQL の呼び出し後に，影響を受けたレコードの数を示します。

ヒント 問い合わせが結果セットを返すかどうかが設計時にはわからない場合（たとえば，実行時にユーザーが動的に問い合わせを行う場合）は，**try...catch** ブロックに両方の問い合わせ実行文を記述してください。Open メソッドの呼び出しは **try** 節に入れます。アクション問い合わせは Open メソッドによってアクティブにされたときに実行されますが，例外はそれに加えて発生するからです。例外をチェックして，それが単に結果セットの不足を示す場合は抑止してください（たとえば，TQuery はこのことを ENoResultSet 例外によって示します）。

単方向結果セットの使い方

問い合わせタイプのデータセットが結果セットを返す場合，その結果セットの最初のレコードへのポインタつまりカーソルも受け取ります。カーソルが指すレコードは現在アクティブなレコードです。現在のレコードは，結果セットのデータソースに関連付けられたデータベース対応コンポーネントに項目値が表示されるレコードです。dbExpress を使用している場合以外は，このカーソルはデフォルトで双方向です。双方向カーソルは，レコードを前方にも後方にも移動できます。双方向カーソルのサポートには，追加の処理がある程度必要なため，問い合わせの速度がやや低下する可能性があります。

結果セットを逆方向に移動する必要がなければ，TQuery と TIBQuery を使えば，かわりに単方向カーソルを要求して，問い合わせの効率を向上させることができます。単方向カーソルを要求するには，UniDirectional プロパティを **true** に設定します。

UniDirectional は問い合わせを準備して実行する前に設定します。次のコードでは，問い合わせの準備と実行の前に UniDirectional を設定しています。

```
if (!CustomerQuery->Prepared)
{
  CustomerQuery->UniDirectional = true;
  CustomerQuery->Prepared = true;
}
CustomerQuery->Open(); // 単方向カーソルを持つ結果セットを返す
```

メモ UniDirectional プロパティを単方向データセットと混同しないでください。単方向データセット（TSQLDataSet，TSQLTable，TSQLQuery，および TSQLStoredProc）は dbExpress を使います。これは単方向カーソルしか返しません。逆向きに移動する機能の制限に加え，単方向データセットはレコードをバッファしないため，フィルタが使用できないなどの，ほかの制限もあります。

ストアドプロシージャタイプのデータセットの使い方

アプリケーションでのストアドプロシージャの使用法は，そのストアドプロシージャのコーディング方法，データを返す方法，使用するデータベースサーバー，およびこれらの要因の組み合わせによって異なります。

一般的にサーバーにあるストアドプロシージャにアクセスするには，アプリケーションは以下のことを行わなければなりません。

1. 適切なデータセットコンポーネントをデータモジュール上かフォーム上に配置して，Name プロパティにアプリケーションに適したユニークな値を設定します。
2. ストアドプロシージャを定義しているデータベースサーバーを設定します。設定の方法は，ストアドプロシージャタイプのデータセットごとに異なっています。しかし通常は，データベース接続コンポーネントを指定します。
   - TStoredProc の場合，DatabaseName プロパティを使用し，TDatabase コンポーネントまたは BDE エリアスを指定する
   - TADOStoredProc の場合，Connection プロパティを使用し，TADOConnection コンポーネントを指定する
   - TSQLStoredProc の場合，SQLConnection プロパティを使用し，TSQLConnection コンポーネントを指定する
   - TIBStoredProc の場合，Database プロパティを使用し，TIBConnection コンポーネントを指定する

   データベース接続コンポーネントの使い方については，第 21 章「データベースへの接続」を参照してください。
3. 実行するストアドプロシージャの指定ほとんどのストアドプロシージャタイプのデータセットの場合，StoredProcName プロパティを設定します。例外は TADOStoredProc で，かわりに ProcedureName プロパティがあります。
4. ストアドプロシージャが返すカーソルがビジュアルデータコントロールで使用される場合は，データソースコンポーネントをフォームやデータモジュールに追加して，DataSet プロパティにストアドプロシージャタイプのデータセットを設定します。データベース対応コンポーネントの DataSource プロパティと DataField プロパティを使用してデータソースに接続します。
5. 必要であればストアドプロシージャに入力パラメータ値を指定します。サーバーがすべてのストアドプロシージャパラメータについての情報を提供するとは限らない場合，パラメータ名やデータ型のような付加的な入力パラメータ情報を用意する必要があります。ストアドプロシージャパラメータを扱う方法については，22-49 ページの「ストアドプロシージャのパラメータの操作」を参照してください。

6. ストアドプロシージャを実行します。カーソルを返すストアドプロシージャの場合は，Active プロパティか Open メソッドを使います。結果を返さない，または出力パラメータだけを返すストアドプロシージャを実行する場合は，実行時に ExecProc メソッドを使います。複数回ストアドプロシージャを実行する予定であれば，Prepare を呼び出して，データアクセス層を初期化し，パラメータ値をストアドプロシージャに結合するとよいでしょう。ストアドプロシージャの準備については，22-52 ページの「結果セットを返さないストアドプロシージャを実行する」を参照してください。
7. 結果を処理します。これらの結果は，結果および出力パラメータとして返すことができます。または，ストアドプロシージャタイプのデータセット内の結果セットとして返すこともできます。一部のストアドプロシージャは，複数のカーソルを返します。付加的なカーソルについての詳細は，22-52 ページの「複数の結果セットを取得する」を参照してください。

ストアドプロシージャのパラメータの操作

ストアドプロシージャには，以下の 4 種類のパラメータを関連付けることができます。

- 入力パラメータ：値をストアドプロシージャに渡すために使います。
- 出力パラメータ：ストアドプロシージャが戻り値をアプリケーションに渡すために使います。
- 入出力パラメータ：値をストアドプロシージャに渡すために使い，ストアドプロシージャが戻り値をアプリケーションに渡すために使います。
- 結果パラメータ：一部のストアドプロシージャがエラー値または状態値をアプリケーションに返すために使います。ストアドプロシージャが返せる結果パラメータは 1 つだけです。

ストアドプロシージャが特定の種類のパラメータを使うかどうかは，データベースサーバーにおけるストアドプロシージャの汎用言語実装と，実際の個々のストアドプロシージャの両方によって決まります。どのサーバーでも，個々のストアドプロシージャには入力パラメータを使用するものと使用しないものがあります。これに対して，サーバーごとにパラメータの使い方が決まっている場合もあります。たとえば，MS-SQL サーバーと Sybase ではストアドプロシージャは必ず結果パラメータを返しますが，InterBase ではストアドプロシージャが結果パラメータを返すことはありません。

ストアドプロシージャパラメータへのアクセスは，Params プロパティ（TStoredProc, TSQLStoredProc，TIBStoredProc の場合）または Parameters プロパティ（TADOStoredProc の場合）によって提供されます。StoredProcName（または ProcedureName）プロパティに値を割り当てると，データセットは自動的に，ストアドプロシージャのパラメータごとにオブジェクトを生成します。一部のデータセットの場合，ストアドプロシージャ名が指定されないときには，実行時にパラメータごとのオブジェクトをプログラミングによって作成する必要があります。ストアドプロシージャを指定せずに手動で TParam または Tparameter オブジェクトを作成すると，1 つのデータセットを任意の数のストアドプロシージャで使用することができます。

メモ 一部のストアドプロシージャは出力パラメータと結果パラメータだけでなくデータセットも返します。アプリケーションはデータセットのレコードをデータベース対応コントロールで表示できますが，出力パラメータと結果パラメータを別々に処理しなければなりません。

設計時のパラメータの設定

設計時にパラメータコレクションエディタでストアドプロシージャパラメータ値を指定できます。パラメータコレクションエディタを表示するには，オブジェクトインスペクタで Params または Parameters プロパティの省略記号ボタンをクリックします。

重要 パラメータコレクションエディタでパラメータを選択し，オブジェクトインスペクタで Value プロパティを設定すると，入力パラメータに値を割り当てることができます。ただし，サーバーから通知される入力パラメータの名前とデータ型は変更しないでください。そうでないと，ストアドプロシージャを実行したときに例外が生成されます。

一部のサーバーは，パラメータの名前もデータ型も示しません。その場合には，パラメータコレクションエディタを使って手動でパラメータを設定しなければなりません。右クリックしてから［追加］を選択して，パラメータを追加します。追加するパラメータごとに，完全に記述しなければなりません。パラメータを追加する必要がない場合でも，それらが正確であることを保証するために，個々のパラメータオブジェクトのプロパティをチェックしてください。

データセットに Params プロパティがある場合（TParam オブジェクト），以下のプロパティを正確に指定しなければなりません。

- Name プロパティは，ストアドプロシージャで定義されたパラメータの名前を示す
- DataType プロパティは，パラメータ値のデータ型を示す。TSQLStoredProc を使用する場合，一部のデータ型では付加的な情報が必要です。
  - NumericScale プロパティは，数値パラメータの小数点以下の桁数を示す
  - Precision プロパティは，数値パラメータの全桁数を示す
  - Size プロパティは，文字列パラメータ内の文字数を示す
- ParamType プロパティは，選択されたパラメータの種類を示す。これは，ptInput（入力パラメータ），ptOutput（出力パラメータ），ptInputOutput（入出力パラメータ）または ptResult（結果パラメータ）のいずれかになります。
- Value プロパティは，選択されたパラメータの値を指定する。出力パラメータと結果パラメータの値は設定できません。この種類のパラメータ値は，ストアドプロシージャの実行によって設定されます。入出力パラメータの場合，アプリケーションが実行時にパラメータを提供するなら，Value を空のままにできます。

データセットに Parameters プロパティがある場合（TParameter オブジェクト），以下のプロパティを正確に指定しなければなりません。

- Name プロパティは，ストアドプロシージャで定義されたパラメータの名前を示す
- DataType プロパティは，パラメータ値のデータ型を示す。いくつかのデータ型については，以下のような，補足的な情報を提供しなければなりません。
  - NumericScale プロパティは，数値パラメータの小数点以下の桁数を示す
  - Precision プロパティは，数値パラメータの全桁数を示す
  - Size プロパティは，文字列パラメータ内の文字数を示す
- ParamType プロパティは，選択されたパラメータの種類を示す。これは，pdInput（入力パラメータ），pdOutput（出力パラメータ），pdInputOutput（入出力パラメータ）または pdReturnValue（結果パラメータ）のいずれかになります。

- Attributes プロパティは，パラメータが受け取る値の型を制御する。Attributes は，psSigned, psNullable，および psLong の組み合わせになり得ます。
- Value プロパティは，選択されたパラメータの値を指定する。出力パラメータと結果パラメータの値は設定できません。入出力パラメータの場合，アプリケーションが実行時にパラメータを提供するなら，Value を空のままにできます。

実行時にパラメータを使用する

一部のデータセットでは，実行時までにストアドプロシージャ名が指定されていない場合，パラメータ用に TParam オブジェクトは自動的に作成されません。プログラミングによって作成する必要があります。これには，新しい TParam オブジェクトをインスタンス化するか，TParams::AddParam メソッドを使用します。

```
TParam *P1, *P2;
StoredProc1->StoredProcName = "GET_EMP_PROJ";
StoredProc1->Params->Clear();
P1 = new TParam(StoredProc1->Params, ptInput);
P2 = new TParam(StoredProc1->Params, ptOutput);
try
{
  StoredProc1->Params->Items[0]->Name = "EMP_NO";
  StoredProc1->Params->Items[1]->Name = "PROJ_ID";
  StoredProc1->ParamByName("EMP_NO")->AsSmallInt = 52;
  StoredProc1->ExecProc();
  Edit1->Text = StoredProc1->ParamByName("PROJ_ID")->AsString;
}
__finally
{
  delete P1;
  delete P1;
}
```

実行時に個々のパラメータオブジェクトを追加する必要がなくても，入力パラメータに値を割り当て，出力パラメータから値を取得するために，個々のパラメータオブジェクトにアクセスしたいこともあるでしょう。データセットの ParamByName メソッドを使用して名前から個々のパラメータにアクセスできます。たとえば次のコードは，入出力パラメータ値を設定してストアドプロシージャを実行し，戻り値を取り出します。

```
SQLDataSet1->ParamByName("IN_OUTVAR")->AsInteger = 103;
SQLDataSet1->ExecSQL();
int Result = SQLDataSet1->ParamByName("IN_OUTVAR")->AsInteger;
```

ストアドプロシージャの準備

問い合わせタイプのデータセットの場合と同様，ストアドプロシージャタイプのデータセットは，ストアドプロシージャを実行する前に準備しておかなければなりません。ストアドプロシージャの準備は，ストアドプロシージャに資源を分配し，パラメータを結合するように，データアクセス層とデータベースサーバーに知らせます。これらの処理により，性能が向上します。

準備する前にストアドプロシージャを実行しようとすると，データセットが自動的に準備を行い，実行後に準備を解除します。ストアドプロシージャを何回も実行する場合は，Prepare プロパティを **true** に設定して，明示的に準備を行う方が効率的です。

```
MyProc->Prepared = true;
```

データセットを明示的に準備する場合，ストアドプロシージャを実行するために割り当てられたリソースは，Prepared を **false** にするまで解放されません。

Oracle のオーバーロードプロシージャを使っていてパラメータを変更する場合など，実行前に必ずデータセットの準備ができているようにするには，Prepared プロパティを **false** に設定します。

結果セットを返さないストアドプロシージャを実行する

ストアドプロシージャがカーソルを返す場合には，データセットからレコードを取り出すのと同じ方法で実行できます。つまり，Active を **true** に設定するか，Open メソッドを呼び出します。

しかし，多くのストアドプロシージャはデータを返しません。または，出力パラメータで結果だけを返します。結果セットを返さないストアドプロシージャを実行するには，ExecProc メソッドを呼び出します。ストアドプロシージャを実行した後に，ParamByName メソッドを使用して，結果パラメータ，または任意の出力パラメータの値を読むこともできます。

```
MyStoredProcedure->ExecProc(); // 結果セットを返さない
Edit1->Text = MyStoredProcedure->ParamByName("OUTVAR")->AsString;
```

メモ TADOStoredProc には ParamByName メソッドはありません。ADO を使用している場合に出力パラメータ値を取得するには，Parameters プロパティを使用して，パラメータオブジェクトにアクセスします。

ヒント プロシージャを複数回実行する場合は，Prepared プロパティを **true** に設定したほうがよいでしょう。

複数の結果セットを取得する

一部のストアドプロシージャは，複数のレコードセットを返します。データセットを開くと，データセットは最初のレコードセットだけを取得します。TSQLStoredProc または TADOStoredProc を使用している場合には，NextRecordSet メソッドか NextRecordset メソッドを呼び出せば，ほかのレコードのセットにアクセスできます。

```
TCustomSQLDataSet *DataSet2 = SQLStoredProc1->NextRecordSet();
```

TSQLStoredProc 内では，NextRecordSet は，次のレコードセットへのアクセスを提供する TCustomSQLDataSet コンポーネントを新たに作成して返します。TADOStoredProc 内では，NextRecordset は，既存の ADO データセットの RecordSet プロパティに割り当てられるインターフェースを返します。どちらのクラスでも，メソッドは，出力パラメータとして返されたデータセットの中のレコードの数を返します。

つまり NextRecordSet か NextRecordset を最初に呼び出すと，2 番目のレコードセットのデータセットを返します。NextRecordSet か NextRecordset をもう一度呼び出すと，3 番目のデータセットを返し，こうしてレコードセットがなくなるまで続きます。それ以上のデータセットがなくなると，NextRecordSet または NextRecordset はヌルを返します。

第23章

項目コンポーネントの操作

この章では，TField オブジェクトとその下位オブジェクトに共通のプロパティ，イベント，およびメソッドについて説明します。項目コンポーネントはデータセット内の個々の項目（列）を表します。またこの章では，項目コンポーネントを使ってアプリケーションでの表示と編集を制御する方法についても説明します。

項目コンポーネントは，必ずデータセットに関連付けられています。アプリケーションでは，TField オブジェクトを直接使用することはありません。そのかわりアプリケーション内で使用する項目コンポーネントは，データセット内の列のデータ型に固有の TField の下位オブジェクトです。項目コンポーネントは，関連するデータセットの特定の列のデータにアクセスするため，TDBEdit や TDBGrid などのデータベース対応コントロールを提供します。

一般的には 1 つの項目コンポーネントは，データ型やサイズなどデータセット内の 1 つの列つまり項目の特性を表します。その上，配置，表示形式，編集形式などの項目の表示特性も表しています。たとえば TFloatField コンポーネントには，データの外観に直接影響する 4 つのプロパティがあります。

表 23.1　データの表示形式に影響する TFloatField プロパティ

プロパティ	目的
Alignment	データの位置合わせを左揃え，中央揃え，右揃えに指定する
DisplayWidth	コントロールに一度に表示する桁数を指定する
DisplayFormat	データの表示形式を指定する（表示する小数点の桁数など）
EditFormat	編集中の値の表示方法を指定する

データセット内のレコードのスクロール時には，項目コンポーネントは現在レコード内の項目値を表示して変更できるようにします。

項目コンポーネントには DisplayWidth や Alignment など，多くの共通プロパティがありますが，TFloatField の Precision のようにデータ型に固有のプロパティもあります。こうしたプロパティはそれぞれ，フォーム上でのデータの表示形式に影響します。また Precision など一部のプロパティは，データを変更または入力するときに，ユーザーがコントロールに入力できる値に影響を与えます。

データセットの項目コンポーネントは動的（データベーステーブルの基本構造に基づいて自動生成される）か，または持続的（項目エディタで設定した特定の項目名とプロパティに基づいて生成される）です。動的項目と持続的項目の長所は違っており，適合するアプリケーションの種類も違います。以降のセクションでは，動的項目と持続的項目について詳細に説明し，選択する上での助言を提供します。

動的項目コンポーネント

動的に生成される項目コンポーネントがデフォルトです。実際どのデータセットでも，初めてデータセットをデータモジュールに配置して，データセットがデータを取り出す方法を指定して開いた場合，全項目コンポーネントは動的項目として始まります。項目コンポーネントが自動生成された場合，項目コンポーネントは動的になります。データセットは基になるデータ内の各列それぞれに，1つの項目コンポーネントを生成します。各列に作成される TField の下位コンポーネントは，データベースやプロバイダコンポーネント（TClientDataSet の場合）から受信した項目型情報によって決まります。

動的項目は一時的なものです。動的項目は，データセットが開いている間だけ存在します。動的項目を使用しているデータセットを開くたびに，データセットの基になっているデータの現在構造に基づいて，完全に新しい動的項目コンポーネントセットが再構築されます。基になっているデータが変更されると，動的項目コンポーネントを使用しているデータセットを次に開いたときに，自動生成される項目コンポーネントもそれに合わせて変更されます。

データの表示や編集に柔軟性を必要とするアプリケーションでは，動的項目を使用します。たとえば，SQL エクスプローラのようなデータベースのブラウズツールを作成するには，各データベーステーブルで列数や列の型が異なるため，動的項目を使わなければなりません。ユーザーとデータの対話のほとんどがグリッドコンポーネント内部で起こり，しかもアプリケーションが使用するデータセットが頻繁に変更されるアプリケーションでも，動的項目を使うと便利です。

アプリケーションで動的項目を使う手順は，次のとおりです。

1. データセットとデータソースをデータモジュールに配置します。
2. データセットとデータを関連付けます。これには，データソースに接続する接続コンポーネントやプロバイダの使用および，データセットが表すデータを指定するプロパティの設定が含まれています。
3. データソースとデータセットを関連付けます。
4. データベース対応コントロールをアプリケーションのフォームに配置して，データモジュールのヘッダーをフォームユニットにインクルードして，各データベース対応コントロールをモジュール内のデータソースに関連付けます。また項目をデータベース対応コントロールに関連付けます。動的項目コンポーネントを使用しているので，指定した項目名がデータセットを開いたときに存在する保証はありません。
5. データセットを開きます。

使いやすさを別にすると，動的項目は限定的です。コードを記述しなければ，動的項目の表示および編集のデフォルト設定を変更することも，動的項目の表示順序を安全に変更することも，データセッ

トの項目へのアクセスを防止することもできません。データセットには計算項目や参照項目など追加項目を作成できず，動的項目のデフォルトのデータ型をオーバーライドすることもできません。データベースアプリケーション内の項目への制御と柔軟性を確保するには，項目エディタを呼び出してデータセット用の持続的項目コンポーネントを作成する必要があります。

持続的項目コンポーネント

デフォルトでは，データセット項目は動的です。そのプロパティと利用可能性は自動的に設定され，変更は不可能です。項目のプロパティとイベントを制御するには，データセット用の持続的項目を作成する必要があります。持続的項目では，以下のことが可能です。

- 設計時や実行時に項目の表示や編集特性を設定したり，変更します。
- 参照項目，計算項目，集合項目など，データセット内の既存項目に基いて新規項目を作成します。
- データ入力を検証します。
- 基になるデータベースの特定の項目へのアクセスを防止するため，持続的コンポーネントのリストから項目コンポーネントを削除します。
- テーブルや基になっているデータセットへの問い合わせの列に基づいて，既存の項目を置換する新規項目を定義します。

設計時には項目エディタを使って，アプリケーションのデータセットで使用される項目コンポーネントの持続的リストを作成します。持続的項目コンポーネントのリストはアプリケーションに格納され，データセットの基になっているデータベース構造が変化しても変化しません。項目エディタで持続的項目を作成した後は，その項目に対し，データ値の変更に応答したり，データ入力を検証するイベントハンドラも作成できます。

メモ データセットに持続的項目を作成する場合，設計時および実行時には選択した項目だけがアプリケーションで利用できます。設計時には，項目エディタを使用してデータセットの持続的項目を追加したり，削除できます。

単一のデータセットが使用する項目はすべて，持続的または動的になります。単一のデータセットでは，項目型を混合できません。データセットに対して持続的項目を作成した後で，それを動的項目に変更したい場合は，データセットからすべての持続的項目を削除しなくてはなりません。動的項目についての詳細は，23-2 ページの「動的項目コンポーネント」を参照してください。

メモ 持続的項目の主要用途の 1 つは，データの表示と外観を制御することです。また，データベース対応グリッド内の列の外観も制御できます。グリッド内の列の外観を制御する方法については，19-16 ページの「カスタマイズされたグリッドの作成」を参照してください。

持続的項目の作成

項目エディタで作成された持続的項目コンポーネントは，基になっているデータに対してプログラムによる効率的で読み出しやすい，型保障されたアクセスを提供します。持続的項目コンポーネントを使うと，アプリケーションを実行するたびに，基礎になるデータベースの物理構造が変更されている場合でも，同じ列と同じ順番を使って表示することが保障されます。特定の項目に依存するデータ

ベース対応のコンポーネントとプログラムコードは，期待通りに動作します。持続的項目コンポーネントの基礎となる列が削除されたり変更されると，存在しない列や一致しない列に対してアプリケーションが実行されることはなく，C++Builder によって例外が生成されます。

データセットに持続的項目を作成する手順は，次のとおりです。

1. データモジュールにデータセットを配置します。
2. データセットを基になっているデータに結合します。一般にこれには，データセットを接続コンポーネントかプロバイダと関連付けて，データを記述するプロパティを指定することが含まれています。たとえば TADODataSet を使用している場合，Connection プロパティに適切に設定された TADOConnection コンポーネントを設定したり，CommandText プロパティに有効な問い合わせを設定できます。
3. データモジュール内のデータセットコンポーネントをダブルクリックして，項目エディタを呼び出します。項目エディタには，タイトルバー，ナビゲータボタン，リストボックスがあります。

   項目エディタのタイトルバーには，データセットの入っているデータモジュールやフォームの名前と，データセット自体の名前が表示されます。たとえば CustomerData データモジュールの Customers というデータセットを開くと，タイトルバーには「CustomerData->Customers」などと表示されます。

   タイトルバーの下にはナビゲータボタンがあります。このボタンでは，設計時にアクティブなデータセットでレコードを 1 つずつスクロールしたり，先頭レコードや最終レコードに移動できます。データセットがアクティブでない場合や空の場合は，ナビゲータボタンは淡色表示になります。データセットが単方向の場合は，最終レコードと前のレコードに移動するボタンも淡色表示になります。

   リストボックスには，データセットの持続的項目コンポーネント名が表示されます。新規データセットで最初に項目エディタを呼び出したときにリストが空のままなのは，そのデータセットの項目コンポーネントが持続的ではなく動的だからです。すでに持続的項目コンポーネントを持っているデータセットで項目エディタを呼び出すと，リストボックスには項目コンポーネント名が表示されます。
4. 項目エディタのコンテキストメニューから，［項目の追加］を選択します。
5. 持続的にする項目を，［項目の追加］ダイアログボックスで選択します。デフォルトでは，ダイアログボックスが表示されたときにすべての項目が選択されています。選択した項目は，持続的項目になります。

［項目の追加］ダイアログボックスが閉じ，選択した項目が項目エディタのリストボックスに表示されます。項目エディタのリストボックスに表示されている項目は，持続型です。データセットがアクティブであれば，リストボックスの上にある［次］と（データセットが単方向でない場合）［最後］のナビゲータボタンが使用可能になることにも注意してください。

その後はデータセットを開くたびに，基になるデータベースの各列に対して動的な項目コンポーネントは作成されません。かわりに，指定した項目に対する持続的コンポーネントが作成されるだけです。

データセットを開くたびに，計算項目でない持続的項目がそれぞれ存在するかどうか，あるいはデータベース内のデータを使ってそうした項目を作成できるかどうかが確認されます。この条件が満たされない場合は，項目が無効であることを警告する例外が生成され，データセットは開かれません。

持続的項目の並べ替え

持続的項目コンポーネントが項目エディタのリストボックスに表示される順序は，データベース対応のグリッドコンポーネントに項目が表示される順序がデフォルトです。リストボックス内で項目をドラッグアンドドロップして，項目の順序を変更できます。

項目の順序を変更する手順は次のとおりです。

1. 項目を選択します。一度に 1 つまたは複数の項目を選択して並べ替えることができます。
2. 選択した項目を新しい位置にドラッグします。

連続していない項目を選択して新しい位置にドラッグすると，それらは連続したブロックとして挿入されます。ブロック内での項目間の順序は変わりません。

あるいは，項目を選択した後〔Ctrl〕+〔↑〕と〔Ctrl〕+〔↓〕を使用して，リスト内の個々の項目の順序を変更できます。

新しい持続的項目の定義

既存のデータセット項目を持続的項目にする以外にも，特別な持続的項目をデータセットに追加として，または別の持続的項目の置き換えとして作成できます。

作成された持続的項目は表示専用です。実行時のデータの内容は，データベースのどこかにすでに存在しているか一時的に存在するだけなので保持されません。データセットの基になっているデータの物理的構造は，いずれにしても変更されません。

新しい持続的項目コンポーネントを作成するには，項目エディタのコンテキストメニューを呼び出して［項目の新規作成］を選択します。［項目の新規作成］ダイアログボックスが表示されます。

［項目の新規作成］ダイアログボックスには 3 つのグループボックスがあります。［項目のプロパティ］，［項目の種類］，および［参照の定義］です。

- ［項目のプロパティ］グループボックスでは，一般的な項目コンポーネント情報を入力できます。項目名を［名前］編集ボックスに入力します。ここで入力する名前は，項目コンポーネントの FieldName プロパティに対応しています。［項目の新規作成］ダイアログボックスでは，この名前を使って［コンポーネント］編集ボックスにコンポーネント名を作成します。［コンポーネント］編集ボックスに表示される名前は，項目コンポーネントの Name プロパティに対応しており，その目的は情報の提供だけです。Name プロパティは，ソースコード中で項目コンポーネントを参照する識別子です。［コンポーネント］編集ボックスに直接入力しても無視され，Name プロパティは変更されません。
- ［項目のプロパティ］グループの［型］コンボボックスでは，項目コンポーネントのデータ型を指定できます。新しく作成する項目コンポーネントには，必ずデータ型を指定しなければなりません。たとえば項目で浮動小数点の通貨値を表示するには，ドロップダウンリストから Currency を選択します。［サイズ］編集ボックスは，文字列項目への表示または入力が可能な最大文字数を指定したり，Bytes 項目と VarBytes 項目のサイズを指定するのに使います。それ以外のデータ型では，［サイズ］は意味がありません。
- ［項目の種類］ラジオグループでは，新しく作成する項目コンポーネントの種類を指定できます。デフォルトの種類は［データ］です。［参照］を選択すると，［参照の定義］グループボックスで

［データセット］と［キー項目］の 2 つの編集ボックスが使用可能になります。計算項目も作成でき，クライアントデータセットを操作している場合はさらに内部計算項目や集合項目が作成できます。次の表に，作成できる項目の種類を示します。

表 23.2 特別な持続的項目の種類

項目の種類	目的
データ	既存の項目を置換する（たとえばデータ型変更のため）
計算項目	データセットの OnCalcFields イベントハンドラが実行時に計算した値を表示する
参照	指定されたデータセットから，実行時に指定した検索条件で値を検索する（単方向データセットではサポートされない）
内部計算項目	実行時にクライアントデータセットが計算して，データとともに格納した値を表示する
集合体	クライアントデータセットのレコードセットのデータの集計値を表示する

［参照の定義］グループボックスは，参照項目の作成専用です。これについて詳しくは，23-8 ページの「参照項目の定義」で説明しています。

データ項目を定義する

データ項目は，データセットの既存の項目を置換します。たとえば，プログラム上の理由で TSmallIntField を TIntegerField に置換するとします。項目のデータ型は直接変更できないため，置換に使う項目を新しく定義しなければなりません。

重要 新しい項目を定義して既存の項目を置換する場合でも，定義する項目のデータ値はデータセットの基になるテーブルの列から取得しなければなりません。

基になるデータセットのテーブルの項目を置換する項目を作成する手順は次のとおりです。

1. データセットに割り当てられている持続的項目のリストから目的の項目を削除し，コンテキストメニューで［項目の新規作成］を選択します。
2. ［項目の新規作成］ダイアログボックスの中の［名前］編集ボックスに，データベーステーブルの既存の項目名を入力します。新しい項目名は入力しないでください。ここでは，新しい項目が値を取り出す元になるデータセットの項目名を指定します。
3. ［型］コンボボックスから，項目の新しいデータ型を選択します。選択するデータ型は，置換する項目のデータ型とは別にする必要があります。文字列項目を，サイズの異なる文字列項目には置換できません。データ型は違う必要がありますが，基になっているテーブルでの実際の項目型と互換の型でなければなりません。
4. 必要であれば，［サイズ］編集ボックスに項目のサイズを入力します。サイズの入力が必要なのは，TStringField，TBytesField，TVarBytesField だけです。
5. まだ選択していなければ，［項目の種類］ラジオグループの［データ］を選択します。
6. ［OK］を選択します。［項目の新規作成］ダイアログボックスは閉じ，手順の 1 で指定した既存の項目は新しく定義したデータ項目で置換されます。データモジュールまたはフォームのヘッダにあるコンポーネント宣言は更新されます。

項目コンポーネントに関連したプロパティやイベントを編集するには，項目エディタのリストボックスでコンポーネント名を選択し，オブジェクトインスペクタでプロパティやイベントを編集します。

項目コンポーネントのプロパティやイベントの編集についての詳細は，23-10 ページの「持続的項目のプロパティとイベントを設定する」を参照してください。

計算項目の定義

計算項目は，実行時にデータセットのイベントハンドラ OnCalcFields が計算した値を表示します。たとえば，ほかの項目値と結合した値を表示する文字列項目を作成できます。

［項目の新規作成］ダイアログボックスで計算項目を作成する手順は次のとおりです。

1. ［名前］編集ボックスに計算項目名を入力します。既存の項目名は入力しないでください。
2. ［型］コンボボックスから，項目のデータ型を選択します。
3. 必要であれば，［サイズ］編集ボックスに項目のサイズを入力します。サイズの入力が必要なのは，TStringField，TBytesField，TVarBytesField だけです。
4. ［項目の種類］ラジオグループで，［計算項目］または［内部計算項目］を選択します。内部計算項目は，クライアントデータセットの処理を行うときに選択します。両計算項目の大きな違いは，内部計算項目用に計算される値がクライアントデータセットのデータの一部として格納されたり，取り出される点です。
5. ［OK］を選択します。新しく定義された計算項目は，自動的に項目エディタのリストボックスの持続的項目リストの最後に追加され，コンポーネントの宣言は自動的にフォームやデータモジュールのヘッダに追加されます。
6. データセットのイベントハンドラ OnCalcFields で，項目値を計算するコードを記述します。項目値を計算するコードの記述についての詳細は，23-7 ページの「計算項目のプログラミング」を参照してください。

メモ 項目コンポーネントに関連したプロパティやイベントを編集するには，項目エディタのリストボックスでコンポーネント名を選択し，オブジェクトインスペクタでプロパティやイベントを編集します。項目コンポーネントのプロパティやイベントの編集については，23-10 ページの「持続的項目のプロパティとイベントを設定する」を参照してください。

計算項目のプログラミング

計算項目を定義したら，項目値を計算するコードを記述しなければなりません。そうしないと，項目は必ずヌル値になります。計算項目のコードはデータセットの OnCalcFields イベントに記述します。

計算項目の値をプログラミングする手順は次のとおりです。

1. オブジェクトインスペクタのドロップダウンリストから，データセットコンポーネントを選択します。
2. オブジェクトインスペクタの［イベント］ページを選択します。
3. OnCalcFields プロパティをダブルクリックしてデータセットコンポーネントの CalcFields 手続きを呼び出すか作成します。
4. 計算項目の値やそのほかのプロパティを設定するコードを記述します。

たとえばデータモジュール CustomerData のテーブル Customers に，CityStateZip という計算項目を作成したとします。CityStateZip は，会社が存在する都市，州，郵便番号を 1 行でデータベース対応コントロールに表示する必要があります。

Customers テーブルの CalcFields 手続きにコードを追加するには，オブジェクトインスペクタのドロップダウンリストから Customers テーブルを選択して［イベント］ページに切り替え，OnCalcFields プロパティをダブルクリックします。

TCustomerData::CustomersCalcFields 手続きが，ユニットのソースコードウィンドウに表示されます。次のコードを，項目を計算する手続きに追加します。

```
CustomersCityStateZip->Value = CustomersCity->Value + AnsiString(", ") +
    CustomersState->Value + AnsiString(" ") + CustomersZip->Value;
```

メモ 内部計算項目の OnCalcFields イベントハンドラを書く場合，クライアントデータセットの State プロパティにチェックマークを付けて，State が dsInternalCalc の時だけ値を再計算させると，処理効率を向上できます。詳細は，27-10 ページの「クライアントデータセットで内部計算項目を使用する」を参照してください。

参照項目の定義

参照項目は読み取り専用項目で，実行時に検索条件に基づいて値を表示します。もっとも簡単なのは，既存の検索項目名，検索する項目値，参照データセット内の値を表示する項目の 3 つを参照項目に渡すことです。

たとえば通信販売のアプリケーションの場合，オペレータは参照項目を使用して，顧客から受け取った郵便番号に対応する都市と州を自動的に設定できます。この場合検索対象の列は ZipTable->Zip で，検索する値は Order->CustZip に入力される顧客の郵便番号です。戻り値は，ZipTable->Zip が Order->CustZip 項目の現在値と一致するレコードの ZipTable->City と ZipTable->State という 2 つの列の値です。

メモ 単方向データセットは，参照項目をサポートしていません。

［項目の新規作成］ダイアログボックス内で参照項目を作成する手順は次のとおりです。

1. 参照項目名を［名前］編集ボックスに入力します。既存の項目名は入力しないでください。
2. ［型］コンボボックスから，項目のデータ型を選択します。
3. 必要であれば，［サイズ］編集ボックスに項目のサイズを入力します。サイズの入力が必要なのは，TStringField，TBytesField，TVarBytesField だけです。
4. ［項目の種類］ラジオグループの［参照］を選択します。［参照］を選択すると，［データセット］と［キー項目］という 2 つのコンボボックスが使用可能になります。
5. ［データセット］コンボボックスのドロップダウンリストから項目値参照の対象となるデータセットを選択します。参照のデータセットは，項目コンポーネントそのもののデータセットとは別でなければなりません。そうでないと実行時に循環参照の例外が生成されます。参照のデータセットを指定すると，［参照側のキー］と［値を返す項目］の 2 つのコンボボックスが使用可能になります。
6. ［キー項目］ドロップダウンリストから，値の照合基準になる現在のデータセットの項目を選択します。複数の項目で照合するには，ドロップダウンリストの選択ではなく，直接項目名を入力します。複数の項目名を指定する場合は，項目名をセミコロンで区切ります。複数の項目を使用する場合は，その項目は持続的項目コンポーネントである必要があります。
7. ［参照側のキー］ドロップダウンリストから手順の 6 で指定した［キー項目］項目を基準に比較するための参照データセットの項目を選択します。複数の項目を指定するには，同じ数だけ参照項目を指定します。複数の項目名を指定する場合は，各項目名をセミコロンで区切ります。

8. ［値を返す項目］ドロップダウンリストから，作成する参照項目の値として返す参照データセットの項目を選択します。

アプリケーションを設計して実行する場合，参照項目の値は計算項目の値が計算される前に決定されます。計算項目は参照項目に基づくことができますが，参照項目が計算項目に基づくことはできません。

LookupCache プロパティを使用すると，参照項目の決定方法をスムーズにします LookupCache は，データセットを初めて開いたときに参照項目の値がメモリにキャッシュされるのか，あるいはデータセットの現在レコードが変更されるたびに参照項目の値が動的に参照されるのかを決めます。LookupDataSet があまり変更されず，参照値の数が少ない場合に参照項目の値をキャッシュするには，LookupCache を **true** に設定します。DataSet を開いたときに LookupKeyFields 値セットに参照値は事前に読み込まれるので，参照値のキャッシングにより処理効率を高速化できます。DataSet の現在レコードが変化すると，項目オブジェクトは LookupDataSet にアクセスせずに，キャッシュでその Value を探すことができます。この処理効率の向上は，LookupDataSet が低速ネットワーク上にあるときに特に顕著です。

ヒント 参照キャッシュを使用すると，二次データセットからではなく，プログラミングによって参照値を提供できます。LookupDataSet プロパティが NULL であることを確認してください。そして LookupList プロパティの Add メソッドを使用して，参照値を入力します。LookupCache プロパティを true に設定します。項目は参照リストを使用しますが，そのとき参照データセットの値で上書きされません。

DataSet の各レコードで KeyFields の値が異なる場合，値をキャッシュで検索することのオーバーヘッドは，キャッシュによって得られるどの処理効率メリットよりも大きくなる可能性があります。値をキャッシュで検索することのオーバーヘッドは，KeyFields で受け取ることのできる値の数が多くなるほど大きくなります。

LookupDataSet が揮発性の場合，参照値のキャッシングにより不正確な結果が生じることがあります。参照キャッシュ内の値を更新するには，RefreshLookupList を呼び出します。RefreshLookupList は LookupList プロパティを生成し直します。LookupList プロパティには，各 LookupKeyFields 値集合の LookupResultField の値が入っています。

実行時に LookupCache を設定する場合は，RefreshLookupList を呼び出してキャッシュを初期化します。

集合項目の定義

集合項目は，クライアントデータセット内で保守される集合体を表示します。集合体は，レコードセットのデータの合計を計算します。保守される集合体の詳細については，27-11 ページの「保守される集合体の使用」を参照してください。

［項目の新規作成］ダイアログボックスで，集合項目を作成する方法は次のとおりです。

1. ［名前］編集ボックスに，データセットの項目名を入力します。既存の項目名は入力しないでください。
2. ［型］コンボボックスから，項目には集合データ型を設定します。
3. ［項目の種類］ラジオグループの［集合］を選択します。
4. ［OK］を選択します。新しく定義した集合項目がクライアントデータセットに自動的に追加され，データセットの Aggregates プロパティが適切な集合値で自動的に更新されます。

5. 集合の計算を新しく作成した集合項目の ExprText プロパティで指定します。総計の定義についての詳細は，27-11 ページの「集合体を指定する」を参照してください。

一度持続的な TAggregateField が作成されると，TDBText コントロールを集合項目に結合できます。すると TDBText コントロールは，基になるクライアントデータセットの現在レコードに対応した集合項目値を表示するようになります。

持続的項目コンポーネントの削除

持続的項目コンポーネントの削除は，テーブル内の利用可能な列の一部にアクセスするときや，独自の持続的項目を定義してテーブル中の列を置き換えるときに役立ちます。データセットの 1 つまたは複数の持続的項目コンポーネントを削除する手順は，次のとおりです。

1. 項目エディタのリストボックスで，削除する項目を選択します。
2. 〔Del〕を押します。

メモ コンテキストメニューを呼び出して［削除］を選択しても，目的の項目を削除できます。

削除した項目はデータセットでは利用できなくなり，データベース対応コントロールで表示できません。間違って削除した持続的項目コンポーネントはいつでも作成し直すことができますが，そのプロパティまたはイベントに対するそれまでの変更はすべて失われます。詳細については，23-3 ページの「持続的項目の作成」を参照してください。

メモ データセットのすべての持続的項目コンポーネントを削除すると，データセットの基になっているデータベーステーブルの各列に対して，動的項目コンポーネントが自動生成されます。

持続的項目のプロパティとイベントを設定する

設計時には，持続的項目コンポーネントのプロパティを設定したり，イベントをカスタマイズできます。プロパティは，データベース対応コンポーネントでの項目の表示方法を制御します。たとえば，その項目を TDBGrid に表示可能か，項目値を変更可能かを制御します。イベントは，項目内のデータの取り出し，変更，設定，検証時の動作を制御します。

項目コンポーネントのプロパティを設定したり，ユーザー定義のイベントハンドラを記述するには，項目エディタでコンポーネントを選択するか，オブジェクトインスペクタのコンポーネントリストで選択します。

設計時に表示プロパティと編集プロパティを設定する

選択した項目コンポーネントの表示プロパティを編集するには，オブジェクトインスペクタウィンドウの［プロパティ］ページに切り替えます。次の表に，編集可能な表示プロパティを示します。

表 23.3 項目コンポーネントのプロパティ

プロパティ	目的
Alignment	データベース対応コンポーネント内の項目内容を左揃え，右揃え，中央揃えにする
ConstraintErrorMessage	制約条件によって編集がクラッシュしたときに表示するテキストを指定する
CustomConstraint	編集中にデータに適用するローカルな制約を指定する

表 23.3 項目コンポーネントのプロパティ（つづき）

プロパティ	目的
Currency	**true** の場合，金額として表示される **false** の場合（デフォルト），金額として表示しない
DisplayFormat	データベース対応コンポーネントでのデータの表示形式を指定する
DisplayLabel	データベース対応グリッドコンポーネントでの項目の列名を指定する
DisplayWidth	項目を表示するグリッド列の幅を文字数で指定する
EditFormat	データベース対応コンポーネントでのデータの編集形式を指定する
EditMask	編集可能項目のデータ入力を指定された型と文字範囲に限定し，項目内に表示される特殊な編集不可能文字（ハイフン，カッコなど）を指定する
FieldKind	作成する項目の種類を指定する
FieldName	項目の値とデータ型の取得元になるテーブルの列の実際の名前を指定する
HasConstraints	項目に課せられた制約条件があるかどうかを示す
ImportedConstraint	データディクショナリか，または SQL サーバーからインポートされた SQL 制約を指定する
Index	データセット内の項目順序を指定する
LookupDataSet	Lookup が **true** のときに，項目値を参照するために使用するテーブルを指定する
LookupKeyFields	参照時に照合する参照データセット内の項目を指定する
LookupResultField	参照データセット内の値をコピーする項目を指定する
MaxValue	数値項目のみ。項目に入力可能な最大値を指定する
MinValue	数値項目のみ。項目に入力可能な最小値を指定する
Name	C++Builder 内の項目コンポーネントの名前を指定する
Origin	基となるデータベースに表示される項目名を指定する
Precision	数値項目のみ。有効桁数を指定する
ReadOnly	**true** の場合，データベース対応コントロール内に項目値は表示されるが，編集できない **false** の場合（デフォルト），項目値の表示と編集を許可する
Size	文字列項目への表示や入力が可能な最大文字数，あるいは TBytesField 項目や TVarBytesField 項目のサイズをバイト数で指定する
Tag	プログラマが必要であればすべてのコンポーネントで使える 32 ビット整数型の項目
Transliterate	**true** の場合（デフォルト），データセットとデータベースとの間でデータが転送されるたびに，各ローカル間での変換が発生する **false** の場合，ロケール変換は発生しない
Visible	**true** の場合（デフォルト），データベース対応グリッド内で項目表示を可能にする **false** の場合，データベース対応グリッドコンポーネント内で項目表示を無効にする ユーザー定義のコンポーネントでは，このプロパティに基づいて表示するかどうかを決定する

すべての項目コンポーネントで，すべてのプロパティが使えるとは限りません。たとえば，TStringField 型の項目コンポーネントには，Currency，MaxValue，DisplayFormat のプロパティはありません。また TFloatField 型のコンポーネントには Size プロパティはありません。

ほとんどのプロパティは設定がそのまま表示に反映されますが，Calculated など一部のプロパティには追加のプログラミングが必要です。また，DisplayFormat，EditFormat，EditMask のように相互関係のあるプロパティでは，設定を調整しなければなりません。DisplayFormat，EditFormat，EditMask の使い方については，23-13 ページの「ユーザー入力の制御とマスク」を参照してください。

実行時の項目コンポーネントプロパティの設定

実行時に項目コンポーネントのプロパティを使用したり，操作できます。持続的項目コンポーネントは，データセット名と項目名を連結した名前を使ってアクセスします。

たとえば次のコードは，Customers テーブルの CityStateZip 項目の ReadOnly プロパティを **true** に設定します。

```
CustomersCityStateZip->ReadOnly = true;
```

また次の文は，Customers テーブルの CityStateZip 項目の Index プロパティを 3 に設定することで，項目順を変更します。

```
CustomersCityStateZip->Index = 3;
```

項目コンポーネントの属性セットの作成

アプリケーションで使用するデータセット内の各項目が，同じ形式のプロパティ（Alignment，DisplayWidth，DisplayFormat，EditFormat，MaxValue，MinValue など）を持っている場合は，まず 1 つの項目に対してプロパティを設定し，そのプロパティを属性セットとしてデータディクショナリに保存すると便利です。データディクショナリに保存された属性セットはほかの項目に簡単に適用できます。

メモ 属性セットとデータディクショナリは，BDE 対応のデータセットのみで使用できます。

データセットの項目コンポーネントに基づいて属性セットを作成する手順は次のとおりです。

1. データセットをダブルクリックして項目エディタを呼び出します。
2. プロパティの設定対象になる項目を選択します。
3. オブジェクトインスペクタで項目の希望のプロパティを設定します。
4. 項目エディタのリストボックスを右クリックしてコンテキストメニューを呼び出します。
5. ［属性の上書き保存］を選択して現在の項目のプロパティ設定を属性セットとしてデータディクショナリに保存します。

属性セットの名前のデフォルトは現在の項目の名前です。別の名前を指定するには，コンテキストメニューの［属性の上書き保存］ではなく，［属性に名前を付けて保存］を選択してください。

新規に作成してデータディクショナリに追加した属性セットは，ほかの持続的項目コンポーネントに関連付けできます。後でその関連付けを解除したとしても，データディクショナリに定義されている属性セットは削除されません。

メモ SQL エクスプローラから属性セットを直接作成することもできます。SQL エクスプローラを使って作成した属性セットは，データディクショナリには追加されますが，項目には適用されません。SQL エクスプローラでは，項目型（TFloatField，TStringField など）およびデータベース対応コントロール（TDBEdit，TDBCheckBox）の 2 つの属性も指定できます。後者の属性は，この属性セットに基づいた項目がフォーム上にドラッグされたときに，自動的にフォーム上に追加されます。詳細は SQL エクスプローラのオンラインヘルプを参照してください。

属性セットと項目コンポーネントの関連付け

アプリケーションで使用するデータセット内の各項目が，同じ形式のプロパティ（Alignment，DisplayWidth，DisplayFormat，EditFormat，MaxValue，MinValue など）を持っていて，それらのプ

ロパティを属性セットとしてデータディクショナリに保存した場合，項目ごとに手作業でプロパティを設定しなくても，保存した属性セットを各項目に対して簡単に適用できます。さらに，データディクショナリの属性の設定を後で変更した場合，次に項目コンポーネントがデータセットに追加されたときにその変更は属性セットに関連するすべての項目に対して自動的に適用されます。

属性セットを項目コンポーネントに適用する手順は次のとおりです。

1. データセットをダブルクリックして項目エディタを呼び出します。
2. 属性セットの適用対象となる項目を選択します。
3. コンテキストメニューを呼び出して［属性の関連付け］を選択します。
4. ［属性の関連付け］ダイアログボックスを使って，適用する属性セットを選択するか入力します。データディクショナリに現在の項目と同じ名前の属性セットがある場合，その集合の名前が編集ボックスに表示されます。

重要 データディクショナリの属性セットが後日変更された場合は，それを使う各項目コンポーネントにその属性セットを再度適用しなければなりません。項目エディタを呼び出して属性を再度適用するときには，データセット内の項目コンポーネントを複数選択することができます。

属性の関連付けの解除

属性セットと項目の関連付けを取り消す場合は，次の手順に従って関連付けを解除できます。

1. 対象項目を持つデータセットで項目エディタを呼び出します。
2. 属性の関連付けの解除対象になる項目を 1 つまたは複数選択します。
3. 項目エディタのコンテキストメニューを呼び出して［属性の関連付けを取り消す］を選択します。

重要 属性セットの関連付けを解除しても項目プロパティは変更されません。項目は属性セット適用時の設定を保持しています。これらのプロパティを変更するには，項目エディタで項目を選択してそのプロパティをオブジェクトインスペクタで設定します。

ユーザー入力の制御とマスク

EditMask プロパティを設定すると，TStringField，TDateField，TTimeField，TDateTimeField，TSQLTimeStampField の各コンポーネントに関連するデータベース対応コンポーネントへユーザーが入力できる値の型と範囲を制御できるようになります。既存のマスクを使用したり，独自のマスクを作成できます。編集マスクを使用，作成する一番簡単な方法は，入力マスクエディタを使うことです。またオブジェクトインスペクタで EditMask 項目に直接マスクを入力できます。

メモ TStringField コンポーネントでは，EditMask プロパティがそのまま表示形式になります。

項目コンポーネントの入力マスクエディタを呼び出す手順は，次のとおりです。

1. 項目エディタまたはオブジェクトインスペクタで，コンポーネントを選択します。
2. オブジェクトインスペクタの［プロパティ］ページをクリックします。
3. オブジェクトインスペクタの EditMask プロパティの値列をダブルクリックするか，省略記号のボタンをクリックします。入力マスクエディタを開きます。

［入力マスク］編集ボックスでは，マスク形式を作成，編集できます。［例］リストボックスでは，定義済みマスクが選択できます。サンプルマスクを選択すると［入力マスク］編集ボックスにそのマス

ク形式が表示され，そこで形式を変更したり，そのまま使用できます。［テスト］編集ボックスでは，マスクについて許されるユーザー入力をテストできます。

［マスク］ボタンを使うと，マスクのカスタムセット（作成してある場合）を［例］リストボックスに読み込んで選択できるようにします。

数値，日付，時刻型の項目にデフォルトの形式を使用する

C++Builder に組み込まれている表示と編集の形式を設定するルーチンは，TFloatField，TCurrencyField，TIntegerField，TSmallIntField，TWordField，TDateField，TDateTimeField，TTimeField の各コンポーネントにデフォルトの形式を与えるインテリジェント機能を備えています。これらのルーチンを使うために必要な操作はありません。

デフォルトの形式は，次のルーチンで設定できます。

表 23.4　項目コンポーネントの形式設定ルーチン

ルーチン	対象
FormatFloat	TFloatField，TCurrencyField
FormatDateTime	TDateField，TTimeField，TDateTimeField
SQLTimeStampToString	TSQLTimeStampField
FormatCurr	TCurrencyField，TBCDField
BcdToStrF	TFMTBCDField

各項目コンポーネントでは，そのコンポーネントのデータ型に適合した形式プロパティしか使用できません。

日付，時刻，通貨，数値についてデフォルトの形式を設定するときの規則はコントロールパネルにある［地域］の設定に基づきます。たとえば米国のデフォルト設定では，TFloatField の Currency プロパティを **true** に設定すると，1234.56 の DisplayFormat は $1234.56 となり，EditFormat は 1234.56 となります。

設計時または実行時には，項目コンポーネントの DisplayFormat プロパティと EditFormat プロパティを編集して，項目のデフォルトの表示設定を変更できます。また OnGetText および OnSetText イベントハンドラを記述して，実行時に項目コンポーネントの形式をカスタマイズできます。

イベントの処理

ほかのコンポーネントと同様，項目コンポーネントにも関連付けられたイベントがあります。これらのイベントのハンドラとして，メソッドを割り当てることができます。ハンドラを記述すると，データベース対応コントロールを通して項目に入力されたデータに影響するイベントの発生に反応して，独自の動作を実行できます。次の表に，項目コンポーネントに関連するイベントの一覧を示します。

表 23.5　項目コンポーネントのイベント

イベント	目的
OnChange	項目値が変更されると呼び出される
OnGetText	表示や編集用に項目コンポーネントの値を取得すると呼び出される
OnSetText	項目コンポーネントの値を設定すると呼び出される
OnValidate	編集や挿入により値が変更されると，項目コンポーネントの値を検証するために呼び出される

OnGetText イベントや OnSetText イベントは主に，組み込み形式機能を越えるユーザー定義の形式設定を実行するときに便利です。OnChange イベントは，メニューやビジュアルコントロールの使用可と使用不可の切り替えなど，データの変更に関連したアプリケーション固有のタスクを実行するのに使います。OnValidate イベントは，データベースサーバーに値を返す前にアプリケーションでデータ入力の検証を制御するときに使います。

項目コンポーネントのイベントハンドラを書く手順は，次のとおりです。

1. コンポーネントを選択します。
2. オブジェクトインスペクタの［イベント］ページを選択します。
3. イベントハンドラの値列をダブルクリックして，ソースコードウィンドウを表示します。
4. ハンドラのコードを作成，編集します。

実行時に項目コンポーネントのメソッドを操作する

実行時に使える項目コンポーネントのメソッドは，項目の値のデータ型を変換する場合や，項目コンポーネントに関連付けられたフォーム上の最初のデータベース対応コントロールにフォーカスを設定する場合に使います。

アプリケーションが BeforePost などのデータセットイベントハンドラでレコード指向のデータ検証を実行する場合，項目に関連付けられたデータベース対応コンポーネントのフォーカスを制御することは重要です。関連付けられたデータベース対応コントロールにフォーカスがあるかどうかに関係なく，レコード内の項目を検証できます。レコード内のある特定の項目の検証に失敗した場合，ユーザーが誤りを修正できるように，違反データを含むデータベース対応コントロールにフォーカスを移動させたい場合があります。

項目の FocusControl メソッドを使ってその項目のデータベース対応コンポーネントに対するフォーカスを制御します。FocusControl メソッドは，項目に関連付けられたフォーム上の最初のデータベース対応コントロールにフォーカスを設定します。項目を検証する前に，イベントハンドラは項目の FocusControl メソッドを呼び出す必要があります。次のコードは，Customers テーブルの Company 項目に対して FocusControl メソッドを呼び出す方法を説明しています。

```
CustomersCompany->FocusControl();
```

次の表では，項目コンポーネントのメソッドの一部とその用途を一覧表示します。メソッドの完全なリストと各メソッドの使い方についての詳細は，オンラインの VCL リファレンスで TField とその下位コンポーネントの項を参照してください。

表 23.6　項目コンポーネントのメソッド

メソッド	目的
AssignValue	項目型に基づいた自動変換関数を使用して，項目値を指定値に設定する
Clear	項目をクリアしてその値を NULL に設定する
GetData	項目から形式設定されていないデータを取得する
IsValidChar	データベース対応のコントロールにユーザーが入力した文字が，この項目に対して有効かどうか判断する
SetData	形式設定されていないデータをこの項目に割り当てる

項目値の表示，変換，アクセス

TDBEdit や TDBGrid などデータベース対応コントロールは，項目コンポーネントに関連付けられた値を自動的に表示します。データセットとコントロールで編集が可能であれば，データベース対応コントロールはデータベースに新しい値や変更した値を送ることもできます。一般にデータベース対応コントロールの組み込みプロパティやメソッドは，プログラムを追加作成しないでもデータセットへ接続したり，値を表示したり，更新できます。データベースアプリケーションで使えるときはいつでも，これらのプロパティやメソッドを使ってください。データベース対応コントロールについての詳細は，第 19 章「データコントロールの使い方」を参照してください。

標準のコントロールは，項目コンポーネントに関連付けられたデータベース値の表示や編集ができます。ただし標準コントロールを使用するには，追加のプログラミング作業が必要なことがあります。たとえば標準コントロールを使用すると，アプリケーションには項目値が変更された場合に，いつコントロールを更新するかを追跡する責任が生まれます。データセットにデータソースコンポーネントがあれば，そのイベントを利用できます。とりわけ OnDataChange イベントでは，コントロール値を更新する時期が分かり，OnStateChange イベントはコントロールを有効 / 無効にする時期を決定するのに役立ちます。これらのイベントの詳細については，19-4 ページの「データソースに仲介された変更に応答する」を参照してください。

次のトピックでは，標準コントロールに表示できるように項目値を処理する方法を説明します。

標準のコントロールを使用した項目コンポーネント値の表示

アプリケーションは，項目コンポーネントの Value プロパティを使ってデータベースの列値にアクセスできます。たとえば次の OnDataChange イベントハンドラは，CustomersCompany 項目値が変更されると TEdit コントロール内のテキストを更新します。

```
void __fastcall TForm1::Table1DataChange(TObject *Sender, TField *Field)
{
  Edit3->Text = CustomersCompany->Value;
}
```

このメソッドは文字列値に対しては正しく機能しますが，ほかのデータ型では変換を処理するために追加のプログラミング作業が必要なことがあります。項目コンポーネントには，このような変換を処理するプロパティが組み込まれています。

メモ 項目値へのアクセスとその設定には，バリアントも使用できます。バリアントを使った項目値へのアクセスとその設定方法についての詳細は，23-18 ページの「デフォルトのデータセットプロパティによる項目値へのアクセス」を参照してください。

項目値の変換

変換プロパティは，データ型を変換しようと試みます。たとえば AsString プロパティは，数値と論理値を文字列表現に変換します。次の表に，項目コンポーネントの変換プロパティの一覧と，項目コンポーネントのクラス別に推奨されるプロパティを示します。

	AsVariant	AsString	AsInteger	AsFloat AsCurrency AsBCD	AsDateTime AsSQLTimeStamp	AsBoolean
TStringField	可	不可	可	可	可	可
TWideStringField	可	可	可	可	可	可
TIntegerField	可	可	不可	可		
TSmallIntField	可	可	可	可		
TWordField	可	可	可	可		
TLargeintField	可	可	可	可		
TFloatField	可	可	可	可		
TCurrencyField	可	可	可	可		
TBCDField	可	可	可	可		
TFMTBCDField	可	可	可	可		
TDateTimeField	可	可		可	可	
TDateField	可	可		可	可	
TTimeField	可	可		可	可	
TSQLTimeStampField	可	可		可	可	
TBooleanField	可	可				
TBytesField	可	可				
TVarBytesField	可	可				
TBlobField	可	可				
TMemoField	可	可				
TGraphicField	可	可				
TVariantField	不可	可	可	可	可	可
TAggregateField	可	可				

表の一部の列は，複数の変換プロパティに関連しています（AsFloat，AsCurrency，および AsBCD）。それは，これらのプロパティの 1 つをサポートしている項目データ型は，他のプロパティもサポートしているためです。

AsVariant プロパティは，全データ型を変換できます。上の表に一覧表示されていないデータ型の内，AsVariant も利用できます（これ以外にはなし）。不安があるときは AsVariant を使うといいでしょう。

変換ができない場合もあります。たとえば AsDateTime を使用して文字列を日付，時刻，日付 / 時刻形式に変換できるのは，文字列値が認識できる日付 / 時刻形式になっている場合だけです。変換が失敗すると，例外が生成されます。

また，常に希望する変換結果が得られるとは限りません。たとえば，TDateTimeField の値を AsFloat で浮動小数点形式に変換する場合は，項目の日付の部分は 1899 年 12 月 31 日から数えた日数に，また時刻の部分は 24 時間を単位とする小数に変換されます。表 23.7 は，特殊な結果を生じる許容変換を示しています。

表 23.7　特殊な変換結果

変換	結果
文字列から論理値	「True」,「False」,「Yes」,「No」を論理値に変換する。それ以外の値では，例外が生成される
浮動小数点形式から整数	浮動小数点値を最も近い整数値に丸める
日付 / 時刻または SQLTimeStamp から浮動小数点値	日付を 1899 年 12 月 31 日から数えた日数に変換し，時間を 24 時間を単位とする分数に変換する
論理値から文字列	論理値を「True」または「False」に変換する

これら以外の変換はまったくできません。そのような変換を試みると，やはり例外が生成されます。

変換が行われるのは，常に代入が実行される前です。たとえば次の文は，CustomersCustNo 項目の値を文字列に変換し，その文字列を編集コントロールのテキストに代入します。

```
Edit1->Text = CustomersCustNo->AsString;
```

逆に次の文は，編集コントロールのテキストを整数として CustomersCustNo 項目に代入します。

```
MyTableMyField->AsInteger = StrToInt(Edit1->Text);
```

デフォルトのデータセットプロパティによる項目値へのアクセス

項目値にアクセスする一般的な方法として，Variants と FieldValues プロパティの使用があります。たとえば次の文は，編集ボックスの値を Customers テーブルの CustNo 項目に入力します。

```
Customers->FieldValues["CustNo"] = Edit2->Text;
```

FieldValues プロパティは Variant 型なので，自動的にほかのデータ型を Variant 値に変換します。

バリアントについての詳細は，オンラインヘルプを参照してください。

データセットの Fields プロパティによる項目値へのアクセス

項目が属しているデータセットコンポーネントの Fields プロパティを使用すると，その項目値にアクセスできます。Fields は，データセット内の全項目リストをインデックス付きで保持しています。Fields プロパティを使用して項目値にアクセスするのが役立つのは，多数の列の繰り返し処理や，アプリケーションが設計時に使えないテーブル操作を行う場合です。

Fields プロパティを使用するには，データセット内の項目の順序とデータ型を知る必要があります。アクセスする項目の指定には，順序数を使います。データセットの最初の項目には 0 という番号が付きます。項目値は，必要であればそれぞれの項目コンポーネントの変換プロパティを使って変換しなければなりません。項目コンポーネントの変換プロパティについて詳しくは，23-16 ページの「項目値の変換」を参照してください。

たとえば次の文は，Customers テーブルの 7 番目の列（Country）の現在値を，編集コントロールに代入します。

```
Edit1->Text = CustTable->Fields->Fields[6]->AsString;
```

逆に，データセットの Fields プロパティを編集する項目に設定すると，その項目に値を代入できます。例を示します。

```
Customers->Edit();
Customers->Insert();
Customers->Fields->Fields[6]->AsString = Edit1->Text;
Customers->Post();
```

データセットの FieldByName メソッドによる項目値へのアクセス

また項目値のアクセスには，データセットの FieldByName メソッドも使用できます。このメソッドは，アクセスしたい項目名はわかっているが，設計時に基になるテーブルにアクセスできない場合に便利です。

FieldByName メソッドを使うには，アクセスする項目の名前とデータセットがわかっていなければなりません。項目名を引数としてメソッドへ渡します。項目値へアクセスしたり，項目値を変更するには，AsString や AsInteger など適切な項目コンポーネント変換プロパティを使用して，結果を変換します。たとえば次の文は，Customers データセットの CustNo 項目値を編集コントロールに代入します。

```
Edit2->Text = Customers->FieldByName("CustNo")->AsString;
```

逆に次のように，値を項目に代入できます。

```
Customers->Edit();
Customers->FieldByName("CustNo")->AsString = Edit2->Text;
Customers->Post();
```

項目のデフォルト値の設定

DefaultExpression プロパティを使用すると，実行時にクライアントデータセットや BDE 対応データセットで項目のデフォルト値を計算する方法を指定できます。DefaultExpression は，項目値を参照しない有効な SQL 値式ならどんな式も可能です。式に数値以外のリテラルが含まれる場合は，引用符で囲まなければなりません。たとえば，時間項目の正午のデフォルト値は，

```
'12:00:00'
```

のように，リテラル値を引用符で囲みます。

メモ 基になっているデータベーステーブルで項目にデフォルト値が定義されている場合，DefaultExpression で指定したデフォルトが優先します。それは，データセットが項目を含むレコードの登録を行うときに，DefaultExpression が適用されてから，その編集レコードがデータベースサーバーに適用されるためです。

制約の操作

クライアントデータセットや BDE 対応データセットの項目コンポーネントでは，SQL サーバー制約を使用できます。また，アプリケーションにローカルなカスタム制約を作成し，これらのデータセットに対して使用することもできます。すべての制約は，項目が格納できる値のスコープまたは範囲に制限を課す条件または規則です。

カスタム制約の作成

カスタム制約はほかの制約のようにサーバーからインポートされる制約ではなく，ローカルアプリケーション内で宣言，実装，実施を行う制約です。そういうものとして，カスタム制約はデータ入力の事前検証の実施を提供することで役立ちますが，サーバーアプリケーションから受け取ったデータやサーバーアプリケーションへ設定されたデータに対してはカスタム制約を適用できません。

カスタム制約を作成するには，制約条件を指定する CustomConstraint プロパティを設定し，実行時にユーザーが制約に違反した場合に表示するメッセージを ConstraintErrorMessage に設定します。

CustomConstraint は，項目値に課されるアプリケーション固有の制約を指定する SQL 文字列です。ユーザーが項目に入力できる値を制限するため，CustomConstraint を設定します。CustomConstraint は，次のように有効な SQL 検索式にします。

```
x > 0 and x < 100
```

項目の値を参照するのに使われる名前は，制約式を通して一貫して使われる限り，予約済みの SQL キーワードでない文字列ならどんな文字列にもできます。

メモ カスタム制約は，BDE 対応データセットとクライアントデータセットのみで使用できます。

カスタム制約は項目値に対して課されるサーバーからの制約以外の制約です。サーバーが課す制約を確認するには，ImportedConstraint プロパティを見ます。

サーバー制約の使い方

実際の SQL データベースはほとんどが制約を使って項目の可能な値に条件を課します。たとえば NULL 値を許容しない項目，ユニークな列値を必要とする項目，あるいは値が 0 より大きく 150 未満でなければならない項目などがあります。このような条件はクライアントアプリケーションにコピーすることもできますが，クライアントデータセットや BDE 対応データセットでは，ImportedConstraint プロパティを使ってサーバーの制約をローカルに利用することができます。

ImportedConstraint プロパティには，項目値を何らかの方法で制限する SQL 節を指定します。例を示します。

```
Value > 0 and Value < 100
```

コメントとしてインポートされた標準外の SQL またはサーバー固有の SQL を，データベースエンジンでは解釈できないので編集する場合を除き，ImportedConstraint の値を変更しないでください。

項目値にほかにも制約を追加するには，CustomConstraint を使います。カスタム制約はインポートされた制約以外に課される制約です。サーバー制約が変化した場合，ImportedConstraint の値は変化しますが，CustomConstraint プロパティに導入された制約は持続します。

ImportedConstraint プロパティから制約を削除した場合，その制約に違反する項目値の妥当性が変わることはありません。制約を削除した結果，その制約はローカルにではなくサーバーによって検査されるようになります。制約をローカルに検査した場合は，違反が見つかったときに表示されるのは ConstraintErrorMessage プロパティで与えたエラーメッセージであって，サーバーからのエラーメッセージではありません。

オブジェクト項目の使い方

オブジェクト項目とは，ほかの単純なデータ型の混合した項目です。これには，ADT（抽象データ型）項目，配列項目，データセット項目，および 参照項目があります。これらの項目型は，子項目やほかのデータセットを含むことも，参照することもあります。

ADT 項目および配列項目は，子項目を持つ項目です。ADT 項目の子項目は，スカラー型またはオブジェクト型です（つまり任意のほかの項目型）。子項目の型は相互に違っていても構いません。配列項目は，同じ型の子項目の配列です。

データセット項目および参照項目は，他のデータセットにアクセスする項目です。データセット項目はネストされた（詳細）データセットへのアクセスを提供し，参照項目はほかの持続的オブジェクト（ADT）へのポインタ（参照）を格納します。

表 23.8　オブジェクト項目コンポーネントの種類

コンポーネント名	目的
TADTField	ADT（抽象データ型）項目を表す
TArrayField	配列項目を表す
TDataSetField	ネストされたデータセットへの参照を含む項目を表す
TReferenceField	ADT へのポインタである REF 項目を表す

項目エディタを使ってオブジェクト項目を含むデータセットに項目を追加すると，型の正しい持続的オブジェクト項目が自動的に作成されます。持続的オブジェクト項目をデータセットに追加すると，自動的にデータセットの ObjectView プロパティが **true** に設定されます。その場合，構成要素の子項目が独立した別の項目であるかのように，項目は平面的ではなく階層的に格納されます。

次のプロパティはすべてのオブジェクト項目に共通し，子項目およびデータセットを処理する機能を提供します。

表 23.9　共通するオブジェクト項目の下位項目プロパティ

プロパティ	目的
Fields	オブジェクト項目に属する子項目を含む
ObjectType	オブジェクト項目を分類する
FieldCount	オブジェクト項目に属す下位項目の数
FieldValues	子項目値へのアクセスを提供する

ADT 項目および配列項目を表示する

ADT 項目と配列項目は，どちらもデータベース対応コントロールを通して表示できる子項目を含みます。

1 つの項目値を表す TDBEdit などデータベース対応コントロールは，子項目値を編集不能のカンマ区切り文字列で表示します。さらに，コントロールの DataField プロパティにオブジェクト項目自身ではなく子項目を設定すると，子項目はほかの通常のデータ項目と同様に表示や編集ができます。

TDBGrid コントロールが ADT 項目と配列項目のデータを表示する方法は，データセットの ObjectView プロパティの値によって変わります。ObjectView が **false** の場合，各子項目は 1 つの列の中に表示されます。ObjectView が **true** の場合，ADT 項目または配列項目は列のタイトルバー内の矢印をクリックすることによって，拡大したり，折りたたんだりできます。項目が拡大されると，各子項目はそれ自身の列とタイトルバーの中に表示されます。ADT または配列が折りたたまれると，その子項目を含む編集不可能なカンマ区切りの文字列の 1 列だけ表示されます。

ADT 項目を操作する

ADT はサーバー上に作成されるユーザー定義済みの項目型で，構造体と類似しています。ADT はほとんど大半のスカラー項目型，配列項目，参照項目，ネストされた ADT を含むことができます。

ADT 項目型のデータにはさまざまな方法でアクセスできますが，次の例では，CityEdit という編集ボックスに子項目値を割り当てて，次の ADT 構造体を使用します。

```
Address
  Street
  City
  State
  Zip
```

持続的項目コンポーネントの使用

ADT 項目値にアクセスする簡単な方法は，持続的項目コンポーネントを使用することです。上の ADT 構造体では，項目エディタを使用して次の持続的項目を Customer テーブルに追加できます。

```
CustomerAddress: TADTField;
CustomerAddrStreet: TStringField;
CustomerAddrCity: TStringField;
CustomerAddrState: TStringField;
CustomerAddrZip: TStringField;
```

持続的項目がある場合，名前を指定すれば ADT 項目の子項目にアクセスできます。

```
CityEdit->Text = CustomerAddrCity->AsString;
```

持続的項目は ADT 子項目にアクセスする簡単な方法ですが，設計時にデータセットの構造が不明な場合は使用できません。持続的項目を使用しないで ADT 子項目にアクセスする場合，データセットの ObjectView プロパティを **true** に設定する必要があります。

データセットの FieldByName メソッドを使用する

子項目の名前を ADT 項目名で修飾することにより，データセットの FieldByName メソッドを使用して ADT 項目の子項目にアクセスできます。

```
CityEdit->Text = Customer->FieldByName("Address.City")->AsString;
```

データセットの FieldValues プロパティを使用する

またデータセットの FieldValues プロパティで修飾された項目名も使用できます。

```
CityEdit->Text = Customer->FieldValues["Address.City"];
```

メモ ADT 子項目値にアクセスするほかのランタイムメソッドと違い，データセットの ObjectView プロパティが **false** でも，FieldValues プロパティは機能します。

ADT 項目の FieldValues プロパティを使用する

子項目値には TADTField の FieldValues プロパティを使ってアクセスできます。FieldValues は Variant を受け入れて返すため，あらゆる種類の項目を処理し，変換できます。インデックスパラメータは，項目のオフセットを指定する整数値です。

```
CityEdit->Text = ((TADTField*)Customer->FieldByName("Address"))->FieldValues[1];
```

ADT 項目の Fields プロパティを使用する

ADT 項目には Fields プロパティがあり，これはデータセットの Fields プロパティと類似しています。データセットの Fields プロパティと同様，それと位置を指定して子項目にアクセスできます。

```
CityEdit->Text = ((TADTField*)Customer->FieldByName("Address"))->Fields->
Fields[1]->AsString;
```

また名前を指定してもアクセスできます。

```
CityEdit->Text = ((TADTField*)Customer->FieldByName("Address"))->
Fields->FieldByName("City")->AsString;
```

配列項目を操作する

配列項目は，同じ型の項目セットで構成されています。項目型はスカラー（浮動小数点，文字列など）であっても，また非スカラー（ADT）であってもかまいませんが，配列の配列項目は許されていません。TDataSet の SparseArrays プロパティは，配列項目の各要素にユニークな TField オブジェクトを作成するかどうかを指定します。

配列項目型のデータには，さまざまな方法でアクセスできます。持続的項目を使用しない場合，データセットの ObjectView プロパティを **true** に設定しないと，配列項目の要素にアクセスできません。

持続的項目の使用

持続的項目と配列項目の個々の配列要素を対応させることができます。たとえば TelNos_Array という要素が 6 つある文字列配列を考えてみましょう。Customer テーブルコンポーネント用に作成された次の持続的項目は，TelNos_Array 項目および 6 つの要素を表します。

```
CustomerTELNOS_ARRAY: TArrayField;
CustomerTELNOS_ARRAY0: TStringField;
CustomerTELNOS_ARRAY1: TStringField;
CustomerTELNOS_ARRAY2: TStringField;
CustomerTELNOS_ARRAY3: TStringField;
CustomerTELNOS_ARRAY4: TStringField;
CustomerTELNOS_ARRAY5: TStringField;
```

持続的項目が指定されると，次のコードは持続的項目を使用して配列要素値を TelEdit という編集ボックスに入力します。

```
TelEdit->Text = CustomerTELNOS_ARRAY0->AsString;
```

配列項目の FieldValues プロパティを使用する

子項目値には，配列項目の FieldValues プロパティを使ってアクセスできます。FieldValues は Variant を受け入れて返すため，あらゆる種類の項目を処理し，変換できます。例を示します。

```
TelEdit->Text = ((TArrayField*)Customer->FieldByName("TelNos_Array"))->FieldValues[1];
```

配列項目の Fields プロパティを使用する

TArrayField には，個々のサブ項目へのアクセスに利用できる Fields プロパティがあります。これは以下に説明されています。ここで配列項目（OrderDates）は，ヌルでない配列要素のあるリストボックスの表示に使用されます。

```
for (int i = 0; i < OrderDates->Size; ++i)
  if (!OrderDates->Fields->Fields[i]->IsNull)
    OrderDateListBox->Items->Add(OrderDates->Fields->Fields[i]->AsString);
```

データセット項目を操作する

データセット項目は，ネストされたデータセット内に格納されたデータへのアクセスを提供します。NestedDataSet プロパティは，ネストされたデータセットを参照します。これでネストされたデータセットの中のデータは，ネストされたデータセットの項目オブジェクトを通してアクセスできます。

データセット項目の表示

TDBGrid コントロールを使用すると，データセット項目に格納されているデータを表示できます。TDBGrid コントロールでは，データセット項目が文字列「(DataSet)」とともにデータセット列の各セルに表示され，実行時には省略記号ボタンがその右側に表示されます。省略記号をクリックすると，現在レコードのデータセット項目に対応するデータセットを表示するグリッドを持つ新しいフォームが表示されます。またこのフォームは，DB グリッドの ShowPopupEditor メソッドを使用して，プログラミングによって表示できます。たとえば，グリッドの中の 7 番目の列がデータセット項目を表す場合，次のコードによって現在のレコードのその項目に対応するデータセットが表示されます。

```
DBGrid1->ShowPopupEditor(DBGrid1->Columns->Items[7], -1, -1);
```

ネストされたデータセット内のデータにアクセスする

データセット項目は通常，データベース対応コントロールに直接結合されていません。むしろ，ネストされたデータセットは単なるデータセットであり，格納されているデータは TDataSet の下位項目を通して取得します。使用するデータセットの種類は，親データセット（データセット項目を所有するデータセット）によって決まります。たとえば，BDE 対応データセットの場合は TNestedTable 使ってデータセット項目のデータを表し，クライアントデータセットの場合は別のクライアントデータセットを使用します。

データセット項目内のデータにアクセスアクセスするには，

1. 親データセットで項目エディタを呼び出して，持続的 TDataSetField オブジェクトを作成します。
2. データセット項目内の値を表すためデータセットを作成します。このデータセットの型は，親データセットに対応したものでなければなりません。
3. 手順 2 で作成されたデータセットの DataSetField プロパティを，手順 1 で作成された持続的データセット項目に設定します。

現在レコードにネストされたデータセット項目に値がある場合，詳細データセットコンポーネントはネストされたデータとともにレコードを含みます。それ以外の場合は，ネストされたデータセットは空になります。

ネストされたデータセットにレコードを挿入する前に，対応するレコードをマスターテーブル内に登録しなければなりません。挿入されたレコードが登録されていない場合，ネストされたデータセットが登録する前に自動的に登録されます。

参照項目を操作する

参照項目は，別の ADT オブジェクトへのポインタつまり参照を格納します。この ADT オブジェクトは，別のオブジェクトテーブルの 1 つのレコードです。参照項目は常にデータセット（オブジェクトテーブル）内の 1 つのレコードを参照します。参照されたオブジェクトの中のデータは，実際にはネストされたデータセットの中に戻りますが，TReferenceField の Fields プロパティを通してアクセスすることもできます。

参照項目を表示する

TDBGrid コントロールでは，参照項目はデータセット列の各セルの中に（Reference）とともに割り当てられ，実行時には右に省略記号ボタンが表示されます。実行時に省略記号をクリックすると，現在レコードの参照項目と対応するオブジェクトを表示するグリッドのある新しいフォームが表示されます。

またこのフォームは，DB グリッドの ShowPopupEditor メソッドを使用して，プログラミングによって表示できます。たとえばグリッド内の 7 番目の列が参照項目を表す場合，次のコードによって現在レコードのその項目に対応するオブジェクトが表示されます。

```
DBGrid1->ShowPopupEditor(DBGrid1->Columns->Items[7], -1, -1);
```

参照項目内のデータにアクセスする

ネストされたデータセットにアクセスするのと同じ方法で，参照項目内のデータにアクセスできます。

1. 親データセットで項目エディタを呼び出して，持続的 TDataSetField オブジェクトを作成します。
2. データセット項目の値を表すため，データセットを作成します。
3. 手順 2 で作成されたデータセットの DataSetField プロパティを，手順 1 で作成された持続的データセット項目に設定します。

参照が割り当てられた場合，参照データセットは参照データとともに 1 つのレコードを含みます。参照がヌルの場合，参照データセットは空になります。

参照項目内のデータにアクセスするため，参照項目の Fields プロパティを使用できます。たとえば次の行は，参照項目 CustomerRefCity のデータを CityEdit という編集ボックスに割り当てます。

```
CityEdit->Text = CustomerADDRESS_REF->NestedDataSet->Fields->Fields[1]->AsString;
```

参照項目の中のデータが編集されると，実際に修正されるのは参照されたデータです。

参照項目を割り当てるには，最初に SELECT 文を使用してテーブルから参照を選択し，次にそれを割り当てます。例を示します。

```
AddressQuery->SQL->Text =
    "SELECT REF(A) FROM AddressTable A WHERE A.City = 'San Francisco'";
AddressQuery->Open();
CustomerAddressRef->Assign(AddressQuery->Fields->Fields[0]);
```

第24章

ボーランドデータベースエンジンの使い方

ボーランドデータベースエンジン（BDE）は，複数のアプリケーションで共有できるデータアクセスメカニズムです。BDE には API 呼び出しの強力なライブラリが定義されており，このライブラリを使って，ローカルおよびリモートのサーバーに対して，その作成，再構築，データの取得，更新などの操作を実行できます。BDE では，各種データベースに接続するドライバを使って，統一されたインターフェースを介してさまざまなデータベースサーバーにアクセスできます。使用する C++Builder のバージョンによって異なりますが，ローカルデータベース（Paradox，dBASE，FoxPro，Access）用のドライバ，リモートデータベース（InterBase，Oracle，Sybase，Informix，Microsoft SQL Server，DB2）用の SQL Link ドライバ，および独自の ODBC ドライバを作成するための ODBC アダプタを利用できます。

BDE ベースのアプリケーションを配布するときは，アプリケーションに BDE を組み込まなければなりません。これによってアプリケーションのサイズと配布の複雑さは増しますが，BDE をほかの BDE ベースアプリケーションと共用でき，データベースに対して多種多様な操作を行うことが可能になります。BDE の API はアプリケーションから直接使用することも可能ですが，このほとんどの機能は，コンポーネントパレットの［BDE］ページから利用することができます。

メモ BDE API についての詳細は，オンラインヘルプファイルの BDE32.hlp を参照してください。このファイルは，ボーランドデータベースエンジンをインストールしたディレクトリにインストールされています。

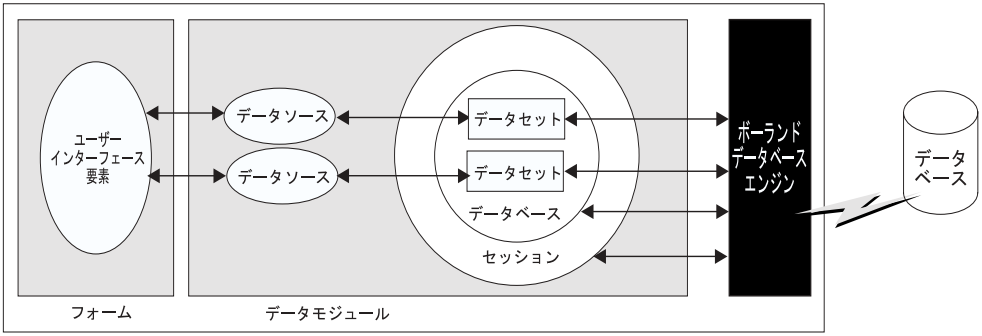
BDE ベースのアーキテクチャ

BDE を利用する場合は，18-6 ページの「データベースアーキテクチャ」に述べられている，データベースの一般的なアーキテクチャを拡張したものをアプリケーションで扱うことになります。すべての C++Builder データベースアプリケーションに共通する，ユーザーインターフェース要素，データソース，データセットに加えて，BDE ベースのアプリケーションでは以下のコンポーネントを使えます。

- トランザクションを制御しデータベース接続を管理するための，1 つまたは複数のデータベースコンポーネント
- データ接続などのデータアクセス操作を切り離し，いくつかのデータベースを管理するための，1 つまたは複数のセッションコンポーネント

BDE ベースアプリケーションでの，コンポーネント間の関係を図 24.1 に示します。

図 24.1　BDE **ベースアプリケーションのコンポーネント**



BDE 対応のデータセットの使い方

BDE 対応データベースは，ボーランドデータベースエンジン（BDE）を使ってデータにアクセスします。このデータセットは，第 22 章「データセットについて」で述べたデータセットの共通機能を継承しており，BDE を使ってその機能を実行します。さらに，すべての BDE データセットには，以下の処理を目的とする，プロパティ，イベント，およびメソッドが追加されています。

- データセットとデータベース接続およびセッション接続との関連付け
- BLOB のキャッシング
- BDE ハンドルの取得

BDE 対応データセットには，次の 3 種類があります。

- TTable：1 つのデータベーステーブルのすべての行と列を表すテーブルタイプのデータセット。テーブルタイプのデータセットに共通する機能については，22-24 ページの「テーブルタイプのデータセットの使い方」を参照してください。TTable に特有な機能については，24-5 ページの「TTable の使い方」を参照してください。
- TQuery：SQL 文をカプセル化し，結果レコードが得られた場合はアプリケーションからそのレコードにアクセスできるようにする問い合わせタイプのデータセット。問い合わせタイプのデータセットに共通する機能については，22-40 ページの「問い合わせタイプのデータセットの使い方」を参照してください。TQuery に特有な機能については，24-8 ページの「TQuery の使い方」を参照してください。
- TStoredProc：データベースサーバー上に定義されたストアドプロシージャを実行するストアドプロシージャタイプのデータセット。ストアドプロシージャタイプのデータセットに共通する機能については，22-48 ページの「ストアドプロシージャタイプのデータセットの使い方」を参照してください。TStoredProc に特有な機能については，24-11 ページの「TStoredProc の使い方」を参照してください。

メモ 以上の 3 種類の BDE 対応データセットのほかに，キャッシュアップデートに使用するための BDE 対応クライアントデータセット（TBDEClientDataSet）もあります。キャッシュアップデートについての詳細は，27-15 ページの「クライアントデータセットをキャッシュアップデートに使用する」を参照してください。

データセットとデータベース接続およびセッション接続との関連付け

BDE 対応データセットがデータベースサーバーからデータを取得するには，データベースとセッションの両方を使用する必要があります。

- データベースは特定のデータベースサーバーへの接続を表します。データベースは，BDE ドライバ，そのドライバを使用する特定のデータベースサーバー，そのデータベースサーバーに接続するための一連の接続パラメータを識別します。それぞれのデータベースサーバーは，TDatabase コンポーネントで表されます。データセットは，フォームまたはデータモジュールに追加する TDatabase コンポーネントを関連付けることができます。また，データベースサーバーを単に名前で特定すると，データベースコンポーネントが C++Builder によって暗黙的に生成されます。ほとんどのアプリケーションでは，TDatabase コンポーネントを明示的に作成することをお勧めします。この場合，このデータベースコンポーネントを使って，ログインプロセスなどの接続の確立方法を細かく制御できるとともに，トランザクションを作成して利用することができます。

  BDE 対応データセットをデータベースに関連付けるには，DatabaseName プロパティを使用します。DatabaseName は文字列で，その内容は，明示的なデータベースコンポーネントを使用するかどうかで異なります。また，明示的なデータベースコンポーネントを使わない場合は，使用するデータベースの種類によって異なります。

  - TDatabase コンポーネントを明示的に使用する場合，DatabaseName は，TDatabase コンポーネントの DatabaseName プロパティの値になります。
  - データベースコンポーネントを暗黙的に使用する場合に，データベースに BDE エリアスがあれば，DatabaseName の値として BDE エリアスを指定できます。BDE エリアスは，データベースとそのデータベースの環境設定情報を表します。エリアスに関連付けられる環境設定情報は，データベースの種類（Oracle，Sybase，InterBase，Paradox，dBASE など）によって異なります。BDE エリアスの作成と管理には，BDE Administrator または SQL エクスプローラを使用します。
  - Paradox または dBASE データベースに対して暗黙的なデータベースコンポーネントを使用する場合は，DatabaseName に単にデータベーステーブルのあるディレクトリを指定することができます。
- セッションコンポーネントを使うと，アプリケーション内の一連のデータベース接続をグローバルに管理できます。BDE 対応データセットをアプリケーションに追加すると，Session というセッションコンポーネントが自動的にアプリケーションに組み込まれます。アプリケーションにデータベースコンポーネントやデータセットコンポーネントを追加すると，そのコンポーネントは自動的にこのデフォルトセッションに関連付けられます。また，セッションコンポーネントは，パスワード保護された Paradox ファイルへのアクセスを制御し，ネットワーク上で Paradox ファイルを共有するためのディレクトリ位置を指定します。データベース接続や Paradox ファイルへのアクセスは，セッションのプロパティ，イベント，およびメソッドを使って制御できます。

デフォルトセッションを使用して，アプリケーション内のすべてのデータベース接続を制御することができます。また，設計時に追加のセッションコンポーネントを追加するか，実行時にそれらを動的に作成して，アプリケーション内のデータベース接続のサブセットを制御することもできます。データセットを，明示的に作成したセッションコンポーネントに関連付けるには，SessionName プロパティを使用します。アプリケーションで明示的なセッションコンポーネントを使わない場合，このプロパティの値を指定する必要はありません。データセットに関連付けられたセッションには，それがデフォルトセッションか，または SessionName プロパティを使って明示的に指定したセッションかにかかわらず，DBSession プロパティを使ってアクセスできます。

メモ セッションコンポーネントを使用する場合，データセットの SessionName プロパティと，そのデータセットが関連付けられているデータベースコンポーネントの SessionName プロパティは一致していなければなりません。

TDatabase と TSession についての詳細は，24-12 ページの「TDatabase 使ってデータベースに接続する」および 24-16 ページの「データベースセッションの管理」を参照してください。

BLOB のキャッシング

すべての BDE 対応データセットには，アプリケーションが BLOB レコードを読むときに BDE によって BLOB 項目をローカルなキャッシュに入れるかどうかを制御する CacheBlobs プロパティがあります。デフォルトでは，CacheBlobs は true で，BDE は BLOB 項目のローカルなコピーをキャッシュに入れます。BLOB をキャッシュに入れると，BDE はユーザーがレコード間をスクロール移動するたびにデータベースサーバーから繰り返し BLOB 項目を取り出すかわりに，BLOB のローカルコピーを保存するので，アプリケーションのパフォーマンスが向上します。

BLOB が頻繁に更新または置換され，また BLOB データの最新内容を表示することがアプリケーションパフォーマンスよりも重要なアプリケーションや環境では，CacheBlobs を **false** に設定することによって常に BLOB 項目の最新内容を表示するようにできます。

BDE ハンドルの取得

BDE 対応データセットは，ボーランドデータベースエンジンの API を直接呼び出さなくても使うことができます。BDE 対応データセットを，データベースコンポーネントとセッションコンポーネントと組み合わせて使用している場合，ほとんどの BDE の機能はカプセル化されています。しかし，BDE の API を直接呼び出すことが必要になった場合，BDE で管理するリソースの BDE ハンドルが必要なこともあります。多くの BDE API では，パラメータとしてこのハンドルを指定する必要があります。

すべての BDE 対応データセットには，BDE ハンドルを実行時に取得するための読み出し専用プロパティが 3 つあります。

- Handle は，データセット内のレコードにアクセスする BDE カーソルへのハンドルです。
- DBHandle は，基になるテーブルまたはストアドプロシージャを含むデータベースへのハンドルです。
- DBLocale は，データセットの BDE 言語ドライバへのハンドルです。ロケールによって，文字列データのソート順および文字セットが制御されます。

これらのプロパティは，データセットが BDE を介してデータベースサーバーに接続されるとき自動的に割り当てられます。BDE API についての詳細は，オンラインヘルプ BDE32.HLP を参照してください。

TTable の使い方

TTable は，基礎になるデータベーステーブルの完全な構造とデータをカプセル化しています。TTable には，TDataSet が持つ基本的な機能とテーブルタイプのデータセットに特有な機能のすべてが実装されています。TTable に特有な機能についての説明は，22-24 ページ以降のテーブルタイプのデータセットも含め，「データセットについて」で説明しているデータベースの一般的な機能に関する知識を前提としています。

TTable は BDE 対応データセットの一種であるため，データベースとセッションに関連付ける必要があります。この関連付けの方法は，24-3 ページの「データセットとデータベース接続およびセッション接続との関連付け」で説明しています。データベースとセッションに関連付けたデータセットは，特定のデータベーステーブルに結びつけることができます。この処理は，TableName プロパティを設定することによって行い，Paradox，dBASE，FoxPro，またはカンマ区切りのテキストのテーブルを使用する場合は TableType プロパティも設定します。

メモ テーブルと，データベース，セッション，またはデータベーステーブルとの関連を変更するときには，テーブルは閉じていなければなりません。また，TableType プロパティを設定するときもテーブルが閉じている必要があります。ただし，テーブルを閉じてこれらのプロパティを変更するには，その前に，保留状態になっているすべての変更を登録または破棄します。キャッシュアップデートが有効な場合，ApplyUpdates メソッドを呼び出し，登録された変更をデータベースに書き込みます。

操作の対象が Paradox，dBASE，FoxPro，カンマ区切り ASCII テキストテーブルなどのローカルデータベーステーブルの場合，TTable は特別な機能をサポートしています。次のトピックでは，この機能を実行するための特別なプロパティとメソッドについて説明します。

さらに TTable コンポーネントでは，BDE のバッチ処理機能（テーブルレベルで，一連のレコードすべて追加，更新，削除，コピーする操作）も利用することができます。この機能ついては，24-8 ページの「別のテーブルからのデータのインポート」で説明しています。

ローカルテーブルのテーブル型の指定

アプリケーションが Paradox テーブル，dBASE テーブル，FoxPro テーブル，カンマ区切り ASCII テキストテーブルにアクセスする場合，BDE は TableType プロパティを使ってテーブルの種類（想定される構造）を決めます。TTable がデータベースサーバー上の SQL テーブルを表している場合は，TableType は使われません。

デフォルトでは TableType は ttDefault に設定されます。TableType が ttDefault の場合，BDE はテーブルの種類をファイル名拡張子から決めます。表 24.1 に，BDE が認識できるファイル名拡張子とそれに対応するテーブルの種類を示します。

表 24.1　BDE が認識するファイル拡張子とその対応テーブルの種類

拡張子	テーブルの種類
なし	Paradox
.DB	Paradox
.DBF	dBASE
.TXT	ASCII テキスト

ローカルの Paradox テーブル，dBASE テーブル，ASCII テキストテーブルが表 24.1 のファイル名拡張子を使う場合，TableType を ttDefault のままにすることができます。これ以外のファイル名拡張子を使う場合は，正しいテーブルの種類を示すように，アプリケーションで TableType を設定しなければなりません。表 24.2 に TableType に設定できる値を示します。

表 24.2　TableType の値

値	テーブルの種類
ttDefault	BDE で自動的に判断されるテーブルの種類
ttParadox	Paradox
ttDBase	dBASE
ttFoxPro	FoxPro
ttASCII	カンマで区切られた ASCII テキスト

ローカルテーブルに対する読み書きの制御

TTable では，ほかのテーブルタイプのデータセットと同様に，アプリケーションの読み書きのアクセス権を ReadOnly プロパティを使って制御できます。

さらに，Paradox，dBASE，および FoxPro の各テーブルについては，ほかのアプリケーションの読み書きアクセス権も TTable で制御できます。Exclusive プロパティは，Paradox，dBASE または FoxPro のテーブルへの排他的な読み書きアクセス権をアプリケーションに持たせるかどうかを制御します。これらのテーブルに排他的に読み書きアクセスができるようにするには，テーブルを開く前にテーブルコンポーネントの Exclusive プロパティを **true** に設定します。排他的アクセスでテーブルを開くと，ほかのアプリケーションはそのテーブルでデータの読み書きができなくなります。排他的アクセス要求は，開こうとするテーブルが使用中のときは認められません。

次の文は排他的アクセスでテーブルを開きます。

```
CustomersTable->Exclusive = true; // 排他的ロックの要求を設定する
CustomersTable->Active = true;    // ここでテーブルを開く
```

メモ　SQL テーブルにも Exclusive プロパティを設定できますが，サーバーによってはテーブルレベルの排他的ロックをサポートしていない場合があります。排他的ロックをサポートしていても，ほかのアプリケーションに対して SQL テーブルからデータを読み出すのを許可している場合もあります。サーバーでのデータベーステーブルの排他的ロックについての詳細は，サーバーのマニュアルを参照してください。

dBASE インデックスファイルを指定する

インデックスを指定するには，ほとんどのサーバーの場合，テーブルタイプのすべてのデータセットに共通するメソッドを使用します。これらのメソッドは，22-25 ページの「インデックスを持つレコードのソート」で説明されています。

しかし，非プロダクションインデックスファイルを使う dBASE テーブルの場合，あるいは dBASE III PLUS スタイルのインデックス（*.NDX）の場合は，上記の方法ではなく，IndexFiles プロパティと IndexName プロパティを使ってインデックスを指定しなければなりません。IndexFiles プロパティで非プロダクションインデックスファイルの名前を設定するか，.NDX ファイルのリストを表示します。次に，IndexName プロパティでインデックスを 1 つ指定します。これにより，インデックスをベースにしてデータセットを有効にソートできます。

設計時に，オブジェクトインスペクタで IndexFiles プロパティの値列にある省略記号ボタンをクリックして，インデックスファイルエディタを呼び出します。非プロダクションインデックスファイルまたは .NDX ファイルを追加するには，［インデックスファイル］ダイアログで［追加］ボタンをクリックして，［開く］ダイアログで目的のファイルを選択します。1 回の操作で選択できる非プロダクションインデックスファイルまたは .NDX ファイルは 1 つだけです。目的のインデックスをすべて追加したら，［インデックスファイル］ダイアログの［OK］をクリックします。

これと同じ操作は，実行時にプログラムで行うことができます。そのためには，文字列リストのプロパティとメソッドを使って IndexFiles プロパティにアクセスします。新しいインデックスを追加する場合は，まずテーブルの IndexFiles プロパティの Clear メソッドを呼び出して，既存のエントリをすべて削除します。追加する非プロダクションインデックスファイルまたは .NDX ファイル 1 つに対して Add メソッドを 1 回ずつ呼び出します。

```
Table2->IndexFiles->Clear();
Table2->IndexFiles->Add("Bystate.ndx");
Table2->IndexFiles->Add("Byzip.ndx");
Table2->IndexFiles->Add("Fullname.ndx");
Table2->IndexFiles->Add("St_name.ndx");
```

目的の非プロダクションインデックスファイルまたは .NDX ファイルを追加すると，そのインデックスファイル内の個々のインデックスの名前が利用可能になり，その名前を IndexName プロパティに代入できます。インデックスタグのリストを取り出すこともできます。それには，GetIndexNames メソッドを使うか，IndexDefs プロパティで TIndexDef オブジェクトを使ってインデックス定義を調べます。.NDX ファイルを適切にリストしておけば，テーブル内でのデータの追加，変更，削除に応じて自動的に .NDX ファイルが更新されます（IndexName プロパティでインデックスを指定したかどうかを問わない）。

下の例では，テーブルコンポーネント AnimalsTable の IndexFiles プロパティを非プロダクションインデックスファイル ANIMALS.MDX に設定し，IndexName プロパティを「NAME」というインデックスタグに設定しています。

```
AnimalsTable->IndexFiles->Add("ANIMALS.MDX");
AnimalsTable->IndexName = "NAME";
```

インデックスファイルの指定が終了すると，非プロダクションインデックスや .NDX インデックスは，ほかのインデックスと同様に機能します。インデックス名を指定すると，テーブル内のデータがソートされ，検索，範囲指定，およびマスタ / 詳細リンク（非プロダクションインデックス場合）の各操作をインデックスに基づいて行えます。インデックスをこれらの用途に使用する方法については，22-24 ページの「テーブルタイプのデータセットの使い方」を参照してください。

TTable コンポーネントで dBASE III PLUS スタイルの .NDX インデックスを使用する場合は，次の 2 点に注意する必要があります。第 1 に，.NDX ファイルをマスター / 詳細リンクのベースとして使うことはできません。第 2 に，IndexName プロパティで .NDX インデックスをアクティブにするとき，インデックス名として設定するプロパティ値に拡張子 .NDX を含めなければなりません。

```
Table1->IndexName = "ByState.NDX";
TVarRec vr = ("NE");
Table1->FindKey(&vr, 0);
```

ローカルテーブルの名前の変更

Paradox または dBASE のテーブルの名前を設計時に変更するには，テーブルコンポーネントを右クリックしてコンテキストメニューから［テーブルの名称変更］を選択します。

Paradox または dBASE のテーブルの名前を実行時に変更するには，テーブルの RenameTable メソッドを呼び出します。たとえば次の文は，Customer テーブルを CustInfo という名前に変更します。

```
Customer->RenameTable("CustInfo");
```

別のテーブルからのデータのインポート

テーブルコンポーネントの BatchMove メソッドを使えば，別のテーブルからデータをインポートできます。BatchMove では以下のことができます。

- 別のテーブルから現在のテーブルにレコードをコピーする
- 別のテーブルと共通するレコードで現在のテーブルを更新する
- 別のテーブルのレコードを現在のテーブルの最後に追加する
- 別のテーブルと共通するレコードを現在のテーブルから削除する

BatchMove には，パラメータを 2 つ指定します。データのインポート元のテーブルの名前と，どのようなインポート操作を実行するかを決めるモード指定です。表 24.3 に設定可能なモードを示します。

表 24.3 BatchMove インポートモード

値	説明
batAppend	インポート元テーブルのすべてのレコードを現在のテーブルの最後に追加する
batAppendUpdate	インポート元テーブルのすべてのレコードを現在のテーブルの最後に追加し，現在のテーブルの既存のレコードをインポート元テーブルの対応するレコードで更新する
batCopy	インポート元テーブルのすべてのレコードを現在のテーブルにコピーする
batDelete	インポート元テーブルと共通するすべてのレコードを現在のテーブルから削除する
batUpdate	現在のテーブルの既存のレコードをインポート元テーブルの対応するレコードで更新する

たとえば次のコードは，現在のテーブルのすべてのレコードを，現在のインデックス内に同じ項目値を持つ，Customer テーブルのレコードで更新します。

```
Table1->BatchMove("CUSTOMER.DB", batUpdate);
```

BatchMove は正常にインポートできたレコードの数を返します。

注意 batCopy モードでレコードをインポートすると，既存のレコードが上書きされます。既存のレコードを保持するには batAppend を使います。

BatchMove で行う処理は，BDE がサポートするバッチ処理機能の一部にすぎません。TBatchMove コンポーネントを使うと，ほかの機能も実行できます。2 つ以上のテーブル間で大量のデータを移動する必要があるときは，テーブルの BatchMove メソッドを呼び出すのではなく，TBatchMove を使ってください。TBatchMove の使い方については，24-47 ページの「TBatchMove の使い方」を参照してください。

TQuery の使い方

TQuery は，1 つのデータ定義言語（DDL）文，または 1 つのデータ操作言語を表します（DML）文（SELECT，INSERT，DELETE，UPDATE，CREATE INDEX，ALTER TABLE コマンドなど）。コマンドに使用される言語はサーバー固有ですが，通常は SQL 言語の SQL-92 標準に準拠しています。TQuery には，TDataSet が持つ基本的な機能と問い合わせタイプのデータセットに特有な機能のすべ

てが実装されています。TQuery に特有な機能についての説明は，22-40 ページ 以降の問い合わせタイプのデータセットも含め，「データセットについて」で説明しているデータベースの一般的な機能に関する知識を前提としています。

TQuery は BDE 対応データセットの一種であるため，通常はデータベースとセッションに関連付ける必要があります。（TQuery を異種間の問い合わせに使用する場合だけは，この必要がありません）。これら関連付けの方法は，24-3 ページの「データセットとデータベース接続およびセッション接続との関連付け」で説明しています。問い合わせの SQL 文は，SQL プロパティを使って指定します。

TQuery コンポーネントは以下にあるデータにアクセスできます。

- ローカル SQL（BDE の一部）を使う Paradox または dBASE テーブル。ローカル SQL は SQL-92 標準のサブセットをサポートする。ほとんどの DML がサポートされ，これらのタイプのテーブルを扱うのに十分な DDL 構文がサポートされている。サポートされている SQL 構文についての詳細は，ローカル SQL のヘルプである LOCALSQL.HLP を参照
- InterBase エンジンを使うローカル InterBase サーバーデータベース。InterBase の SQL-92 標準の SQL 構文サポートと拡張構文サポートについては，『InterBase 言語リファレンス』を参照
- Oracle，Sybase，MS-SQL Server，Informix，DB2，InterBase などのリモートデータベースサーバー上のデータベース。リモートサーバーにアクセスするには，適切な SQL Link ドライバとクライアントソフトウェア（ベンダーが供給したもの）をインストールする必要がある。これらのサーバーがサポートする標準の SQL 構文はどれも使うことができる。SQL の構文，制約，拡張機能については，サーバーのマニュアルを参照

異種間の問い合わせの作成

TQuery は複数の種類のサーバーまたはテーブル（たとえば，Oracle テーブルのデータと Paradox テーブルのデータ）に対する異種間の問い合わせもできます。異種間の問い合わせを実行すると，BDE がローカル SQL を使って問い合わせを解析して処理します。BDE がローカル SQL を使うため，サーバー独自に拡張された SQL 構文はサポートされません。

異種間の問い合わせを実行する手順は次のとおりです。

1. BDE Administrator または SQL エクスプローラを使って，問い合わせでアクセスするデータベースごとに BDE エリアスを別々に定義します。
2. TQuery の DatabaseName プロパティは空白のままにしてください。使用する 2 つのデータベースの名前は SQL 文で指定します。
3. SQL プロパティで，実行する SQL 文を指定します。この文の各テーブル名の前には，そのテーブルがあるデータベースの BDE エリアスを，コロンで囲んで記述します。この参照全体を引用符で囲んでください。
4. 問い合わせ用のパラメータを Params プロパティに設定します。
5. 初めて問い合わせを実行する前に Prepare を呼び出して，その問い合わせを実行するための準備をします。
6. 実行する問い合わせの種類に応じて Open または ExecSQL を呼び出します。

たとえば，CUSTOMER テーブルを持つ Oracle データベースに Oracle1 というエリアスを定義し，ORDERS テーブルを持つ Sybase データベースに Sybase1 というエリアスをそれぞれ定義するとします。この 2 つのテーブルに対する単純な問い合わせは次のようになります。

```
SELECT Customer.CustNo, Orders.OrderNo
FROM ":Oracle1:CUSTOMER"
  JOIN ":Sybase1:ORDERS"
    ON (Customer.CustNo = Orders.CustNo)
WHERE (Customer.CustNo = 1503)
```

異種間の問い合わせでデータベースを指定するのに，BDE エリアスを使う代わりに TDatabase コンポーネントを使うこともできます。TDatabase を通常どおりにデータベースを指し示すように設定し，TDatabase::DatabaseName を任意のユニークな値に設定して，その値を SQL 文の中で BDE エリアス名の代わりに使います。

編集可能な結果セットの取得

データベース対応コントロールでユーザーが編集可能な結果セットを要求するには，問い合わせコンポーネントの RequestLive プロパティを **true** に設定します。RequestLive を **true** に設定することでライブ結果セットが保証されるわけではありません。しかし，BDE はできるだけ要求に応えようとします。問い合わせがローカル SQL パーサーを使っているかサーバーの SQL パーサーを使っているかにより，ライブ結果セット要求はある程度制限されます。

- 異種間の問い合わせのように，テーブル名の前にデータベースの BDE エリアスが記述されている問い合わせ，および Paradox や dBASE に対して実行される問い合わせは，BDE がローカル SQL を使って解析します。ローカル SQL パーサーを使う問い合わせの場合，BDE は単一テーブルとマルチテーブルの両方の問い合わせに対して更新可能なライブ結果セットを拡張してサポートします。ローカル SQL を使用する場合は，問い合わせに以下のものが含まれていなければ，単一のテーブルまたはビューに対する問い合わせでライブ結果セットが返されます。
  - SELECT 文の DISTINCT 節
  - 結合（内部，外部，または UNION）
  - 集合関数（GROUP BY 節または HAVING 節を伴う場合も伴わない場合も）
  - 更新できない基本テーブルまたはビュー
  - 副問い合わせ
  - インデックスに基づいていない ORDER BY 節
- リモートデータベースサーバーに対する問い合わせの場合は，サーバーがその問い合わせを解析します。RequestLive プロパティが **true** に設定されている場合には，BDE がデータの変更をテーブルに反映するためにローカル SQL 標準を使う必要があるので，SQL 文は，サーバーによる制限に加えて，ローカル SQL 標準にも従っていなければなりません。問い合わせに以下のものが含まれていなければ，単一のテーブルまたはビューに対する問い合わせでライブ結果セットが返されます。
  - SELECT 文の DISTINCT 節
  - 集合関数（GROUP BY 節または HAVING 節を伴う場合も伴わない場合も）
  - 複数の基本テーブルまたは更新可能なビューへの参照
  - FROM 節またはほかのテーブルの中のテーブルを参照している副問い合わせ

アプリケーションがライブ結果セットを要求し，受け取る場合，その問い合わせコンポーネントの CanModify プロパティは **true** に設定されます。問い合わせがライブ結果セットを返した場合でも，

リンクされた項目がその中に含まれているかまたは更新する前にインデックスを切り替えると，問い合わせの結果セットを直接更新できなくなることがあります。そのような場合，結果セットを読み出し専用結果セットとして扱い，それに応じた方法で更新する必要があります。

アプリケーションがライブ結果セットを要求しても，SELECT 文の構文でそれが許されない場合，BDE は以下のどちらかを返します。

- 読み出し専用結果セット（Paradox または dBASE に対する問い合わせの場合）
- エラーコード（リモートサーバーに対する SQL 問い合わせの場合）

読み出し専用結果セットの更新

キャッシュアップデートを使うと，読み出し専用の問い合わせ結果から元テーブルへの更新を行うことができます。

クライアントデータセットを使って更新データをキャッシュする場合，クライアントデータセットまたはそれに関連したプロバイダによって，更新を適用するための SQL 文が自動的に生成されます。ただし，これは，その問い合わせが複数のテーブルを表していない場合に限られます。問い合わせが複数のテーブルを表す場合は，更新データを適用する方法を指定しなければなりません。

- すべての更新を適用するデータベーステーブルが 1 つだけの場合は，OnGetTableName イベントハンドラ内でそのテーブルを指定し，更新します。
- 更新の適用処理をより細かく制御する必要がある場合は，その問い合わせをアップデートオブジェクト（TUpdateSQL）に関連付けることができます。プロバイダは，自動的にこのアップデートオブジェクトを使って更新を適用します。

  1. 問い合わせの UpdateObject プロパティに，使用する TUpdateSQL を設定することにより，問い合わせにアップデートオブジェクトを関連付けます。
  2. そのアップデートオブジェクトの ModifySQL，InsertSQL，DeleteSQL の各プロパティに，問い合わせのデータを適切に更新するための SQL 文を設定します。

BDE を使ってキャッシュアップデートを実行している場合は，アップデートオブジェクトを使用しなければなりません。

メモ アップデートオブジェクトの使い方については，24-39 ページの「アップデートオブジェクトを使ったデータセットの更新」を参照してください。

TStoredProc の使い方

TStoredProc は，ストアドプロシージャを表します。TStoredProc には，TDataSet が持つすべての基本機能と，ストアドプロシージャタイプのデータセットに特有な機能のほとんどが実装されています。TStoredProc に特有な機能についての説明は，22-48 ページ 以降のストアドプロシージャタイプのデータセットも含め，「データセットについて」で説明しているデータベースの一般的な機能に関する知識を前提としています。

TStoredProc は BDE 対応データセットの一種であるため，データベースとセッションに関連付ける必要があります。この関連付けの方法は，24-3 ページの「データセットとデータベース接続およびセッション接続との関連付け」で説明しています。データベースとセッションに関連付けたデータ

セットは，StoredProcName プロパティを設定することにより，特定のストアドプロシージャに結びつけることができます。

TStoredProc は，ほかのストアドプロシージャタイプのデータセットとは以下の点が異なります。

- パラメータの結合方法を細かく制御できる
- Oracle のオーバーロードストアドプロシージャをサポートしている

パラメータの結合

ストアドプロシージャを準備して実行した場合，その入力パラメータは自動的にサーバーのパラメータに結合されます。

TStoredProc では，ParamBindMode プロパティを使って，パラメータをどのようにサーバーのパラメータと結合させるかを指定できます。ParamBindMode のデフォルト設定は pbByName で，名前を使ってストアドプロシージャコンポーネントのパラメータとサーバーのパラメータの対応がとられます。これが一番簡単なパラメータ結合の方法です。

一部のサーバーは，順序値，つまりストアドプロシージャ内のパラメータの順序によるパラメータ結合もサポートしています。この場合，パラメータコレクションエディタでのパラメータの指定順が重要です。最初に指定したパラメータがサーバーの最初の入力パラメータと一致し，2 番目のパラメータがサーバーの 2 番目の入力パラメータと一致する，という具合になります。サーバーが順序値によるパラメータ結合をサポートしている場合は，ParamBindMode を pbByNumber に設定できます。

ヒント ParamBindMode を pbByNumber に設定したい場合は，正しいパラメータ型を正しい順序で指定する必要があります。指定するパラメータの正しい順序と型を調べるために，SQL エクスプローラでサーバーのストアドプロシージャのソースコードを表示できます。

Oracle のオーバーロードストアドプロシージャ

Oracle サーバーでは，ストアドプロシージャのオーバーロードができます。オーバーロードプロシージャとは同じ名前を持つ異なるプロシージャのことです。ストアドプロシージャコンポーネントの Overload プロパティを使うと，アプリケーションで実行するプロシージャを指定できます。

Overload が 0（デフォルト）の場合，オーバーロードはないと想定されます。Overload が 1 の場合，ストアドプロシージャコンポーネントは Oracle サーバーで同じオーバーロード名を持つ最初のストアドプロシージャを実行します。Overload が 2 の場合は，2 番目のストアドプロシージャが実行されるというようになります。

メモ オーバーロードストアドプロシージャは異なる入力パラメータと出力パラメータをとることがあります。詳細は Oracle サーバーのドキュメントを参照してください。

TDatabase 使ってデータベースに接続する

C++Builder アプリケーションがボーランドデータベースエンジン（BDE）を使ってデータベースに接続する場合，その接続は TDatabase コンポーネントによってカプセル化されます。データベースコンポーネントは，1 つのデータベースへの接続を BDE セッションのコンテキストで表します。

TDatabase が実行するタスクの多くは，ほかのデータベース接続コンポーネントのタスクと同じであり，TDatabase が持つプロパティ，メソッド，イベントなど多くは，ほかのデータベース接続コンポー

ネントと共有しています。これらの共通点については，第 21 章「データベースへの接続」で説明しています。

これらの共通するプロパティ，メソッド，およびイベントに加え，TDatabase には，BDE 固有の機能も多く追加されています。以下のトピックでは，この機能について説明します。

データベースとセッションとの関連付け

すべてのデータベースコンポーネントは BDE セッションと関連付けなければなりません。この関連付けは，SessionName を使って確立します。設計時に初めてデータベースコンポーネントを作成したときには，SessionName は，「Default」に設定されています。これは，そのコンポーネントがグローバルな Session 変数が示す，デフォルトのセッションに関連付けられていることを意味します。

マルチスレッドの BDE アプリケーションや再入可能な BDE アプリケーションでは，複数のセッションが必要なことがあります。複数のセッションが必要な場合は，セッションごとに TSession コンポーネントを追加します。そして，データセットの SessionName プロパティにセッションコンポーネントの SessionName プロパティを設定することによってデータセットを関連付けます。

実行時には，データベースの Session プロパティを読み出すことによって，関連付けられているセッションコンポーネントにアクセスできます。SessionName が空白または「Default」の場合，Session プロパティは，グローバルの Session 変数が参照しているのと同じ TSession インスタンスを参照します。Session を使うと，アプリケーションはセッションの実際の名前を知らずにデータベースコンポーネントの親セッションコンポーネントのプロパティ，メソッド，イベントにアクセスできます。

BDE セッションについては，24-16 ページの「データベースセッションの管理」を参照してください。

データベースコンポーネントを暗黙的に使用している場合は，データセットの SessionName プロパティで指定されたセッションが，そのデータベースコンポーネントのセッションになります。

データベースとセッションの相互作用

一般に，セッションコンポーネントの各プロパティにはグローバルなデフォルトの動作があり，実行時に自動的に作成されるすべての暗黙的データベースコンポーネントに，その動作が適用されます。たとえば，セッションの KeepConnections プロパティを制御することにより，関連するデータセットを閉じた場合にもデータベース接続を維持するのか（デフォルト），またはすべてのデータセットが閉じたら接続を切断するのかが決まります。同様に，セッションのデフォルトの OnPassword イベントにより，パスワードの必要なサーバー上のデータベースにアプリケーションが接続しようとしたときに標準のパスワード入力ダイアログボックスが表示されます。

セッションメソッドの使われ方はやや異なります。データベースコンポーネントが明示的に作成されたかデータセットによって暗黙的に作成されたかにかかわらず，TSession のメソッドはすべてのデータベースコンポーネントに影響します。たとえば，セッションメソッド DropConnections は，セッションのデータベースコンポーネントに属するすべてのデータセットを閉じ，個々のデータベースコンポーネントの KeepConnection プロパティが **true** であっても，すべてのデータベース接続を切断します。

データベースコンポーネントのメソッドは，特定のデータベースコンポーネントに関連付けられているデータセットに対してだけ適用されます。たとえば，データベースコンポーネント Database1 がデフォルトセッションに関連付けられているとします。Database1->CloseDataSets は，Database1 に関連付けられたデータセットだけを閉じます。デフォルトセッション内でほかのデータベースコンポーネントに属して開いているデータセットは，開いたままになります。

データベースの識別

AliasName と DriverName は互いに排他的な関係にありますが，両方とも TDatabase コンポーネントの接続先のサーバーを識別します。

- AliasName は，データベースコンポーネントに使う既存の BDE エリアスの名前を表します。BDE エリアスは，特定のデータベースコンポーネントにデータセットコンポーネントをリンクできるように，データセットコンポーネント用の以後のドロップダウンリストに表示されます。ドライバ名は BDE エリアスの一部なので，データベースコンポーネントに AliasName を指定すると，すでに DriverName に代入されている値はクリアされます。

  BDE エリアスの作成と編集にはデータベースエクスプローラまたは BDE Administrator を使います。BDE エリアスの作成と管理については，これらのユーティリティのオンラインマニュアルを参照してください。

- DriverName は BDE ドライバの名前です。ドライバ名は BDE エリアスの 1 パラメータですが，DatabaseName プロパティを使ってデータベースコンポーネントのローカル BDE エリアスを作成するときにエリアスのかわりにドライバ名を指定できます。DriverName を指定すると，すでに AliasName に代入されている値はクリアされます。これは，指定したドライバ名が，AliasName で指定されている BDE エリアスに含まれているドライバ名と矛盾することを避けるためです。

DatabaseName では，データベース接続に独自の名前を付けることができます。指定する名前は，AliasName または DriverName とは別の名前であり，アプリケーションに対してローカルになります。DatabaseName は BDE エリアス名や，Paradox または dBASE ファイルの場合は絶対パス名を指定できます。AliasName と同様に，DatabaseName はデータセットコンポーネント用のドロップダウンリストに表示され，データセットコンポーネントとデータベースコンポーネントのリンクに使用されます。

設計時に BDE エリアスの指定，BDE ドライバの割り当て，ローカル BDE エリアスの作成をするには，データベースコンポーネントをダブルクリックしてデータベースプロパティエディタを起動します。

DatabaseName は，プロパティエディタの［名前］編集ボックスで入力できます。Alias プロパティ用の［エリアス名］コンボボックスには，既存の BDE エリアス名を入力するかまたはドロップダウンリストの既存のエリアスを選択します。DriverName プロパティ用の［ドライバ名］コンボボックスには，既存の BDE ドライバ名を入力するかまたはドロップダウンリストの既存のドライバ名を選択します。

メモ データベースプロパティエディタでは，BDE の接続パラメータを表示および設定したり，LoginPrompt プロパティと KeepConnection プロパティの状態を設定することもできます。接続パラメータについては，下記の「BDE エリアスのパラメータの設定」を参照してください。LoginPrompt については，21-4 ページの「サーバーログインの制御」を参照してください。KeepConnection については，24-15 ページの「TDatabase を使って接続を開く」を参照してください。

BDE エリアスのパラメータの設定

設計時には，以下の 3 通りの方法で接続パラメータを作成または編集できます。

- データベースエクスプローラまたは BDE Administrator を使って BDE エリアスとパラメータを作成または修正する。これらのユーティリティについての詳細は，それぞれのオンラインヘルプを参照してください。

- オブジェクトインスペクタの中で Params プロパティをダブルクリックして文字列リストエディタを起動する
- データモジュールまたはフォームの中でデータベースコンポーネントをダブルクリックしてデータベースプロパティエディタを起動する

これらの方法はすべて，データベースコンポーネントの Params プロパティを編集します。Params は，データベースコンポーネントに関連付けられた BDE エリアスのデータベース接続パラメータの文字列リストです。通常の接続パラメータには，パス名，サーバー名，スキーマキャッシングサイズ，言語ドライバ，SQL 問い合わせモードなどがあります。

データベースプロパティエディタを初めて起動したとき，BDE エリアスのパラメータは表示されません。現在の設定を表示するには，［デフォルト］をクリックします。［パラメータの変更］メモボックスに現在のパラメータが表示されます。既存のエントリを編集したり，新しいエントリを追加できます。既存のパラメータをクリアするには，［クリア］をクリックします。［OK］をクリックすると変更が有効になります。

実行時にアプリケーションがエリアスパラメータを設定するには，Params プロパティを直接編集しなければなりません。BDE で SQL Link を使用するためのパラメータについては，SQL Link のオンラインヘルプを参照してください。

TDatabase を使って接続を開く

データベース接続コンポーネントに共通する操作ですが，TDatabase を使ってデータベースに接続するには，Connected プロパティを **true** に設定するか，Open メソッドを呼び出します。この手順については，21-3 ページの「データベースサーバーへの接続」で述べます。いったん確立したデータベース接続は，アクティブなデータセットが少なくとも 1 つある限りは維持されます。アクティブなデータセットがなくなると，データベースコンポーネントの KeepConnection プロパティが **true** に設定されていない場合は，接続が切断されます。

アプリケーションでリモートデータベースサーバーに接続する場合，アプリケーションは，BDE と Borland SQL Link ドライバを使って接続を確立します（BDE は指定された ODBC ドライバとも通信できます）。接続する前に，SQL Link ドライバまたは ODBC ドライバをアプリケーション用に設定しなければなりません。SQL Link と ODBC の各パラメータは，データベースコンポーネントの Params プロパティに格納されます。SQL Link のパラメータについては，オンラインヘルプ SQLLNK32.HLP を参照してください。Params プロパティの編集については，24-14 ページの「BDE エリアスのパラメータの設定」を参照してください。

ネットワークプロトコルを操作する

SQL Link ドライバまたは ODBC ドライバを適切に設定する一環として，サーバーで使われるネットワークプロトコル（SPX/IPX や TCP/IP など）を指定しなければなりません。ほとんどの場合，プロトコルの環境設定は，サーバーのクライアントセットアップソフトウェアを使用して行われます。ODBC の場合は，ODBC ドライバマネージャを使って設定内容をチェックする必要があります。

サーバーとクライアント間の初期接続の確立には，問題が生じる場合があります。問題がある場合は，次のチェックリストをその解決に役立ててください。

- サーバークライアント側の接続が適切に設定されているか？

- 接続のための DLL とデータベースドライバに検索パスが通っているか？
- TCP/IP を使っている場合
  - TCP/IP 通信ソフトウェアをインストールしてあるか，また適切な WINSOCK.DLL をインストールしてあるか？
  - クライアントの HOSTS ファイルにサーバーの IP アドレスを登録してあるか？
  - ドメインネームサービス（DNS）を適切に設定してあるか？
  - サーバーの PING が可能か？

トラブルシューティングについては，SQL Link のオンラインヘルプ SQLLNK32.HLP および各サーバーのマニュアルを参照してください。

ODBC を使う

アプリケーションでは，Btrieve などの ODBC データソースを使えます。ODBC ドライバ接続には，以下の 3 つが必要です。

- ベンダー提供の ODBC ドライバ
- Microsoft ODBC ドライバマネージャ
- BDE 環境設定ユーティリティ

ODBC ドライバ接続の BDE エリアスの設定には BDE Administrator を使います。詳細については，BDE Administrator のオンラインヘルプファイルを参照してください。

データモジュールでデータベースコンポーネントを使う

データベースコンポーネントはデータモジュールに問題なく追加できます。ただし，データベースコンポーネントがあるデータモジュールをオブジェクトリポジトリに入れた場合，および他のユーザーがこれらのデータモジュールから継承できるようにしたい場合は，名前の衝突を避けるためにデータベースコンポーネントの HandleShared プロパティを **true** に設定してください。

データベースセッションの管理

BDE 対応アプリケーションのデータベース接続，ドライバ，カーソル，問い合わせなどは，1 つまたは複数の BDE セッションのコンテキスト内で管理されます。セッションはデータベース接続などの一連のデータベースアクセス処理を分離しますが，アプリケーションの別のインスタンスを開始する必要はありません。

すべての BDE ベースのデータベースアプリケーションには，デフォルトの BDE セッションをカプセル化する，Session というセッションコンポーネントが自動的に提供されます。アプリケーションにデータベースコンポーネントを追加すると，そのコンポーネントはデフォルトのセッションに自動的に関連付けられます（その SessionName は「Default」であることに注意してください）。デフォルトセッションでは，別のセッションに関連付けられていないすべてのデータベースコンポーネントを，それらが暗黙的か持続的かにかかわらず，グローバルに制御できます。暗黙的なデータベースコンポーネントは，開発者が作成するデータベースコンポーネントに関連付けられていないデータセットを開く実行時にデフォルトセッションによって作成されます。持続的なコンポーネントは，アプリケーションで明示的に作成されます。設計時にはデータモジュールまたはフォームにデフォルトセッ

ションは表示されませんが，実行時にはコードでデフォルトセッションのプロパティやメソッドにアクセスできます。

アプリケーションが以下のことをする必要がなければ，デフォルトセッションを使うためにコードを記述する必要はありません。

- デフォルトセッションを明示的にアクティブまたは非アクティブにして，このセッションのデータベースが開く機能を有効または無効にする
- デフォルトセッションのプロパティを変更する。暗黙的に生成されるデータベースコンポーネントのデフォルトのプロパティを指定する場合など
- デフォルトセッションのメソッドを実行する。データベース接続の管理（例 : ユーザーの操作に応じてデータベース接続を開くまたは閉じる）を実行する場合など
- デフォルトセッションのイベントに応答する。パスワード保護された Paradox テーブルや dBASE テーブルにアプリケーションがアクセスを試みる場合など
- Paradox のディレクトリ位置を設定する。NetFileDir プロパティを設定して，ネットワーク上の Paradox テーブルにアクセスする場合や，PrivateDir プロパティをローカルドライブに設定してパフォーマンスを向上させる場合など
- デフォルトセッションを使用するデータベースとデータセットの，データベース接続の環境状態を示す可能性のある BDE エリアスを管理する

データベースコンポーネントを設計時にアプリケーションに追加するか実行時に動的に作成する場合，明確にほかのセッションに関連付けないと，それらのデータベースコンポーネントは自動的にデフォルトセッションに関連付けられます。データベースコンポーネントに関連付けられていないデータセットを開こうとすると，自動的に次の処理が行われます。

- 実行時にそのデータセット用のデータベースコンポーネントが作成される
- デフォルトセッションに関連付けられる
- デフォルトセッションのプロパティに基づいてデータベースコンポーネントのいくつかの主なプロパティが初期化される。これらのプロパティの中でもっとも重要なのが KeepConnections です。KeepConnections は，アプリケーションがどのようなときにデータベース接続を維持または切断するかを指定します。

デフォルトのセッションは，そのままでほとんどのアプリケーションで変更することなく使うことができます。作成するデータベースコンポーネントを明示的に指定したセッションに関連付けなければならないのは，デフォルトセッションがすでに開いているデータベースに対して，そのコンポーネントが同時に問い合わせを実行する場合だけです。この場合，それぞれの問い合わせが，問い合わせ自身のセッション下で実行されなければなりません。マルチスレッドのデータベースアプリケーションではスレッドごとにセッションを持つため，この場合も複数のセッションが必要になります。

必要であればアプリケーションで追加セッションコンポーネントを作成できます。BDE ベースのデータベースアプリケーションには，すべてのセッションコンポーネントを管理できる，Sessions というセッションリストコンポーネントが自動的に提供されます。複数セッションの管理については，24-28 ページの「複数のセッションの管理」を参照してください。

セッションコンポーネントをデータモジュールに入れても問題はありません。ただし，セッションコンポーネントが 1 つでも含まれているデータモジュールをオブジェクトリポジトリに入れた場合，

ユーザーがそれから継承するときに名前が衝突しないように，AutoSessionName プロパティを **true** に設定してください。

セッションのアクティブ化

Active は論理型のプロパティであり，セッションに関連付けられたデータベースコンポーネントとデータセットコンポーネントが開いているかどうかを表します。このプロパティを使うと，セッションのデータベース接続とデータセット接続の現在の状態を読み出したり変更したりできます。Active が **false**（デフォルト）の場合，セッションに関連付けられたすべてのデータベースとデータセットは閉じています。**true** の場合，データベースとデータセットは開いています。

セッションは，最初に作成されるときと，その後で Active プロパティが **false** から **true** に変更されるたびにアクティブになります。（たとえば，セッションに関連づけられているデータベースまたはデータセットが開いていて，現在その他に開いているデータベースまたはデータセットがない場合）Active を **true** に設定すると，セッションの OnStartup イベントが発生し，Paradox のディレクトリ位置が BDE に登録され，そのセッション内で使用可能な BDE エリアスを決定する ConfigMode プロパティが登録されます。OnStartup イベントハンドラを記述すると，NetFileDir，PrivateDir，ConfigMode の各プロパティが登録される前に，それぞれを初期化することができます。OnStartup イベントハンドラには，このようなセッション開始時の特定の動作を指定できます。NetFileDir プロパティと PrivateDir プロパティについては，24-24 ページの「Paradox のディレクトリ位置の指定」を参照してください。ConfigMode については，24-24 ページの「BDE エリアスの操作」を参照してください。

セッションがアクティブになると，OpenDatabase メソッドを呼び出してそのセッションのデータベース接続を開くことができます。

データモジュールまたはフォームに入れるセッションコンポーネントの場合，開いているデータベースまたはデータセットがあるときに Active を **false** に設定すると，それらのデータベースやデータセットは閉じます。実行時にデータベースやデータセットを閉じると，それらに関連付けられているイベントが発生する場合があります。

メモ デフォルトセッションの Active を設計時に **false** に設定することはできません。実行時にデフォルトセッションを閉じることはできますが，できるだけ閉じないでください。

デフォルトセッション以外の場合，セッションの Open メソッドと Close メソッドを使っても，そのセッションを実行時にアクティブや非アクティブにできます。たとえば，次の 1 行のコードだけで，セッションで開いているすべてのデータベースとデータセットが閉じます。

```
Session1->Close();
```

このコードは Session1 の Active プロパティを **false** に設定します。セッションの Active プロパティが **false** の場合，その後でアプリケーションがデータベースまたはデータセットを開こうとすると，Active が **true** に再設定され，セッションの OnStartup イベントハンドラがある場合は，そのハンドラが呼び出されます。実行時のセッションの再アクティブ化を明示的に記述することもできます。次のコードは Session1 を再びアクティブにします。

```
Session1->Open();
```

メモ セッションがアクティブな場合，データベース接続を個別に開いたり閉じたりすることもできます。詳細については，24-20 ページの「データベース接続を閉じる」を参照してください。

デフォルトのデータベース接続動作の指定

KeepConnections は，実行時に作成される暗黙的なデータベースコンポーネントの KeepConnection プロパティのデフォルト値を与えます。KeepConnection は，データベースコンポーネント用に確立されたデータベース接続を，そのデータベース接続のすべてのデータセットが閉じられたときにどうするかを決めます。True（デフォルト）の場合，アクティブなデータセットがなくても持続的なデータベース接続が維持されます。**false** の場合，すべてのデータセットが閉じると，ただちにデータベース接続も切断されます。

メモ データモジュールまたはフォームに明示的に入れるデータベースコンポーネントの接続の持続性は，そのデータベースコンポーネントの KeepConnection プロパティによって制御されます。設定が異なる場合，データベースコンポーネントの KeepConnection がセッションの KeepConnections プロパティより常に優先します。セッション内のデータベース接続を個別に制御する方法については，24-19 ページの「データベース接続の管理」を参照してください。

リモートサーバー上のデータベースに関連付けられたすべてのデータセットを頻繁に開いたり閉じたりするアプリケーションの場合，KeepConnections を常に **true** に設定しておく必要があります。**true** の場合，セッションの存続中に接続を 1 度開いて閉じるだけですむので，ネットワークのトラフィックが減り，データアクセス速度が向上します。True でない場合は，接続を閉じたり再確立するたびに，データベースのアタッチおよびデタッチのオーバーヘッドが発生します。

メモ セッションの KeepConnections が **true** の場合でも，DropConnections メソッドを呼び出せば，すべての暗黙的データベースコンポーネントに関連する非アクティブなデータベース接続を閉じて解放することができます。DropConnections については，24-20 ページの「非アクティブなデータベース接続の切断」を参照してください。

データベース接続の管理

セッションコンポーネント内のデータセット接続は，セッションコンポーネントから管理できます。セッションコンポーネントのプロパティとメソッドを使って操作できる内容は，以下のとおりです。

- データベース接続を開く
- データベース接続を閉じる
- 非アクティブなすべての一時データベース接続を閉じて解放する
- 特定のデータベース接続を検索する
- 開いているすべてのデータベース接続に対して処理を繰り返す

データベース接続を開く

セッション内でデータベース接続を開くには，OpenDatabase メソッドを呼び出します。OpenDatabase は，開きたいデータベースの名前を 1 つパラメータとして取ります。この名前は BDE エリアスまたはデータベースコンポーネントの名前です。Paradox または dBASE では，完全に限定されたパス名の場合もあります。たとえば次の文は，デフォルトセッションを使い，BCDEMOS エリアスが指すデータベースのデータベース接続を開こうとします。

```
TDatabase *BCDemosDatabase = Session->OpenDatabase("BCDEMOS");
```

OpenDatabase は，セッションがまだアクティブでなければアクティブにしてから，指定されたデータベース名がセッションのいずれかのデータベースコンポーネントの DatabaseName プロパティと一致するかどうかを検査します。既存のデータベースコンポーネントに，指定された名前に一致するものがなかった場合，OpenDatabase は，その名前を使って一時データベースコンポーネントを作成し

ます。最後に，そのデータベースコンポーネントの Open メソッドを呼び出してサーバーに接続します。OpenDatabase を呼び出すと，そのデータベースの参照カウンタが 1 つ増えます。参照カウンタが 0 でない限り，データベースは開いたままです。

データベース接続を閉じる

データベース接続を個別に閉じるには，CloseDatabase メソッドを呼び出します。CloseDatabase を呼び出すと，OpenDatabase を呼び出したときに 1 つ増えていたデータベースの参照カウンタの値が，1 つ減ります。参照カウンタが 0 になると，データベースが閉じられます。CloseDatabase は，閉じるデータベースを 1 つパラメータとして取ります。OpenDatabase メソッドを使ってデータベースを開いていた場合は，そのメソッドの戻り値をこのパラメータに設定できます。

```
Session->CloseDatabase(BCDemosDatabase);
```

指定したデータベース名が一時的（暗黙的）なデータベースコンポーネントに関連付けられていて，セッションの KeepConnections プロパティが false の場合，その一時データベースコンポーネントは解放され，接続が完全に閉じます。

メモ KeepConnections が **false** の場合，一時データベースコンポーネントに関連付けられている最後のデータセットが閉じると，その一時データベースコンポーネントが自動的に閉じ，解放されます。アプリケーションは，この時点より前にいつでも CloseDatabase を呼び出して強制的に閉じることができます。KeepConnections が **true** の場合に一時データベースコンポーネントを解放するには，その一時データベースコンポーネントの Close メソッドを呼び出してから，セッションの DropConnections メソッドを呼び出します。

メモ 持続的なデータベースコンポーネントの CloseDatabase を呼び出しても，実際には接続は閉じません。接続を閉じるには，データベースコンポーネントの Close メソッドを直接呼び出します。

セッション内のすべてのデータベース接続を閉じるには，次の 2 通りの方法があります。

- セッションの Active プロパティを **false** に設定する
- セッションの Close メソッドを呼び出す

Active を **false** に設定すると，自動的に Close メソッドが呼び出されます。Close は，一時データベースコンポーネントを解放してから持続的な各データベースコンポーネントの Close メソッドを呼び出すことによって，アクティブなすべてのデータベースから切断します。最後に，Close がセッションの BDE ハンドルを NULL に設定します。

非アクティブなデータベース接続の切断

セッションの KeepConnections プロパティが **true**（デフォルト）の場合は，一時データベースコンポーネントが使っていたデータセットがすべて閉じても，一時データベースコンポーネントのデータベース接続が維持されます。DropConnections メソッドを呼び出すと，これらの接続を閉じて，セッションの非アクティブなすべての一時データベースコンポーネントを解放できます。たとえば次のコードは，デフォルトセッションの非アクティブな一時データベースコンポーネントをすべて解放します。

```
Session->DropConnections();
```

アクティブなデータセットが 1 つでもあれば，DropConnections を呼び出しても一時データベースコンポーネントは切断も解放もされずにアクティブを保ちます。このようなコンポーネントを解放するには Close を呼び出します。

データベース接続の検索

特定のデータベースコンポーネントがすでにセッションに関連付けられているかどうかを確認するには，セッションの FindDatabase メソッドを使います。FindDatabase は，検索するデータベースの名前を 1 つパラメータとして取ります。この名前は BDE エリアスまたはデータベースコンポーネントの名前です。Paradox または dBASE の場合は，完全に限定されたパス名でもかまいません。

FindDatabase は，一致するものが見つかった場合はそのデータベースコンポーネントを返し，それ以外の場合は NULL を返します。

次のコードは，BCDEMOS エリアスを使ってデフォルトセッションのデータベースコンポーネントを検索し，見つからなければデータベースコンポーネントを作成して開きます。

```
TDatabase *DB = Session->FindDatabase("BCDEMOS");
if ( !DB )                                  // セッションにデータベースがなければ
  DB = Session->OpenDatabase("BCDEMOS"); // 作成して開く
if (DB && DB->Connected)
{
  if (!DB->InTransaction)
  {
    DB->StartTransaction();
    ...
  }
}
```

データベースコンポーネントに対する処理の繰り返し

セッションコンポーネントの Databases と DatabaseCount という 2 つのプロパティを使うと，セッションに関連付けられているすべてのアクティブなデータベースコンポーネントに対して処理を繰り返し実行できます。

Databases は，セッションに関連付けられている，現在アクティブなすべてのデータベースコンポーネントの配列です。DataSetCount は，その配列中のデータセットの数です。セッションの存続期間中に接続を開くかまたは閉じると，Databases と DatabaseCount の値が変わります。たとえば，セッションの KeepConnections プロパティが **false** で，すべてのデータベースコンポーネントが実行時に必要に応じて作成される場合，データベースを開くたびに，DatabaseCount は 1 ずつ増えます。また，作成されたデータベースを閉じるたびに，DatabaseCount は 1 ずつ減ります。DatabaseCount がゼロの場合，そのセッションには現在アクティブなデータベースコンポーネントはありません。

次のサンプルコードは，デフォルトセッション内でアクティブな各データベースの KeepConnection プロパティを **true** に設定します。

```
if (Session->DatabaseCount > 0)
  for (int MaxDbCount = 0; MaxDbCount < Session->DatabaseCount; MaxDbCount++)
    Session->Databases[MaxDbCount]->KeepConnection = true;
```

パスワード保護された Paradox テーブルと dBASE テーブルの操作

セッションコンポーネントには，パスワード保護された Paradox テーブルと dBASE テーブルのパスワードを持たせることができます。セッションにパスワードを追加すると，パスワードで保護されたテーブルをアプリケーションで開くことができます。追加したパスワードをセッションから削除すると，再びそのパスワードを追加するまでは，パスワードを使用しているテーブルをアプリケーションで開くことはできません。

AddPassword メソッドを使用する

AddPassword メソッドは，アクセスにパスワードが必要な暗号化された Paradox テーブルや dBASE テーブルを開く前に，アプリケーションがセッションにパスワードを提供することを可能にします。パスワードをセッションに追加していない場合は，パスワード保護されたテーブルをアプリケーションが開こうとしたときに，ユーザーに対してパスワード入力を求めるダイアログボックスが表示されます。

AddPassword は，パラメータとしてパスワードの文字列をとります。AddPassword は必要なだけ何回でも呼び出し，パスワードを（一度に 1 つずつ）追加していくことによって，別々のパスワードで保護された複数のテーブルにアクセスできます。

```
AnsiString PassWrd;
PassWrd = InputBox("Enter password", "Password:", "");
Session->AddPassword(Passwrd);
try
{
  Table1->Open();
}
catch (...)
{
  ShowMessage('Could not open table!');
  Application->Terminate();
}
```

メモ 上記の InputBox 関数の使用は，デモンストレーションだけを目的としています。実際のアプリケーションでは，PasswordDialog 関数やカスタムフォームなど，入力と同時にパスワードを隠すパスワード入力機能を使用します。

PasswordDialog 関数ダイアログの Add ボタンには，AddPassword メソッドと同じ効果があります。

```
if (PasswordDialog(Session))
  Table1->Open();
else
  ShowMessage("No password given, could not open table!");
```

RemovePassword および RemoveAllPasswords メソッドを使用する

RemovePassword はそれまでに追加されたパスワードの 1 つをメモリから削除します。
RemovePassword は，パラメータとして削除するパスワードの文字列をとります。

```
Session->RemovePassword("secret");
```

RemoveAllPasswords は，それまでに追加されたすべてのパスワードをメモリから削除します。

```
Session->RemoveAllPasswords();
```

GetPassword メソッドと OnPassword イベントを使用する

OnPassword イベントを利用すると，Paradox テーブルと dBASE テーブルのパスワードが要求されたときに，そのパスワードをアプリケーションからどのように提示するかを制御できます。デフォルトのパスワード処理動作をオーバーライドしたい場合は，OnPassword イベントのハンドラを作成します。このハンドラを作成しなければ，パスワード入力用のデフォルトのダイアログが表示されるだけで，特別な動作は行われません。テーブルを開く試みが成功するか，または例外が送出されるだけです。

OnPassword イベントのハンドラを作成する場合は，そのイベントハンドラで次の 2 つのことを行ってください。1 つは AddPassword メソッドを呼び出すこと，もう 1 つは，イベントハンドラの

Continue パラメータを t **rue** に設定することです。AddPassword メソッドは，テーブルのパスワードとして使用する文字列をセッションに渡します。Continue パラメータは，テーブルを開く試みで，これ以上パスワード要求する必要がないことを C++Builder に伝えます。Continue のデフォルト値は **false** なので，明示的に **true** に設定する必要があります。イベントハンドラが実行を完了した後に Continue が **false** である場合，OnPassword イベントが再び発生します。AddPassword を使って有効なパスワードを渡しておいた場合も同様です。イベントハンドラが実行を完了した後に Continue が **true** でも，AddPassword を使って渡した文字列が無効なパスワードであれば，テーブルを開く試みは失敗し，例外が生成されます。

OnPassword イベントは，次の 2 つの状況で発生します。1 つは，有効なパスワードがセッションに渡されていない状態で，パスワード保護されたテーブル（dBASE または Paradox）を開こうとしたときに発生します。（対象テーブルの有効なパスワードが設定してあれば，OnPassword イベントは発生しません）。

もう 1 つは，GetPassword メソッドの呼び出しがあったときです。GetPassword は，セッションに OnPassword イベントハンドラが設定されている場合は OnPassword イベントを生成し，設定されていない場合はデフォルトのパスワードダイアログを表示します。このメソッドの戻り値は，OnPassword イベントハンドラまたはデフォルトのダイアログでパスワードがセッションに追加された場合は **true**，追加されなかった場合は **false** になります。

次の例では，Password メソッドがデフォルトセッションの OnPassword イベントハンドラに指定されます。この処理は，グローバルな Session オブジェクトの OnPassword プロパティに Password メソッドを代入することによって行われます。

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Session->OnPassword = Password;
}
```

Password メソッドでは，InputBox 関数を使ってユーザーにパスワードを要求します。ダイアログに入力されたパスワードは，プログラム上で AddPassword メソッドを使ってセッションに渡します。

```
void __fastcall TForm1::Password(TObject *Sender, bool &Continue)
{
  AnsiString PassWrd = InputBox("Enter password", "Password:", "");
  Session->AddPassword(PassWrd);
  Continue = (PassWrd > "");
}
```

下に示すように，パスワード保護されたテーブルを開こうとすると，OnPassword イベントが発生します（よってイベントハンドラの Password が実行される）。OnPassword イベント用のハンドラでユーザーがパスワードの入力を求められたとしても，無効なパスワードを指定するなどの不都合が発生すると，テーブルを開けないことがあります。

```
void __fastcall TForm1::OpenTableBtnClick(TObject *Sender)
{
  try
  {
    // この行によって OnPassword イベントが発生する
    Table1->Open();
  }
  // テーブルを開けない場合は，例外
  catch (...)
```

```
  {
    ShowMessage("Could not open table!");
    Application->Terminate();
  }
}
```

Paradox のディレクトリ位置の指定

セッションコンポーネントの NetFileDir と PrivateDir という 2 つのプロパティは，Paradox テーブルを扱うアプリケーション専用です。

NetFileDir は，Paradox のネットワークコントロールファイル PDOXUSRS.NET を格納するディレクトリを指定します。PDOXUSRS.NET は，ネットワークドライブ上での Paradox テーブルの共有を管理するファイルです。Paradox テーブルの共有が必要なすべてのアプリケーションが，ネットワークコントロールファイルの同一のディレクトリ（通常はネットワークファイルサーバー上のディレクトリ）を指定しなければなりません。データベースの特定のエリアスについては，ボーランドデータベースエンジン（BDE）の環境設定ファイルから NetFileDir の値が生成されます。開発者が NetFileDir を設定する場合，その値が BDE の環境設定の値に優先するので，正しい値かどうか必ず確認してください。

設計時には，NetFileDir の値をオブジェクトインスペクタで指定できます。実行時にコードで NetFileDir を設定したり変更することもできます。次のコードは，アプリケーションの起動ディレクトリ位置をデフォルトセッションの NetFileDir に設定します。

```
Session->NetFileDir = ExtractFilePath(ParamStr(0));
```

メモ NetFileDir を変更できるのは，アプリケーションで開いている Paradox ファイルがない場合だけです。実行時に NetFileDir を変更する場合は，その値が，ネットワークユーザーが共有する正しいネットワークディレクトリを指しているかどうか確認してください。

PrivateDir は，ローカル SQL 文を処理するために BDE が生成するファイルなどの一時テーブル処理ファイルを格納するディレクトリを指定します。PrivateDir プロパティに値が指定されていない場合，BDE は初期化時に自動的にカレントディレクトリを使います。アプリケーションをネットワークファイルサーバーから直接実行する場合は，データベースを開く前に PrivateDir をローカルドライブに設定することにより実行時のパフォーマンスを向上できます。

メモ 設計時には PrivateDir を設定しないで IDE でセッションを開いてください。PrivateDir が設定されていると，IDE からアプリケーションを実行したときにディレクトリが使用中であるというエラーが生成されます。

次のコードは，デフォルトセッションの PrivateDir プロパティの設定をユーザーの C: \ TEMP ディレクトリに変更します。

```
Session->PrivateDir = "C:\\TEMP";
```

重要 PrivateDir はドライブ上のルートディレクトリに設定しないでください。常にサブディレクトリを指定してください。

BDE エリアスの操作

セッションに関連付けられたデータベースコンポーネントは，それぞれ BDE エリアスを持ちます（ただし，Paradox や dBASE のテーブルにアクセスする場合は，必要であれば，エリアスのかわりに完全に限定されたパス名が使われることがあります）。セッションは，存続期間中にエリアスを作成，変更，削除できます。

AddAlias メソッドは SQL データベースサーバーの新しい BDE エリアスを作成します。AddAlias はパラメータを 3 つ取ります。エリアスの名前を含む文字列，使用する SQL Link ドライバを指定する文字列，およびエリアスのパラメータが入る文字列リストの 3 つです。たとえば次の文は，AddAlias を使用して InterBase サーバーにアクセスするための新しいエリアスをデフォルトセッションに追加します。

```
TStringList *AliasParams = new TStringList();
try
{
  AliasParams->Add("OPEN MODE=READ");
  AliasParams->Add("USER NAME=TOMSTOPPARD");
  AliasParams->Add("SERVER NAME=ANIMALS:/CATS/PEDIGREE.GDB");
  Session->AddAlias("CATS", "INTRBASE", AliasParams);
  ...
}
catch (...)
{
  delete AliasParams;
  throw;
}
delete AliasParams;
```

AddStandardAlias は，Paradox，dBASE，または ASCII の各テーブルの新しい BDE エリアスを作成します。AddStandardAlias が取る文字列パラメータは，エリアス名，アクセスする Paradox テーブルまたは dBASE テーブルへの完全に限定されたパス，および拡張子を持たないテーブルを開こうとするときに使うデフォルトドライバの名前の 3 つです。たとえば次の文は，AddStandardAlias を使用して Paradox テーブルにアクセスするための新しいエリアスを作成します。

```
Session->AddStandardAlias("MYBCDEMOS", "C:\\TESTING\\DEMOS\\", "Paradox");
```

セッションにエリアスを追加すると BDE によってそのコピーがメモリに格納されますが，そのエリアスを使えるのは，追加したセッションと，ほかのセッションではその ConfigMode プロパティに cfmPersistent が含まれているものに限られます。ConfigMode は，そのセッション内のデータベースで使用可能なエリアスの種類を示します。デフォルトは cmAll です。これは集合 [cfmVirtual, cfmPersistent, cfmSession] に変換されます。ConfigMode が cmAll の場合，そのセッションから参照できるのは，セッション内で作成されたすべてのエリアス（cfmSession），ユーザーのシステム上の BDE 環境設定ファイルにあるすべてのエリアス（cfmPersistent），および BDE がメモリ内に保持するすべてのエリアス（cfmVirtual）です。ConfigMode を変更すると，セッション内のデータベースが使用可能な BDE エリアスを制限することができます。たとえば，ConfigMode を cfmSession に設定すると，セッションはセッション内で作成されるエリアスしか見えなくなります。BDE 環境設定ファイル内とメモリ内にあるその他のすべてのエリアスが使えなくなります。

新規に作成したエリアスをすべてのセッションとほかのアプリケーションが使えるようにするには，セッションの SaveConfigFile メソッドを使います。SaveConfigFile は，メモリ内のエリアスを BDE 環境設定ファイルに書き込みます。BDE 対応のほかのアプリケーションは，この BDE 環境設定ファイルからエリアスを読み出して使えます。

エリアスを作成した後，ModifyAlias を呼び出すとエリアスのパラメータを変更できます。ModifyAlias は 2 つのパラメータを取ります。変更するエリアスの名前と，変更するパラメータ値を含む文字列リストの 2 つです。たとえば次の文は，ModifyAlias を使用してデフォルトセッション内で CATS エリアスの OPEN MODE パラメータを READ/WRITE に変更します。

```
TStringList *List = new TStringList();
List->Clear();
List->Add("OPEN MODE=READ/WRITE");
Session->ModifyAlias("CATS", List);
delete List;
```

これまでにセッション内に作成したエリアスを削除するには，DeleteAlias メソッドを呼び出します。DeleteAlias は，削除するエリアスの名前を 1 つパラメータとして取ります。DeleteAlias を使うと，セッションがそのエリアスを使えなくなります。

メモ SaveConfigFile を呼び出して BDE 環境設定ファイルに書き込んであるエリアスは，DeleteAlias を実行しても環境設定ファイルから削除されません。エリアスを環境設定ファイルから削除するには，DeleteAlias を呼び出した後でもう一度 SaveConfigFile を呼び出します。

セッションコンポーネントの以下の 5 つのメソッドを使うと，パラメータ情報とドライバ情報も含めて，BDE のエリアスについての情報を取り出せます。

- GetAliasNames：セッションがアクセスできるエリアスのリストを返す
- GetAliasParams：指定したエリアスのパラメータのリストを返す
- GetAliasDriverName：エリアスが使う BDE ドライバの名前を返す
- GetDriverNames：セッションが利用できるすべての BDE ドライバのリストを返す
- GetDriverParams：指定したドライバのドライバパラメータを返す

セッションの情報を取得するメソッドの使い方の詳細については，下記の「セッションについての情報の取り出し」を参照してください。BDE エリアス，およびエリアスを扱う SQL リンクドライバについては，BDE のオンラインヘルプ BDE32.HLP を参照してください。

セッションについての情報の取り出し

セッションの情報用メソッドを使うと，セッションとセッションのデータベースコンポーネントについての情報を取り出せます。たとえば，セッションで既知のすべてのエリアスの名前を取り出すメソッドや，セッションが使う特定のデータベースコンポーネントに関連付けられたテーブルの名前を取り出すメソッドがあります。表 24.4 に，セッションコンポーネントの情報用メソッドの概要を示します。

表 24.4 セッションコンポーネントのデータベース関連情報メソッド

メソッド	目的
GetAliasDriverName	データベースの特定のエリアスの BDE ドライバを取り出す
GetAliasNames	データベースの BDE エリアスのリストを取り出す
GetAliasParams	データベースの特定の BDE エリアスのパラメータリストを取り出す
GetConfigParams	BDE 環境設定ファイルから環境設定情報を取り出す
GetDatabaseNames	BDE エリアスのリストと現在使われているすべての TDatabase コンポーネントの名前を取り出す
GetDriverNames	現在インストールされているすべての BDE ドライバの名前を取り出す
GetDriverParams	特定の BDE ドライバのパラメータリストを取り出す
GetStoredProcNames	特定のデータベースのすべてのストアドプロシージャの名前を取り出す
GetTableNames	特定のデータベースの特定のパターンに一致するすべてのテーブルの名前を取り出す
GetFieldNames	特定のデータベースの特定のテーブル内のすべての項目の名前を取り出す

GetAliasDriverName を除き，これらのメソッドは，アプリケーションが宣言して管理する文字列リストに値のセットを返します (GetAliasDriverName は，セッションが使っている特定のデータベースコンポーネントの現在の BDE ドライバの名前を 1 つの文字列で返します)。

たとえば次のコードは，すべてのデータベースコンポーネント名と，デフォルトセッションで既知のエリアスを取り出します。

```
TStringList *List = new TStringList();
try
{
  Session->GetDatabaseNames(List);
  ...
}
catch (...)
{
  delete List;
  throw;
}
delete List;
```

セッションの追加

デフォルトセッションを補足するセッションを作成できます。設計時にデータモジュール（またはフォーム）にセッションを追加します。オブジェクトインスペクタで追加したセッションのプロパティを設定し，イベントハンドラを記述し，追加したセッションのメソッドを呼び出すコードを記述できます。実行時にセッションを作成し，セッションのプロパティを設定して，セッションのメソッドを呼び出すこともできます。

メモ アプリケーションが 1 つのデータベースに対して同時に問い合わせを実行しない場合，またはアプリケーションがマルチスレッドでない場合，セッションを追加するかどうかは任意です。

セッションコンポーネントを実行時に動的に作成し追加する手順は次のとおりです。

1. TSession 変数を宣言します。
2. 新しいセッションをインスタンス化するには，**new** 演算子を使います。この演算子は TSession コンストラクタを呼び出して，新しいセッションを作成し，初期化します。コンストラクタによって，セッションのデータベースコンポーネントの空のリストが設定され，KeepConnections プロパティが **true** に設定され，アプリケーションのセッションリストコンポーネントで管理するセッションリストにセッションが追加されます。
3. 新しいセッションの SessionName プロパティにユニークな名前を設定します。このプロパティは，データベースコンポーネントをセッションに関連付けるのに使われます。SessionName プロパティの詳細については，24-28 ページの「セッション名の指定」を参照してください。
4. セッションをアクティブにし，必要であればセッションのプロパティを調整します。

セッションを作成したり開いたりする処理は，TSessionList の OpenSession メソッドを使っても実行できます。OpenSession を使うと，まだセッションが存在していない場合にだけセッションが作成されるため，OpenSession を使う方が **new** 演算子を使うよりも安全です。OpenSession については，24-28 ページの「複数のセッションの管理」を参照してください。

次のコードは新しいセッションコンポーネントを作成して名前を割り当て，それに続くデータベース処理のためにセッションを開きます。使用後に，セッションは Free メソッドの呼び出しによって破棄されます。

メモ デフォルトセッションは絶対に削除しないでください。

```
TSession *SecondSession = new TSession(Form1);
try
{
  SecondSession->SessionName = "SecondSession";
  SecondSession->KeepConnections = false;
  SecondSession->Open();
  ...
}
__finally
{
  delete SecondSession;
}
```

セッション名の指定

セッションの SessionName プロパティは，データベースとデータセットをセッションに関連付けられるようにセッションに名前を付けるのに使います。デフォルトセッションの SessionName は「Default」です。作成する追加セッションコンポーネントの SessionName プロパティには，それぞれユニークな値を設定しなければなりません。

データベースとデータセットの各コンポーネントには，セッションコンポーネントの SessionName プロパティに対応する SessionName プロパティがあります。データベースまたはデータセットコンポーネントの SessionName プロパティを空白にしておくと，それらのコンポーネントは自動的にデフォルトセッションに関連付けられます。データベースまたはデータセットコンポーネントの SessionName には，作成したセッションコンポーネントの SessionName に対応する名前を割り当てることもできます。

次のコードは，デフォルトの TSessionList コンポーネント Sessions の OpenSession メソッドを使って新しいセッションコンポーネントを開き，そのコンポーネントの SessionName を「InterBaseSession」に設定し，セッションをアクティブにして，既存のデータベースコンポーネント Database1 をこのセッションに関連付けます。

```
TSession *IBSession = Sessions->OpenSession("InterBaseSession");
Database1->SessionName = "InterBaseSession";
```

複数のセッションの管理

データベース処理を実行する複数のスレッドを使う単独のアプリケーションを作成する場合，スレッドごとに追加セッションを 1 つ作成しなければなりません。コンポーネントパレットの［BDE］ページには，設計時にデータモジュールまたはフォームに追加できるセッションコンポーネントがあります。

重要 セッションコンポーネントを追加する場合，デフォルトセッションの SessionName プロパティと衝突しないように，ユニークな値を SessionName プロパティに設定する必要もあります。

設計時に TSession コンポーネントを入れると，実行時にアプリケーションに必要になるスレッド（および，それによるセッション）の数が静的であると仮定されます。しかし，アプリケーションがセッションを動的に作成しなければならないことがよくあります。セッションを動的に作成するには，実行時にグローバルオブジェクト Sessions の OpenSession メソッドを呼び出します。

OpenSession には，アプリケーションのすべてのセッション名でユニークなセッション名のパラメータが 1 つ必要です。次のコードは，ユニークに生成される名前を持つ新しいセッションを動的に作成してアクティブにします。

```
Sessions->OpenSession("RunTimeSession" + IntToStr(Sessions->Count + 1));
```

この文では，現在のセッション数を取り出して，その値に 1 を足すことによって，新しいセッションのユニークな名前を生成しています。実行時にセッションを動的に作成して破棄する場合，このコード例は期待どおりに動作しないことに注意してください。この例は，複数のセッションを管理するために Sessions のプロパティとメソッドをどう使うかを示したものです。

Sessions は TSessionList 型の変数で，BDE ベースのデータベースアプリケーション用に自動的にインスタンス化されます。Sessions のプロパティとメソッドを使ってマルチスレッドデータベースアプリケーション内の複数のセッションを管理します。TSessionList コンポーネントのプロパティとメソッドの概要を表 24.5 に示します。

表 24.5　TSessionList のプロパティとメソッド

プロパティ / メソッド	目的
Count	セッションリスト内のセッション数（アクティブと非アクティブの両方）を返す
FindSession メソッド	指定された名前のセッションを検索し，そのセッションを指すポインタを返す。指定された名前のセッションがない場合は NULL を返す。空白のセッション名を渡した場合は，デフォルトセッション Session を返す
GetSessionNames メソッド	現在インスタンス化されているすべてのセッションコンポーネントの名前の文字列リストを返す。最低でも 1 つの文字列（デフォルトセッションの「Default」）が必ず追加される。
List プロパティ	指定されたセッション名のセッションコンポーネントを返す。その名前のセッションがない場合，例外が生成される
OpenSession メソッド	指定されたセッション名で新しいセッションを作成してアクティブにするか，その名前の既存のセッションを再びアクティブにする
Sessions プロパティ	順序値でセッションリストにアクセスする

マルチスレッドアプリケーションでの Sessions のプロパティとメソッドの例として，データベース接続を開くときの動作について考えます。接続がすでに存在するかどうかを確認するには，Sessions プロパティを使って，セッションリストにある各セッションをデフォルトセッションから順にたどっていきます。セッションコンポーネントごとに，Databases プロパティを調べて，問題のデータベースが開いているかを確認します。必要なデータベースをすでにほかのスレッドが使っている場合，リスト内の次のセッションを調べます。

既存のスレッドがデータベースを使っていない場合，そのセッション内で接続を開けます。

一方，既存のすべてのスレッドがデータベースを使っている場合は，別のデータベース接続を開くための新しいセッションを開かなければなりません。

それぞれのスレッドがデータモジュールの独自のコピーを含むマルチスレッドアプリケーション内の 1 つのセッションを含むデータモジュールをレプリケートする場合は，AutoSessionName プロパティを使ってデータモジュール内のすべてのデータセットが確実に正しいセッションを使うようにすることができます。AutoSessionName を **true** に設定すると，実行時に作成されるときにそれ自体のユニークな名前が動的に生成されます。次にこの名前がデータモジュール内のすべてのデータセットに割り当てられ，明示的に設定されたセッション名がすべてオーバーライドされます。これにより，そ

れぞれのスレッドがそれ自身のセッションを持ち，それぞれのデータセットがそれ自身のデータモジュールの中のセッションを確実に使用できます。

BDE でトランザクションを使用する

デフォルトでは，BDE はアプリケーションに暗黙のトランザクション制御を提供します。アプリケーションが暗黙のトランザクション制御の下にある場合，基になったデータベースに書き込まれるデータセットのレコードごとに別々のトランザクションが使われます。暗黙のトランザクション制御を使うとレコード更新の衝突が最小限になり，データベースのビューの一貫性が保証されます。しかし，データベースへの各データ行の書き込みがそれぞれのトランザクション内で行われるため，ネットワークトラフィックが過大になったり，アプリケーションの処理効率が低下することがあります。また，暗黙のトランザクション制御では，複数のレコードにわたる論理演算は保護されません。

明示的なトランザクション制御では，トランザクションを開始，コミット，ロールバックするのにもっとも効果的なタイミングを選択できます。マルチユーザー環境のアプリケーションを開発するとき，特にアプリケーションが SQL サーバーに対して実行されるときは，トランザクションを明示的に制御しなければなりません。

BDE ベースのデータベースアプリケーションでは，以下の 2 通りのどちらかの方法でトランザクションを明示的に制御できます。

- データベースコンポーネントを使ってトランザクションを制御する。データベースコンポーネントのメソッドとプロパティを使う一番の利点は，特定のデータベースまたはサーバーに依存しない，簡潔で移植性のあるアプリケーションを作成できることです。この方法を使ってトランザクションを制御する機能は，すべてのデータベース接続コンポーネントでサポートされています。その内容については，21-6 ページの「トランザクションの管理」で説明しています。
- 問い合わせコンポーネントでパススルー SQL を使用することによって SQL 文を直接リモート SQL または ODBC サーバーに渡す。パススルー SQL を使う一番の利点は，スキーマキャッシングなど，特定のデータベースサーバーの高度なトランザクション管理機能を使えることです。サーバーのトランザクション管理モデルの利点については，各データベースサーバーのマニュアルを参照してください。パススルー SQL の使い方については，下記の「パススルー SQL を使用する」を参照してください。

ローカルデータベースを扱う場合には，データベースコンポーネントを使用する方法でのみ明示的なトランザクションを作成できます（ローカルデータベースではパススルー SQL がサポートされていません）。ただし，ローカルトランザクションの使用には制限があります。ローカルトランザクションの使い方については，24-31 ページの「ローカルトランザクションを使用する」を参照してください。

メモ キャッシュアップデートを利用すると，必要なトランザクションの数を最小限に抑えられます。キャッシュアップデートについての詳細は，「クライアントデータセットをキャッシュアップデートに使用する」および 24-32 ページの「BDE を使ってキャッシュアップデートを実行する」を参照してください。

パススルー SQL を使用する

パススルー SQL を使ってリモートデータベースサーバーに SQL トランザクション制御文を直接送信するには，TQuery，TStoredProc，TUpdateSQL のいずれかのコンポーネントを使います。BDE は SQL 文を処理しません。パススルー SQL を使うと，サーバーのトランザクション制御が標準的でない場合は特に，それらのトランザクション制御の利点を直接利用できます。

トランザクションを制御するためにパススルー SQL を使うための必要条件を以下に示します。

- 適切な SQL Link ドライバをインストールする。「デフォルト」インストールを選択して C++Builder をインストールした場合，すべての SQL Link ドライバがすでに正しくインストールされている
- ネットワークプロトコルを設定する。詳細については，ネットワーク管理者に問い合わせてください
- リモートサーバーのデータベースにアクセスする
- SQL エクスプローラを使って SQLPASSTHRUMODE を NOT SHARED に設定する。SQLPASSTHRUMODE には，BDE とパススルー SQL 文が同一のデータベース接続を共有できるかどうかを指定する。ほとんどの場合，SQLPASSTHRUMODE は SHARED AUTOCOMMIT に設定されます。ただし，トランザクション制御文を使うときはデータベース接続を共用できない。SQLPASSTHRUMODE についての詳細は，BDE Administrator のヘルプファイルを参照

メモ SQLPASSTHRUMODE が NOT SHARED の場合は，SQL トランザクション文をサーバーに渡すデータセットと，渡さないデータセットに，それぞれ別のデータベースコンポーネントを使わなければなりません。

ローカルトランザクションを使用する

BDE は，Paradox，dBASE，Access，および FoxPro のテーブルに対するローカルトランザクションをサポートしています。コーディングの面では，ローカルトランザクションでもリモートデータベースサーバーにあるトランザクションでも違いはありません。

メモ Paradox，dBASE，Access，および FoxPro のローカルテーブルでトランザクションを使う場合，TransIsolation はデフォルトの tiReadCommitted ではなく tiDirtyRead に設定してください。ローカルテーブルで TransIsolation が tiDirtyRead 以外に設定されると BDE エラーが返されます。

ローカルテーブルに対してトランザクションを開始すると，そのテーブルに対して実行された更新がログに記録されます。ログの各レコードには，レコードの以前のレコードバッファが記録されます。トランザクションがアクティブな場合，更新されるレコードは，そのトランザクションがコミットされるかロールバックされるまでロックされます。ロールバック時には，更新したレコードに対して以前のレコードバッファが使われ，レコードが更新前の状態に復元されます。

ローカルトランザクションは，SQL サーバーまたは ODBC ドライバに対するトランザクションよりも制限があります。特に，ローカルトランザクションには以下の制限があります。

- クラッシュの自動復元はない
- データ定義文はサポートされていない

- 一時テーブルに対してはトランザクションを実行できない
- TransIsolation レベルは tiDirtyRead 以外に設定してはならない
- Paradox の場合，ローカルトランザクションを実行できるのは，有効なインデックスがあるテーブルに対してだけである。インデックスがない Paradox テーブルでは，データをロールバックできない
- 少数のレコードしかロックおよび変更できない。Paradox テーブルでは 255 レコードまで，dBASE テーブルでは 100 レコード までに限られる
- BDE ASCII ドライバに対してはトランザクションを実行できない
- トランザクション中にテーブル上でカーソルを閉じると，以下の場合を除いて，トランザクションはロールバックされる
  - 複数のテーブルを開いているとき
  - 変更しなかったテーブル上でカーソルを閉じるとき

BDE を使ってキャッシュアップデートを実行する

キャッシュアップデートを行う方法として推奨するのは，クライアントデータセット（TBDEClientDataSet）を使う方法と，データセットプロバイダを使って BDE データセットをクライアントデータセットに接続する方法です。クライアントデータセットを使った場合の利点については，27-15 ページの「クライアントデータセットをキャッシュアップデートに使用する」で説明しています。

ただし，単純な場合には，BDE を使ってキャッシュアップデートを行うことができます。BDE 対応データセットと TDatabase コンポーネントには，キャッシュアップデートを扱うためのプロパティ，メソッド，イベントが組み込まれています。これらのほとんどは，キャッシュアップデートをクライアントデータセットで行う場合にクライアントデータセットとデータセットプロバイダで使用するプロパティ，メソッド，およびイベントに 1 対 1 で対応しています。次の表に，これらのプロパティ，イベント，メソッドと，TBDEClientDataSet のプロパティ，イベント，メソッドの一覧を示します。

表 24.6　キャッシュアップデートのプロパティ，イベント，メソッド

BDE 対応データセット (または TDatabase)	TBDEClientDataSet	目的
CachedUpdates	クライアントデータセットの場合は不要。更新内容は常にキャッシュされる	当該データセットについてキャッシュアップデートが有効かどうか調べる
UpdateObject	BeforeUpdateRecord イベントハンドラを使用する。TClientDataSet を使う場合は，BDE 対応のソースデータセットの UpdateObject プロパティを使用する	読み出し専用データセットを更新するためのアップデートオブジェクトを指定する
UpdatesPending	ChangeCount	データベースに適用しなければならない更新レコードがローカルキャッシュ内にあるかどうか示す

表 24.6 キャッシュアップデートのプロパティ，イベント，メソッド（つづき）

BDE 対応データセット (または TDatabase)	TBDEClientDataSet	目的
UpdateRecordTypes	StatusFilter	キャッシュアップデートを適用するときに，見えるようにする更新レコードの種類を示す
UpdateStatus	UpdateStatus	レコードが変更されていない，変更された，挿入された，または削除されたかどうかを示す
OnUpdateError	OnReconcileError	更新エラーを処理するためのイベント（レコード単位）
OnUpdateRecord	BeforeUpdateRecord	更新を処理するためのイベント（レコード単位）
ApplyUpdates ApplyUpdates（データベース）	ApplyUpdates	ローカルキャッシュ内のレコードをデータベースに適用する
CancelUpdates	CancelUpdates	すべての未処理の更新を適用しないでローカルキャッシュから削除する
CommitUpdates	Reconcile	更新の適用が成功した後，アップデートキャッシュをクリアする
FetchAll	GetNextPacket（および PacketRecords）	データベースレコードを編集および更新のためにローカルキャッシュにコピーする
RevertRecord	RevertRecord	更新がまだ適用されていなければ，現在のレコードへの更新内容を元に戻す

キャッシュアップデート処理の概要については，27-16 ページの「キャッシュアップデートの使い方の概要」を参照してください。

メモ クライアントデータセットを使ってキャッシュアップデートを行う場合も，24-39 ページ のアップデートオブジェクトの節が参考になることがあります。ストアドプロシージャまたは複数テーブルの問い合わせを使って更新を適用する場合は，アップデートオブジェクトを，TBDEClientDataSet または TDataSetProvider の BeforeUpdateRecord イベントハンドラ内で使用することができます。

BDE ベースのキャッシュアップデートを有効にする

キャッシュアップデートに BDE を使用するには，BDE 対応データセットがキャッシュアップデートを行うことを，そのデータセットで示す必要があります。この処理は，CachedUpdates プロパティを True に設定することで行います。キャッシュアップデートを使用可能にすると，すべてのレコードのコピーがローカルメモリにキャッシュされます。ユーザーはデータのこのローカルコピーを表示し，編集します。変更，挿入，削除もメモリにキャッシュされます。この編集内容は，アプリケーションによってデータベースサーバーに適用されるまで，メモリ内に蓄積されます。変更したレコードがデータベースに正しく適用されると，これらの変更のレコードがキャッシュから解放されます。

データセットに変更内容をキャッシュする処理は，CachedUpdates を **false** に設定するまで続きます。キャッシュしていた変更内容を適用しても，その時点での変更内容がデータベースに書き込まれメモリからクリアされるだけで，キャッシュアップデートの処理はその後も続けられます。
CancelUpdates を呼び出して変更を取り消すと，その時点でキャッシュ内にある変更内容はすべて削除されますが，その後の変更内容は引き続きデータセットによってキャッシュされます。

メモ CachedUpdates を false に設定してキャッシュアップデートを使用不可にすると，まだ適用していない保留中の変更は破棄され，通知も発行されません。変更内容が消失してしまうことを防ぐには，キャッシュアップデートを使用不可にする前に UpdatesPending プロパティを検査します。

BDE ベースのキャッシュアップデートを適用する

アップデートを適用する処理は，アプリケーションがエラーから適切に回復できるように，データベースのトランザクションのコンテキストの下で 2 段階の処理に分けて行われます。データベースコンポーネントを使ったトランザクション処理については，21-6 ページの「トランザクションの管理」を参照してください。

トランザクション制御の下で更新を行った場合，以下の処理が行われます。

1. データベーストランザクションを開始する。
2. キャッシュされている更新レコードがデータベースに書き込まれる（第 1 段階）。OnUpdateRecord イベントが用意されていれば，データベースに書き込むレコードごとに 1 回ずつ OnUpdateRecord イベントが発生します。レコードをデータベースに適用するときにエラーが発生した場合は，OnUpdateError イベントが用意されていれば OnUpdateError イベントが発生します。
3. トランザクションは，書き込みが正常に完了した場合はコミットされ，失敗した場合はロールバックされる。

   データベースの書き込みに成功すると，以下の段階に移行します。

   - データベースの変更がコミットされ，データベーストランザクションが終了する
   - キャッシュアップデートがコミットされ，内部キャッシュバッファがクリアされる（第 2 段階）

   データベースの書き込みに失敗すると，以下の段階に移行します。

   - データベースの変更がロールバックされ，データベーストランザクションが終了する
   - キャッシュアップデートはコミットされず，そのまま内部キャッシュに残る

OnUpdateRecord イベントハンドラの作成とその使い方については，24-36 ページの「OnUpdateRecord イベントハンドラの作成」を参照してください。キャッシュアップデートを適用するときに発生する更新エラーの処理については，24-38 ページの「キャッシュアップデートエラーの処理」を参照してください。

メモ キャッシュアップデートを適用する処理は，マスター / 詳細関係を持つ複数のデータセットを扱う場合には特に注意を要します。これは，各データセットに更新を適用する順序が問題になるためです。削除レコードを処理する場合以外は，詳細テーブルより先にマスターテーブルを更新しなければなりません（削除レコードの場合はこの順序が守られるため問題にはなりません）。このような制約があるため，マスター / 詳細フォームでキャッシュアップデートを行う場合は，クライアントデータセットの使用を強くお勧めします。クライアントデータセットでは，マスター / 詳細関係に伴う順序の問題がすべて自動的に処理されます。

BDE ベースで更新を適用するには，2 通りの方法があります。

- データベースコンポーネントを使用し，その ApplyUpdates メソッドを呼び出すことによって更新を適用する。これはもっとも簡単な方法です。この場合，更新プロセスのトランザクションを管

理し更新の終了時にデータセットのキャッシュをクリアする処理が，すべてデータベースによって行われます。

- 単一データセットの更新を適用する場合は，そのデータセットの ApplyUpdates メソッドと CommitUpdates メソッドを呼び出して行う。データセットレベルで更新を適用する場合は，その更新処理をまとめたトランザクションを明示的にコードで表すと同時に，CommitUpdates を明示的に呼び出してキャッシュ内の更新内容をコミットする必要があります。

重要 ライブ結果セットを返さないストアドプロシージャまたは SQL 問い合わせからの更新を適用するには，TUpdateSQL を使って更新の実行方法を指定しなければなりません。結合（複数のテーブルが関係する問い合わせ）を更新する場合は，関係するテーブルごとに TUpdateSQL オブジェクトを 1 つずつ準備し，OnUpdateRecord イベントハンドラを使ってこれらのオブジェクトを呼び出して，更新を行わなければなりません。詳しくは，24-39 ページの「アップデートオブジェクトを使ったデータセットの更新」を参照してください。

データベースを使ってキャッシュアップデートを適用する

キャッシュアップデートをデータベース接続のコンテキストで 1 つまたは複数のデータセットに適用するには，そのデータベースコンポーネントの ApplyUpdates メソッドを呼び出します。次のコードは，ボタンクリックイベントに応答して CustomersQuery データセットの更新を適用します。

```
void __fastcall TForm1::ApplyButtonClick(TObject *Sender)
{
  // Paradox, dBASE, FoxPro などのローカルデータベースでは
  // TransIsolation を tiDirtyRead に設定する
  if (!Database1->IsSQLBased && Database1->TransIsolation != tiDirtyRead)
    Database1->TransIsolation = tiDirtyRead;
  Database1->ApplyUpdates(&CustomersQuery,0);
}
```

上記のシーケンスは，自動的に生成されたトランザクションのコンテキストで，キャッシュアップデートをデータベースに書き込みます。成功した場合，トランザクションのコミット，さらにキャッシュアップデートのコミットが続きます。失敗した場合，トランザクションはロールバックされ，アップデートキャッシュの内容は変更されません。後者の場合は，データセットの OnUpdateError イベントを通じてキャッシュアップデートエラーをアプリケーションで処理しなければなりません。更新エラーの処理については，24-38 ページの「キャッシュアップデートエラーの処理」を参照してください。

データベースコンポーネントの ApplyUpdates メソッドを呼び出すことの主な利点は，そのデータベースに関連するデータセットコンポーネントをいくつでも更新できることです。データベースの ApplyUpdates メソッドの 2 つの引数は，TDBDataSet の配列と配列の最後のデータセットのインデックスです。複数のデータセットに更新を適用するには，データセットにポインタのローカル配列を作成します。たとえば，次のコードは 2 つの問い合わせの更新を適用します。

```
TDBDataSet* ds[] = {CustomerQuery, OrdersQuery};
if (!Database1->IsSQLBased && Database1->TransIsolation != tiDirtyRead)
  Database1->TransIsolation = tiDirtyRead;
Database1->ApplyUpdates(ds,1);
```

データセットコンポーネントメソッドによるキャッシュアップデートの適用

個々の BDE 対応データセットについての更新は，そのデータセットの ApplyUpdates メソッドと CommitUpdates メソッドを使って，直接適用できます。これらのメソッドはどちらも更新プロセスの 1 つの段階をカプセル化します。

1. ApplyUpdates が，キャッシュに入っている変更をデータベースに書き込む（第 1 段階）
2. CommitUpdates はデータベースの書き込みに成功した場合，内部キャッシュをクリアする（第 2 段階）

次のコードは，CustomerQuery データセットについて，トランザクション内での更新の適用のしかたを示します。

```
void __fastcall TForm1::ApplyButtonClick(TObject *Sender)
{
  Database1->StartTransaction();
  try
  {
    if (!Database1->IsSQLBased && Database1->TransIsolation != tiDirtyRead)
      Database1->TransIsolation = tiDirtyRead;
    CustomerQuery->ApplyUpdates(); // 更新内容をデータベースに書き込む
    Database1->Commit(); // 成功したら，変更をコミットする
  }
  catch (...)
  {
    Database1->Rollback(); // 失敗したら，変更を元に戻す
    throw; // CommitUpdates の呼び出しを防止するために例外を再生成する
  }
  CustomerQuery->CommitUpdates(); // 成功したら，内部キャッシュをクリアする
}
```

ApplyUpdates の呼び出し中に例外が生成された場合，データベーストランザクションはロールバックされます。トランザクションのロールバックでは，基礎になるデータベーステーブルは変更されません。**try...catch** ブロック内部の throw 文は例外を再生成するので，CommitUpdates の呼び出しは防止されます。CommitUpdates が呼び出されないため，エラー条件の処理と可能であれば更新の再試行ができるように更新の内部キャッシュはクリアされません。

OnUpdateRecord イベントハンドラの作成

BDE 対応データセットがそのキャッシュアップデートを適用するときは，キャッシュ内のすべての変更レコードを，基になったテーブル内の対応するレコードに適用する試みが繰り返して行われます。変更，削除，または新規挿入された各レコードについて更新が適用されようとするとき，データセットコンポーネントの OnUpdateRecord イベントが発生します。

OnUpdateRecord イベントにハンドラを定義すると，現在のレコードの更新が実際に適用される直前にアクションを実行できます。そのようなアクションとして，データの検証，ほかのテーブルの更新，特殊パラメータの置換，複数のアップデートオブジェクトの実行などがあります。OnUpdateRecord イベントのハンドラによって，更新プロセスの制御性が高まります。

OnUpdateRecord イベントハンドラの概略コードを次に示します。

```
void __fastcall TForm1::DataSetUpdateRecord(TDataSet *DataSet,
  TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
```

```
{
  // ここで更新を実行
}
```

DataSet パラメータは更新のあるキャッシュされたデータセットを指定します。

UpdateKind パラメータは，現在のレコードについて実行する必要がある更新の種類を表します。UpdateKind の値は，ukModify，ukInsert，および ukDelete です。アップデートオブジェクトを使用する場合は，更新を適用するときにこのパラメータをアップデートオブジェクトに渡す必要があります。更新の種類に基づいてハンドラが何か特殊な処理を実行する場合は，このパラメータを調べる必要もあります。

UpdateAction パラメータは，更新を適用したかどうかを示します。UpdateAction の値は，uaFail（デフォルト），uaAbort，uaSkip，uaRetry，uaApplied です。イベントハンドラが更新の適用に成功した場合は，終了前にこのパラメータを uaApplied に変更します。現在のレコードを更新しないことに決めた場合は，値を uaSkip に変更して未適用の変更をキャッシュに保持します。UpdateAction の値を変更しない場合，そのデータセットの更新操作全体が中止され，例外が発生します。UpdateAction を uaAbort に設定すると，エラーメッセージを発生させないように指定できます（サイレント例外）。

これらのパラメータ以外に，現在のレコードに関連した項目コンポーネントの OldValue プロパティや NewValue プロパティを使用したいと考えるでしょう。OldValue は，データベースから取得した元の項目値です。このプロパティは，更新するデータベースレコードを検索する場合に使用します。NewValue は，適用しようとしている更新内容に含まれている編集済みの値です。

重要 OnUpdateError イベントハンドラや OnCalcFields イベントハンドラの場合と同様に，OnUpdateRecord イベントハンドラは，データセット内の現在のレコードを変更するようなメソッドを決して呼び出してはなりません。

次の例は，このハンドラのパラメータとプロパティの使い方を示しています。この例では，TTable コンポーネントの UpdateTable を使って更新を適用します。実際にはアップデートオブジェクトを使用する方が簡単ですが，テーブルを使うことで動作を詳しく説明しています。

```
void __fastcall TForm1::EmpAuditUpdateRecord(TDataSet *DataSet,
  TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
{
  if (UpdateKind == ukInsert)
  {
    TVarRec values[2];
    for (int i = 0; i < 2; i++)
      values[i] = DataSet->Fields->Fields[i]->NewValue;
    UpdateTable->AppendRecord(values, 1);
  }
  else
  {
    TLocateOptions lo;
    lo.Clear();
    if (UpdateTable->Locate("KeyField", DataSet->Fields->Fields[0]->OldValue, lo))
      switch (UpdateKind)
      {
       case ukModify:
          UpdateTable->Edit();
          UpdateTable->Fields->Fields[1]->Value = DataSet->Fields->Fields[1]->Value;
          UpdateTable->Post();
          break;
```

```
        case ukDelete:
          UpdateTable->Delete();
          break;
      }
    }
    UpdateAction = uaApplied;
  }
```

キャッシュアップデートエラーの処理

ボーランドデータベースエンジン（BDE）は，更新を適用しようとするとき，特にユーザーによる更新の衝突がないかやその他の条件をチェックし，エラーがあれば報告します。データセットコンポーネントの OnUpdateError イベントを使うと，エラーを検出してエラーに応答できます。キャッシュアップデートを使う場合は，このイベントのためのハンドラを作成してください。作成しない場合にエラーが発生すると，更新操作全体が失敗に終わります。

OnUpdateError イベントハンドラの概略コードを次に示します。

```
void __fastcall TForm1::DataSetUpdateError(TDataSet *DataSet,
  EDatabaseError *E, TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
{
  // ここでエラーに対処する ...
}
```

DataSet へは更新の適用先のデータセットへの参照が渡されます。エラーの処理時には，このデータセットを使って新しい値と古い値にアクセスできます。各レコード内の項目の元の値は，読み出し専用の TField プロパティ OldValue に格納されます。変更された値は TField プロパティの NewValue に格納されます。イベントハンドラで更新値を調べたり変更するためには，これらの値を利用する方法しかありません。

注意 現在のレコードを変更するデータセットメソッド（Next や Prior など）は呼び出さないでください。呼び出すと，イベントハンドラが無限ループに入ります。

E パラメータは通常 EDBEngineError 型です。この例外の型からは，ユーザーに表示できるエラーメッセージをエラーハンドラで抽出できます。たとえば次のコードは，ダイアログボックスのキャプションにエラーメッセージを表示するのに使えます。

```
ErrorLabel->Caption = E->Message;
```

このパラメータは，更新エラーの実際の原因を調べる場合にも役立ちます。EDBEngineError から特定のエラーコードを抽出し，それに基づいて適切な処理を実行できます。

UpdateKind パラメータは，エラーを生成した更新の種類を示します。実行中の更新の種類に基づいてエラーハンドラで特別な処理を実行するのでなければ，コードでこのパラメータを使うことはないでしょう。

次の表に UpdateKind の値の一覧を示します。

表 24.7　UpdateKind の値

値	説明
ukModify	既存のレコードの編集でエラーになった
ukInsert	新規レコードの挿入でエラーになった
ukDelete	既存のレコードの削除でエラーになった

UpdateAction は，イベントハンドラが終了した後で進めるべき更新処理の内容を BDE に指示します。更新エラーハンドラが初めて呼び出されたときに，このパラメータの値は必ず uaFail に設定されます。エラーを引き起こしたレコードのエラー条件とその修正のための措置に基づき，通常はハンドラの終了前に UpdateAction を別の値に設定します。

- エラーハンドラの呼び出しの原因となったエラー条件をエラーハンドラで修正できる場合は，UpdateAction を終了時にとるべき適切な操作に設定します。エラー条件を修正できた場合は，UpdateAction を uaRetry に設定してそのレコードの更新を再度適用します。
- uaSkip に設定すると，エラーを引き起こした行の更新はスキップされ，そのレコードの更新はほかのすべての更新が完了した後もキャッシュ内に残ります。
- uaFail と uaAbort は両方とも更新操作全体を終了させます。uaFail は例外を生成してエラーメッセージを表示します。uaAbort はサイレント例外を生成します（エラーメッセージを表示しません）。

次のコードに示す OnUpdateError イベントハンドラは，更新エラーがキー違反に関係するかどうかを調べ，関係する場合には UpdateAction パラメータを uaSkip に設定します。

```
// この例では，ユニットファイルで BDE.hpp をインクルードする
void __fastcall TForm1::DataSetUpdateError(TDataSet *DataSet,
  EDatabaseError *E, TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
{
  UpdateAction = uaFail; // 更新が失敗した状態に初期化する
  if (E->ClassNameIs("EDBEngineError"))
  {
    EDBEngineError *pDBE = (EDBEngineError *)E;
    if (pDBE->Errors[pDBE->ErrorCount - 1]->ErrorCode == DBIERR_KEYVIOL)
      UpdateAction = uaSkip; // キー違反の場合，単にこのレコードをスキップする
  }
}
```

メモ キャッシュアップデートの適用中にエラーが発生すると，例外が送出されてエラーメッセージが表示されます。ApplyUpdates が **try...catch** 構造内で呼び出されなければ，OnUpdateError イベントハンドラ内からユーザーにエラーメッセージが表示され，アプリケーションで同じメッセージが 2 度表示される恐れがあります。エラーメッセージが繰り返されるのを防ぐには，UpdateAction を uaAbort に設定して，システムによって生成されるエラーメッセージの表示をオフにします。

アップデートオブジェクトを使ったデータセットの更新

「ライブ」でないストアドプロシージャや問い合わせを BDE 対応データセットが表している場合，データセットから直接的に更新を適用することはできません。このようなデータセットでは，クライアントデータセットを使ってキャッシュアップデートを行うときに問題が生じることもあります。キャッシュアップデートに使用するデータセットが BDE かクライアントデータセットかにかかわらず，この問題を持つデータセットはアップデートオブジェクトを使って処理することができます。

1. クライアントデータセットを使用する場合は，TBDEClientDataSet ではなく TClientDataSet を外部プロバイダコンポーネントとともに使用します。こうすることによって，BDE 対応のソースデータセットの UpdateObject プロパティを設定できます（手順 3）。
2. TUpdateSQL コンポーネントを BDE 対応データセットと同じデータモジュールに追加します。

3. BDE 対応データセットコンポーネントの UpdateObject プロパティを，そのデータモジュールの TUpdateSQL コンポーネントに設定します。
4. 更新の実行に必要な SQL 文を，アップデートオブジェクトの ModifySQL，InsertSQL，および DeleteSQL プロパティを使って指定します。これらの文を作成する際は，SQL 更新エディタを利用できます。
5. データセットを閉じます。
6. このデータセットコンポーネントの CachedUpdates プロパティを true に設定するか，データセットプロバイダを使ってデータセットをクライアントデータセットにリンクします。
7. データセットを再度開きます。

メモ 場合によっては，複数のアップデートオブジェクトが必要なこともあります。たとえば，複数のテーブルの結合を更新する場合や，複数のデータセットのデータを表すストアドプロシージャを更新する場合は，更新するテーブルごとに TUpdateSQL が 1 つずつ必要です。複数のアップデートオブジェクトを使用する場合，単に UpdateObject プロパティを設定するだけでは，アップデートオブジェクトをデータセットに関連付けることはできません。このような場合は，OnUpdateRecord イベントハンドラ（キャッシュアップデートに BDE を使う場合）または BeforeUpdateRecord イベントハンドラ（クライアントデータセットを使う場合）からアップデートオブジェクトを手動で呼び出す必要があります。

アップデートオブジェクトは，実際には 3 つの TQuery コンポーネントをカプセル化しています。各問い合わせコンポーネントはそれぞれが単一の更新処理を実行します。1 つ目の問い合わせコンポーネントが既存のレコード変更用の SQL 文 UPDATE を提供し，2 つ目がテーブルへの新規レコード追加用の INSERT 文を提供します。3 つ目の問い合わせコンポーネントがテーブルからのレコード削除用の DELETE 文を提供します。

データモジュールにアップデートコンポーネントを追加する場合，カプセル化された TQuery コンポーネントは見えません。これらは，SQL 文を指定する 3 つの更新プロパティに基づいて，実行時にアップデートコンポーネントによって作成されます。

- ModifySQL は UPDATE 文を指定する
- InsertSQL は INSERT 文を指定する
- DeleteSQL は DELETE 文を指定する

実行時，アップデートコンポーネントが更新の適用に使われるとき，アップデートコンポーネントは，次のことを行います。

1. 現在のレコードに対して行われた処理（変更，挿入，または削除）に基づいて，実行する SQL 文を選択します。
2. パラメータ値を SQL 文に渡します。
3. 指定された更新を実行する SQL 文を用意して実行します。

アップデートコンポーネントの SQL 文の作成

関連付けられたデータセット内のレコードを更新するために，アップデートオブジェクトは 3 つの SQL 文のいずれかを使います。アップデートオブジェクトでは 1 つのテーブルしか更新できないため，オブジェクトの更新文はそれぞれが同じベーステーブルを参照していなければなりません。

この 3 つの SQL 文は，更新のためにキャッシュに入れられたレコードを削除，挿入，および変更します。これらの文は，アップデートオブジェクトの DeleteSQL，InsertSQL，ModifySQL の各プロパティに設定する必要があります。これらの値は，設計時に設定することも，実行時に設定することもできます。たとえば次のコードは，実行時に DeleteSQL プロパティの値を指定します。

```
UpdateSQL->DeleteSQL->Clear();
UpdateSQL->DeleteSQL->Add("DELETE FROM Inventory I");
UpdateSQL->DeleteSQL->Add("WHERE (I.ItemNo = :OLD_ItemNo)");
```

設計時には，SQL 更新エディタを利用して，更新を適用するための SQL 文を作成できます。

データセットの元の項目値と変更された項目値を参照するパラメータがある場合，アップデートオブジェクトではパラメータが自動的にバインドされます。したがって SQL 文を作成するときには，通常は一定の形式の名前を使ってパラメータを記述します。これらのパラメータの使い方については，24-42 ページの「SQL 更新文でのパラメータ置換について」を参照してください。

SQL 更新エディタの使い方

アップデートコンポーネントの SQL 文を作成する手順は次のとおりです。

1. オブジェクトインスペクタで，データセットの UpdateObject プロパティのドロップダウンリストから，アップデートオブジェクトの名前を選択します。この手順は，次の手順で呼び出す SQL 更新エディタが，SQL 生成オプションに使う適切なデフォルト値を決められるようにします。
2. アップデートオブジェクトを右クリックし，コンテキストメニューから［UpdateSQL の設定］を選択します。この操作で，SQL 更新エディタが表示されます。このエディタによって，アップデートオブジェクトの ModifySQL，InsertSQL，DeleteSQL の各プロパティの SQL 文が，基になるデータセットと入力する値に基づいて作成されます。

SQL 更新エディタは 2 つのページで構成されています。［オプション］ページは，このエディタを最初に呼び出したときに表示されるページです。［テーブル名］コンボボックスは，更新するテーブルを選択するために使います。テーブル名を指定すると，［キー項目］リストボックスと［更新する項目］リストボックスに，使用可能な列のリストが表示されます。

［更新する項目］リストボックスは，どの列が更新対象であるかを示します。最初にテーブルを指定したときは，［更新する項目］リストボックスのすべての列が選択されて含まれています。希望する項目を複数選択できます。

［キー項目］リストボックスは，更新中にキーとして使う列を指定するのに使います。Paradox，dBASE，FoxPro の場合，ここに指定する列は既存のインデックスに対応していなければなりませんが，リモート SQL データベースの場合はその必要はありません。［キー項目］で設定を行うかわりに［主インデックスを選択］ボタンをクリックすると，一次インデックスに基づいて更新用のキー項目を選択できます。［データセットのデフォルト］を選択すると，選択リスト内のすべての項目がキーとして選択され，すべてが更新用に選択されている元の状態に戻ります。

項目名を引用符で囲む必要のあるサーバーの場合，［項目名を " で括る］チェックボックスにチェックマークを付けます。

テーブルを指定したら，キー列を選択し，更新列を選択し，［SQL 文を生成］を選択して，アップデートコンポーネントの ModifySQL，InsertSQL，および DeleteSQL プロパティに関連付ける予備的な SQL 文を生成します。ほとんどの場合，自動的に生成された SQL 文の微調整が必要になります。

生成した SQL 文を表示して変更するには，必要に応じて［SQL］ページを選択します。SQL 文を生成してこのページを選択した場合，ModifySQL プロパティの文が［SQL 文］メモボックスにすでに表示されています。このボックス内の文を希望どおりに編集できます。

重要 生成した SQL 文は更新文の作成用の土台であることに注意してください。正しく実行させるには変更する必要があるかもしれません。たとえば，NULL 値を含むデータを扱う場合，生成された項目変数を使わないで，WHERE 節を

```
WHERE field IS NULL
```

のように変える必要があります。それぞれの文を受け入れる前に直接自分でテストしてください。

［SQL 文の種類］ラジオボタンは，生成した SQL 文の種類を切り替えて各文を編集するために使います。

SQL 文を TUpdateSQL コンポーネントの SQL プロパティと関連付けるには，［OK］をクリックします。

SQL 更新文でのパラメータ置換について

SQL 更新文では特殊な形のパラメータ置換を使用していて，レコードの更新で古い項目値または新しい項目値のどちらかを代入できるようになっています。SQL 更新エディタはその文を生成するときに，どの項目値を使うかを決めます。SQL 更新文を書くとき，使用する項目値を指定してください。

パラメータ名がテーブルの列名と一致すると，当該レコードのキャッシュアップデート内の項目の新しい値が自動的にパラメータ値として使われます。パラメータ名が文字列「OLD_」が前に付く列名と一致した場合は，項目の古い値が使われます。たとえば，以下の SQL 更新文では，パラメータ :LastName には，挿入されたレコードについてキャッシュアップデート内の新しい項目値が自動的に代入されます。

```
INSERT INTO Names
(LastName, FirstName, Address, City, State, Zip)
VALUES (:LastName, :FirstName, :Address, :City, :State, :Zip)
```

新しい項目値は通常，InsertSQL 文や ModifySQL 文の中で使われます。変更されたレコードの更新では，キャッシュアップデートの新しい項目値が UPDATE 文に使われ，更新対象のベーステーブルの古い項目値が置換されます。

削除レコードの場合は新しい値はないので，DeleteSQL プロパティが「:Old_FieldName」構文を使います。古い項目値も，通常，変更または削除更新用の SQL 文の WHERE 節で，どのレコードを更新または削除するか判断するために使用されます。

UPDATE または DELETE SQL 更新文の WHERE 節には，少なくとも，キャッシュデータを使って更新する対象のベーステーブル内でレコードをユニークに識別できるだけの数のパラメータを指定してください。たとえば，顧客リストで顧客の名字だけを使っても，ベーステーブル内で正しいレコードをユニークに識別するには不十分です。「Smith」の名字のレコードはいくつもあるかもしれません。パラメータとして名字，名前，それに電話番号を使えば，十分な組み合わせになります。もっと好ましいのは，顧客番号のような固有の項目値を使うことです。

メモ 作成した SQL 文の中のパラメータが，編集した項目値も元の項目値も参照していない場合，アップデートオブジェクトではそのパラメータにどの値を割り当てればよいかが判断できません。このような場合は，アップデートオブジェクトの Query プロパティを使って手動で設定できます。詳細は，24-47 ページの「アップデートコンポーネントの Query プロパティの使い方」を参照してください。

SQL 更新文を書く

設計時には，SQL 更新エディタを使って DeleteSQL，InsertSQL，ModifySQL の各プロパティの SQL 文を記述することができます。SQL 更新エディタを使わないときや，生成された文を修正するときには，以下で述べる要領に従って，ベーステーブル内のレコードを削除，挿入，または変更する文を記述してください。

DeleteSQL プロパティは，DELETE コマンドを使った SQL 文だけを含んでいなければなりません。更新対象のベーステーブル名を，FROM 節に指定する必要があります。アップデートキャッシュ内で削除されたレコードに対応するベーステーブル内のレコードのみを削除するには，SQL 文で WHERE 節を使います。WHERE 節のパラメータとして，キャッシュアップデートレコードに対応するベーステーブル内のレコードをユニークに識別するための 1 つまたは複数の項目を使用します。項目名の頭に「OLD_」を付けたのと同じ名前のパラメータならば，キャッシュアップデートレコードの対応する項目からそのパラメータに自動的に値が割り当てられます。パラメータの名前がそれ以外ならば，自分でパラメータ値を指定しなければなりません。

```
DELETE FROM Inventory I
WHERE (I.ItemNo = :OLD_ItemNo)
```

テーブルの種類によっては，レコードの識別に使われる項目がヌル値のときにベーステーブル内でレコードを見つけられません。その場合，それらのレコードの削除更新は失敗します。これに対処するため，NULL を含む可能性のある項目について，IS NULL 述語を使って条件を追加します（NULL 以外の値の場合の条件に加えて）。たとえば，FirstName 項目の値が NULL のことがある場合，次のようにします。

```
DELETE FROM Names
WHERE (LastName = :OLD_LastName) AND
  ((FirstName = :OLD_FirstName) OR (FirstName IS NULL))
```

InsertSQL 文は，INSERT コマンドを使った SQL 文だけで構成されなければなりません。更新対象のベーステーブルの名前を，INTO 節で指定しなければなりません。VALUES 節には，パラメータをカンマで区切ったリストを指定します。パラメータ名が項目名と同じならば，パラメータにはキャッシュアップデートレコードから自動的に値が割り当てられます。パラメータの名前がそれ以外ならば，自分でパラメータ値を指定しなければなりません。パラメータのリストは，新規挿入レコードの項目値を指定します。文の中でリストされている項目の数と同じ数の値パラメータがなければなりません。

```
INSERT INTO Inventory
(ItemNo, Amount)
VALUES (:ItemNo, 0)
```

ModifySQL 文は，UPDATE コマンドを使った SQL 文だけで構成されていなければなりません。更新対象のベーステーブル名を，FROM 節に指定する必要があります。SET 節で 1 つまたは複数の値を割り当てます。SET 節で割り当てた値が，項目と同じパラメータ名ならば，パラメータには自動的にキャッシュ内の更新レコード内の同名項目から値が割り当てられます。どの項目名とも名前が一致しないほかのパラメータ名を使って，項目値を追加で割り当てることもできます。ただし，この値は自分で手入力で指定しなければなりません。DeleteSQL 文と同様に，更新対象のベーステーブル内でレコードをユニークに識別するために，項目名の頭に「OLD_」を付けたパラメータ名を使って WHERE 節を指定します。次の更新文では，パラメータ :ItemNo には自動的に値が割り当てられますが，:Price は自動的ではありません。

```
UPDATE Inventory I
SET I.ItemNo = :ItemNo, Amount = :Price
WHERE (I.ItemNo = :OLD_ItemNo)
```

上の SQL で，アプリケーションのエンドユーザーが既存のレコードを変更する場合を例として考えてみましょう。ItemNo 項目の元の値は 999 です。キャッシュデータセットに接続されたグリッドで，エンドユーザーは ItemNo 項目の値を 123 に，Amount を 20 に変えました。ApplyUpdates メソッドが呼び出されると，この SQL 文は，:OLD_ItemNo パラメータで古い項目値を使うので，ベーステーブル内で ItemNo 項目が 999 のすべてのレコードに影響します。これらのレコードで，ItemNo 項目の値は 123 になり（:ItemNo パラメータを使用，値はグリッドから），Amount は 20 になります。

複数のアップデートオブジェクトの使用

アップデートデータセットで参照される複数のベーステーブルを更新する場合，更新する各ベーステーブルについて 1 つずつアップデートオブジェクトを使う必要があります。データセットコンポーネントの UpdateObject を使うとそのデータセットに 1 つのアップデートオブジェクトしか関連付けられないので，各アップデートオブジェクトの DataSet プロパティをデータセットの名前に設定することによってそれぞれをデータセットと関連付けます。

ヒント 複数個のアップデートオブジェクトを使用する場合は，TClientDataSet ではなく TBDEClientDataSet を外部プロバイダとともに使用できます。これは，ソースデータセットの UpdateObject プロパティを設定する必要がないためです。

アップデートオブジェクトの DataSet プロパティは，設計時のオブジェクトインスペクタでは使用できません。このプロパティは，実行時にのみ設定できます。

```
UpdateSQL1->DataSet = Query1;
```

アップデートオブジェクトはこのデータセットを使って，パラメータ置換の際の元の項目値と更新された項目値を取得します。データセットが BDE 対応データセットの場合は，更新を適用する際のセッションとデータベースもこのデータセットを基にして特定します。パラメータ置換が正しく機能するためには，更新された項目値を含むデータセットがアップデートオブジェクトの DataSet プロパティに設定されていなければなりません。BDE 対応データセットを使ってキャッシュアップデートを行う場合は，このデータセットは，その BDE 対応データセットです。クライアントデータセットを使用する場合，このデータセットは，BeforeUpdateRecord イベントハンドラにパラメータとして与えるクライアントデータセットです。

アップデートオブジェクトがデータセットの UpdateObject プロパティに代入されていないときには，その SQL 文は，ApplyUpdates を呼び出しても自動的に実行されません。レコードを更新するには，OnUpdateRecord イベントハンドラ（キャッシュアップデートに BDE を使う場合）または BeforeUpdateRecord イベントハンドラ（クライアントデータセットを使う場合）からアップデートオブジェクトを手動で呼び出す必要があります。これらのイベントハンドラでは，少なくとも以下の処理を実行する必要があります。

- クライアントデータセットを使ってキャッシュアップデートを行っている場合は，アップデートオブジェクトの DatabaseName プロパティと SessionName プロパティに，ソースデータセットの DatabaseName プロパティと SessionName プロパティが設定されていることを確認する
- イベントハンドラから，アップデートオブジェクトの ExecSQL メソッドまたは Apply メソッドを必ず呼び出す。これによって，更新を必要とする各レコードのアップデートオブジェクトが実行されます。更新文の実行については，下記の「SQL 文の実行」を参照してください。
- イベントハンドラの UpdateAction パラメータを uaApplied に設定する（OnUpdateRecord の場合）か，または Applied パラメータを **true** に設定（BeforeUpdateRecord の場合）する

ほかにオプションとして，各レコードの更新によってデータ検証，データ変更，その他の操作も行えます。

注意 アップデートオブジェクトの ExecSQL メソッドまたは Apply メソッドを OnUpdateRecord イベントハンドラから呼び出す場合は，データセットの UpdateObject プロパティをそのアップデートオブジェクトに設定しないようにしてください。そのように設定してしまうと，各レコードの更新が再度試みられます。

SQL 文の実行

複数のアップデートオブジェクトを使用する場合，その UpdateObject プロパティを設定しても，アップデートオブジェクトはデータセットに関連付けられません。その結果，更新の適用時に適切な文が自動的には実行されません。このような場合は，アップデートオブジェクトをコードで明示的に実行しなければなりません。

アップデートオブジェクトを実行するには，2 通りの方法があります。どちらの方法を利用するかは，項目値を表すパラメータを SQL 文で使用するかどうかで決まります。

- 実行する SQL 文がパラメータを使用する場合は，Apply メソッドを呼び出す
- 実行する SQL 文がパラメータを使用しない場合は，より効率的な ExecSQL メソッドを呼び出す

メモ SQL 文で使用するパラメータが組み込み型のパラメータ（元の項目値と更新された項目値）以外の場合は，手動でその値を指定する必要があります（Apply メソッドに用意されているパラメータの置換機能は利用できません）。手動でパラメータの値を指定する方法については，24-47 ページの「アップデートコンポーネントの Query プロパティの使い方」を参照してください。

アップデートオブジェクトの SQL 文のパラメータについて，そのデフォルトの置換機能については，24-42 ページの「SQL 更新文でのパラメータ置換について」を参照してください。

Apply メソッドの呼び出し

アップデートコンポーネントの Apply メソッドは，現在のレコードについて手動で更新を適用します。このプロセスには次の 2 つの手順が関係します。

1. レコードの元の項目値と変更された項目値を，SQL 文の該当するパラメータに割り当てます。
2. SQL 文を実行します。

Apply メソッドは，アップデートキャッシュ内の現在のレコードについて更新を適用するときに呼び出します。Apply メソッドは，ほとんどの場合，データセットの OnUpdateRecord イベントハンドラ，またはプロバイダの BeforeUpdateRecord イベントハンドラから呼び出されます。

注意 データセットとアップデートオブジェクトの関連付けにデータセットの UpdateObject プロパティを使う場合，Apply メソッドは自動的に呼び出されます。この場合，OnUpdateRecord イベントのハンドラで Apply を呼び出すと，現在のレコードの更新を再度適用しようとするので，呼び出さないでください。

OnUpdateRecord イベントハンドラは，適用する必要がある更新の種類を，TUpdateKind 型のパラメータ UpdateKind で示します。このパラメータを Apply メソッドに渡して，どの更新 SQL 文を使用するか指示しなければなりません。この内容を BeforeUpdateRecord イベントハンドラを使った以下のコードで示します。

```
void __fastcall TForm1::BDEClientDataSet1BeforeUpdateRecord(TObject *Sender,
   TDataSet *SourceDS, TCustomClientDataSet *DeltaDS, TUpdateKind UpdateKind, bool &Applied)
```

```
{
  UpdateSQL1->DataSet = DeltaDS;
  TDBDataSet *pSrcDS = dynamic_cast<TDBDataSet *>(SourceDS);
  UpdateSQL1->DatabaseName = pSrcDS->DatabaseName;
  UpdateSQL1->SessionName = pSrcDS->SessionName;
  UpdateSQL1->Apply(UpdateKind);
  Applied = true;
}
```

ExecSQL メソッドの呼び出し

アップデートコンポーネントの ExecSQL メソッドは，現在のレコードについて手動で更新を適用します。Apply メソッドとは異なり ExecSQL の場合は，SQL 文を実行する前に，そのパラメータの結合が行われません。ExecSQL メソッドは，ほとんどの場合，OnUpdateRecord イベントハンドラ（BDE を使用する場合），BeforeUpdateRecord イベントハンドラ（クライアントデータセットを使用する場合）から呼び出されます。

ExecSQL ではパラメータの値が割り当てられないため，アップデートオブジェクトの SQL 文にパラメータが含まれない場合に主に使われます。パラメータがない場合でも Apply を使うことはできますが，ExecSQL ではパラメータがチェックされないので，ExecSQL の方が効率的です。

SQL 文にパラメータがある場合にも ExecSQL を呼び出すことはできますが，それは，前もってパラメータに値を明示的に割り当てた場合に限られます。BDE を使ってキャッシュアップデートを行う場合は，アップデートオブジェクトの DataSet プロパティを設定し，その SetParams メソッドを呼び出すことによって，パラメータを明示的に結合できます。クライアントデータセットを使ってキャッシュアップデートを行う場合は，TUpdateSQL が管理する基になる問い合わせオブジェクトに対してパラメータを渡す必要があります。その手順については，24-47 ページの「アップデートコンポーネントの Query プロパティの使い方」を参照してください。

注意 データセットとアップデートオブジェクトの関連付けにデータセットの UpdateObject プロパティを使う場合，ExecSQL メソッドは自動的に呼び出されます。この場合，OnUpdateRecord イベントまたは BeforeUpdateRecord イベントのハンドラで ExecSQL を呼び出すと，現在のレコードの更新を再度適用しようとするので，呼び出さないでください。

OnUpdateRecord イベントと BeforeUpdateRecord イベントのハンドラは，適用する必要がある更新の種類を，TUpdateKind 型のパラメータ UpdateKind で示します。このパラメータを ExecSQL メソッドに渡して，どの更新 SQL 文を使用するか指示しなければなりません。この内容を BeforeUpdateRecord イベントハンドラを使った以下のコードで示します。

```
void __fastcall TForm1::BDEClientDataSet1BeforeUpdateRecord(TObject *Sender,
   TDataSet *SourceDS, TCustomClientDataSet *DeltaDS, TUpdateKind UpdateKind, bool &Applied)
{
  TDBDataSet *pSrcDS = dynamic_cast<TDBDataSet *>(SourceDS);
  UpdateSQL1->DatabaseName = pSrcDS->DatabaseName;
  UpdateSQL1->SessionName = pSrcDS->SessionName;
  UpdateSQL1->ExecSQL(UpdateKind);
  Applied = true;
}
```

更新プログラムの実行中に例外が生成された場合は，定義されていれば，OnUpdateError イベントで実行が継続します。

アップデートコンポーネントの Query プロパティの使い方

アップデートコンポーネントの Query プロパティを使用すると，DeleteSQL 文，InsertSQL 文，および ModifySQL 文を実装する問い合わせコンポーネントにアクセスできます。ただし，これらの問い合わせで実行する文は，DeleteSQL，InsertSQL，ModifySQL の各プロパティを使って指定可能であり，その文の実行は，アップデートオブジェクトの Apply メソッドまたは ExecSQL メソッドを呼び出すことによって可能なため，ほとんどのアプリケーションでは，これらの問い合わせコンポーネントに直接アクセスする必要はありません。しかし，問い合わせコンポーネントを直接操作することが必要な場合もあります。特に，アップデートオブジェクトが持つ，新旧の項目値へのパラメータの自動結合機能を利用するのではなく，SQL 文のパラメータを独自に指定したい場合などには，Query プロパティを使用することができます。

メモ Query プロパティには，実行時にのみアクセスできます。

Query プロパティは，TUpdateKind の値を基にしてインデックスが付けられます。

- インデックス ukModify を使用すると，既存レコードを更新する問い合わせにアクセスします。
- インデックス ukInsert を使用すると，新規レコードを挿入する問い合わせにアクセスします。
- インデックス ukDelete を使用すると，レコードを削除する問い合わせにアクセスします。

以下に，自動的に結合することのできないパラメータ値を，Query プロパティを使って指定する方法を示します。

```
void __fastcall TForm1::BDEClientDataSet1BeforeUpdateRecord(TObject *Sender,
  TDataSet *SourceDS, TCustomClientDataSet *DeltaDS, TUpdateKind UpdateKind, bool &Applied)
{
  UpdateSQL1->DataSet = DeltaDS;  // 自動的なパラメータ置換のために必要
  TQuery *pQuery = UpdateSQL1->Query[UpdateKind];  // 問い合わせにアクセス
  // 問い合わせが正しい DatabaseName と SessionName を持つようにする
  TDBDataSet *pSrcDS = dynamic_cast<TDBDataSet *>(SourceDS);
  pQuery->DatabaseName = pSrcDS->DatabaseName;
  pQuery->SessionName = pSrcDS->SessionName;
  // ここでカスタムパラメータの値を代入
  pQuery->ParamByName("TimeOfLastUpdate")->Value = Now();
  UpdateSQL1->Apply(UpdateKind); // ここで自動置換して実行
  Applied = true;
}
```

TBatchMove の使い方

TBatchMove は，データセットの複製，一方のデータセットから別のデータセットへのレコードの追加，一方のデータセットのレコードによる別のデータセットのレコードの更新，別のデータセットのレコードに一致するレコードの削除ができるボーランドデータベースエンジン（BDE）の各機能をカプセル化します。TBatchMove は以下の場合にもっともよく使われます。

- 分析やほかの処理のためにサーバーからローカルデータソースにデータをダウンロードする
- アップサイジング処理の一環としてデスクトップのデータベースをリモートサーバーのテーブルへ移行する

バッチ移動コンポーネントはソーステーブルに対応する転送先テーブルを作成することができ，その際に列名とデータ型を自動的にマッピングします。

バッチ移動コンポーネントの作成

バッチ移動コンポーネントを作成する手順は次のとおりです。

1. レコードのインポート元となるテーブルコンポーネントまたは問い合わせコンポーネント（ソースデータセット）を，フォームまたはデータモジュール上に置きます。
2. レコードの移動先となるデータセット（転送先データセット）を，フォームまたはデータモジュール上に置きます。
3. コンポーネントパレットの［BDE］ページにある TBatchMove コンポーネントをデータモジュールまたはフォームに入れ，その Name プロパティをアプリケーションに適したユニークな値に設定します。
4. バッチ移動コンポーネントの Source プロパティを，レコードのコピー元，追加元，または更新元のテーブルの名前に設定します。テーブルは利用可能なデータセットコンポーネントのドロップダウンリストから選択できます。
5. Destination プロパティを，作成，追加，または更新するテーブルの名前に設定します。対象のテーブルは，利用可能なデータセットコンポーネントのドロップダウンリストから選択できます。
   - 追加，更新，削除の場合，Destination プロパティは既存のデータベーステーブルの名前でなければなりません。
   - テーブルのコピーで Destination が既存のテーブル名を表している場合にバッチ移動を実行すると，転送先テーブル内の現在のデータがすべて上書きされます。
   - 既存のテーブルをコピーしてまったく新しいテーブルを作成する場合，作成されるテーブルには，コピー先のテーブルコンポーネントの Name プロパティに指定されている名前が付きます。作成されるテーブルの種類は，DatabaseName プロパティで指定したサーバーに適した構造の種類になります。
6. Mode プロパティを実行する操作の種類を示すように設定します。有効な操作は，batAppend（デフォルト），batUpdate，batAppendUpdate，batCopy，および batDelete です。これらのモードについては，24-49 ページの「バッチ移動モードの指定」を参照してください。
7. 必要であれば，Transliterate プロパティを設定します（オプション）。Transliterate が **true**（デフォルト）の場合，文字データは，必要に応じて Source データセットの文字セットから Destination データセットの文字セットに変換されます。
8. 必要であれば，列のマッピングを Mappings プロパティで設定します（オプション）。ソーステーブルと転送先テーブルでの位置に基づいて列がバッチ移動で照合される場合は，このプロパティを設定する必要はありません。列のマッピングについての詳細は，24-50 ページの「データ型のマッピング」を参照してください。
9. 必要であれば，ChangedTableName，KeyViolTableName，および ProblemTableName プロパティを指定します（オプション）。バッチ処理中に検出された問題のあるレコードは，ProblemTableName で指定されたテーブルに保存されます。バッチ移動によって Paradox テーブルを更新している場合は，KeyViolTableName で指定したテーブルにキー違反が報告されるようにすることができます。ChangedTableName には，バッチ移動処理の結果，転送先テーブルで変更があったすべてのレコードが示されます。これらのプロパティを指定しないと，以上のエラーテー

ブルは作成も使用もされません。バッチ移動エラーの処理についての詳細は，24-51 ページの「バッチ移動エラーの処理」を参照してください。

バッチ移動モードの指定

Mode プロパティはバッチ移動コンポーネントが実行する処理を指定します。

表 24.8　バッチ移動モード

プロパティ	目的
batAppend	転送先テーブルにレコードを追加する
batUpdate	ソーステーブルのレコードで対応する転送先テーブルのレコードを更新する。更新は転送先テーブルのインデックスに基づいて行われる
batAppendUpdate	一致するレコードが転送先テーブルに存在する場合は，それを更新する。存在しない場合は，レコードを転送先テーブルに追加する
batCopy	ソーステーブルの構造に基づいて転送先テーブルを作成する。転送先テーブルがすでに存在する場合，そのテーブルは削除され，新たに作成される
batDelete	ソーステーブルのレコードと一致する転送先テーブルのレコードを削除する

レコードの追加

データを追加するには，ターゲットデータセットは既存のテーブルを表していなければなりません。追加処理では，必要に応じてターゲットデータセットに適したデータ型とデータサイズにデータが変換されます。変換できない場合は例外が生成され，データは追加されません。

レコードの更新

データを更新するには，ターゲットデータセットは既存のテーブルを表していなければならず，レコードを照合できるインデックスが定義されていなければなりません。一次インデックスフィールドを照合に使う場合，ソースデータセット内のレコードのインデックスフィールドと一致するインデックスフィールドを持つターゲットデータセット内のレコードが，ソースデータで上書きされます。更新処理では，必要に応じてターゲットデータセットに適したデータ型とデータサイズにデータが変換されます。

レコードの追加更新

データを追加および更新するには，ターゲットデータセットは既存のテーブルを表していなければならず，レコードを照合できるインデックスが定義されていなければなりません。一次インデックスフィールドを照合に使う場合，ソースデータセット内のレコードのインデックスフィールドと一致するインデックスフィールドを持つターゲットデータセット内のレコードが，ソースデータで上書きされます。一致するレコードがターゲットデータセット内にない場合は，ソースデータセットのデータがターゲットデータセットに追加されます。追加更新処理では，必要に応じてターゲットデータセットに適したデータ型とデータサイズにデータが変換されます。

データセットのコピー

ソースデータセットをコピーするには，ターゲットデータセットは既存のテーブルを表すものであってはいけません。既存のテーブルを表している場合にバッチ移動処理を行うと，そのテーブルがソースデータセットのコピーで上書きされます。

ソースデータセットとターゲットデータセットで使用されているデータベースエンジンが異なる場合（たとえば Paradox と InterBase など），可能な限りソースデータセットに近い構造を持つターゲットデータセットが作成され，必要であればデータ型とデータサイズの変換が自動的に実行されます。

メモ TBatchMove では，インデックス，制約，ストアドプロシージャなどのメタデータ構造はコピーされません。したがって，メタデータオブジェクトは，必要ならサーバー上または BDE Administrator で作成しなければなりません。

レコードの削除

ターゲットデータセット内のデータを削除するには，そのデータセットが既存のテーブルを表していなければならず，レコードを照合できるインデックスが定義されていなければなりません。一次インデックスフィールドを照合に使う場合，ソースデータセット内のレコードのインデックスフィールドと一致するインデックスフィールドを持つターゲットデータセット内のレコードが，転送先テーブルから削除されます。

データ型のマッピング

batAppend モードでは，バッチ移動コンポーネントは，ソーステーブルの列のデータ型に基づいて転送先テーブルを作成します。列と型は，ソーステーブルと転送先テーブルでの位置に基づいて照合されます。つまり，ソーステーブルの最初の列は転送先テーブルの最初の列と照合され，ほかの列も同様に照合されます。

デフォルトの列マッピングを変更するには，Mappings プロパティを使います。Mappings プロパティは，列マッピングのリスト（1 行に 1 つずつ）です。このリストは 2 つの形式のどちらかで示されます。ソーステーブルの列を転送先テーブルの同じ名前の列にマッピングするには，一致する列名を指定する単純なリストを使えます。たとえば，次のマッピングはソーステーブルの ColName という列が転送先テーブルの同じ名前の列にマッピングされるように指定しています。

```
ColName
```

ソーステーブルの SourceColName という列を転送先テーブルの DestColName という列にマッピングする場合は，次のような構文になります。

```
DestColName = SourceColName
```

ソースとターゲットの列のデータの型が同一でない場合，バッチ移動処理は「最適な方法」を試します。必要であれば文字データ型を切り詰め，可能な場合は制限付きで変換を実行しようとします。たとえば，CHAR(10) 列を CHAR(5) 列にマッピングすると，ソースの列で最後の 5 文字が切り詰められます。

変換の例として，文字データ型のソース列を整数型のターゲット列にマッピングした場合を考えます。バッチ移動処理では，文字値「5」が対応する整数値に変換されます。値を変換できないとエラーになります。エラーについての詳細は，24-51 ページの「バッチ移動エラーの処理」を参照してください。

異なるテーブル型の間でデータを移動する場合，バッチ移動コンポーネントはデータ型をデータセットのサーバーのタイプに基づいて適切に変換します。種類の異なるサーバー間でのマッピングに関する最新の表については，BDE のオンラインヘルプを参照してください。

メモ SQL サーバーデータベースへのデータのバッチ移動には，データベースサーバーと，適切な SQL Link がインストールされた C++Builder が必要です。サードパーティの適切な ODBC ドライバがインストールされていれば，ODBC を使用できます。

バッチ移動の実行

実行時にバッチ処理を実行するには，Execute メソッドを使います。たとえば，バッチ移動コンポーネントの名前が BatchMoveAdd の場合は，次の文でバッチ処理を実行します。

```
BatchMoveAdd->Execute();
```

設計時にバッチ移動コンポーネントを右クリックしてコンテキストメニューから［実行］を選択するという方法でも，バッチ移動を実行できます。

MovedCount プロパティはバッチ移動の実行時に移動されたレコードの数を記録します。

RecordCount プロパティは，移動するレコードの最大数を指定します。RecordCount が 0 の場合，ソースデータセット内の最初のレコードから始まって，すべてのレコードが移動されます。RecordCount が正の数の場合，ソースデータセット内の現在のレコードから始まって，最大で RecordCount 個のレコードが移動されます。ソースデータセット内の現在のレコードから最後のレコードまでの数より RecordCount の方が大きい場合は，ソースデータセットの終わりに達した時点でバッチ移動は終了します。MoveCount を調べると，実際に転送されたレコードの数を確認できます。

バッチ移動エラーの処理

バッチ移動処理で発生する可能性のあるエラーには，データ型変換エラーと整合性違反の 2 種類があります。TBatchMove には，エラー処理の報告と制御を行うためのプロパティがいくつか用意されています。

AbortOnProblem プロパティは，データ型変換エラーが発生したときに処理を中止するかどうかを指定します。AbortOnProblem が true の場合，エラーが発生するとバッチ移動処理は中止されます。false の場合は継続されます。ProblemTableName に指定したテーブルを調べると，どのレコードが問題を引き起こしたかを確認できます。

AbortOnKeyViol プロパティは，Paradox キー違反が発生したときに処理を中止するかどうかを指定します。

ProblemCount プロパティは，転送先テーブル内での処理でデータが失われたレコードの数を示します。AbortOnProblem が **true** の場合はエラーが発生すると処理が中止されるので，この数は 1 になります。

以下のプロパティを使うと，バッチ移動コンポーネントはバッチ移動処理を記述した追加のテーブルを作成できます。

- ChangedTableName を指定すると，更新または削除処理の結果，転送先テーブルで変更されたすべてのレコードを収めたローカル Paradox テーブルが作成されます。
- KeyViolTableName を指定すると，Paradox テーブルの操作時にキー違反を引き起こしたソーステーブル内のすべてのレコードを収めたローカル Paradox テーブルが作成されます。

AbortOnKeyViol が true の場合は最初の問題が起きたときに処理が中止されるので，このテーブルには最大でも 1 つのエントリしか保存されません。

- ProblemTableName を指定すると，データ型変換エラーが原因で転送先テーブルに登録できなかったすべてのレコードを収めたローカル Paradox テーブルが作成されます。たとえば，転送先テーブルに収まるようにソーステーブルのレコードのデータを切り詰める必要があった場合は，そのレコードがこのローカルテーブルに収められます。AbortOnProblem が **true** の場合は最初の問題が起きたときに処理が中止されるので，このテーブルには最大でも 1 つのエントリしか保存されません。

メモ ProblemTableName を指定しなかった場合には，レコード内のデータは切り詰められて転送先テーブルに入れられます。

データディクショナリ

BDE を使ってデータにアクセスする場合は，アプリケーションからデータディクショナリにアクセスできます。データディクショナリはカスタマイズ可能な格納領域をアプリケーションとは別に提供するもので，データの外観および内容を記述する拡張項目属性セットの作成が可能になります。

たとえば，頻繁に財務アプリケーションを開発する場合は，各種の通貨表示形式を記述する特殊な項目属性セットを数多く作成する可能性があります。設計時に開発アプリケーションのデータセットを作成する場合は，オブジェクトインスペクタを使って各データセットに通貨型項目を手作業で設定するのではなく，データディクショナリに設定されている拡張項目属性セットに通貨型項目を関連付けられます。データディクショナリを使うと，作成するアプリケーション内およびアプリケーション間でデータの外観が一貫したものになります。

クライアント / サーバー環境では，データディクショナリをリモートサーバーに置いて情報共有に備えることができます。

設計時に項目エディタで拡張項目属性セットを作成する方法，およびアプリケーションのデータセットを通してそれらの属性セットを項目に関連付ける方法については，23-12 ページの「項目コンポーネントの属性セットの作成」を参照してください。SQL エクスプローラおよびデータベースエクスプローラを使ったデータディクショナリと拡張項目属性の作成についての詳細は，それぞれのオンラインヘルプを参照してください。

データディクショナリのプログラミングインターフェースは, drintf ヘッダーファイル（include \ VCL ディレクトリ内）にあります。このインターフェースは，次のメソッドを提供します。

表 24.9 データディクショナリのインターフェース

ルーチン	使い方
DictionaryActive	データディクショナリがアクティブかどうか示す
DictionaryDeactivate	データディクショナリを非アクティブにする
IsNullID	指定された ID がヌル ID かどうか示す
FindDatabaseID	エリアスで指定されたデータベースの ID を返す
FindTableID	指定されたデータベースのテーブル ID を返す
FindFieldID	指定されたテーブル内の項目の ID を返す

表 24.9 データディクショナリのインターフェース（つづき）

ルーチン	使い方
FindAttrID	名前で指定された属性セットの ID を返す
GetAttrName	ID で指定された属性セットの名前を返す
GetAttrNames	ディクショナリ内の各属性セットに対してコールバックを実行する
GetAttrID	指定した項目の属性セットの ID を返す
NewAttr	項目コンポーネントから新規の属性セットを作成する
UpdateAttr	項目のプロパティと一致するように属性セットを更新する
CreateField	保存されている属性に基づいて項目コンポーネントを作成する
UpdateField	指定された属性セットと一致するように項目のプロパティを変更する
AssociateAttr	指定された項目 ID に属性セットを関連付ける
UnassociateAttr	項目 ID への属性セットの関連付けを解除する
GetControlClass	指定された属性 ID のコントロールクラスを返す
QualifyTableName	ユーザー名によって完全に限定されたテーブル名を返す
QualifyTableNameByName	ユーザー名によって完全に限定されたテーブル名を返す
HasConstraints	ディクショナリ内にデータセットの制約があるかどうか示す
UpdateConstraints	データセットのインポートされた制約を更新する
UpdateDataset	データセットをディクショナリ内の現在の設定と制約に更新する

BDE の操作用ツール

データアクセスのメカニズムとして BDE を利用するメリットの 1 つに，豊富なサポートユーティリティが C++Builder に付属していることが挙げられます。このユーティリティには，次のものがあります。

- **SQL エクスプローラ**および**データベースエクスプローラ**：C++Builder には，バージョン（版）によってこの 2 つのどちらかのアプリケーションが付属しています。どちらのエクスプローラでも，以下の処理が可能です。
  - 既存のデータベーステーブルとデータベース構造を検査する。SQL エクスプローラでは，リモート SQL データベースに対する検査と問い合わせが可能
  - テーブルにデータを設定する
  - データディクショナリ内に拡張項目属性セットを作成する。または拡張項目属性セットをアプリケーションで項目に関連付ける
  - BDE エリアスを作成し管理する

  SQL エクスプローラでは，以下の処理も可能です。
  - ストアドプロシージャなどの SQL オブジェクトをリモートデータベースサーバー上に作成する
  - リモートデータベースサーバー上の SQL オブジェクトのテキストを再構築して表示する
  - SQL スクリプトを実行する

- **SQL モニタ**：リモートデータベースサーバーと BDE との間の通信をすべて監視できます。通信されるメッセージにフィルタをかけて，関心があるカテゴリのメッセージのみを監視できます。SQL モニタは，作成するアプリケーションのデバッグに非常に役立ちます。
- **BDE Administrator**：新規のデータベースドライバを追加し，既存ドライバのデフォルト環境を設定し，新規の BDE エリアスを作成できます。
- **データベースデスクトップ**：Paradox または dBASE のテーブルを扱う場合，そのデータの表示と編集，新規テーブルの作成，既存テーブルの再構成の各処理を実行できます。データベースデスクトップを利用すると，TTable コンポーネントのメソッドを使う場合より細かな制御が可能になります（たとえば，有効性のチェックや言語ドライバが指定可能です）。BDE の API を直接呼び出す場合を除くと，データベースデスクトップは，Paradox と dBASE のテーブルを再構築する唯一のメカニズムです。

第25章

ADO コンポーネントの操作

dbGo コンポーネントを利用すると，ADO フレームワーク経由のデータアクセスが可能になります。ADO（ActiveX Data Objects）は，OLE DB プロバイダを介してデータにアクセスする COM オブジェクトを集めたものです。dbGo コンポーネントは，これらの ADO オブジェクトを C++Builder データベースアーキテクチャ内にカプセル化します。

ADO をベースにしているアプリケーションの ADO 層は，Microsoft ADO 2.1，データストアアクセス用の OLE DB プロバイダまたは ODBC ドライバ，使用される特定のデータベースシステムのクライアントソフトウェア（SQL データベースの場合），アプリケーションからアクセス可能なデータベースバックエンドシステム（SQL データベースシステム用），およびデータベースで構成されます。ADO ベースのアプリケーションが完全に機能するには，そのアプリケーションからこれらのすべてにアクセスできなければなりません。

ADO オブジェクトの中でも特に目立つものは，Connection，Command，および Recordset オブジェクトです。これらの ADO オブジェクトは，TADOConnection，TADOCommand，ADO データセットの各コンポーネントでカプセル化されます。ADO フレームワークの「ヘルパー」オブジェクトには，Field オブジェクトや Properties オブジェクトなどもありますが，dbGo アプリケーションでは通常，直接的には使用しません。また，これらのオブジェクトをカプセル化する専用コンポーネントもありません。

この章では，dbGo コンポーネントを紹介し，そのコンポーネントによって C++Builder データベースの共通アーキテクチャに付加される，このコンポーネント独自の機能について説明します。dbGo コンポーネントに特有な機能の説明は，データベース接続コンポーネントとデータセットの一般的な機能について理解していることを前提としています。データベース接続コンポーネントとデータセットについては，それぞれ第 21 章「データベースへの接続」および第 22 章「データセットについて」を参照してください。

ADO コンポーネントの概要

コンポーネントパレットの［ADO］ページには，複数の dbGo コンポーネントがあります。これらのコンポーネントを使用して，ADO データストアへ接続したり，コマンドを実行したり，ADO フレームワークを使用するデータベースにあるテーブルからデータを取得したりすることができます。dbGo コンポーネントを利用するためには，ADO 2.1（またはそれ以降）がホストコンピュータにインストールされている必要があります。加えて，対象のデータベースシステム用のクライアントソフトウェア（Microsoft SQL Server など）と，そのデータベースシステム専用の OLE DB ドライバまたは ODBC ドライバがインストールされていなければなりません。

ほとんどの dbGo コンポーネントには，ほかのデータアクセスメカニズムで使用されるコンポーネントと同等の機能を持つ，データベース接続コンポーネント（TADOConnection）や各種データセットがあります。さらに dbGo には，TADOCommand も用意されています。この単純なコンポーネントはデータセットではありませんが，ADO データストア上で実行する SQL コマンドを表します。

次の表に，ADO コンポーネントを示します。

表 25.1　ADO コンポーネント

コンポーネント	使い方
TADOConnection	ADO データストアとの接続を確立するデータベース接続コンポーネント。複数の ADO データセットコンポーネントおよびコマンドコンポーネントがこの接続を共有して，コマンドの実行，データの取得，およびメタデータの操作を行える
TADODataSet	データを取得および操作するための主要データセット。TADODataSet は 1 つまたは複数のテーブルからデータを取得できる。データストアに直接接続することも，TADOConnection コンポーネントを使用することも可能
TADOTable	1 つのデータベーステーブルに基づいたレコードセットを取得し操作するための，テーブルタイプのデータセット。TADOTable は，データストアに直接接続することも，TADOConnection コンポーネントを使用することも可能
TADOQuery	適正な SQL 文によって作成されたレコードセットを取得し操作するための，問い合わせタイプのデータセット。TADOQuery は，データ定義言語（DDL）SQL 文も実行できる。データストアに直接接続することも，TADOConnection コンポーネントを使用することも可能
TADOStoredProc	ストアドプロシージャを実行するための，ストアドプロシージャタイプのデータセット。TADOStoredProc で実行するストアドプロシージャは，データを取得するタイプでも取得しないタイプでもよい。データストアに直接接続することも，TADOConnection コンポーネントを使用することも可能
TADOCommand	コマンド（結果セットを返さない SQL 文）を実行するための単純なコンポーネント。TADOCommand は，これをサポートするデータセットコンポーネントとともに使用することも，テーブルからデータセットを取得することもできる。また，データストアに直接接続することも，TADOConnection コンポーネントを使用することも可能

ADO データストアへの接続

dbGo アプリケーションでは，データストアに接続しデータにアクセスする OLE DB プロバイダとの通信に，Microsoft ActiveX Data Objects（ADO）2.1 を使用します。データストアが表すことのできるものの 1 つは，データベースです。ADO ベースのアプリケーションでは，ADO 2.1 がクライアント

コンピュータにインストールされていなければなりません。ADO と OLE DB は，Microsoft によって供給され，Windows にインストールされます。

ADO プロバイダは，ネイティブ OLE DB ドライバから ODBC ドライバまでの各種アクセスのいずれかを行います。これらのドライバは，クライアントコンピュータにインストールされていなければなりません。各種データベース用の OLE DB ドライバは，データベースベンダーまたはサードパーティによって供給されます。アプリケーションが Microsoft SQL Server や Oracle のような SQL データベースを使用する場合は，そのデータベースシステム用のクライアントソフトウェアもクライアントコンピュータにインストールされていなければなりません。クライアントソフトウェアは，データベースベンダーによって供給され，データベースシステム CD（またはディスク）からインストールされます。

アプリケーションをデータストアに接続するには，ADO 接続コンポーネント（TADOConnection）を使用します。利用可能な ADO プロバイダの中から 1 つを選択して，そのプロバイダを使用するように ADO 接続コンポーネントを設定します。ADO のコマンドコンポーネントとデータセットコンポーネントでは，それぞれの ConnectionString プロパティを使って接続を直接確立できるので TADOConnection はかならずしも必要ではありませんが，TADOConnection を使用すると，1 つの接続を複数の ADO コンポーネントで共有できます。この場合，リソースの消費量が減少するとともに，複数のデータセットにまたがるトランザクションを作成できます。

ほかのデータベース接続コンポーネントと同様に，TADOConnection では以下の操作がサポートされています。

- 接続の制御
- サーバーログインの制御
- トランザクションの管理
- 関連付けられたデータセットを操作する
- サーバーにコマンドを送信する
- メタデータの取得

以上の機能はデータセット接続コンポーネントのすべてに共通するものですが，TADOConnection には以下の独自の機能もあります。

- 接続を微調整するためのさまざまなオプション
- 接続を使用するコマンドオブジェクトのリストを作成する機能
- 一般的なタスク実行時の追加イベント

TADOConnection の使用によるデータストアへの接続

TADOConnection を使用すると，1 つのデータストア接続を，1 まはた複数の ADO データセットコンポーネントと ADO コマンドコンポーネントで共有できます。このためには，データセットコンポーネントとコマンドコンポーネントを，それぞれの Connection プロパティを使って接続コンポーネントに関連付けます。設計時には，オブジェクトインスペクタで Connection プロパティのドロップダウンリストから目的の接続コンポーネントを選択します。実行時には，参照を Connection プロパティに代入します。たとえば，次の行では，TADODataSet コンポーネントが TADOConnection コンポーネントに関連付けられています。

```
ADODataSet1->Connection = ADOConnection1;
```

この接続コンポーネントによって，ADO 接続オブジェクトが表されます。接続オブジェクトを使って接続を確立するには，まず接続対象とするデータストアを特定する必要があります。通常，この情報は ConnectionString プロパティを使って指定します。ConnectionString は，セミコロンを区切り記号として使用する文字列で，指定する 1 つまたは複数の接続パラメータを記述します。このパラメータに，接続情報を持つファイルまたは ADO プロバイダのいずれかの名前，およびデータストアを識別するための参照値を指定することによって，データストアが特定されます。これらの情報を指定するには，以下の定義済みパラメータを使用します。

パラメータ	説明
Provider	接続に使用するローカルな ADO プロバイダの名前
Data Source	データストア名
File name	接続情報を持つファイルの名前
Remote Provider	リモートマシン上の ADO プロバイダの名前
Remote Server	リモートサーバー名（リモートプロバイダを使用する場合）

したがって，ConnectionString は一般には次のようになります。

```
Provider=MSDASQL.1;Data Source=MQIS
```

メモ Provider プロパティを使って ADO プロバイダを指定する場合は，ConnectionString の接続パラメータとして Provider や Remote Provider を記述する必要はありません。同様に，DefaultDatabase プロパティを使用する場合，Data Source パラメータは指定する必要がありません。

以上に挙げたパラメータのほかに，ConnectionString には，使用する特定の ADO プロバイダに特有な接続パラメータも記述できます。たとえば，ログイン情報をコードに埋め込む場合は，このような追加接続パラメータとしてユーザー ID やパスワードを記述できます。

設計時には，接続文字列エディタを使用し，一覧から接続要素（プロバイダやサーバー）を選択して接続文字列を構築できます。オブジェクトインスペクタで ConnectionString プロパティの省略記号ボタンをクリックすると，接続文字列エディタが起動します。このエディタは，ActiveX のプロパティエディタで，ADO によって提供されます。

ConnectionString プロパティ（およびオプションの Provider プロパティ）の設定処理が終了したら，ADO 接続コンポーネントを使って，ADO データストアへの接続を確立または切断できます（ただし，その前にほかのプロパティを使って接続を微調整することが必要な場合もあります）。TADOConnection を使用していると，データセットへの接続時または切断時に，すべてのデータベース接続コンポーネントに共通するイベント以外にも，いくつかの追加イベントに応答できます。この追加イベントについては，25-7 ページの「接続確立時のイベント」および 25-8 ページの「切断時のイベント」で説明しています。

メモ 接続コンポーネントの Connected プロパティを **true** に設定することで明示的に接続をアクティブにしていない場合でも，最初のデータセット接続コンポーネントが開かれたとき，または ADO コマンドコンポーネントを使って初めてコマンドを実行したときに，接続は自動的に確立されます。

接続オブジェクトへのアクセス

基になる ADO 接続オブジェクトにアクセスするには，TADOConnection の ConnectionObject プロパティを使用します。この参照を使用して，基になる ADO 接続オブジェクトのプロパティにアクセスしたりメソッドを呼び出したりできます。

基になる ADO 接続オブジェクトを使うためには，ADO オブジェクト全般についてと，特に ADO 接続オブジェクトについての，実践的な知識が必要です。Connection オブジェクトの操作について十分理解していない限り，Connection オブジェクトを使用することはお勧めできません。ADO Connection オブジェクトの使用についての詳しい情報は，Microsoft Data Access SDK のヘルプを参照してください。

接続の微調整

単に ADO コマンドコンポーネントと ADO データセットコンポーネントの接続文字列を指定するのではなく，TADOConnection を使ってデータストアへの接続を確立した場合，接続の状態や特性を細かく制御できるという利点があります。

接続を強制的に非同期にする

接続を非同期にする場合は，ConnectOptions プロパティを使用します。接続を非同期にすると，接続が完全に開かれるのを待たなくてもアプリケーションが処理を続けることができます。

デフォルトでは，ConnectionOptions は coConnectUnspecified に設定され，サーバーが最適な接続タイプを決定するようになっています。接続を明示的に非同期にするには，ConnectOptions を coAsyncConnect に設定します。

次に例として示すルーチンは，指定された接続コンポーネントでの非同期接続を有効および無効にします。

```
void __fastcall TForm1::AsyncConnectButtonClick(TObject *Sender)
{
  ADOConnection1->Close();
  ADOConnection1->ConnectOptions = coAsyncConnect;
  ADOConnection1->Close();
}
void __fastcall TForm1::ServerChoiceConnectButtonClick(TObject *Sender)
{
  ADOConnection1->Close();
  ADOConnection1->ConnectOptions = coConnectUnspecified;
  ADOConnection1->Close();
}
```

タイムアウトの制御

ConnectionTimeout プロパティと CommandTimeout プロパティを使用すると，コマンドの実行時や接続の確立時に，どれだけの時間が経過したらその処理を失敗とみなして中止するかを制御できます。

ConnectionTimeout は，データストアへの接続を試行してどれだけの時間が経過したらタイムアウトにするかを秒単位で指定します。ConnectionTimeout で指定された時間内に接続が成功しなかった場合，その接続の試みは中止されます。

```
ADOConnection1->ConnectionTimeout = 10; // 秒数
ADOConnection1->Close();
```

ConnectionTimeout は，コマンドの試行がタイムアウトになるまでの時間を秒単位で指定します。Execute メソッドの呼び出しによって開始されたコマンドが，CommandTimeout で指定された時間内に正常終了しなかった場合，そのコマンドは取り消され，ADO は例外を発生させます。

```
ADOConnection1->ConnectionTimeout = 10;
ADOConnection1->Execute("DROP TABLE Employee1997", cmdText, TExecuteOptions());
```

接続でサポートされている操作の種類の指定

ADO 接続は，特定のモードで確立されます。このモードは，ファイルを開くときのモードと類似したものです。この接続モードによって，接続のアクセス権，つまりその接続を使って実行することが可能な操作の種類（読み出し，書き込みなど）が決まります。

接続モードを指定するには，Mode プロパティを使用します。表 25.2 に設定可能な値を示します。

表 25.2　ADO 接続モード

接続モード	説明
cmUnknown	接続のアクセス権が未設定，またはアクセス権を決定できない
cmRead	接続のアクセス権は読み出し専用
cmWrite	接続のアクセス権は書き込み専用
cmReadWrite	接続のアクセス権は読み書き可
cmShareDenyRead	読み出し専用アクセス権でほかから接続が開かれることを拒否
cmShareDenyWrite	書き込み専用アクセス権でほかから接続が開かれることを拒否
cmShareExclusive	ほかから接続が開かれることを拒否
cmShareDenyNone	どのアクセス権が使用された場合でも，ほかから接続が開かれることを拒否

Mode として設定可能な値は，基になる ADO 接続オブジェクトの Mode プロパティの ConnectModeEnum 値に対応します。この値についての詳細は，Microsoft Data Access SDK のヘルプを参照してください。

接続がトランザクションを自動的に開始するかどうかを指定する

保持されるコミットと保持される中止を接続コンポーネントで使用するかどうかを制御するには，Attributes プロパティを使います。接続コンポーネントで保持されるコミットを使用する場合は，アプリケーションがトランザクションをコミットするたびに，新しいトランザクションが自動的に開始されます。接続コンポーネントで保持される中止を使用する場合は，アプリケーションがトランザクションをロールバックするたびに，新しいトランザクションが自動的に開始されます。

Attributes には，定数 xaCommitRetaining と xaAbortRetaining の一方または両方を指定しても，あるいはどちらも指定しなくてもかまいません。Attributes に xaCommitRetaining が含まれている場合は，接続は保持されるコミットを使用します。Attributes に xaAbortRetaining が含まれている場合は，接続は保持される中止を使用します。

保持されるコミットと保持される中止が有効になっているかどうかを確認するには，Contains メソッドを使います。保持されるコミットまたは保持される中止を有効にするには，該当する値を Attributes プロパティに加えます。無効にするには，その値を差し引きます。次のルーチン例では，ADO 接続コンポーネントでの保持されるコミットを，それぞれ有効および無効にします。

```
void __fastcall TForm1::RetainingCommitsOnButtonClick(TObject *Sender)
{
  ADOConnection1->Close();
  if (!ADOConnection1->Attributes.Contains(xaCommitRetaining))
    ADOConnection1->Attributes = TXactAttributes() << xaCommitRetaining;
  ADOConnection1->Open();
}
void __fastcall TForm1::RetainingCommitsOffButtonClick(TObject *Sender)
{
  ADOConnection1->Close();
  if (ADOConnection1->Attributes.Contains(xaCommitRetaining))
    ADOConnection1->Attributes = TXactAttributes() >> xaCommitRetaining;
  ADOConnection1->Open();
}
```

接続のコマンドへのアクセス

ほかのデータベース接続コンポーネントの場合と同様に，接続に関連付けられているデータセットには DataSets プロパティと DataSetCount プロパティを使ってアクセスできます。ただし，dbGo には TADOCommand オブジェクトも用意されています。このオブジェクトは，データセットではありませんが，接続コンポーネントと似た関係を持っています。

TADOConnection の Commands プロパティと CommandCount プロパティを使用すると，DataSets プロパティと DataSetCount プロパティを使って関連するデータセットにアクセスする場合と同じ方法で，関連する ADO コマンドオブジェクトにアクセスできます。ただし，DataSets と DataSetCount では，アクティブなデータベースのリストだけが得られるのに対して，Commands と CommandCount の場合は，接続コンポーネントに関連付けられているすべての TADOCommand コンポーネントへの参照が得られます。

Commands は，ADO コマンドコンポーネントへの参照の，ゼロで始まる配列です。CommandCount プロパティは，Commands に含まれるすべてのコマンドの総数を示します。この 2 つのプロパティを使って，接続コンポーネントを使用するすべてのコマンドを繰り返して処理できます。その処理を以下のコードで説明します。

```
for (int i = 0; i < ADOConnection2->CommandCount; i++)
  ADOConnection2->Commands[i]->Execute();
```

ADO 接続イベント

どのデータベース接続コンポーネントでも発生する一般のイベントに加えて，TADOConnection ではいくつかの追加イベントも通常の使用時に生成されます。

接続確立時のイベント

すべてのデータベース接続コンポーネントに共通する BeforeConnect イベントと AfterConnect イベントに加えて，TADOConnection では接続の確立時に OnWillConnect イベントと OnConnectComplete イベントも生成されます。これらのイベントは，BeforeConnect イベントの後で発生します。

- OnWillConnect は，ADO プロバイダが接続を確立する前に発生します。このイベントを利用すると，接続文字列に最終変更を加えたり，独自にログインサポートを処理する場合はユーザー名や

パスワードを指定したり，非同期接続を強制したり，さらには接続が開かれる前にその接続処理を中止したりすることもできます。

- OnConnectComplete は接続が開かれた後で発生します。TADOConnection は非同期接続を表すこともできるため，AfterConnect イベントではなく，OnConnectComplete イベントを使用してください。OnConnectComplete は，接続が開かれた後か，またはエラーのために接続を開くことができなかった後で発生しますが，AfterConnect は，接続コンポーネントが ADO プロバイダに対して接続を開くよう指示した後で発生します。この場合，必ずしも接続が開かれた後であるとは限りません。

切断時のイベント

すべてのデータベース接続コンポーネントに共通する BeforeDisconnect イベントと AfterDisconnect イベントに加えて，TADOConnection では接続が閉じられた後に OnDisconnect イベントも生成されます。OnDisconnect は，接続が閉じられた後で，かつ関連したデータセットが閉じられる前に発生します。このイベントは AfterDisconnect イベントの前に発生します。

トランザクションの管理時のイベント

ADO 接続コンポーネントは，トランザクション関連プロセスが完了したことを検出するための多数のイベントを提供します。これらのイベントは，BeginTrans，CommitTrans，および RollbackTrans メソッドによって開始されたトランザクションプロセスがデータストアでいつ正常終了したかを知らせます。

- OnBeginTransComplete イベントは，BeginTrans メソッドが呼び出された後，データストアがトランザクションを正常に開始したときに発生します。
- OnCommitTransComplete イベントは，CommitTrans の呼び出しによってトランザクションが正常にコミットされたときに発生します。
- OnRollbackTransComplete イベントは，RollbackTrans の呼び出しによってトランザクションが正しく破棄されたときに発生します。

その他のイベント

ADO 接続コンポーネントには，以上のほかに 2 つのイベントがあります。このイベントを利用して，基になる ADO 接続オブジェクトから得られた通知に対応することができます。

- OnExecuteComplete イベントは，たとえば *Execute* メソッドを実行した後など，接続コンポーネントがデータストア上でコマンドを実行した後で発生します。OnExecuteComplete は，実行処理が正常終了したかどうかを示します。
- OnInfoMessage イベントは，ある操作が終了し，基になる接続オブジェクトが詳細な情報を提供したときに発生します。OnInfoMessage イベントハンドラは，ADO エラーオブジェクトへのインターフェースを受け取ります。このオブジェクトには，詳細情報，および操作が正常に終了したかどうかを示すステータスコードが入っています。

ADO データセットの使い方

ADO データセットコンポーネントは，ADO レコードセットオブジェクトをカプセル化します。このコンポーネントは，第 22 章「データセットについて」で述べたデータセットの共通の機能を継承しており，ADO を使ってその機能を実行します。ADO データセットを使用するには，この共通の機能について理解しておく必要があります。

データセットが共通に持っている機能に加え，すべての ADO データセットには，以下の処理を目的とする，プロパティ，イベント，およびメソッドが追加されています。

- ADO データベースへの接続
- 基になるレコードセットオブジェクトへのアクセス
- ブックマークに基づくレコードのフィルタ処理
- レコードを非同期で取得
- バッチアップデートの実行（更新内容のキャッシュ）
- ディスク上のファイルにデータを保存

ADO データセットには以下の 4 種類があります。

- TSQLTable。テーブルタイプのこのデータセットによって，1 つのデータベーステーブルのすべての行と列が表されます。TADOTable など，テーブルタイプのデータセットの使い方については，22-24 ページの「テーブルタイプのデータセットの使い方」を参照してください。
- TSQLQuery。問い合わせタイプのこのデータセットによって SQL 文がカプセル化され，結果レコードが得られた場合はアプリケーションからアクセスできるようになります。TADOQuery など，問い合わせタイプのデータセットの使い方については，22-40 ページの「問い合わせタイプのデータセットの使い方」を参照してください。
- TSQLStoredProc。ストアドプロシージャタイプのこのデータセットは，データベースサーバー上に定義されたストアドプロシージャを実行します。TADOStoredProc など，ストアドプロシージャタイプのデータセットの使い方については，22-48 ページの「ストアドプロシージャタイプのデータセットの使い方」を参照してください。
- TADODataSet。汎用データセットで，ほかの 3 種類のデータセットの機能を持っています。TADODataSet に特有な機能については，25-15 ページの「TADODataSet の使い方」を参照してください。

メモ ADO を使ってデータベース情報にアクセスするときに，カーソルを返さない SQL コマンドを表す目的であれば，TADOQuery などのデータセットを使用する必要はありません。このような場合は，データセットではなく単純なコンポーネントの TADOCommand を使うことができます。TADOCommand についての詳細は，25-17 ページの「コマンドオブジェクトの使い方」を参照してください。

ADO データセットをデータストアに接続する

ADO データセットは，ADO データストアに集合的にまたは個別に接続できます。

データセットを集合的に接続する場合，各データセットの Connection プロパティを TADOConnection コンポーネントに設定します。各データセットは，ADO 接続コンポーネントの接続を使用します。

```
ADODataSet1->Connection = ADOConnection1;
ADODataSet2->Connection = ADOConnection1;
...
```

複数のデータセットを集合的に接続することの利点として，次のことがあげられます。

- データセットの間で接続オブジェクトの属性を共有できる
- TADOConnection の接続を 1 つだけセットアップすればよい
- データセットがトランザクションに参加できる

TADOConnection の使い方については，25-2 ページの「ADO データストアへの接続」を参照してください。

データセットを個別に接続する場合は，各データセットの ConnectionString プロパティを設定します。ConnectionString を使用したデータセットは，アプリケーション内のほかのデータセットとは独立した，データストアへの独自の接続を確立します。

ADO データセットの ConnectionString プロパティは，TADOConnection の ConnectionString プロパティと同じように機能します。このプロパティは，接続パラメータをセミコロンで区切って記述したもので，たとえば次のようになります。

```
ADODataSet1->ConnectionString = "Provider=YourProvider;Password=SecretWord;";
ADODataSet1->ConnectionString += "User ID=JaneDoe;SERVER=PURGATORY;";
ADODataSet1->ConnectionString += "UID=JaneDoe;PWD=SecretWord;";
ADODataSet1->ConnectionString += "Initial Catalog=Employee";
```

設計時には，接続文字列エディタを使って接続文字列を容易に作成できます。接続文字列についての詳細は，25-3 ページの「TADOConnection の使用によるデータストアへの接続」を参照してください。

レコードセットの操作

Recordset プロパティを使うと，データセットコンポーネントの基になる ADO レコードセット オブジェクトに直接アクセスできます。このオブジェクトを使って，アプリケーションからレコードセットオブジェクトのプロパティにアクセスしたり，メソッドを呼び出したりできます。Recordset を使って基になる ADO レコードオブジェクトに直接アクセスするには，ADO オブジェクト一般的についてと，特に ADO レコードセットオブジェクトの詳細についての実践的な知識が必要です。レコードセットオブジェクトの操作をよく理解していなければ，レコードセットオブジェクトを直接使用することはお勧めできません。ADO レコードセットオブジェクトの使用に関する詳細については，Microsoft Data Access SDK ヘルプを参照してください。

RecordsetState プロパティは，基になるレコードセットオブジェクトの現在の状態を示します。RecordsetState は，ADO レコードセットオブジェクトの State プロパティに相当します。RecordsetState の値は，stOpen，stExecuting，stFetching のいずれかです (RecordsetState プロパティと類似した TObjectState では，ほかの値も定義されていますが，レコードセットに関連するのは，stOpen，stExecuting，stFetching に限られています)。値が stOpen の場合は，レコードセットが現在アイドルであることを表します。値が stExecuting の場合は，レコードセットでコマンドが実行中であることを表します。値が stFetching の場合は，レコードセットが，関連する 1 つまたは複数のテーブルから行を取得中であることを表します。

RecordsetState の値は，データセットの現在の状態に依存するアクションを実行するときに使用します。たとえば，データを更新するルーチンでは，RecordSetState プロパティを調べることによって，

データセットがアクティブになっているか，そして接続やデータ取得といったほかのアクティビティを行っていないかを確認できます。

ブックマークに基づくレコードのフィルタ処理

ブックマークを使って特定のレコードにマークを付け，そのレコードを返すという機能は，データセットが共通に持っている機能ですが，ADO データセットでもこの機能が利用できます。また，ほかのデータセットと同様に ADO データセットの場合も，フィルタを使ってデータセット内の使用可能なレコードを制限することができます。これらの機能に加え ADO データセットでは，データセットの共通機能である上記の 2 つの機能を組み合わせ，ブックマークによって特定されたレコードセットにフィルタを適用することができます。

ブックマークのセットにフィルタを適用する手順は次のとおりです。

1. Bookmark メソッドを使って，フィルタ処理の対象のデータセットに入れるレコードにマークを付けます。
2. FilterOnBookmarks メソッドを呼び出して，ブックマークを付けたレコードのみが表示されるように，データセットにフィルタを適用します。

このプロセスを以下に示します。

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  TBookmarkStr BM1;
  TBookmarkStr BM2;
  BM1 = ADODataSet1->Bookmark;
  BMList->Add(BM1);
  ADODataSet1->MoveBy(3);
  BM2 = ADODataSet1->Bookmark;
  BMList->Add(BM2);
  ADODataSet1->FilterOnBookmarks(ARRAYOFCONST((BM1,BM2)));
}
```

上記の例で，BMList というリストオブジェクトにブックマークを追加していることに注意してください。後で不要になったブックマークを，アプリケーションで解除できるようにするためには，この操作が必要です。

ブックマークの使い方についての詳細は，22-9 ページの「レコードにマークを付けて戻る」を参照してください。ほかの種類のフィルタについての詳細は，22-12 ページの「フィルタを使って編集する」を参照してください。

レコードを非同期で取得する

ほかのデータセットとは異なり，ADO データセットではデータを非同期で取得できます。このため，データストアのデータがデータセットに設定されている処理の間も，アプリケーションは別のタスクを実行できます。

データセットがデータを取得するタイプの場合にデータを非同期で取得するどうかを制御するには，ExecuteOptions プロパティを使用します。Open を呼び出すか，Active を true に設定したときにどのようにレコードを取得するかが ExecuteOptions によって制御されます。データセットが問い合わせまたはストアドプロシージャを表していて，レコードが返されない場合は，ExecSQL や ExecProc を呼

び出したときにその問い合わせやストアドプロシージャがどのように実行されるかが ExecuteOptions によって制御されます。

ExecuteOptions は，以下の値を組み合わせたものです（0 個または複数個の組み合わせも可）。

表 25.3　ADO データセットの実行オプション

実行オプション	説明
eoAsyncExecute	コマンドの操作またはデータの取得操作が非同期で実行される
eoAsyncFetch	データセットは，まず CacheSize プロパティで指定されたレコード数を同期的に取得し，次に残りの行を非同期で取得する
eoAsyncFetchNonBlocking	現在の実行スレッドが，非同期のデータ取得処理やコマンド実行処理によって阻止されない
eoExecuteNoRecords	データを返さないコマンドやストアドプロシージャ。取得された行があったとしても，その行は破棄され返されない

バッチアップデートの使い方

更新内容をキャッシュするには，データセットプロバイダを使って ADO データセットをクライアントデータセットに接続する方法もあります。この方法については，27-15 ページの「クライアントデータセットをキャッシュアップデートに使用する」で説明しています。

ただし，ADO データセットコンポーネントの場合，このコンポーネント独自のバッチアップデートという方法を使ってキャッシュアップデートに対応することができます。次の表は，キャッシュアップデートについて，クライアントデータセットを使用する場合と，バッチアップデート機能を使用する場合を比較しています。

表 25.4　キャッシュアップデートについての ADO / クライアントデータセットの比較

ADO データセット	TClientDataSet	説明
LockType	使用せず：クライアントデータセットは常に更新内容をキャッシュする	データセットをバッチアップデートモードで開くかどうかを指定する
CursorType	使用せず：クライアントデータセットは常にメモリ内のデータのスナップショットを処理する	サーバー上の変更内容から ADO データセットをどのように分離するかを指定する
RecordStatus	UpdateStatus	現在の行について，発生したアップデートの状態を示す（アップデートが発生している場合）。RecordStatus の場合は，UpdateStatus より詳細な情報が得られる
FilterGroup	StatusFilter	利用可能なレコードの種類を指定する。FilterGroup の場合は，より多くの情報が得られる
UpdateBatch	ApplyUpdates	キャッシュ内の更新内容をデータベースサーバーに適用する。ApplyUpdates と異なり，UpdateBatch では更新内容の種類を限定して適用できる
CancelBatch	CancelUpdates	保留状態の更新内容を破棄し，元の値を取得する。CancelUpdates と異なり，CancelBatch では更新内容の種類を限定して取り消すことができる

ADO データセットコンポーネントのバッチアップデート機能を使うときは，次のことを行います。

- データセットをバッチアップデートモードで開く
- 個々の行のアップデートステータスを検査する
- アップデートステータスを基に複数行をフィルタにかける
- ベーステーブルにバッチアップデートを適用する
- バッチアップデートを取り消す

データセットをバッチアップデートモードで開く

ADO データセットをバッチアップデートモードで開くには，以下の条件が満たされていなければなりません。

1. コンポーネントの CursorType プロパティが ctKeySet（デフォルトのプロパティ値）または ctStatic になっていること
2. LockType プロパティが ltBatchOptimistic であること
3. コマンドが SELECT 問い合わせであること

データセットコンポーネントをアクティブにする前に，CursorType および LockType プロパティを上記のように設定します。コンポーネントの CommandText プロパティ（TADODataSet）または SQL プロパティ（TADOQuery）に SELECT 文を代入します。TADOStoredProc コンポーネントの場合，ProcedureName に，結果セットを返すストアドプロシージャの名前を設定します。これらのプロパティは，設計時にオブジェクトを使って設定するか，実行時にプログラムによって設定します。次の例に，バッチアップデートモードのための TADODataSet コンポーネントの準備を示します。

```
ADODataSet1->CursorLocation = clUseClient;
ADODataSet1->CursorType = ctStatic;
ADODataSet1->LockType = ltBatchOptimistic;
ADODataSet1->CommandType = cmdText;
ADODataSet1->CommandText = "SELECT * FROM Employee";
```

データセットをバッチアップデートモードで開いた後は，データへの変更はすべて，ベーステーブルに直接適用せずに，キャッシュに入れられます。

個々の行のアップデートステータスを検査する

特定の行のアップデートステータスを調べるには，その行を現在行にしてから ADO データコンポーネントの RecordStatus プロパティを検査します。RecordStatus は，現在の行だけの現在のアップデートステータスを表します。

```
switch (ADOQuery->RecordStatus)
{
  case rsUnmodified:
    StatusBar1->Panels->Items[0]->Text = "Unchanged record";
    break;
  case rsModified:
    StatusBar1->Panels->Items[0]->Text = "Changed record";
    break;
  case rsDeleted:
    StatusBar1->Panels->Items[0]->Text = "Deleted record";
    break;
  case rsNew:
    StatusBar1->Panels->Items[0]->Text = "New record";
    break;
}
```

アップデートステータスを基に複数行をフィルタにかける

レコードセットにフィルタをかけて，同じアップデートステータスを持つ行のグループだけを表示するには，FilterGroup プロパティを使用します。FilterGroup に，どのアップデートステータスの行を表示するかを示す TFilterGroup 定数を設定します。値が fgNone（このプロパティのデフォルト値）なら，フィルタは適用されず，どのアップデートステータスの行もすべて表示されます（削除マーク付きの行を除く）。次の例は，保留状態のバッチアップデート行だけを表示します。

```
FilterGroup := fgPendingRecords;
Filtered = true;
```

メモ FilterGroup プロパティが機能するためには，ADO データセットコンポーネントの Filtered プロパティが **true** に設定されていなければなりません。

ベーステーブルにバッチアップデートを適用する

まだ適用も取り消しもされていない保留状態のデータ変更を適用するには，UpdateBatch メソッドを呼び出します。変更されて適用された行は，変更内容がレコードセットの基のベーステーブルに書き込まれます。キャッシュ内の削除マーク付き行は，ベーステーブル内の対応する行を削除します。挿入レコード（キャッシュ内に存在するがベーステーブルには存在しない）は，ベーステーブルに追加されます。行の変更は，ベーステーブル内の対応する行の列を，キャッシュ内の新しい行値に変更します。

CloseDatabase は，TAffectRecords の値を 1 つパラメータとして取ります。渡された値が arAll 以外なら，保留中の変更のサブセットのみが適用されます。下の例では，現在アクティブな行だけが適用されます。

```
ADODataSet1->UpdateBatch(arCurrent);
```

バッチアップデートを取り消す

まだ取り消しも適用もされていない保留中のデータ変更を取り消すには，CancelBatch メソッドを呼び出します。保留中のバッチアップデートを取り消すと，変更されていた行の項目値は，CancelBatch または UpdateBatch が呼び出されたことがあれば最後に呼び出されたときの直前の値，または現在の保留状態の変更バッチ以前の元の値に戻されます。

CancelBatch は，TAffectRecords の値を 1 つパラメータとして取ります。渡された値が arAll 以外なら，保留中の変更のサブセットのみが取り消されます。次の例は，保留中の変更のすべてを取り消します。

```
ADODataSet1->CancelBatch(arAll);
```

ファイルからのデータ読み込みとファイルへのデータ保存

ADO データセットコンポーネント経由で取得されたデータは，後で同じまたは別のコンピュータ上で取得できるようにファイルに保存することができます。データは，ADTG と XML の 2 つの形式のいずれかで保存されます。ADO では，この 2 つのファイル形式のみがサポートされています。ただし，両形式が ADO のすべてのバージョンでサポートされているとは限りません。使用しているバージョンの ADO のドキュメントで，保存ファイル用にサポートされている形式を確認してください。

ファイルにデータを保存するには，SaveToFile メソッドを使います。SaveToFile は，2 つのパラメータを取ります。データの保存先のファイル名を示すパラメータと，データの保存形式（ADTG または XML）を示すパラメータです。保存するファイル形式を指定するには，Format パラメータを

pfADTG または pfXML に設定します。すでに存在するファイルを FileName パラメータで指定すると，SaveToFile は EOleException を生成します。

ファイルからデータを取り出すには，LoadFromFile メソッドを使います。LoadFromFile には，読み出すファイルの名前をパラメータとして指定します。ほかのパラメータはありません。指定するファイルが存在しなかった場合，LoadFromFile は EOleException 例外を生成します。LoadFromFile メソッドが呼び出されるとき，データセットコンポーネントは自動的にアクティブになります。

次の例の最初のメソッドでは，TADODataSet コンポーネント ADODataSet1 によって取得されたデータセットをファイルに保存します。ターゲットファイルは，ローカルドライブに保存される SaveFile という名前の ADTG ファイルです。2 つ目のメソッドは，この保存されたファイルを TADODataSet コンポーネント ADODataSet2 に読み込みます。

```
void __fastcall TForm1::SaveBtnClick(TObject *Sender)
{
  if (FileExists("c:\\SaveFile"))
  {
    DeleteFile("c:\\SaveFile");
    Statusbar1->Panels->Items[0]->Text = "Save file deleted!";
  }
  ADODataSet1->SaveToFile("c:\\SaveFile");
}
void __fastcall TForm1::LoadBtnClick(TObject *Sender)
{
  if (FileExists("c:\\SaveFile"))
    ADODataSet1->LoadFromFile("c:\\SaveFile");
  else
    Statusbar1->Panels->Items[0]->Text = "Save file does not exist!";
}
```

データの保存用と読み込み用のデータセットは，必ずしも上の例のように同じフォーム上にある必要はありません。同じアプリケーション内や，同じコンピュータ上である必要さえもありません。そのため，ブリーフケースのように，あるコンピュータから別のコンピュータにデータを転送することが可能です。

TADODataSet の使い方

TADODataSet は，ADO データストアのデータを扱うための汎用データセットです。ほかの ADO データセットコンポーネントとは異なり，TADODataSet は，テーブル，問い合わせ，ストアドプロシージャのいずれのタイプでもありません。このデータセットは，これらの任意のタイプとして機能することができます。

- テーブルタイプのデータセットのように，TADODataSet では，1 つのデータベーステーブル内のすべての行と列を表すことができます。この用途で使用するには，CommandType プロパティに cmdTable を設定し，CommandText プロパティにテーブル名を設定します。TADODataSet では，以下のテーブルタイプのタスクをサポートしています。
  - インデックスを割り当て，レコードをソートしたり，レコードベース検索の基礎を作成したりする。22-25 ページの「インデックスを持つレコードのソート」で説明している標準的なインデックスのプロパティとメソッドに加え，TADODataSet では Sort プロパティを設定すること

により，一時的なインデックスを使用してソート操作を実行できます。Seek メソッドを使用して実行されるインデックスベースの検索には，現在のインデックスが使われます。

- データセットを空にする。DeleteRecords メソッドを使用する場合は削除するレコードを指定できるため，他のテーブルタイプのデータセットが持つメソッドに比べて，処理をきめ細かく制御できます。

TADODataSet でサポートされているテーブルタイプのタスクは，CommandType を cmdTable に設定していない場合でも利用できます。

- 問い合わせタイプのデータセットのように，TADODataSet では，このデータセットを開いたときに実行される SQL コマンドを 1 つ指定できます。この用途で使用するには，CommandType プロパティに cmdText を設定し，CommandText プロパティに実行する SQL コマンドを設定します。設計時には，オブジェクトインスペクタで CommandText プロパティをダブルクリックし，コマンドテキストエディタ上で SQL コマンドを構成できます。TADODataSet では，以下の問い合わせタイプのタスクをサポートしています。
  - 問い合わせテキスト内でパラメータを使用する。問い合わせパラメータについての詳細は，22-43 ページの「問い合わせでパラメータを使用する」を参照してください。
  - パラメータを使ってマスター / 詳細関係を設定する。この方法については，22-45 ページの「マスター / 詳細関係をパラメータを使用して確立する」を参照してください。
  - Prepared プロパティを **true** に設定し，あらかじめ問い合わせを準備しておくことによってパフォーマンスを改善する
- ストアドプロシージャタイプのデータセットのように，TADODataSet では，このデータセットを開いたときに実行されるストアドプロシージャを指定できます。この用途で使用するには，CommandType プロパティに cmdStoredProc を設定し，CommandText プロパティにストアドプロシージャ名を設定します。TADODataSet では，以下のストアドプロシージャタイプのタスクをサポートしています。
  - ストアドプロシージャのパラメータを操作する。ストアドプロシージャのパラメータについての詳細は，22-49 ページの「ストアドプロシージャのパラメータの操作」を参照してください。
  - 複数の結果セットを取得する。この方法については，22-52 ページの「複数の結果セットを取得する」を参照してください。
  - Prepared プロパティを **true** に設定し，あらかじめストアドプロシージャを準備しておくことによってパフォーマンスを改善する

さらに，TADODataSet では，ファイルに保存されているデータも操作できます。このためには，CommandType プロパティを cmdFile に設定し，CommandText プロパティにファイル名を設定します。

CommandText と CommandType プロパティを設定するには，その前に Connection または ConnectionString プロパティを設定して，TADODataSet をデータストアに接続する必要があります。この手順については，25-9 ページの「ADO データセットをデータストアに接続する」で述べます。ほかの方法として，RDS DataSpace オブジェクトを使って TADODataSet を ADO ベースのアプリケーションサーバーに接続することもできます。RDS DataSpace オブジェクトを使用するには，RDSConnection プロパティに TRDSConnection オブジェクトを設定します。

コマンドオブジェクトの使い方

ADO 環境では，コマンドは，プロバイダ固有のアクション要求をテキストで表現したものです。一般に，これらはデータ定義言語（DDL）およびデータ操作言語（DML）SQL 文です。コマンドで使用される言語は，プロバイダ固有ですが，通常は SQL 言語の SQL-92 標準に準拠しています。

TADOQuery を使用するといつでもコマンドを実行できますが，特に結果セットを返さないコマンドの場合は，データセットコンポーネントを使用することに伴うオーバーヘッドを避けたいことがあります。このような場合は，TADOCommand コンポーネントを使うことができます。これは，一度に 1 つのコマンドを実行する目的で使用する軽量オブジェクトです。TADOCommand は基本的に，データ定義言語（DDL）SQL 文のような，結果セットを返さないコマンドの実行用に意図されています。ただし，これの Execute メソッドのオーバーロードバージョンを介してなら，ADO データセットコンポーネントの RecordSet プロパティに代入した形で結果セットを返すことができます。

データセットの標準メソッドを使ってデータの取得，レコードの操作，データの編集などの処理を実行することができない点を除けば，全体として TADOCommand の操作は TADODataSet の操作とよく似ています。TADOCommand オブジェクトをデータストアに接続する方法は，ADO データセットの場合と同じです。詳しくは，25-9 ページの「ADO データセットをデータストアに接続する」を参照してください。

以降のセクションでは，TADOCommand を使ってコマンドを指定し実行する方法について詳細に説明します。

コマンドの指定

TADOCommand コンポーネントのコマンドは，CommandText プロパティを使って指定します。TADODataSet と同様に TADOCommand の場合も，CommandType プロパティに応じて各種コマンドを指定で行きます。CommandType として設定可能な値には，cmdText（コマンドが SQL 文の場合），cmdTable（テーブル名の場合），および cmdStoredProc（コマンドがストアドプロシージャ名の場合）があります。設計時には，オブジェクトインスペクタのリストから適切なコマンドタイプを選択します。実行時には，CommandType プロパティに TCommandType 型の値を代入します。

```
ADOCommand1->CommandText = "AddEmployee";
ADOCommand1->CommandType = cmdStoredProc;
...
```

特にタイプが指定されなければ，サーバーは，CommandText 内のコマンドに基づいて最適のタイプを決定します。

CommandText には，パラメータ付きの SQL 問い合わせや，パラメータを使用するストアドプロシージャの名前を記述できます。この場合はパラメータを使用した部分について，コマンドを実行する前にそのパラメータの値を提示する必要があります。詳細は，25-19 ページの「コマンドパラメータの扱い方」を参照してください。

Execute メソッドの使い方

TADOCommand のコマンドを実行するには，その前に TADOCommand がデータストアに正常に接続されていなければなりません。接続を確立する処理は，ADO データセットの場合と同じです。詳しくは，25-9 ページの「ADO データセットをデータストアに接続する」を参照してください。

コマンドを実行するには，Execute メソッドを呼び出します。Execute メソッドはオーバーロードされているため，コマンドのもっとも適切な実行方法を選択できます。

パラメータの不要なコマンドの場合や，コマンドの処理対象となったレコード数を知る必要がない場合は，以下のように，パラメータなしで Execute を呼び出します。

```
ADOCommand1->CommandText = "UpdateInventory";
ADOCommand1->CommandType = cmdStoredProc;
ADOCommand1->Execute();
```

Execute には，バリアント配列を使ってパラメータ値を指定したり，コマンドの処理対象となったレコード数を得ることができるバージョンもあります。

結果セットを返すコマンドの実行についての詳細は，25-18 ページの「コマンドによる結果セットの取得」を参照してください。

コマンドの取り消し

コマンドを非同期で実行している場合は，Execute を呼び出した後で Cancel メソッドを呼び出すと，その実行処理を中止できます。

```
void __fastcall TDataForm::ExecuteButtonClick(TObject *Sender)
{
  ADOCommand1->Execute();
}
void __fastcall TDataForm::CancelButtonClick(TObject *Sender)
{
  ADOCommand1->Execute();
}
```

Cancel メソッドは，保留中のコマンドがあって，そのコマンドが非同期で実行された（Execute メソッドの ExecuteOptions パラメータに eoAsynchExecute が指定されている）場合にだけ機能します。コマンドが保留中とみなされるのは，Execute メソッドが呼び出された後，コマンドがまだ終了せず，タイムアウトにもなっていない場合です。

CommandTimeout で指定されている秒数が経過するまでに，コマンドが終了せず，また取り消しもされなかった場合，そのコマンドはタイムアウトになります。デフォルトでは，30 秒後にコマンドがタイムアウトします。

コマンドによる結果セットの取得

結果セットを返すかどうかで使用する実行メソッドが異なる TADOQuery コンポーネントと違い，TADOCommand では，結果セットを返すかどうかに関係なく，常に Execute メソッドを使ってコマ

ンドを実行します。コマンドが結果セットを返す場合，Execute は ADO _RecordSet インターフェースへのインターフェースを返します。

このインターフェースは，ADO データセットの RecordSet プロパティに代入すると，もっとも簡単に扱えます。

たとえば，次のコードは TADOCommand（ADOCommand1）を使って，結果セットを返す SELECT 問い合わせを実行しています。続いて，この結果セットは，TADODataSet コンポーネント（ADODataSet1）の RecordSet プロパティに代入されています。

```
ADOCommand1->CommandText = "SELECT Company, State ";
ADOCommand1->CommandText += "FROM customer ";
ADOCommand1->CommandText += "WHERE State = :StateParam";
ADOCommand1->CommandType = cmdText;
ADOCommand1->Parameters->ParamByName("StateParam")->Value = "HI";
ADOCommand1->Recordset = ADOCommand1->Execute();
```

結果セットが ADO データセットの Recordset プロパティに代入されるとすぐに，そのデータセットが自動的にアクティブになり，データが使用可能になります。

コマンドパラメータの扱い方

TADOCommand オブジェクトでは，次の 2 通りの方法でパラメータを使用できます。

- パラメータを含む問い合わせを，CommandText プロパティで指定する。TADOCommand でのパラメータ付き問い合わせは，ADO データセットでパラメータ付き問い合わせを使用する場合と同じように機能します。パラメータ付き問い合わせについての詳細は，22-43 ページの「問い合わせでパラメータを使用する」を参照してください。
- パラメータを使用するストアドプロシージャを，CommandText プロパティで指定する。ストアドプロシージャのパラメータは，TADOCommand と ADO データセットでほとんど同じように機能します。ストアドプロシージャのパラメータについての詳細は，22-49 ページの「ストアドプロシージャのパラメータの操作」を参照してください。

TADOCommand のパラメータの値を指定する方法としては，Execute メソッドを呼び出すときに指定する方法と，Parameters プロパティを使ってあらかじめ指定しておく方法の 2 通りがあります。

Execute メソッドは，パラメータ値のセットをバリアント配列として受け取れるようにオーバーロードされています。この方法は，Parameters プロパティのセットアップに伴うオーバーヘッドがないため，パラメータの値をすばやく提供する場合に便利です。

```
Variant Values[2];
Values[0] = Edit1->Text;
Values[1] = Date();
ADOCommand1.Execute(VarArrayOf(Values,1));
```

ストアドプロシージャを扱い，出力パラメータが返される場合は，上記の方法ではなく，Parameters プロパティを使わなければなりません。出力パラメータを読み出す必要がない場合でも，Parameters プロパティを使うようにします。そうすることにより，パラメータを設計時に提供することが可能になるとともに，データセットのパラメータを操作するのと同じ方法で TADOCommand プロパティを操作できます。

CommandText プロパティを設定すると，問い合わせ内のパラメータまたはストアドプロシージャが使用するパラメータに基づいて，Parameters プロパティが自動的に更新されます。設計時には，オブジェクトインスペクタで Parameters プロパティの省略記号ボタンをクリックすることにより，パラメータエディタを使ってパラメータにアクセスできます。実行時には，TParameter のプロパティとメソッドを使って各パラメータの値を設定（または取得）します。

```
ADOCommand1->CommandText = "INSERT INTO Talley ";
ADOCommand1->CommandText += "(Counter) ";
ADOCommand1->CommandText += "VALUES (:NewValueParam)";
ADOCommand1->CommandType = cmdText;
ADOCommand1->Parameters->ParamByName("NewValueParam")->Value = 57;
ADOCommand1->Execute();
```

第26章

単方向データセットの使い方

dbExpress は軽量のデータベースドライバのセットで，SQL データベースサーバーに高速にアクセスできます。dbExpress には，サポートするデータベースごとにドライバが用意されており，dbExpress の統一的なクラスでサーバー固有のソフトウェアを扱えます。dbExpress を使ったデータベースアプリケーションを配布する際は，作成したアプリケーションファイルに 1 つの dll（サーバー固有のドライバ）を付属するだけで済みます。

dbExpress では，単方向のデータセットを使ってデータベースにアクセスできます。単方向データセットは，迅速で軽く，オーバーヘッドも最小でデータベース情報にアクセスできるように設計されています。ほかのデータセットの場合と同様に，単方向データセットを使って SQL コマンドをデータベースサーバーに送信し，コマンドがレコードセットを返すと，そのレコードにアクセスするカーソルを取得できます。ただし，単方向のデータセットで取得できるのは単方向のカーソルのみです。単方向データセットはデータのバッファリングを行わないため，ほかのデータセットより高速で，リソースをあまり消費しません。ただし，レコードがバッファリングされないため，単方向データセットはほかのデータセットほど柔軟ではありません。TDataSet で導入されている機能の多くは，単方向データセットに実装されていないか，例外を生成させます。例を示します。

- サポートされているナビゲーションメソッドは，First メソッドと Next メソッドだけです。他のほとんどのナビゲーションメソッドは例外を生成します。ブックマークのサポートに関するメソッドなど一部のメソッドは何もしません。
- 編集では編集内容の保持にバッファを必要とするので，組み込みの編集サポートはありません。CanModify プロパティは常に **False** なので，このデータセットを編集モードにはできません。ただし単方向データセットでも，SQL UPDATE コマンドを使用してデータを更新したり，dbExpress 対応のクライアントデータセットを使用するかデータセットをクライアントデータセットに接続して，従来と同様に編集したりすることはできます（18-10 ページの「ほかのデータセットに接続する」を参照してください）。
- フィルタはサポートしていません。フィルタでは複数のレコードが処理されますが，それにはバッファリングが必要だからです。単方向データセットにフィルタをかけると，例外が生成されます。そのかわり表示されるデータの制限は，データセットのデータを定義する SQL コマンドを使用して設定しなければなりません。

- 参照項目はサポートしていません，参照項目を含む複数レコードを保持するにはバッファリングを必要とするからです。単方向データセット上で参照項目を設定しても正しく動作しません。

これらの制限にもかかわらず，単方向データセットはデータにアクセスする強力な方法です。単方向データセットは，データにアクセスするもっとも高速なメカニズムであり，容易に利用し配布することができます。

単方向データセットの種類

コンポーネントパレットの［dbExpress］ページには，TSQLDataSet，TSQLQuery，TSQLTable，TSQLStoredProc の 4 種類の単方向データセットがあります。

この中では，TSQLDataSet がもっとも一般的です。dbExpress を通して利用可能な任意のデータを表すため，または dbExpress を通してアクセスされるデータベースにコマンドを送信するため，SQL データセットを使用できます。新しいデータベースアプリケーションでデータベーステーブルを扱う場合は，このコンポーネントを使用するとよいでしょう。

TSQLQuery は，SQL 文をカプセル化する問い合わせタイプのデータセット です。このデータセットを使って，アプリケーションから結果レコードにアクセスできます。問い合わせタイプのデータセットの使い方については，22-40 ページの「問い合わせタイプのデータセットの使い方」を参照してください。

TSQLTable は，テーブルタイプのデータセット です。このデータセットは，1 つのデータベーステーブルのすべての行と列を表します。テーブルタイプのデータセットの使い方については，22-24 ページの「テーブルタイプのデータセットの使い方」を参照してください。

TSQLStoredProc は，ストアドプロシージャタイプのデータセットです。このデータセットは，データベースサーバー上に定義されたストアドプロシージャを実行します。ストアドプロシージャタイプのデータセットの使い方については，22-48 ページの「ストアドプロシージャタイプのデータセットの使い方」を参照してください。

メモ ［dbExpress］ページには，単方向データセットではありませんが，TSQLClientDataSet もあります。このデータセットは，内部的に単方向データセットを使用してデータにアクセスするクライアントデータセットです。

データベースサーバーへの接続

単方向データセットを使うときの最初の手順は，データベースサーバーに接続することです。設計時には，データセットのデータベースサーバーへの接続がアクティブになると，オブジェクトインスペクタはほかのプロパティ値のドロップダウンリストを提供できるようになります。たとえばストアドプロシージャを表している場合，オブジェクトインスペクタでは接続をアクティブにしないと，サーバーで利用可能なストアドプロシージャをリスト表示できません。

データベースサーバーへの接続コンポーネントは，別個の TSQLConnection コンポーネントで表されます。TSQLConnection の扱いは，ほかのデータベース接続コンポーネントの場合と同じです。データベース接続コンポーネントについては，第 21 章「データベースへの接続」を参照してください。

TSQLConnection を使って単方向データセットをデータベースサーバーに接続するには，SQLConnection プロパティを設定します。設計時には，オブジェクトインスペクタのドロップダウンリストから SQL 接続コンポーネントを選択できます。実行時に指定する場合，接続がアクティブになったことを確認してください。

```
SQLDataSet1->SQLConnection = SQLConnection1;
SQLConnection1->Connected = true;
```

処理するデータが 1 つのデータベースサーバー上にある場合は，通常，アプリケーション内のすべての単方向データセットで 1 つの接続コンポーネントを共用します。ただし，サーバーの制約によって 1 つの接続で複数の文が許されない場合，データセットごとに異なる接続を使用する必要があるかもしれません。そのデータベースサーバーでデータセットごとに異なる接続が必要かどうかは，MaxStmtsPerConn プロパティを読み出して確認してください。1 つの接続を通じて実行できる文の数がサーバーで制限されている場合，TSQLConnection は必要に応じてデフォルトで接続を生成します。使用する接続を厳密に追跡する場合は，AutoClone プロパティを **false** に設定してください。

SQLConnection プロパティを設定するには，その前に TSQLConnection コンポーネントを設定して，データベースサーバーと必要な接続パラメータ（サーバー上のデータベース，サーバーを実行しているコンピュータのホスト名，ユーザー名，パスワードなど）を特定する必要があります。

TSQLConnection の設定

TSQLConnection で接続を開くには，そのデータベース接続に必要な情報を指定する必要があります。つまり，使用するドライバと，そのドライバに渡す接続パラメータの両方を指定しなければなりません。

ドライバの識別

ドライバは DriverName プロパティで識別されます。この値は，インストールされている dbExpress ドライバの名前（INTERBASE，ORACLE，MYSQL，DB2 など）です。このドライバ名は 2 つのファイルに関連付けられています。

- dbExpress ドライバ。これは，dbexpint.dll，dbexpora.dll，dbexpmys.dll，dbexpdb2.dll などのダイナミックリンクライブラリです。
- クライアント側でのサポート用にデータベースのベンダーから提供されるダイナミックリンクライブラリ

この 2 つのファイルとデータベース名との関係は，dbxdrivers.ini というファイルに保存されます。このファイルは，dbExpress ドライバをインストールするときに更新されます。DriverName の値を指定すると，SQL 接続コンポーネントによって dbxdrivers.ini からファイルが自動的に検索されるため，通常はこれらのファイルを気にかける必要はありません。DriverName プロパティを設定すると，TSQLConnection によって，関連する dll の名前が LibraryName と VendorLib プロパティに自動的に設定されます。いったん LibraryName と VendorLib が設定されると，アプリケーションは dbxdrivers.ini

に依存しません。（つまり，実行時に DriverName プロパティを設定しない限り，アプリケーションの配布時に dbxdrivers.ini を付属させる必要はありません）。

接続パラメータの指定

Params プロパティは，名前 / 値のペアが記述された文字列リストです。それぞれのペアは，Name=Value の形式で記述されます。Name はパラメータの名前で，Value は割り当てる値です。

具体的にどのパラメータが必要なのかは，使用するデータベースによって異なります。ただし，Database パラメータはどのサーバーの場合でも必要になります。その値は使用するサーバーによって異なります。たとえば，InterBase では，Database は .gdb ファイルの名前であり，ORACLE や DB2 では，それぞれ TNSNames.ora 内のエントリとクライアント側のノード名です。

一般的なパラメータとしては，ほかにも User_Name（ログイン時に使用する名前），Password（User_Name に対するパスワード），HostName（サーバーが存在するマシンの名前または IP アドレス），TransIsolation（使用するトランザクションが，他のトランザクションによって実行された変更を認識するレベル）などがあります。ドライバ名を指定すると，そのドライバに必要なすべてのパラメータが Params プロパティに読み込まれ，デフォルト値に初期化されます。

Params は文字列リストであり，設計時にパラメータを編集できます（オブジェクトインスペクタで Params プロパティをダブルクリックして，文字列リストエディタで編集します）。実行時には，Params::Values プロパティを使って，各パラメータに値を割り当てます。

接続に名前を付ける

DatabaseName と Params プロパティだけで接続を指定する方法はいつでも実行することができますが，一定の組み合わせに対して名前を付け，その名前で接続を識別する方が簡単です。dbExpress のデータベースとパラメータの組み合わせに対して名前を付けることができます。このデータは dbxconnections.ini というファイルに保存されます。この組み合わせを，接続名と呼びます。

いったん接続名を定義すると，ConnectionName プロパティに有効な接続名を設定するだけでデータベース接続を識別できます。ConnectionName を設定すると，DriverName と Params プロパティは自動的に設定されます。ConnectionName を設定して Params プロパティを編集すると，パラメータの値を保存されているパラメータと一時的に異なる状態にできますが，DriverName プロパティを変更すると，Params と ConnectionName プロパティは両方ともクリアされます。

接続名の利点が発揮されるのは，アプリケーションの開発時に使うデータベース（たとえば Local InterBase）と，実行時に使うデータベース（たとえば ORACLE）が異なる場合です。このような場合，システム上の DriverName と Params について，アプリケーションの開発時と実行時でその値が異なることになります。このようなときには，2 つの dbxconnections.ini ファイルを使って，接続の記述を切り替えることができます。開発時には，開発時バージョンの dbxconnections.ini から DriverName と Params をアプリケーションで読み込みます。アプリケーションを配布してからは，「実際の」データベースに対応した別バージョンの dbxconnections.ini からこの値を読み込みます。ただしこのためには，実行時に DriverName と Params プロパティを再度読み込むように接続コンポーネントに指示しなければなりません。これには 2 つの方法があります。

- LoadParamsOnConnect プロパティを **true** に設定する。こうすることによって，接続が開かれるときに DriverName と Params が dbxconnections.ini 中の ConnectionName に関連付けられている値に TSQLConnection によって自動的に設定されます

- LoadParamsFromIniFile メソッドを呼び出す。このメソッドは，DriverName と Params を，dbxconnections.ini ファイル（または別に指定されたファイル）中の ConnectionName に関連付けられている値に設定します。このメソッドを使うと，接続を開く前に特定のパラメータ値を変更できます。

接続エディタの使い方

接続名と，それに関連するドライバおよび接続パラメータとの関係は，dbxconnections.ini ファイルに保存されます。この関連付けは，接続エディタを使って作成し変更できます。

接続エディタを起動するには，TSQLConnection コンポーネントをダブルクリックします。接続エディタに，使用可能なドライバのドロップダウンリスト，現在選択されているドライバの接続名リスト，および現在選択されている接続名に対する接続パラメータリストが表示されます。

このダイアログボックスでドライバと接続名を指定すると，使用する接続を指定できます。目的の環境を選択したら，［接続の確認］ボタンをクリックして設定に誤りがないかどうかを確認します。

さらに，このダイアログボックスから dbxconnections.ini 中の名前付き接続を編集することもできます。

- パラメータテーブルのパラメータ値を編集すると，現在選択している名前付き接続を変更できます。［OK］をクリックしてダイアログボックスを終了すると，新しく設定した値が dbxconnections.ini に保存されます。
- ［接続の追加］ボタンをクリックすると，新規の名前付き接続を定義できます。表示されたダイアログボックスで，使用するドライバと新規の接続名を指定します。接続に名前を付けたら，パラメータを編集して目的の接続を指定します。［OK］ボタンをクリックして，新規の接続を dbxconnections.ini に保存します。
- ［接続の削除］ボタンをクリックすると，現在選択している名前付き接続を dbxconnections.ini から削除できます。
- ［コネクション名の変更］ボタンをクリックすると，現在選択している名前付き接続の名前を変更できます。パラメータに対して編集した内容のすべてが，［OK］ボタンをクリックしたときに新規の名前で保存されることに注意してください。

表示データの指定

単方向データセットが表すデータを指定する方法はいくつかあります。どの方法を選択するかは，使用している単方向データセットの種類と，情報を取り出すのが，単一のデータベーステーブルからか，問い合わせの結果か，あるいはストアドプロシージャからかによって決まります。

TSQLDataSet コンポーネントを使う場合は，データセットがデータを取得するソースを CommandType プロパティで指定します。CommandType は，次の値を取ることができます。

- ctQuery：CommandType が ctQuery の場合，TSQLDataSet は指定した問い合わせを実行します。問い合わせが SELECT コマンドの場合，データセットは結果のレコードセットになります。
- ctTable：CommandType が ctTable の場合，TSQLDataSet はすべてのレコードを指定したテーブルから取得します。

- ctStoredProc：CommandType が ctStoredProc の場合，TSQLDataSet はストアドプロシージャを実行します。ストアドプロシージャがカーソルを返した場合，データセットには返されたレコードが入ります。

メモ また，単方向データセットには，サーバー上で利用可能なデータについてのメタデータを入力できます。その手順については，26-12 ページの「メタデータを単方向データセットに取得する」を参照してください。

問い合わせの結果を表示する

問い合わせの使用は，レコードセットを指定するもっとも一般的な方法です。問い合わせは，簡単に言えば SQL で記述されたコマンドです。問い合わせの結果を表すのに，TSQLDataSet または TSQLQuery が使用できます。

TSQLDataSet を使用する場合，CommandType プロパティを ctQuery に設定し，問い合わせ文のテキストを CommandText プロパティに入力します。TSQLQuery を使用する場合は，かわりに SQL プロパティに問い合わせを入力します。これらのプロパティは，汎用データセットや問い合わせタイプのデータセットのすべてで同じように機能します。この内容については，22-41 ページの「問い合わせの指定」で詳しく説明されています。

問い合わせの設定時にパラメータ（変数）を使用すると，設計時または実行時にその値を変化させることができます。パラメータでは，SQL 文に表示されるデータ値を置換できます。問い合わせにパラメータを使用し，その値を指定する方法については，22-43 ページの「問い合わせでパラメータを使用する」を参照してください。

SQL には，UPDATE 問い合わせのように，サーバー上で何らかの操作を行うだけでレコードを返さない問い合わせを定義します。この種の問い合わせについては，26-9 ページの「レコードを返さないコマンドの実行」で説明しています。

テーブルのレコードを表す

基になっている 1 つのデータベーステーブル内の全項目と全レコードを表す場合，SQL を直接記述しなくても，TSQLDataSet または TSQLTable を使って問い合わせを生成させることができます。

メモ サーバーの処理効率が問われるときは，自動生成問い合わせにたよらず，明示的に問い合わせを作成するとよいでしょう。自動生成問い合わせは，テーブル内の全項目を 1 つずつ指定する代わりにワイルドカードを使用します。このためサーバー側の処理効率がわずかに遅くなることもあります。自動生成される問い合わせに使われるワイルドカード（*）は，サーバー上の項目が変更されたときの対応性に優れています。

TSQLDataSet を使用してテーブルを表す

単一データベーステーブルの全項目，全レコードを取り出す問い合わせを TSQLDataSet に生成させるには，CommandType プロパティを ctTable に設定します。

CommandType が ctTable の場合，TSQLDataSet は 2 つのプロパティ値に基づいて問い合わせを生成します。

- CommandText は，TSQLDataSet オブジェクトが表すデータベーステーブル名を指定します。
- SortFieldNames は，データのソートに使用する項目名をソートキーの順で列記します。

次に指定例を示します。

```
SQLDataSet1->CommandType = ctTable;
SQLDataSet1->CommandText = "Employee";
SQLDataSet1->SortFieldNames = "HireDate,Salary"
```

TSQLDataSet は次の問い合わせを生成します。これは，Employee テーブルの全レコードを HireDate と Salary の順でソートします。

```
select * from Employee order by HireDate, Salary
```

TSQLTable を使用してテーブルを表す

TSQLTable を使用する場合は，TableName プロパティを使ってテーブルを指定します。

データセット内の項目の順序を指定するには，インデックスを指定する必要があります。これには 2 つの方法があります。

- IndexName プロパティに，サーバー上で定義されたインデックス名を設定する
- IndexFieldNames プロパティに，ソートする項目名のリストをセミコロンで区切って設定する。IndexFieldNames は TSQLDataSet の SortFieldNames プロパティと同じように機能しますが，区切り記号はカンマではなくセミコロンを使用します。

ストアドプロシージャの結果を表示する

ストアドプロシージャは，いくつかの SQL 文のセットに名前を付けて SQL サーバーに保存したものです。実行するストアドプロシージャを指定する方法は，使用している単方向データセットの種類によって異なります。

TSQLDataSet を使用している場合，次のようにストアドプロシージャを指定します。

- CommandType プロパティに ctStoredProc を設定する
- CommandText プロパティの値として，ストアドプロシージャ名を指定する

```
SQLDataSet1->CommandType = ctStoredProc;
SQLDataSet1->CommandText = "MyStoredProcName";
```

TSQLStoredProc を使用している場合は，次のようにストアドプロシージャ名を StoredProcName プロパティ値として指定するだけで十分です。

```
SQLStoredProc1->StoredProcName = "MyStoredProcName";
```

ストアドプロシージャの名前を指定した後で，ストアドプロシージャの入力パラメータ値の指定が必要なこともあります。また，ストアドプロシージャの実行後に，出力パラメータの値をアプリケーションで取得することが必要な場合もあります。ストアドプロシージャのパラメータの扱いについては，22-49 ページの「ストアドプロシージャのパラメータの操作」を参照してください。

データの取得

データソースを指定したら，アプリケーションでアクセスする前にデータを取得する必要があります。データセットがデータを取得したら，データソースを通してデータセットにリンクされたデータベース対応のコントロールは自動的にデータ値を表示し，プロバイダを通してデータセットにリンクされたクライアントデータセットにはレコードを入力できます。

ほかのデータセットと同様，単方向データセットにはデータを取得する 2 つの方法があります。

- Active プロパティを **true** に設定する（設計時にオブジェクトインスペクタを使用するか，または実行時にコードで設定する）

```
CustTable->Active = true;
```

- 実行時に Open メソッドを呼び出す

```
CustQuery->Open();
```

Active プロパティや Open メソッドは，サーバーからレコードを取得する単方向データセットに対して使用します。そのレコードが SELECT 問い合わせ（CommandType が ctTable の場合は自動生成問い合わせを含む）から得られるか，ストアドプロシージャから得られるかには関係しません。

データセットの準備

問い合わせやストアドプロシージャをサーバー上で実行する前に，それを「準備する」必要があります。データセットの準備とは，dbExpress とサーバーが SQL 文とパラメータのリソースを割り当てるという意味です。CommandType が ctTable の場合，これはデータセットが SELECT 問い合わせを生成するときです。サーバーによってバインドされないパラメータは，この時点で問い合わせに追加されます。

Active を **true** に設定するか，Open メソッドを呼び出したときに，単方向データセットは自動的に準備されます。データセットを閉じると，文を実行するために割り当てられていたリソースは解放されます。問い合わせまたはストアドプロシージャを複数回実行する場合は，最初に開く前に明示的にデータセットを準備することにより処理効率を改善できます。明示的にデータセットを準備するには，その Prepared プロパティを **true** に設定します。

```
CustQuery->Prepared = true;
```

データセットを明示的に準備する場合，文を実行するために割り当てられたリソースは，Prepared を **false** にするまで解放されません。

パラメータ値や SortFieldNames プロパティを変更する場合など，実行前に必ずデータセットの準備ができているようにするには，Prepared プロパティを **false** に設定します。

複数データセットの取得

一部のストアドプロシージャは，複数のレコードセットを返します。データセットを開くと，データセットは最初のレコードセットだけを取得します。それ以外のレコードセットにアクセスするには，次のように NextRecordSet メソッドを呼び出します。

```
TCustomSQLDataSet *DataSet2 = SQLStoredProc1->NextRecordSet(nRows);
```

NextRecordSet は，次のレコードセットへのアクセスを提供する新規に作成された TCustomSQLDataSet コンポーネントを返します。つまり NextRecordSet を初めて呼び出すと，2 番目のレコードセットのデータセットを返します。NextRecordSet をもう一度呼び出すと 3 番目のデータセットを返し，こうしてレコードセットがなくなるまで続きます。それ以上のデータセットがなくなると，NextRecordSet は NULL を返します。

レコードを返さないコマンドの実行

単方向データセットが表す問い合わせやストアドプロシージャがレコードを返さない場合でも，単方向データセットを使用できます。そのコマンドには，SELECT 文以外のデータ定義言語（DDL）文またはデータ操作言語（DML）文を使用する文があります（たとえば，INSERT，DELETE，UPDATE，CREATE INDEX，ALTER TABLE コマンドはレコードを返しません）。コマンドに使用される言語はサーバー固有ですが，通常は SQL 言語の SQL-92 標準に準拠しています。

実行する SQL コマンドは，使用しているサーバーに受け入れられる必要があります。単方向データセットは SQL を評価せず，実行もしません。単方向データセットは，単にコマンドをサーバーに渡すだけです。

メモ コマンドがレコードを返さない場合は，単方向データセットを使用する必要はありません。レコードセットへのアクセスを提供するデータセットメソッドは必要ないからです。データベースサーバーに接続する SQL 接続コンポーネントは，サーバー上でコマンドを直接実行するために使用できます。詳しくは，21-10 ページの「サーバーにコマンドを送信する」を参照してください。

実行するコマンドの指定

単方向データセットでは，実行するコマンドの指定方法は，コマンドがデータセットを返す場合も返さない場合も同じです。つまり次のようになります。

TSQLDataSet を使用する場合，コマンドを指定するために CommandType および CommandText プロパティを使用します。

- CommandType が ctQuery の場合，CommandText はサーバーに渡す SQL 文です。
- CommandType が ctStoredProc の場合，CommandText は実行するストアドプロシージャ名です。

TSQLQuery を使用する場合は，サーバーに渡す SQL 文を指定するために SQL プロパティを使用します。

TSQLStoredProc を使用する場合は，実行するストアドプロシージャの名前を指定するために StoredProcName プロパティを使用します。

レコードを取り出しているときと同じ方法でコマンドを指定するように，レコードを返す問い合わせやストアドプロシージャと同じ方法で問い合わせパラメータやストアドプロシージャパラメータを扱うことができます。詳しくは，22-43 ページの「問い合わせでパラメータを使用する」および 22-49 ページの「ストアドプロシージャのパラメータの操作」を参照してください。

コマンドの実行

レコードを返さない問い合わせやストアドプロシージャを実行する場合には，Active プロパティと Open メソッドは使用しません。そのかわり次のようにします。

- データセットが TSQLDataSet か TSQLQuery のインスタンスの場合は，ExecSQL メソッドを使用する

```
FixTicket->CommandText = "DELETE FROM TrafficViolations WHERE (TicketID = 1099)";
FixTicket->ExecSQL();
```

- データセットが TSQLStoredProc のインスタンスの場合は，ExecProc メソッドを使用する

```
SQLStoredProc1->StoredProcName = "MyCommandWithNoResults";
SQLStoredProc1->ExecProc();
```

ヒント 問い合わせやストアドプロシージャを複数回実行する場合は，Prepared プロパティを true に設定したほうがよいでしょう。

サーバーメタデータの作成と変更

データを返さないコマンドのほとんどは，データの編集に使われる INSERT，DELETE，UPDATE などのコマンドと，サーバー上でテーブルやインデックス，ストアドプロシージャなどのエンティティの作成や変更を行うコマンドの 2 つのカテゴリに分類されます。

編集のために明示的に SQL コマンドを使用しない場合，単方向データセットをクライアントデータセットにリンクして，編集に関する SQL コマンドをすべて生成させることができます（18-12 ページの「同じアプリケーションで別のデータセットにクライアントデータセットを接続する」を参照）。実際，これは推奨されるアプローチです。データベース対応のコントロールは，TClientDataSet などのデータセットを通して編集を行うように設計されているからです。

しかし，サーバー上でアプリケーションがメタデータを作成または変更できる唯一の方法は，コマンドの送信です。すべてのデータベースドライバが同じ SQL 構文をサポートしているわけではありません。データベースの種類ごとにサポートされる SQL 構文やデータベースの種類の違いについては，このマニュアルの範囲外なので説明しません。特定のデータベースシステムの SQL 実装に関する最新の詳しい解説は，そのシステムに添付されているドキュメントを参照してください。

一般に，データベーステーブルを作成するには CREATE TABLE 文を，テーブルに新しいインデックスを作成するには CREATE INDEX 文を使います。CREATE DOMAIN，CREATE VIEW，CREATE SCHEMA，CREATE PROCEDURE などの CREATE 文がサポートされていれば，各種のメタデータオブジェクトの追加にはそれを使用します。

各 CREATE 文に対応して，メタデータオブジェクトを削除する DROP 文として，DROP TABLE，DROP VIEW，DROP DOMAIN，DROP SCHEMA，および DROP PROCEDURE があります。

テーブルの構造を変更するには ALTER TABLE 文を使用します。ALTER TABLE には，テーブル内に新しい要素を作成する ADD 節とテーブル内の要素を削除する DROP 節があります。たとえば，ADD COLUMN 節は新規の列をテーブルに追加し，DROP CONSTRAINT はテーブルから既存の制約を削除します。

たとえば次の文は，InterBase データベース上に GET_EMP_PROJ というストアドプロシージャを作成します。

```
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
  FOR SELECT PROJ_ID
  FROM EMPLOYEE_PROJECT
  WHERE EMP_NO = :EMP_NO
  INTO :PROJ_ID
  DO
    SUSPEND;
END
```

次のコードは，このストアドプロシージャを作成するために TSQLDataSet を使用します。データセットがストアドプロシージャ定義内のパラメータ（:EMP_NO と :PROJ_ID）を，ストアドプロシージャを作成する問い合わせのパラメータと混同しないように，ParamCheck プロパティを使用していることに注目してください。

```
SQLDataSet1->ParamCheck = false;
SQLDataSet1->CommandType = ctQuery;
SQLDataSet1->CommandText = "CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT) RETURNS
(PROJ_ID CHAR(5)) AS BEGIN FOR SELECT PROJ_ID FROM EMPLOYEE_PROJECT WHERE EMP_NO =
:EMP_NO INTO :PROJ_ID DO SUSPEND; END";
SQLDataSet1->ExecSQL();
```

マスター / 詳細のリンクカーソルのセットアップ

単方向データセットを詳細セットとして，カーソルのリンクを使ってマスター / 詳細関係を設定する方法は 2 通りあります。使用するメソッドは，使用している単方向データセットの種類によります。関係を設定した場合，単方向データセット（1 対多関係の「多」）が提供するアクセスは，マスターセット（1 対多関係の「1」）の現在のレコードに対応するレコードだけです。

TSQLDataSet と TSQLQuery では，パラメータ付き問い合わせを使ってマスター / 詳細関係を確立する必要があります。これは，このような関係を問い合わせタイプのデータセットについて作成するときに常に使用する方法です。問い合わせタイプのデータセットでのマスター / 詳細関係の作成については，22-45 ページの「マスター / 詳細関係をパラメータを使用して確立する」を参照してください。

詳細セットが TSQLTable のインスタンスのときにマスター / 詳細関係を設定するには，ほかのテーブルタイプデータセットのときと同じように，MasterSource プロパティおよび MasterFields プロパティを使用します。テーブルタイプのデータセットでのマスター / 詳細関係の作成については，22-45 ページの「マスター / 詳細関係をパラメータを使用して確立する」を参照してください。

スキーマ情報にアクセスする

サーバー上で利用可能なデータに関する情報は，2 通りの方法で得ることができます。スキーマ情報またはメタデータと呼ばれるこの情報には，テーブルやストアドプロシージャがサーバー上で利用可

能な情報および，そのテーブルやストアドプロシージャについての情報があります（テーブル内の項目，定義されているインデックス，ストアドプロシージャが使用するパラメータなど）。

このメタデータをもっとも容易に取得するには，TSQLConnection のメソッドを使用します。このメソッドを実行すると，既存の文字列リストオブジェクトまたはリストオブジェクトに，テーブル名，ストアドプロシージャ名，またはインデックス名，あるいはパラメータの記述子が得られます。この方法は，ほかのデータベース接続コンポーネントでリストにメタデータを設定する方法と同じです。これらのメソッドは，21-13 ページの「メタデータの取得」で説明されています。

さらに詳細なスキーマ情報が必要な場合は，メタデータを単方向データセットに入れることができます。この場合は単純なリストではなく，単方向データセットには，それぞれのレコードが 1 つのテーブル，ストアドプロシージャ，インデックス，項目，またはパラメータを表す形でスキーマ情報が設定されます。

メタデータを単方向データセットに取得する

データベースサーバーのメタデータを単方向データセットに入れるには，まず必要なデータを SetSchemaInfo メソッドを使用して指定しなければなりません。SetSchemaInfo は 3 つのパラメータを取ります。

- 取得するスキーマ情報（メタデータ）の種類：これには，テーブルリスト（stTables），システムテーブルリスト（stSysTables），ストアドプロシージャリスト（stProcedures），テーブルの項目リスト（stColumns），インデックスリスト（stIndexes），またはストアドプロシージャが使用するパラメータリスト（stProcedureParams）があります。各種の情報は，リスト内のアイテムを表すために使用する項目セットが違います。このデータセットの構造の詳細については，26-13 ページの「メタデータデータセットの構造」を参照してください。
- テーブル名またはストアドプロシージャ名（項目，インデックス，ストアドプロシージャパラメータの情報を取得する場合）：これら以外のスキーマ情報を取得する場合，このパラメータは NULL です。
- 返された名前に一致するパターン：このパターンは，「Cust%」などの SQL パターンで，ワイルドカードとして「%」（任意の長さの任意の文字）と「_」（任意の 1 文字）を使用します。パターン内でパーセントまたは下線記号を使用するには，その文字を 2 つ表記します（%% または _ _）。パターンを使用しない場合，このパラメータは NULL です。

メモ テーブルについてのスキーマ情報（stTables）を取得する場合，SQL 接続の TableScope プロパティの値に応じて得られたスキーマ情報は，通常のテーブル，システムテーブル，ビュー，シノニムについてのものです。

次の呼び出しは，全システムテーブル（メタデータが入力されているサーバーのテーブル）を列挙するテーブルを要求しています。

```
SQLDataSet1->SetSchemaInfo(stSysTable, "", "");
```

SetSchemaInfo への呼び出し後データセットを開くと，得られたデータセットには各テーブルのレコードがあり，そのレコードにはテーブル名，種類，スキーマ名などの列があります。サーバーがメタデータを保存するのにシステムテーブルを使用しない場合（たとえば MySQL），データセットを開いてもレコードはありません。

前の例は，最初のパラメータにだけ使用できます。次に，「MyProc」という名前のストアドプロシージャの入力パラメータのリストを取得するケースを考えてみましょう。ここでは，ストアドプロシージャを書いたプログラマが，パラメータに入力パラメータか出力パラメータを示すプレフィクス（「inName」や「outValue」など）を付けているものとします。SetSchemaInfo は，次のように呼び出します。

```
SQLDataSet1->SetSchemaInfo(stSysTable, "", "");
```

得られるデータセットは，各パラメータのプロパティを記述する列のある入力パラメータのテーブルです。

メタデータのデータセットを使用後にデータを取得する

SetSchemaInfo の呼び出しの後，データセットを使用して問い合わせまたはストアドプロシージャを実行するには 2 つの方法があります。

- CommandText プロパティを変更して，データを取得する問い合わせ，テーブル，またはストアドプロシージャを指定する
- 先頭のパラメータを stNoSchema に設定して，SetSchemaInfo を呼び出す。この場合，データセットは CommandText の現在値で指定されたデータの取得に変わります。

メタデータデータセットの構造

TSQLDataSet を使用してアクセスできるメタデータのそれぞれの種類に対して，要求された種類のアイテムについての情報が入力された定義済みの列（項目）セットがあります。

テーブル情報

テーブル情報（stTables または stSysTables）を要求すると，得られたデータセットには各テーブルについてのレコードがあります。レコードには次の列があります。

表 26.1　テーブルをリストアップしたメタデータのテーブル列

列名	項目型	内容
RECNO	ftInteger	各レコードを一意に識別するレコード番号
CATALOG_NAME	ftString	テーブルが含まれているカタログ（データベース）名。これは SQL 接続コンポーネントの Database パラメータと同じ
SCHEMA_NAME	ftString	テーブルの所有者を表すスキーマ名
TABLE_NAME	ftString	テーブル名。この項目によりデータセットのソート順が決まる
TABLE_TYPE	ftInteger	テーブルの種類を表す。次の値の 1 つ以上の合計 1: テーブル 2: ビュー 4: システムテーブル 8: シノニム 16: 一時テーブル 32: ローカルテーブル

ストアドプロシージャ情報

ストアドプロシージャ情報（stProcedures）を要求する場合，得られたデータセットには，各ストアドプロシージャについてのレコードがあります。レコードには次の列があります。

表 26.2 ストアドプロシージャをリストするメタデータのテーブル列

列名	項目型	内容
RECNO	ftInteger	各レコードを一意に識別するレコード番号
CATALOG_NAME	ftString	ストアドプロシージャが含まれているカタログ（データベース）名。これは SQL 接続コンポーネントの Database パラメータと同じ
SCHEMA_NAME	ftString	ストアドプロシージャの所有者を表すスキーマ名
PROC_NAME	ftString	ストアドプロシージャ名。この項目によりデータセットのソート順が決まる
PROC_TYPE	ftInteger	ストアドプロシージャの種類を表す。次の値の 1 つ以上の合計 1: プロシージャ（戻り値なし） 2: 関数（値を返す） 4: パッケージ 8: システムプロシージャ
IN_PARAMS	ftSmallint	入力パラメータの数
OUT_PARAMS	ftSmallint	出力パラメータの数

項目情報

指定したテーブルの項目情報（stColumns）を要求した場合，得られるデータセットには，各項目に対するレコードがあります。レコードには次の列があります。

表 26.3 項目をリストアップしたメタデータのテーブル列

列名	項目型	内容
RECNO	ftInteger	各レコードを一意に識別するレコード番号
CATALOG_NAME	ftString	項目をリストアップしたテーブルが含まれているカタログ（データベース）名。これは SQL 接続コンポーネントの Database パラメータと同じ
SCHEMA_NAME	ftString	項目の所有者を表すスキーマ名
TABLE_NAME	ftString	項目があるテーブル名
COLUMN_NAME	ftString	項目名この値により，データセットのソート順が決まる
COLUMN_POSITION	ftSmallint	テーブル内の列位置
COLUMN_TYPE	ftInteger	項目値の種類を識別します。次の値の 1 つ以上の合計です。 1: 行 ID 2: 行バージョン 4: 自動インクリメント項目 8: デフォルト値のある項目
COLUMN_DATATYPE	ftSmallint	列のデータ型。これは sqllinks.h で定義されている項目型を表す論理定数値
COLUMN_TYPENAME	ftString	データ型を表す文字列。これは，COLUMN_DATATYPE と COLUMN_SUBTYPE に入力されている情報と同じだが，一部の DDL 文で使用されている形式になる
COLUMN_SUBTYPE	ftSmallint	列のデータ型のサブタイプ。これは sqllinks.pas で定義されているサブタイプを表す論理定数値

表 26.3 項目をリストアップしたメタデータのテーブル列（つづき）

列名	項目型	内容
COLUMN_PRECISION	ftInteger	項目型のサイズ（文字列内の文字数，バイト型項目のバイト数，BCD 値の有効桁数，ADT 項目のメンバー数など）
COLUMN_SCALE	ftSmallint	BCD 値では小数点の右の桁数，ADT 項目と配列項目では下位オブジェクト数
COLUMN_LENGTH	ftInteger	項目値の保存に必要なバイト数
COLUMN_NULLABLE	ftSmallint	項目を空にできるかどうかを示す論理値（0 の場合は項目に値が必要）

インデックス情報

テーブルのインデックス情報（stIndexes）を要求した場合，得られるデータセットには各項目に対するレコードがあります。マルチレコードインデックスは，複数のレコードを使用して記述されます。データセットには次の列があります。

表 26.4 インデックスをリストアップしたメタデータのテーブル列

列名	項目型	内容
RECNO	ftInteger	各レコードを一意に識別するレコード番号
CATALOG_NAME	ftString	インデックスが含まれているカタログ（データベース）名。これは SQL 接続コンポーネントの Database パラメータと同じ
SCHEMA_NAME	ftString	インデックスの所有者を表すスキーマ名
TABLE_NAME	ftString	インデックスが定義されているテーブル名
INDEX_NAME	ftString	インデックス名。この項目によりデータセットのソート順が決まる
PKEY_NAME	ftString	一次キー名を指定する
COLUMN_NAME	ftString	インデックス内の項目（列）名
COLUMN_POSITION	ftSmallint	インデックス内の項目位置
INDEX_TYPE	ftSmallint	インデックスの種類を表す。次の値の 1 つ以上の合計 1: 非ユニーク 2: ユニーク 4: 一次キー
SORT_ORDER	ftString	インデックスが昇順（a）と降順（d）のどちらでソートされるかを示す
FILTER	ftString	インデックス付きのレコードを制限するフィルタ条件を記述する

ストアドプロシージャのパラメータ情報

ストアドプロシージャのパラメータ情報（stProcedureParams）を要求した場合，得られるデータセットには，各パラメータに対するレコードがあります。レコードには次の列があります。

表 26.5 パラメータをリストアップしたメタデータのテーブル列

列名	項目型	内容
RECNO	ftInteger	各レコードを一意に識別するレコード番号
CATALOG_NAME	ftString	ストアドプロシージャが含まれているカタログ（データベース）名。これは SQL 接続コンポーネントの Database パラメータと同じ
SCHEMA_NAME	ftString	ストアドプロシージャの所有者を表すスキーマ名
PROC_NAME	ftString	パラメータを含むストアドプロシージャ名
PARAM_NAME	ftString	パラメータ名。この項目によりデータセットのソート順が決まる

表 26.5 パラメータをリストアップしたメタデータのテーブル列（つづき）

列名	項目型	内容
PARAM_TYPE	ftSmallint	パラメータの種類を表す。これは TParam オブジェクトの ParamType プロパティと同じ
PARAM_DATATYPE	ftSmallint	パラメータのデータ型。これは sqllinks.h で定義されている項目型を表す論理定数値
PARAM_SUBTYPE	ftSmallint	パラメータのデータ型のサブタイプ。これは sqllinks.h で定義されているサブタイプを表す論理定数値
PARAM_TYPENAME	ftString	データ型を表す文字列。これは，PARAM_DATATYPE と PARAM_SUBTYPE に入力されている情報と同じだが，一部の DDL 文で使用されている形式になる
PARAM_PRECISION	ftInteger	浮動小数点値の最大桁数。文字列項目またはバイト型項目の場合は最大バイト数
PARAM_SCALE	ftSmallint	浮動小数点値の小数点以下の桁数
PARAM_LENGTH	ftInteger	パラメータ値の保存に必要なバイト数
PARAM_NULLABLE	ftSmallint	パラメータを空にできるかどうかを示す論理値（0 の場合はパラメータに値が必要）

dbExpress アプリケーションのデバッグ

データベースアプリケーションのデバッグ時には，接続コンポーネントを通してデータベースサーバーとやり取りする SQL メッセージ，およびプロバイダコンポーネントや dbExpress ドライバによって自動的に生成されるメッセージを監視するとよいかもしれません。

TSQLMonitor を使って SQL コマンドを監視する

TSQLConnection は，その関連コンポーネントである TSQLMonitor を使って，これらのメッセージを捕捉し文字列のリストに保存します。TSQLMonitor の機能と BDE で使用する SQLMonitor ユーティリティの機能はよく似ていますが，TSQLMonitor が監視するのは 1 つの TSQLConnection コンポーネントに関連するコマンドのみであり，dbExpress が管理するすべてのコマンドではありません。

TSQLMonitor を使用する手順は次のとおりです。

1. TSQLMonitor コンポーネントを，SQL コマンドを監視する TSQLConnection コンポーネントがあるフォームまたはデータモジュールに追加します。
2. SQLConnection プロパティを TSQLConnection コンポーネントに設定します。
3. SQL モニタの Active プロパティを **true** に設定します。

SQL コマンドをサーバーに送信すると，SQLMonitor の TraceList プロパティが自動的に更新され，捕捉されたすべての SQL コマンドがリストに記録されます。

FileName プロパティに値を指定し，AutoSave プロパティを **true** に設定しておくと，このリストをファイルに保存できます。AutoSave が設定してあると，新しいメッセージが記録されるたびに TraceList プロパティの内容がファイルに保存されます。

メッセージが記録されるたびに発生するファイル保存処理のオーバーヘッドを避けたい場合は，OnLogTrace イベントハンドラを使うと，一定数のメッセージが記録された時点でのみファイルを保存するように動作させることができます。たとえば次のイベントハンドラは，メッセージが 10 個記録されると TraceList の内容を保存し，ログをクリアします。そのため，リストのサイズが大きくなりすぎることはありません。

```
void __fastcall TForm1::SQLMonitor1LogTrace(TObject *Sender, void *CBInfo)
{
  TSQLMonitor *pMonitor = dynamic_cast<TSQLMonitor *>(Sender);
  if (pMonitor->TraceCount == 10)
  {
    // 一意なファイル名を作成する
    AnsiString LogFileName = "c: \\ log";
    LogFileName = LogFileName + IntToStr(pMonitor->Tag);
    LogFileName = LogFileName + ".txt"
    pMonitor->Tag = pMonitor->Tag + 1;
    // ログの内容を保存してリストをクリア
    pMonitor->SaveToFile(LogFileName);
    pMonitor->TraceList->Clear();
}
```

メモ このイベントハンドラを実際に使う場合は，アプリケーションの終了時に，10 個に満たないメッセージを含む部分的なリストも保存する必要があります。

コールバックを使って SQL コマンドを監視する

TSQLMonitor を利用しない方法もあります。SQL 接続コンポーネントの SetTraceCallbackEvent メソッドを使うと，SQL コマンドを追跡する方法をアプリケーションでカスタマイズできます。SetTraceCallbackEvent のパラメータは，TSQLCallbackEvent 型のコールバックと，そのコールバック関数に渡すユーザー定義の値の 2 つです。

コールバック関数は，CallType と CBInfo の 2 つのパラメータを取ります。

- CallType は未使用
- CBInfo は構造を指すポインタ。このレコードには，カテゴリ（CallType と同じ），SQL コマンドのテキスト，および SetTraceCallbackEvent メソッドに渡すユーザー定義の値を記述します

コールバックは CBRType 型の値を返します。通常は cbrUSEDEF です。

SQL 接続コンポーネントがコマンドをサーバーに送信するか，サーバーからエラーメッセージが返されるたびに，作成したコールバックが dbExpress ドライバから呼び出されます。

注意 TSQLConnection オブジェクトに TSQLMonitor コンポーネントが関連付けられている場合は，SetTraceCallbackEvent を呼び出さないでください。TSQLMonitor はコールバックのメカニズムを使って動作しますが，TSQLConnection が一度にサポートするコールバックは 1 つに限られています。

第27章

クライアントデータセットの使い方

クライアントデータセットは，すべてのデータをメモリ上に保持している特別なデータセットです。メモリに格納されているデータの操作は，midas.dll によってサポートされます。クライアントデータセットでデータを保管するために使用する形式は，自己包含型で容易に移送できる形式であり，これによりクライアントデータベースでは次のことが行えます。

- ディスク上の専用ファイルを読み書きする。ファイルベースのデータセットとして動作する。このメカニズムをサポートしているプロパティおよびメソッドは，27-32 ページの「クライアントデータセットでファイルデータを使用する」で説明されています。
- データベースサーバーからのデータをキャッシュアップデートする。キャッシュアップデートをサポートするクライアントデータセットの機能についての詳細は，27-15 ページの「クライアントデータセットをキャッシュアップデートに使用する」に記載されている
- 多層アプリケーションのクライアント部分のデータを表す。このように機能するには，クライアントデータセットは，27-23 ページの「クライアントデータセットでデータプロバイダを使用する」で説明されているように，外部プロバイダを操作する必要がある。多層データベースアプリケーションのについての詳細は，第 29 章「多層アプリケーションの作成」に記載されている
- データセット以外のソースからのデータを表す。クライアントデータセットでは，外部プロバイダからのデータを使用できるため，特別なプロバイダにより各種の情報源をクライアントデータセットを操作するように適応させることができる。たとえば，XML プロバイダを使用することにより，クライアントデータセットが XML ドキュメント内の情報を表すようにできる

ファイルベースのデータ，キャッシュアップデート，外部プロバイダからのデータ（XML ドキュメントの操作や多層アプリケーションでの使用など），または「ブリーフケースモデル」アプリケーションなどの方法の組み合わせにクライアントデータセットを使用するしないに関係なく，クライアントデータセットでデータの操作に関してサポートしているさまざまな機能を利用することができます。

クライアントデータセットを使用するデータ操作

ほかのデータセットと同様クライアントデータセットを使用すると，データソースコンポーネントを使用してデータベース対応コントロールにデータを供給できます。データベース対応コントロールでデータベース情報を表示する方法については，第 19 章「データコントロールの使い方」を参照してください。

クライアントデータセットでは，TDataSet から継承したすべてのプロパティおよびメソッドを実装します。この汎用データセットの動作についての詳細な説明は，第 22 章「データセットについて」を参照してください。

さらに，クライアントデータセットでは，次のようなテーブル型のデータセットに共通の多数の機能も実装します。

- インデックスを持つレコードのソート
- インデックスを使ってレコードを検索する方法
- 範囲でレコードを制限する
- マスター / 詳細関係の作成
- 読み書きのアクセスを制御する
- 基になるデータセットを作成する
- データセットを空にする
- クライアントデータセットの同期をとる

これらの機能についての詳細は，22-24 ページの「テーブルタイプのデータセットの使い方」を参照してください。

クライアントデータセットは，すべてのデータをメモリ上に保持しているという点でほかのデータセットと異なります。このため，一部のデータベース機能をサポートするには追加の機能や配慮を必要とすることがあります。この章では，クライアントデータセットで導入されるこれらの一般的な機能および違いの一部について説明します。

クライアントデータセット内のデータナビゲーション

アプリケーションが標準のデータベース対応コントロールを使用する場合，ユーザーはクライアントデータセットのレコードの間を，コントロールの組み込み動作を使用して移動できます。プログラムによってレコード間を移動することも可能で，その場合は First，Last，Next，Prior などの標準データセットメソッドを使います。これらのメソッドについての詳細は，22-5 ページの「データセットの操作」を参照してください。

大半のデータセットとは異なり，クライアントデータセットは RecNo プロパティを使うことによって，データセット内の特定のレコードにカーソルを置くこともできます。通常，アプリケーションは現在のレコードのレコード番号を調べるために RecNo を使います。クライアントデータセットは RecNo に特定のレコード番号を設定することによって，そのレコードを現在のレコードにすることができます。

表示されるレコードを制限する

一時的にデータの一部しか使用できないように制限するため，アプリケーションは範囲とフィルタを使用できます。範囲またはフィルタを適用すると，クライアントデータセットはメモリキャッシュ内にある全データを表示しなくなります。そのかわりに，範囲やフィルタの条件に適合するデータだけを表示します。フィルタの使い方については，22-12 ページの「フィルタを使って編集する」を参照してください。範囲についての詳細は，22-29 ページの「範囲でレコードを制限する」を参照してください。

たいていのデータセットでは，フィルタ文字列は，解析されて SQL コマンドにされ，それがデータベースサーバー上で実行されます。そのため，フィルタ文字列にどんな操作が使用されるかは，サーバーの SQL 方言によって制限されます。クライアントデータセットは自身のフィルタサポートを実装しており，ほかのデータセットよりも多くの操作が可能です。たとえば，クライアントデータセットを使うとき，部分文字列を返す文字列演算子や，日付 / 時刻値を解析する演算子，その他多くをフィルタ式の中で使用できます。クライアントデータセットは，BLOB 項目について，または ADT 項目や配列項目などの複雑な項目タイプについてのフィルタが可能です。

クライアントデータセットでフィルタに使用できる各種の演算子および関数を，フィルタをサポートする他のデータセットとの比較とともに，以下に示します。

表 27.1　クライアントデータセット内のフィルタサポート

演算子 または関数	例	他のデータベースでのサポート	コメント
比較			
=	State = 'CA'	可	
<>	State <> 'CA'	可	
>=	DateEntered >= '1/1/1998'	可	
<=	Total <= 100,000	可	
>	Percentile > 50	可	
<	Field1 < Field2	可	
BLANK	State <> 'CA' or State = BLANK	可	フィルタ中に明示的に設定しない限り，空白レコードは現れない
IS NULL	Field1 IS NULL	不可	
IS NOT NULL	Field1 IS NOT NULL	不可	
論理演算子			
and	State = 'CA' and Country = 'US'	可	
or	State = 'CA' or State = 'MA'	可	
not	not (State = 'CA')	可	
算術演算子			
+	Total + 5 > 100	ドライバに依存する	数値，文字列，または日付（時刻）+ 数値に適用する
-	Field1 - 7 <> 10	ドライバに依存する	数値，日付，または日付（時刻）- 数値に適用する
*	Discount * 100 > 20	ドライバに依存する	数値のみに適用する

表 27.1 クライアントデータセット内のフィルタサポート（つづき）

演算子または関数	例	他のデータベースでのサポート	コメント
/	Discount > Total / 5	ドライバに依存する	数値のみに適用する
文字列関数			
Upper	Upper(Field1) = 'ALWAYS'	不可	
Lower	Lower(Field1 + Field2) = 'josp'	不可	
Substring	Substring(DateFld,8) = '1998' Substring(DateFld,1,3) = 'JAN'	不可	値は，第 2 引数の位置から終わりまで，または第 3 引数の文字数分まで。先頭文字は位置 1 である
Trim	Trim(Field1 + Field2) Trim(Field1, '-')	不可	先頭と末尾にある第 3 引数で指定された文字列を削除する。第 3 引数が指定されていない場合は，空白を削除する
TrimLeft	TrimLeft(StringField) TrimLeft(Field1, '$') <> ''	不可	「Trim」を参照
TrimRight	TrimRight(StringField) TrimRight(Field1, '.') <> ''	不可	「Trim」を参照
日付 / 時刻関数			
Year	Year(DateField) = 2000	不可	
Month	Month(DateField) <> 12	不可	
Day	Day(DateField) = 1	不可	
Hour	Hour(DateField) < 16	不可	
Minute	Minute(DateField) = 0	不可	
Second	Second(DateField) = 30	不可	
GetDate	GetDate - DateField > 7	不可	現在の日付と時刻を表す
Date	DateField = Date(GetDate)	不可	日付 / 時刻値の日付部分を返す
Time	TimeField > Time(GetDate)	不可	日付 / 時刻値の時刻部分を返す
その他			
Like	Memo LIKE '%filters%'	不可	ESC 句なしの SQL-92 に似た動作をする。BLOB 項目に適用する場合，FilterOptions で大小文字の区別が決定される
In	Day(DateField) in (1,7)	不可	SQL-92 に似た動作をする。第 2 引数は，すべて同じ型の値のリストである
*	State = 'M*'	可	部分比較用のワイルドカード

範囲またはフィルタを適用しても，クライアントデータセットはまだメモリ上にすべてのレコードを保存しています。範囲またはフィルタは，クライアントデータセットのデータ間を移動したりデータを表示したりするコントロールがどのレコードを使用できるかできないか決めるだけです。

メモ プロバイダからデータを取得するときに，プロバイダにパラメータを渡すことによりクライアントデータセットに保管するデータを制限することもできます。詳細は，27-28 ページの「パラメータでレコードを制限する」を参照してください。

データの編集

クライアントデータセットは，データをメモリ内のデータパケットとして表します。このパケットは，クライアントデータセットの Data プロパティの値です。ただし，デフォルトでは編集内容は Data プロパティに保存されません。そのかわり，ユーザーまたはプログラムによって行われる挿入，削除，および変更は，Delta プロパティで表される内部変更ログに保存されます。変更ログの使用には 2 つの目的があります。

- 更新内容をデータベースサーバーまたは外部プロバイダコンポーネントに適用するには，変更ログが必要である
- 変更ログによって，変更を元に戻す機能を洗練された形でサポートできる

LogChanges プロパティを使用すると，ログへの記録を停止できます。LogChanges が **true** の場合，変更内容はログに記録されます。LogChanges が **false** の場合，変更は Data プロパティに対して直接行われます。ファイルベースのアプリケーションでは，元に戻す機能のサポートを必要としない場合に変更ログを使用不可にできます。

変更ログに格納された編集内容は，アプリケーションによって削除されるまで，変更ログに残ります。アプリケーションは，次の場合に編集内容を削除します。

- 変更を元に戻す
- 変更を保存する

メモ クライアントデータセットをファイルに保存しても，変更ログの編集内容は削除されません。データセットを再読み込みするとき，Data と Delta プロパティの内容はデータが保存されたときと同じです。

変更を元に戻す

レコードのオリジナルの内容が変更されないまま Data に残っていても，ユーザーがレコードを編集してそのレコードを放置し，再びそのレコードに戻ると，最後に変更したレコード内容が表示されます。ユーザーまたはアプリケーションが 1 つのレコードを何回も編集すると，それぞれのレコードの変更内容が変更ログ内に別々のエントリとして格納されます。

レコードに対する全変更内容を格納することによって，レコードの以前の状態を復元する必要が生じた場合に，元に戻す操作を複数レベルでサポートできます。

- レコードに対する最後の変更内容を削除するには，UndoLastChange を呼び出します。UndoLastChange は論理パラメータ FollowChange をとります。このパラメータは，復元されたレコード上にカーソルを戻すか（**true**），カーソルは現在レコードに置いたままにするか（**false**）を指示します。1 つのレコードに対して複数の変更がある場合，UndoLastChange を呼び出すたびに編集内容が 1 つずつ削除されます。UndoLastChange は，成功したか失敗したかを示す論理値を返します。削除が成功した場合，UndoLastChange は **true** を返します。ChangeCount プロパティを使用すると，元に戻す変更がまだあるかどうか調べることができます。ChangeCount は，変更ログに保存された変更の数を示します。
- 1 つのレコードに対する変更内容を 1 つずつ削除するかわりに，すべての変更を一度に削除できます。1 つのレコードに対する変更内容をすべて削除するには，レコードを選択してから RevertRecord を呼び出します。RevertRecord は，現在のレコードに対するすべての変更内容を変更ログから削除します。

- 削除されたレコードを復元するには，まず，StatusFilter プロパティを [usDeleted] に設定します。これで，削除されたレコードが「見える」ようになります。次に，復元するレコードに移動し，RevertRecord を呼び出します。最後に，StatusFilter プロパティを [usModified, usInserted, usUnmodified] に設定します。これで，データセットの編集済みバージョン（今は復元されたレコードが入っている）が再び表示されます。
- 編集中の任意の時点で，変更ログの現在の状態を SavePoint プロパティを使用して保存できます。SavePoint を読み込むと，変更ログ内での現在位置を示すマーカーが返されます。後で，SavePoint を読んだ後に起こったすべての変更を元に戻したいときは，SavePoint を前に読み込んだ値に設定します。アプリケーションは，複数のセーブポイント値を取得できます。ただし，変更ログのセーブポイントまでのバックアップをとると，アプリケーションが読んだそれ以降のすべてのセーブポイントの値は無効です。
- 変更ログに記録されたすべての変更内容を破棄するには，CancelUpdates を呼び出します。CancelUpdates は変更ログをクリアし，全レコードに対するすべての編集内容を完全に破棄します。CancelUpdates を呼び出す場合は注意が必要です。CancelUpdates を呼び出した後は，ログ内に存在していた変更内容は回復できなくなります。

変更の保存

クライアントデータセットが変更ログから変更内容を取り込む場合，クライアントデータセットがデータをファイルに保存しているか，プロバイダから得たデータを表しているのかによって，クライアントデータセットは異なるメカニズムを使います。どちらのメカニズムが使用されても，全更新内容が取り込まれると変更ログは自動的に空になります。

ファイルベースのアプリケーションでは，変更内容を Data プロパティで表されるローカルキャッシュに簡単にマージできます。これらのアプリケーションでは，ほかのユーザーが行った変更をローカルな編集内容に反映させることを心配する必要はありません。変更ログを Data プロパティにマージするには，MergeChangeLog メソッドを呼び出します。27-33 ページの「変更内容をデータにマージする」でこの方法について説明します。

キャッシュアップデートを行ったり，外部プロバイダコンポーネントからのサーバーデータを表すためにクライアントデータセットを使用していると，MergeChangeLog は使用できません。更新されたレコードがデータベース（またはソースデータセット）内に保存されているデータになるには，変更ログ内の情報が必要です。かわりに ApplyUpdates を呼び出して変更内容をデータベースサーバーまたはソースデータセットに書き込み，変更内容がデータベースに正常にコミットされたときだけ Data プロパティを更新することができます。この過程についての詳細は，27-20 ページの「更新を適用する」を参照してください。

データ値を制限する

クライアントデータセットでは，データに対してユーザーが行った編集結果に制約を強制できます。これらの制約は，ユーザーが変更を変更ログに登録しようとしたときに適用されます。いつでもカスタム制約を指定できます。これによって，ユーザーがクライアントデータセットに登録する値に対してアプリケーションで定義した自分独自の制限を加えることができます。

さらに，クライアントデータセットが，BDE を使用してアクセスされるサーバーデータを表すときには，データベースサーバーからインポートされたデータ制約も実行します。クライアントデータセットで外部プロバイダコンポーネントを操作する場合，これらの制約をクライアントデータセットに送信するかどうかをプロバイダ側で制御し，使用するかどうかをクライアントデータセット側で制御できます。プロバイダで制約がデータパケットに含まれるかどうかを制御する方法についての詳細は，28-12 ページの「サーバー制約の処理」を参照してください。クライアントデータベース側でサーバー制約の実行をオフにする方法とその理由についての詳細は，27-29 ページの「サーバーからの制約の処理」を参照してください。

カスタム制約の指定

クライアントデータセットの項目コンポーネントのプロパティを使用して，データユーザーが入力できる内容に，独自の制約を課すことができます。項目コンポーネントには，制約の指定に使用できるプロパティが 2 つあります。

- DefaultExpression プロパティは，ユーザーが項目値を入力しなかった場合に項目に割り当てられるデフォルト値を定義します。データベースサーバーやソースデータセットでも項目にデフォルト式を割り当てる場合，更新データがデータベースサーバーまたはソースデータセットに戻される前に式の割り当てが行われるので，クライアントデータセット側のデフォルト式が優先します。
- CustomConstraint プロパティを使用すると，項目値がポストされる前に満足させなければならない制約条件を割り当てることができます。この方法で定義されるカスタム制約は，サーバーからインポートされる制約に加えて適用されます。項目コンポーネントに対するカスタム制約の操作については，23-20 ページの「カスタム制約の作成」を参照してください。

また，クライアントデータセットの Constraints プロパティを使ってレコードレベルの制約を作成することができます。Constraints は TCheckConstraint オブジェクトのコレクションであり，その各オブジェクトが個別の条件を表します。TCheckConstraint オブジェクトの CustomConstraint プロパティを使用すると，レコードのポスト時にチェックされる独自の制約を追加できます。

ソートとインデックス付け

インデックスの使用は，アプリケーションにとっていくつかのメリットがあります。

- クライアントデータセットで，データ位置を迅速に特定できる
- 更新範囲を編集可能なレコードに制限できる
- アプリケーションでは，参照テーブルやマスター / 詳細フォームなど他のデータセットとの関係を設定できる
- インデックスによってレコードの表示順序が指定される

クライアントデータセットがサーバーデータを表すか，または外部プロバイダを使う場合，受信するデータに基づいて，デフォルトのインデックスとソート順が継承されます。デフォルトインデックスは DEFAULT_ORDER と呼ばれます。この順序付けは使用できますが，デフォルトインデックスの変更や削除はできません。

デフォルトインデックスのほかに，クライアントデータセットは別のインデックスを保持します。このインデックスは CHANGEINDEX と呼ばれ，変更ログ（Delta プロパティ）に格納された変更レ

コードを対象とします。CHANGEINDEX は，Delta で指定された変更が適用された場合に，クライアントデータセット内の全レコードを出現順に並べます。CHANGEINDEX は，DEFAULT_ORDER から継承した順序付けに基づいています。DEFAULT_ORDER と同様，CHANGEINDEX インデックスの変更や削除はできません。

ほかのインデックスを使用したり，独自にインデックスを作成できます。以降のセクションでは，クライアントデータセットに対してインデックスを作成 / 使用する方法を説明します。

メモ テーブル型データセットのインデックスで資料を検討したい場合もあります。この資料はクライアントデータセットに適用されます。この資料については，22-25 ページの「インデックスを持つレコードのソート」と 22-29 ページの「範囲でレコードを制限する」で説明しています。

新しいインデックスの追加

クライアントデータセットにインデックスを追加する方法は 3 通りあります。

- 実行時にクライアントデータセット内のレコードをソートする一時的なインデックスを作成するには，IndexFieldNames プロパティを使用します。項目名をセミコロンで区切って指定します。リスト内で指定した項目名の順序でインデックスの順序が決まります。

  これは，インデックスを追加する方法としては最もパワーの低い方法です。この方法では降順インデックスも，大文字小文字を無視するインデックスも指定できないし，作成されるインデックスはグループ化をサポートしません。この方法で作成したインデックスは，データセットを閉じた後は持続せず，クライアントデータセットをファイルに保存してもインデックスは保存されません。

- 実行時にグループ化として使用可能なインデックスを作成するには，AddIndex を呼び出します。AddIndex を使用すると，次のようなインデックスのプロパティを指定できます。
  - インデックス名。これは，実行時にインデックスを切り替えるのに使用されます。
  - インデックスを構成する項目：インデックスはこれらの項目を使用してレコードをソートし，これらの項目で特定の値を持つレコードを検索します。
  - インデックスによるレコードのソート方法：デフォルトでは，インデックスは昇順になります（マシンのロケールに基づく）。このデフォルトのソート順は，大文字と小文字を区別します。オプションの設定により，インデックス全体が大文字小文字の区別を無視したり降順にソートするようにできます。または，大文字小文字の区別をしないでソートする項目リストや，降順にソートする項目リストを指定できます。
  - インデックスのグループ化サポートのデフォルトレベル

  AddIndex メソッドを使用して作成したインデックスは，クライアントデータセットを閉じた後は保持されません（つまり，クライアントデータセットを再び開いた時点で，インデックスは失われています）。データセットが閉じているときには，AddIndex は呼び出せません。AddIndex を使用して追加されたインデックスは，クライアントデータセットをファイルに保存するときに保存されません。

- 3 番目の方法は，クライアントデータセットの作成時にインデックスを作成することです。クライアントデータセットを作成する前に，IndexDefs プロパティを使用して目的のインデックスを指定します。CreateDataSet を呼び出すと，元となるデータセットとともにインデックスが作成され

ます。クライアントデータセットの作成についての詳細は，22-37 ページの「テーブルの作成と削除」を参照してください。

AddIndex と同様，データセットと一緒に作成したインデックスもグループ化をサポートし，項目ごとに昇順 / 降順のソートを指定でき，項目ごとに大文字小文字を区別させたり，無視させることができます。この方法で作成したインデックスは常に保持され，クライアントデータセットをファイルに保存するときに保存されます。

ヒント クライアントデータセットでは，内部的に計算される項目を使ってインデックス付けとソートができます。

インデックスの削除と切り替え

クライアントデータセットに作成したインデックスを削除するには，DeleteIndex を呼び出して，削除するインデックス名を指定します。DEFAULT_ORDER インデックスと CHANGEINDEX インデックスは削除できません。

複数のインデックスが使用可能な場合に，別のインデックスを使うには，IndexName プロパティを使って目的のインデックスを選択します。設計時には，オブジェクトインスペクタで IndexName プロパティのドロップダウンボックスに表示される使用可能なインデックスの中から選択できます。

インデックスを使用してデータをグループ化する

クライアントデータセットでインデックスを使用するとき，レコードのソート順は自動的に決まります。この順序のため，通常隣り合うレコードでは，インデックスを構成する項目値が重複します。たとえば，次の SalesRep 項目と Customer 項目に基づいてインデックスが設定されている受注テーブルを見てみましょう。

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

ソート順のせいで，SalesRep 列の値が重複しています。SalesRep が 1 のレコードでは，隣接する Customer 列の値が重複しています。つまりデータは SalesRep によってグループ化され，SalesRep のグループ内では Customer によってグループ化されます。それぞれのグループ化にはレベルがあります。この例では SalesRep グループはレベル 1（ほかのグループにネストされていない），Customer グループはレベル 2（レベル 1 のグループ内にネストされている）です。グループ化レベルは，インデックス内の項目順序に対応しています。

クライアントデータセットでは，現在レコードが任意のグループ化レベル内のどこにあるか特定できます。これによってアプリケーションは，レコードがグループ内の先頭にあるか中央か，それとも最後にあるかによって，レコード表示を変えることができます。たとえばグループの先頭レコードだけ項目値を表示し，重複する値を表示しないようにできます。前のテーブルでこれを行うと，結果は次のようになります。

SalesRep	Customer	OrderNo	Amount
1	1	5	100
		2	50
	2	3	200
		6	75
2	1	1	10
	3	4	200

現在レコードがグループ内のどこに位置するか知るには，GetGroupState メソッドを使います。GetGroupState は，グループレベルを指定する整数を受け取り，現在レコードがグループ内のどこにあるか示す値を返します（先頭レコード，最終レコード，またはそのどれでもない）。

インデックスを作成する場合，インデックスの項目数までの範囲でサポートするグループ化レベルを指定できます。GetGroupState は，インデックスが追加項目に基づいてレコードをソートしても，そのレベルを超えるグループ情報は提供できません。

計算値を表す

ほかのデータセットと同様に，クライアントデータセットに計算項目を追加できます。計算項目の値は，通常同じレコード内のほかの項目に基づいて動的に計算されます。計算項目の使い方については，23-7 ページの「計算項目の定義」を参照してください。

しかしクライアントデータセットは，内部計算項目を使うことにより，項目が計算される時期を最適化できます。内部計算項目については，後述の「クライアントデータセットで内部計算項目を使用する」を参照してください。

また保守される集合体を使用して，複数レコード値を集計する計算値を作成するよう，クライアントデータセットに指定できます。保守される集合体について詳しくは，27-11 ページの「保守される集合体の使用」を参照してください。

クライアントデータセットで内部計算項目を使用する

ほかのデータセットではレコードが変更されたり，ユーザーが現在レコードの項目を編集するたびに，アプリケーションは計算項目の値を計算しなければなりません。これは OnCalcFields ハンドラによって行われます。

クライアントデータセットでは，計算値をクライアントデータセットのデータに保存することによって，計算項目が再計算される回数を最小限にすることができます。計算値がクライアントデータセットに保存される場合，ユーザーが現在レコードを編集したときには再計算が必要ですが，アプリケーションは現在レコードが変わるたびに再計算する必要はありません。計算値をクライアントデータセットのデータに保存するには，計算項目ではなく内部計算項目を使います。

内部計算項目は計算項目と同様，OnCalcFields イベントハンドラで計算されます。ただし，クライアントデータセットの State プロパティにチェックマークを付けることによって，イベントハンドラを最適化できます。State が dsInternalCalc のときは，内部計算項目を再計算する必要があります。State が dsCalcFields のときは，普通の計算項目だけを再計算する必要があります。

内部計算項目を使用するには，クライアントデータセットを作成する前に，項目を内部計算項目として定義しなければなりません。持続的項目を使用するか項目定義を使用するかに応じて，次の方法のいずれかを使用します。

- 持続的項目を使用する場合，項目エディタで InternalCalc を選択することによって，項目を内部計算項目として定義できます。
- 項目定義を使用する場合，該当する項目定義の InternalCalcField プロパティを **true** に設定します。

メモ ほかの種類のデータセットも，内部計算項目を使用します。ただし，ほかのデータセットでは，計算は OnCalcFields イベントハンドラでは行われません。かわりに，BDE またはリモートデータベースサーバーによって自動的に計算されます。

保守される集合体の使用

クライアントデータセットは，レコードグループ別でのデータ集計をサポートしています。データセット内のデータを編集するとこれらの集計は自動的に更新されるので，集計されたデータを「保守される集合体」と呼びます。

最も単純な形の場合，保守される集合体により，クライアントデータセット内のある列の合計などの情報が取得できます。保守される集合体には柔軟性があって，さまざまな集計をサポートしており，グループ化をサポートしているインデックス項目によって定義されたグループ別に小計を求めることができます。

集合体を指定する

クライアントデータセットのレコードを集計するよう指定するには，Aggregates プロパティを指定します。Aggregates は，集合指定（TAggregate）のコレクションです。集合指定をクライアントデータセットに追加するには設計時にコレクションエディタを使用するか，実行時に Aggregates の Add メソッドを使います。集合体の項目コンポーネントを作成する場合，項目エディタで集合値用の持続的項目を作成します。

メモ 集合項目を作成すると，該当する集合体オブジェクトがクライアントデータセットの Aggregates プロパティに自動的に追加されます。持続的集合項目を作成するときには，それらを明示的に追加しないでください。持続的集合項目の作成については，23-9 ページの「集合項目の定義」を参照してください。

各集合体の Expression プロパティは，その集合体が表す集計計算を示します。Expression は，次のような単純な集計式の場合があります。

```
Sum(Field1)
```

また，次のような数値項目の情報を組み合わせる複雑な式のこともあります。

```
Sum(Qty * Price) - Sum(AmountPaid)
```

集合体の式には，表 27.2 に示す集計演算子が 1 個以上含まれます。

表 27.2 保守される集合体の集計演算子

演算子	使い方
Sum	数値型の項目 / 式の値を合計する
Avg	数値型または日付 / 時刻型の項目 / 式の平均値を計算する
Count	項目 / 式の値が空でない件数を示す。
Min	文字列型，数値型，日付 / 時刻型の項目 / 式について，最小値を示す
Max	文字列型，数値型，日付 / 時刻型の項目 / 式について，最大値を示す

集計演算子は，項目値に対して，またはフィルタを作成するのと同じ演算子と項目値からできた式に対して作用します（ただし集計演算子のネストはできません）。式を作成するには，集計値と別の集計値に，または集計値と定数に対して演算子を使います。ただし，集計値と項目値を組み合わせることはできません。そのような式はあいまいになるからです（どのレコードが項目値を指定するか示されない）。これらの規則の例を，次の式で示します。

```
Sum(Qty * Price)                { 有効 -- 項目を使った式の集計 }
Max(Field1) - Max(Field2)       { 有効 -- 集計値を使用した式 }
Avg(DiscountRate) * 100         { 有効 -- 集計値と定数を使用した式 }
Min(Sum(Field1))                { 無効 -- ネストされた集計 }
Count(Field1) - Field2          { 無効 -- 集計値と項目を使用した式 }
```

レコードグループの集計

デフォルトでは，保守される集合体は，クライアントデータセット内のすべてのレコードを集計するように計算されます。ただし，それにかわって，グループ内のレコードだけを集計するように指定できます。これによって，項目値が共通するレコードグループごとに小計を出すなどの中間集計ができます。

レコードグループについて保守される集合体を指定する前に，適切なグループ化をサポートするインデックスを使用しなければなりません。グループ化のサポートについての詳しくは，27-9 ページの「インデックスを使用してデータをグループ化する」を参照してください。

集計の目的に合うようにデータをグループ化するインデックスが整ったら，集合体の IndexName および GroupingLevel プロパティを指定して，使用するインデックスと，そのインデックスのどのグループまたはサブグループによって集計するレコードを定義するかを示します。

たとえば，受注テーブルが SalesRep によってグループ化され，SalesRep の中でさらに Customer によってグループ化されている場合，次のような部分データを考えてみましょう。

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

次のコードは，各営業担当者（SalesRep）の総売上金額を示す保守される集合体を設定します。

```
Agg->Expression = "Sum(Amount)";
Agg->IndexName = "SalesCust";
Agg->GroupingLevel = 1;
Agg->AggregateName = "Total for Rep";
```

ある営業担当者のそれぞれの顧客を集計する集合体を追加するには，保守される集合体をレベル 2 で作成します。

レコードグループについて集計する保守される集合体は，特定のインデックスと関連付けられます。Aggregates プロパティには，別のインデックスを使用する集合体を指定できます。ただしデータセット全体を集計する集合体と，現在のインデックスを使用する集合体だけが有効です。現在のインデックスを変更すると，有効な集合体も変更されます。ある時点でどの集合体が有効か知るには，ActiveAggs プロパティを使います。

集合体の値を取得する

保守される集合体の値を取得するには，その集合体を表す TAggregate オブジェクトの Value メソッドを呼び出します。Value メソッドは，クライアントデータセットの現在レコードを含むグループについて，保守される集合体の値を返します。

クライアントデータセット全体について集計するときは，いつでも Value を呼び出して保守される集合体の値を取得できます。ただし，グループ化された情報を集計するときは，必ず現在レコードが集計したいグループ内にあるように注意する必要があります。このためグループの先頭レコードに移動したときとか，グループの最終レコードに移動したときなど，明確に指定されたときに集合体の値を求めるとよいでしょう。GetGroupState メソッドを使用すると，現在レコードがどのグループ内にあるか調べられます。

保守された集合体をデータベース対応コントロールで表示するには，項目エディタを使用して持続的集合項目コンポーネントを作成します。項目エディタで集合項目を指定すると，クライアントデータセットの Aggregates は自動的に更新されて，該当する集合指定が含められます。AggFields プロパティには新規の集合項目コンポーネントが含まれ，FindField メソッドはそれを返します。

データを別のデータセットからコピーする

設計時にデータを別のデータセットからコピーするには，クライアントデータセットを右クリックして［ローカルデータの割り当て］を選択します。ダイアログが表示されて，プロジェクトで使用できる全データセットが一覧表示されます。コピーするデータと構造を選択してから，［OK］を選択します。コピー元のデータセットをコピーすると，クライアントデータセットは自動的にアクティブになります。

実行時に別のデータセットからコピーするには，そのデータを直接割り当てるか，またはコピー元が別のクライアントデータセットなら，カーソルをコピーします。

データを直接割り当てる

別のデータセットからクライアントデータセットにデータを割り当てるには，クライアントデータセットの Data プロパティを使います。Data は，OleVariant 形式のデータパケットです。データパケットは，別のクライアントデータセット，またはプロバイダを使用してそれ以外のデータセットか

ら取得できます。いったんデータパケットが Data に割り当てられると，データソースコンポーネントによってクライアントデータセットに接続されたデータベース対応コントロールに，その内容が自動的に表示されます。

サーバーデータを表すクライアントデータセットまたは外部プロバイダコンポーネントを使うクライアントデータセットを開くと，データパケットが自動的に Data に割り当てられます。

クライアントデータセットがプロバイダを使わないときは，別のクライアントデータセットからデータを次のようにコピーできます。

```
ClientDataSet1->Data = ClientDataSet2->Data;
```

メモ 別のクライアントデータセットの Data プロパティをコピーするとき，変更ログも一緒にコピーしますが，そのコピーには適用されていたフィルタや範囲は反映されません。フィルタや範囲を含めるには，コピー元データセットからカーソルをコピーしなければなりません。

クライアントデータセット以外のデータセットからコピーする場合，データセットプロバイダコンポーネントを作成し，それをコピー元データセットにリンクすると，データをコピーできます。

```
TempProvider = new TDataSetProvider(Form1);
TempProvider->DataSet = SourceDataSet;
ClientDataSet1->Data = TempProvider->Data;
delete TempProvider;
```

メモ Data プロパティに直接割り当てる場合，新規データパケットは既存のデータにマージされません。そのかわりに，以前のすべてのデータが置換されます。

別のデータセットからデータをコピーするのではなく変更をマージしたい場合は，プロバイダコンポーネントを使用しなければなりません。前例のようにデータセットプロバイダを作成しますが，データセットプロバイダをマージ先のデータセットに結び付けて，データプロパティをコピーするかわりに ApplyUpdates メソッドを使用します。

```
TempProvider = new TDataSetProvider(Form1);
TempProvider->DataSet = ClientDataSet1;
TempProvider->ApplyUpdates(SourceDataSet->Delta, -1, ErrCount);
delete TempProvider;
```

クライアントデータセットのカーソルをコピーする

クライアントデータセットでは，CloneCursor メソッドを使用すると，実行時にデータの 2 番目のビューを扱えるようになります。CloneCursor によって，2 番目のクライアントデータセットは元のクライアントデータセットのデータを共有できます。これは，元のデータをすべてコピーするよりコストがかかりませんが，データが共用されるため，2 番目のクライアントデータセットを変更すると，元のクライアントデータセットにも影響してしまいます。

CloneCursor は 3 つのパラメータを取ります。最初のパラメータ Source には，コピーするクライアントデータセットを指定します。残りの 2 つのパラメータ（Reset と KeepSettings）は，データ以外の情報をコピーするかどうかを示します。この情報とは，フィルタ，現在のインデックス，マスターテーブルへのリンク（元のデータセットが詳細セットの場合），ReadOnly プロパティ，および接続コンポーネントまたはプロバイダへのリンクです。

Reset と KeepSettings が **false** の場合，コピーされたクライアントデータセットが開かれ，コピー元データセットの設定値を使用してコピー先のプロパティが設定されます。Reset が **true** ならば，コピー先データセットのプロパティにはデフォルト値が適用されます（インデックスもフィルタもな

し，マスターテーブルなし，ReadOnly は **false**，指定されている接続コンポーネントもプロバイダもなし）。KeepSettings が **true** ならば，コピー先データセットのプロパティは変更されません。

アプリケーション固有の情報をデータに追加する

アプリケーション開発者は，クライアントデータセットの Data プロパティにカスタム情報を追加できます。この情報はデータパケットとバンドルされるので，データをファイルまたはストリームに保存するときに一緒に保存されます。データを別のデータセットにコピーするときに，それもコピーされます。オプションで，この情報を Delta プロパティに含めることができ，プロバイダはクライアントデータセットから更新内容を受け取るときにこの情報を読めるようにできます。

アプリケーション固有の情報を Data プロパティと一緒に保存するには，SetOptionalParam メソッドを使います。このメソッドを使用すると，このデータを含んでいる OleVariant を特定の名前で保存できます。

このアプリケーション固有の情報を取り出すには，GetOptionalParam メソッドを使って，その情報が格納されたときに使われた名前を渡します。

クライアントデータセットをキャッシュアップデートに使用する

デフォルトでは，たいていのデータセットのデータを編集して，レコードを削除または登録するごとに，データセットにより，トランザクションの生成，そのレコードの削除，またはデータベースサーバーへのそのレコードの書き込みが行われます。データベースに変更内容を書き込むことときに問題が発生すると，アプリケーションに対し直ちに，レコードを登録したときにデータセットで例外が発生したと通知されます。

データセットでリモートデータベースサーバーを使っている場合，この方法だと，現在のレコードを編集した後新しいレコードに移動するごとにアプリケーションとサーバーとの間にネットワークトラフィックが発生するためパフォーマンスが低下することがあります。ネットワークトラフィックを最小限に抑えるために，ローカルでキャッシュアップデートを使用するようにします。キャッシュアップデートを使うと，アプリケーションで，データベースから取り出したデータをキャッシュに入れてローカルで編集してから，キャッシュ内の更新内容を 1 回のトランザクションでデータベースに適用できます。キャッシュアップデートを使用すると，データセットに対する更新内容（変更内容の登録やレコードの削除）は，データセットの基となるテーブルに直接書き込むのではなくローカルのキャッシュに記憶されます。変更を完了したら，アプリケーションは，キャッシュに入っている変更内容をデータベースに書き込んでキャッシュをクリアするメソッドを呼び出します。

キャッシュアップデートによって，トランザクション時間を最小限にし，ネットワークトラフィックを減らすことができます。ただし，キャッシュに入れられているデータは，アプリケーションに対してローカルであり，トランザクション制御下にはありません。したがって，ローカルなメモリ上にコピーされたデータに対して作業している間に，ほかのアプリケーションがそのデータの基となるデータベーステーブルのデータを変更することもあります。また，キャッシュアップデートを適用するまでは変更を見ることはできません。そのため，キャッシュアップデートは，変わりやすいデータを扱

うアプリケーションには適していません。データベースに変更をマージしようとするとあまりにも多くの矛盾をまねいたりするからです。

BDE や ADO にはキャッシュアップデートの代替メカニズムが用意されていますが，キャッシュアップデートにクライアントデータセットを使うと，次のようないくかのメリットがあります。

- データセットがマスター / 詳細関係でリンクされているときの更新の処理がユーザーに代わって行われます。このため，複数のリンクされたデータセットへの更新は正しい順序で適用されます。
- クライアントデータセットを使用すると，ユーザーによる更新プロセスの制御が最大になります。レコードを更新するために生成される SQL に影響を及ぼすプロパティを設定したり，マルチテーブルの結合からレコードを更新するときに使用するテーブルを指定することができるほか，BeforeUpdateRecord イベントハンドラから手動で更新することさえできます。
- キャッシュアップデートをデータベースサーバーに適用したときにエラーが発生すると，クライアントデータセット（およびデータセットプロバイダ）のみから，データセットからの元の（未編集の）値と失敗した更新の新しい（編集された）値に加えてデータベースサーバー上の現在のレコード値に関する情報が得られます。
- クライアントデータセットを使って，更新全体がロールバックされるまでの許容更新エラー数を指定できます。

キャッシュアップデートの使い方の概要

キャッシュアップデートを使うには，アプリケーションで以下の処理をする必要があります。

1. **編集するデータを指示します。**方法は，使用しているデータセットの種類によります。
   - TClientDataSet を使用している場合は，編集するデータを表すプロバイダコンポーネントを指定します。これについては，27-24 ページの「プロバイダを指定する」で説明しています。
   - 特定のデータアクセスメカニズムと関連するクライアントデータセットを使用している場合は，次のようにする必要があります。
     - データベースサーバーを識別するには，DBConnection プロパティを適切な接続コンポーネントに設定します。
     - 表示するデータを指定するには，CommandText プロパティと CommandType プロパティを指定します。CommandType は，CommandText が，実行する SQL 文か，ストアドプロシージャ名か，またはテーブル名かを示します。CommandText が問い合わせまたはストアドプロシージャの場合，Params プロパティを使用して出力パラメータを与えます。
     - 必要であれば，Options プロパティを使用して，ネストされた詳細セットと BLOB データがデータパケットに含まれるか別々に取得されるか，特定の種類の編集（挿入，変更，削除など）を無効にするかどうか，1 回の更新で複数のサーバーレコードを更新するか，更新の適用時にクライアントデータセットのレコードをリフレッシュするか，などの動作を指定します。Options は，プロバイダの Options プロパティと同一です。したがって，関連していないオプションや適切でないオプションも設定できます。たとえば，クライアントデータセットは持続的項目のあるデータセットからそのデータを取得しないので，poIncFieldProps を加える理由はありません。逆に，デフォルトで含まれる poAllowCommandText は除外したくありません。これは，CommandText プロパティを無効

にするからです。クライアントデータセットではこのプロパティを使用して必要なデータを指定します。プロバイダの Options プロパティについては，28-5 ページの「データパケットに影響するオプションを設定する」を参照してください。

2. **データを表示して編集します。**新規レコードの挿入を許可し，既存レコードの削除をサポートします。各レコードの元のコピーとそのレコードへの編集は，どちらもメモリに格納されます。この手順は，27-5 ページの「データの編集」で説明します。
3. **必要に応じて，追加レコードを取得します。**デフォルトで，クライアントデータセットにより，すべてのレコードが取り出され，メモリに格納されます。データセット内に多数のレコードまたは大きい BLOB 項目を持つレコードがある場合は，これを変更して，クライアントデータセットが表示できるだけのレコードを取得し，必要に応じて再度取得するようにできます。レコード取得プロセスの制御方法についての詳細は，27-25 ページの「ソースデータセットまたはドキュメントにデータを要求する」を参照してください。
4. **必要ならば，レコードをリフレッシュします。**時間が経つうちに，他のユーザがデータベースサーバー上のデータを変更する可能性があります。その結果，クライアントデータセットのデータがサーバー上のデータからますます逸脱し，更新を適用するときにエラーが発生する可能性が高まります。まだ編集していないレコードをリフレッシュすることで，このような問題の影響を緩和できます。詳細は，27-30 ページの「レコードのリフレッシュ」を参照してください。
5. **ローカルにキャッシュした記録をデータベースに適用する**か，更新を取り消します。データベースに書き込むどのレコードでも，BeforeUpdateRecord イベントが発生します。データベースに個々のレコードを書き込んでいるときにエラーが発生した場合は，OnUpdateError イベントが発生することにより，アプリケーションは可能であればエラーを訂正し，更新を継続できます。更新が終了した場合，正しく適用された更新はすべてローカルキャッシュからクリアされます。データベースへの更新の適用のしかたについては，27-19 ページの「レコードの更新」を参照してください。

   更新を適用する代わりに，アプリケーション側でその更新を取り消し，変更内容をデータベースに書き込まずに変更ログを空にすることもできます。CancelUpdates メソッドを呼び出して，更新を取り消すことができます。キャッシュ内のすべての削除されたレコードが復元され，変更されたレコードは元の値に戻され，新たに挿入されたレコードは単に消されます。

キャッシュアップデートの対象となるデータセットの種類の選択

C++Builder には，キャッシュアップデート用の専用クライアントデータセットコンポーネントが付属しています。それぞれのクライアントデータベースは，特定のデータアクセスメカニズムと関連しています。これらについては，表 27.3 で説明されています。

表 27.3　キャッシュアップデート専用のクライアントデータセット

クライアントデータセット	データアクセスメカニズム
TBDEClientDataSet	BDE（ボーランドデータベースエンジン）
TSQLClientDataSet	dbExpress
TIBClientDataSet	InterBase Express

さらに，汎用クライアントデータセット（TClientDataSet）を外部プロバイダおよびソースデータセットとともに使用してキャッシュアップデートを実行できます。TClientDataSet を外部プロバイダとともに使用する方法についての詳細は，27-23 ページの「クライアントデータセットでデータプロバイダを使用する」を参照してください。

メモ それぞれのデータアクセスメカニズムと関連する専用クライアントデータセットは実際には，プロバイダとソースデータセットも使用します。ただし，プロバイダとソースデータセットは両方とも，クライアントデータセットの内部にあります。

専用クライアントデータセットのうちの 1 つを使用してキャッシュアップデートするのが一番簡単です。しかし，TClientDataSet を外部プロバイダとともに使用した方が好ましい場合もあります。

- 専用クライアントデータセットのないデータアクセスメカニズムを使用している場合は，TClientDataSet を外部プロバイダコンポーネントとともに使用する必要があります。たとえば，XML ドキュメントやカスタムデータセットのデータを使用する場合です。
- マスター / 詳細関係で関係しているテーブルを操作している場合は，TClientDataSet を使用し，プロバイダを使って，マスター / 詳細関係でリンクされている 2 つのソースデータセットのマスターテーブルに接続する必要があります。クライアントデータセットでは，詳細データセットをネストされたデータセット項目とみなします。これは，マスターと詳細テーブルへの更新を正しい順序で実行できるようにするために必要な方法です。
- クライアントデータセットとプロバイダの間の通信に応答するイベントハンドラのコードを作成する場合（たとえば，クライアントデータセットがプロバイダからレコードを取得する前後），TClientDataSet を外部プロバイダコンポーネントとともに使用してください。専用クライアントデータセットは，更新を適用する際に最も重要なイベントを公開しますが（OnReconcileError, BeforeUpdateRecord および OnGetTableName），クライアントデータセットとそのプロバイダの間の通信に関連するイベントは公開しません。主に多層アプリケーションを対象としているからです。
- BDE を使用するときに，アップデートオブジェクトを使用する必要があれば外部プロバイダとソースデータセットを使用するとよいでしょう。TBDEClientDataSet の BeforeUpdateRecord イベントハンドラからアップデートオブジェクトのコードを作成することは可能ですが，ソースデータセットの UpdateObject プロパティを割り当てるだけにしたほうが簡単です。アップデートオブジェクトの使い方についての詳細は，24-39 ページの「アップデートオブジェクトを使ったデータセットの更新」を参照してください。

変更されたレコードを示す

ユーザーがクライアントデータセットを編集している間，行った編集に関するフィードバックを返すようにすると便利な場合があります。これは，特に，ユーザーが移動して［Undo］ボタンをクリックして特定の編集結果を元に戻すのを許可する場合に有用です。

UpdateStatus メソッドと StatusFilter プロパティは，実行された更新の内容に関するフィードバックを返すときに使用できます。

- UpdateStatus は，現在のレコードに対し更新があればその更新の種類を示します。次の値のいずれかです。
  - usUnmodified は，現在のレコードに変更がないことを示す

  - usModified は，現在のレコードが編集されたことを示す
  - usInserted は，ユーザーによって挿入されたレコードを示す
  - usDeleted は，ユーザーによって削除されたレコードを示す
- StatusFilter で，表示する変更ログの更新の種類を指定します。StatusFilter によるキャッシュレコードの操作方法は，フィルタによるふつうのデータの操作方法とほぼ同じです。StatusFilter は集合なので，次の値の任意の組み合わせを含むことができます。
  - usUnmodified は，変更されていないレコードを示す
  - usModified は，変更されたレコードを示す
  - usInserted は，挿入されたレコードを示す
  - usDeleted は，削除されたレコードを示す

  デフォルトでは，StatusFilter は集合［usModified，usInserted，usUnmodified］です。usDeleted をこの集合に追加して，削除されたレコードに関するフィードバックも返すようにできます。

メモ UpdateStatus および StatusFilter も，BeforeUpdateRecord および OnReconcileError イベントハンドラで使用できます。BeforeUpdateRecord についての詳細は，27-21 ページの「更新適用時の介入」を参照してください。OnReconcileError についての詳細は，27-22 ページの「更新エラーの調停」を参照してください。

次の例は，UpdateStatus メソッドを使用するレコードのアップデートステータスに関するフィードバックを返す方法を示しています。StatusFilter プロパティを変更して usDeleted を含むようにし，削除したレコードをデータセット内で表示できるようにしていると仮定します。さらに，計算項目を「Status」という名前のデータセットに追加していると仮定します。

```
void __fastcall TForm1::ClientDataSet1CalcFields(TDataSet *DataSet)
{
  switch (DataSet->UpdateStatus())
  {
    case rsUnmodified:
      ClientDataSet1Status->Value = NULL; break;
    case usModified:
      ClientDataSet1Status->Value = "M"; break;
    case usInserted:
      ClientDataSet1Status->Value = "I"; break;
    case usDeleted:
      ClientDataSet1Status->Value = "D"; break;
  }
}
```

レコードの更新

変更ログの内容はクライアントデータセットの Delta プロパティにデータパケットとして格納されます。Delta 内の変更を確定するには，クライアントデータセットが変更内容をデータベース（またはソースデータセットまたは XML ドキュメント）に適用しなければなりません。

クライアントがサーバーに更新内容を適用するときは，次の過程で行われます。

1. クライアントアプリケーションは，クライアントデータセットオブジェクトの ApplyUpdates メソッドを呼び出します。この ApplyUpdates メソッドは，クライアントデータセットの Delta プロ

パティの内容を（内部または外部）プロバイダに渡します。Delta は，クライアントデータセットで更新，挿入，削除されたレコードが入っているデータパケットです。

2. プロバイダは更新内容を適用しますが，解決できない問題レコードはキャッシュしています。サーバーが更新内容を適用する方法については，28-8 ページの「クライアントの更新リクエストに応答する」を参照してください。
3. プロバイダは処理できない全レコードを，Result データパケットでクライアントデータセットに返します。この Result データパケットには未更新のレコードがすべて入っています。エラーメッセージやエラーコードなどのエラー情報も入っています。
4. クライアントデータセットは，Result データパケットにして返された更新エラーを，1 レコードずつ整合させようとします。

更新を適用する

クライアントデータセットのデータのローカルコピーへの変更は，クライアントアプリケーションがそのデータセットの ApplyUpdates メソッドを呼び出すまで，データベースサーバー（または XML ドキュメント）には送られません。ApplyUpdates は変更ログの変更内容を取り出し，プロバイダにデータパケットとして送信します。このデータパケットは Delta と呼ばれます。（たいていのクライアントデータセットを使用するときには，プロバイダはクライアントデータセットの内部に組み込まれていることに注意してください。）

ApplyUpdates は 1 つのパラメータ MaxErrors を取ります。MaxErrors は，プロバイダが更新プロセスを停止するまでの最大許容エラー数を示します。MaxErrors が 0 の場合に更新エラーが発生すると，ただちに更新プロセス全体が中止されます。変更内容はデータベースに書き込まれず，クライアントデータセットの変更ログは変更されません。MaxErrors が -1 の場合，エラーはいくつでも許容され，変更ログには正常に適用できなかったレコードがすべて含まれます。MaxErrors が正の値で，MaxErrors で許容される数より多くのエラーが発生した場合，すべての更新が中断されます。エラーの数が MaxErrors の指定値より少ない場合は，更新が正しく適用されたすべてのレコードが，自動的にクライアントデータセットの変更ログから消去されます。

ApplyUpdates は実際に検出したエラーの数を返します。この値は必ず MaxErrors に 1 を加えた値以下でなければなりません。この戻り値は，データベースに書き込まれなかったレコード数を表します。

クライアントデータセットの ApplyUpdates メソッドは，次のように処理されます。

1. メソッドは間接的に，プロバイダの ApplyUpdates メソッドを呼び出します。プロバイダの ApplyUpdates メソッドは，更新内容をデータベース，ソースデータセット，または XML ドキュメントに書き込み，発生したエラーを修正するよう試みます。エラーのため適用できないレコードは，クライアントデータセットに戻されます。
2. それからクライアントデータセットの ApplyUpdates メソッドは，Reconcile メソッドを呼び出して，問題レコードを調停しようとします。Reconcile はエラー処理ルーチンで，OnReconcileError イベントハンドラを呼び出します。エラーを修正するため，OnReconcileError イベントハンドラのコードを記述しておかなければなりません。OnReconcileError の使用について詳しくは，27-22 ページの「更新エラーの調停」を参照してください。
3. 最後に Reconcile は正常に適用された変更内容を変更ログから削除し，Data を更新して新たに更新されたレコードを反映させます。Reconcile が完了すると，ApplyUpdates は発生したエラーの数を報告します。

重要 プロバイダ側で，更新を適用する方法を決定できない場合もあります（たとえば，ストアドプロシージャまたはマルチテーブル結合から更新を適用するとき）。クライアントデータセットおよびプロバイダコンポーネントにより，こうした状況に対応できるイベントを生成します。詳細は，以下の「更新適用時の介入」を参照してください。

ヒント プロバイダがステートレスアプリケーションサーバー上にある場合，更新内容の適用の前後で持続的ステート情報について交信したい場合があります。TClientDataSet は，更新内容を送る前に BeforeApplyUpdates イベントを受け取り，このイベントは持続的ステート情報をサーバーに送信できるようにします。更新を適用した後（ただし調停処理の前）に，TClientDataSet はアプリケーションサーバーから返される持続的なステート情報に応答できる AfterApplyUpdates イベントを受け取ります。

更新適用時の介入

クライアントデータセットによって更新が適用されるときに，プロバイダ側ではデータベースサーバーまたはソースデータセットへの挿入，削除，および変更の書き込みを処理する方法を決定します。TClientDataSet を外部プロバイダコンポーネントとともに使用する際に，そのプロバイダのプロパティとイベントを使用して，更新の適用方法に影響を及ぼすことができます。これらについては，28-8 ページの「クライアントの更新リクエストに応答する」で説明しています。

ただし，プロバイダが内部にあれば，データアクセスメカニズムと関連するクライアントデータセットの場合のように，プロパティを設定したり，イベントハンドラを用意したりできません。そのため，クライアントデータセットでは，1 つのプロパティと，内部プロバイダが更新を適用する方法に影響を及ぼす 2 つのイベントを公開します。

- UpdateMode は，更新内容の適用のためプロバイダが作成する SQL 文で，レコードを検索するために使用する項目を制御します。UpdateMode は，プロバイダの UpdateMode プロパティと同一です。プロバイダの UpdateMode プロパティについての詳細は，28-9 ページの「更新の適用方法に影響を与える」を参照してください。
- OnGetTableName でを使用すると，更新を適用するデータベーステーブル名を指定できます。これによりプロバイダは，CommandText によって指定されたストアドプロシージャまたは問い合わせからデータベーステーブルを識別できない場合に，更新用の SQL 文を生成できます。たとえば，問い合わせが 1 つのテーブルの更新しか必要としないマルチテーブルの結合を実行する場合，OnGetTableName イベントハンドラを指定すると，内部プロバイダは更新を正しく適用できます。

  OnGetTableName イベントハンドラには，内部プロバイダコンポーネント，サーバーからデータを取得した内部データセット，および生成された SQL 内で使用するテーブル名を返すパラメータの 3 つのパラメータがあります。
- BeforeUpdateRecord はデルタパケット内の各レコードで発生します。このイベントにより，レコードを挿入，削除，変更する直前に変更することができます。また（たとえば複数テーブルの更新が必要となるマルチテーブルの結合など）プロバイダが正しい SQL を生成できない場合に，更新を適用する独自の SQL 文を実行する手段が提供されます。

  BeforeUpdateRecord イベントハンドラには，内部プロバイダコンポーネント，サーバーからデータを取得した内部データセット，更新することになっているレコード上に配置されているデルタパケット，更新が挿入であるか，削除であるか，変更であるかを示す値，およびイベントハンドラが更新を実行したかどうかを示す値を返すパラメータの 5 つのパラメータがあります。簡単にするため，例では，SQL 文を項目値のみを必要とするグローバル変数であると仮定しています。

```
void __fastcall TForm1::SQLClientDataSet1BeforeUpdateRecord(TObject *Sender,
  TDataSet *SourceDS, TCustomClientDataSet *DeltaDS, TUpdateKind UpdateKind, bool &Applied)
{
  TSQLConnection *pConn := (dynamic_cast<TCustomSQLDataSet *>(SourceDS)->SQLConnection);
  char buffer[256];
  switch (UpdateKind)
  case ukModify:
    // 1 つめのデータセット : Fields[1] を更新, WHERE 句で Fields[0] を使用
    sprintf(buffer, UpdateStmt1, DeltaDS->Fields->Fields[1]->NewValue,
            DeltaDS->Fields->Fields[0]->OldValue);
    pConn->Execute(buffer, NULL, NULL);
    // 2 つめのデータセット : Fields[2] を更新, WHERE 句で Fields[3] を使用
    sprintf(buffer, UpdateStmt2, DeltaDS->Fields->Fields[2]->NewValue,
            DeltaDS->Fields->Fields[3]->OldValue);
    pConn->Execute(buffer, NULL, NULL);
    break;
  case ukDelete:
    // 1 つめのデータセット : WHERE 句で Fields[0] を使用
    sprintf(buffer, DeleteStmt1, DeltaDS->Fields->Fields[0]->OldValue);
    pConn->Execute(buffer, NULL, NULL);
    // 2 つめのデータセット : WHERE 句で Fields[3] を使用
    sprintf(buffer, DeleteStmt2, DeltaDS->Fields->Fields[3]->OldValue);
    pConn->Execute(buffer, NULL, NULL);
    break;
  case ukInsert:
    // 1 つめのデータセット : Fields[0] と Fields[1] の値
    sprintf(buffer, UpdateStmt1, DeltaDS->Fields->Fields[0]->NewValue,
            DeltaDS->Fields->Fields[1]->NewValue);
    pConn->Execute(buffer, NULL, NULL);
    // 2 つめのデータセット : Fields[2] と Fields[3] の値
    sprintf(buffer, UpdateStmt2, DeltaDS->Fields->Fields[2]->NewValue,
            DeltaDS->Fields->Fields[3]->NewValue);
    pConn->Execute(buffer, NULL, NULL);
    break;
}
```

更新エラーの調停

更新プロセスで発生するエラーを処理するイベントには 次の 2 種類があります。

- 更新プロセスで，内部プロバイダは，処理できない更新が見つかるごとに OnUpdateError イベントを発生します。OnUpdateError イベントハンドラで発生した問題を解決すると，エラーは ApplyUpdates メソッドに渡される最大エラー数に達しません。このイベントは，内部プロバイダを使用するクライアントデータセットに対してのみ発生します。TClientDataSet を使用している場合，代わりにプロバイダコンポーネントの OnUpdateError イベントを使用できます。
- 更新操作全体が終了した後，プロバイダがデータベースサーバーに適用できなかったすべてのレコードについて，クライアントデータセットにより OnReconcileError イベントが発生します。

OnReconcileError または OnUpdateError イベントハンドラは必ず記述しなければなりません。適用できなかった返されたレコードを破棄するだけの場合にも必要です。これら 2 つのイベントのイベントハンドラの動作は同じです。これらのハンドラには次のパラメータがあります。

- DataSet：適用できなかった更新レコードを含んでいるクライアントデータセット。このデータセットのメソッドを使用して，問題レコードについての情報を取得し，問題修正のためにレコードを編集できます。特に，問題の原因を突き止めるのに現在のレコードの項目の CurValue,

OldValue，および NewValue プロパティを使用できます。ただし，現在のレコードを変更するためのクライアントデータセットのメソッドはイベントハンドラ内で呼び出してはいけません。

- E：発生した問題を表す EReconcileError オブジェクト。エラーメッセージを抽出したり，更新エラーの原因を特定するため，この例外を使用できます。
- UpdateKind：エラーを生成した更新の種類。UpdateKind の値は，ukModify（変更された既存のレコードを更新しようとして問題が発生），ukInsert（新規レコードを挿入しようとして問題が発生），または ukDelete（既存のレコードを削除しようとして問題が発生）です。
- Action：イベントハンドラが終了するときに実行するアクションを指示する reference パラメータ。イベントハンドラでは，このパラメータは次のいずれかに設定します。
  - このレコードを変更ログに残したままスキップする（rrSkip または raSkip）
  - 調停操作全体を停止する（rrAbort または raAbort）
  - 失敗した変更内容を，サーバーからの対応するレコードにマージする（rrMerge または raMerge）。これは，サーバーレコードが，クライアントデータセットのレコードで行なわれた項目の変更内容を含んでいない場合にだけ有効です。
  - 変更ログ内の現在の更新内容を，イベントハンドラ内のレコードの値（すでに修正されているもの）で置換する（rrApply または raCorrect）
  - エラーを完全に無視する（rrIgnore）。これは，OnUpdateError イベントハンドラの場合のみ可能性があり，イベントハンドラが更新をデータベースサーバーに戻す場合を対象としています。更新されたレコードは，変更ログから削除され，Data にマージされます。プロバイダが更新を適用した場合と同様です。
  - クライアントデータセット側でレコードを元の値に戻すことによって，レコードの変更を取り消す（raCancel）。これは，OnReconcileError イベントハンドラの場合のみ可能性があります。
  - 現在のレコード値を，サーバー側のレコードと一致するように更新する（raRefresh）。これは，OnReconcileError イベントハンドラの場合のみ可能性があります。

次のコードは OnReconcileError イベントハンドラの 1 例で，これは objrepos ディレクトリに入っている RecError ユニットのエラー調停ダイアログを使用します。（このダイアログを使うには，ソースユニットに RecError.hpp を含めます）。

```
void __fastcall TForm1::ClientDataSetReconcileError(TCustomClientDataSet *DataSet,
    EReconcileError *E, TUpdateKind UpdateKind, TReconcileAction &Action)
{
  Action = HandleReconcileError(this, DataSet, UpdateKind, E);
}
```

クライアントデータセットでデータプロバイダを使用する

クライアントデータセットでは，次の場合にプロバイダを使用して，データを与え，更新を適用します。

- データベースサーバーまたは他のデータセットからの更新をキャッシュする
- データを XML ドキュメントで表す

- 多層アプリケーションのクライアント部分のデータを格納する

TClientDataSet 以外のクライアントデータセットに対して，このプロバイダは内部にあり，したがって，アプリケーションから直接にアクセスすることはできません。TClientDataSet では，プロバイダはクライアントデータセットを外部データソースにリンクする外部コンポーネントです。

外部プロバイダコンポーネントは，クライアントデータセットとして同じアプリケーション内に常駐するか，別のシステム上で動作している別のアプリケーションの一部であることもできます。プロバイダコンポーネントについての詳細は，第 28 章「プロバイダコンポーネントの使い方」を参照してください。プロバイダが他のシステム上の別のアプリケーションにあるアプリケーションについての詳細は，第 29 章「多層アプリケーションの作成」を参照してください。

（内部または外部）プロバイダを使用する場合，クライアントデータセットは常にキャッシュアップデートを実行します。この機能についての詳細は，27-15 ページの「クライアントデータセットをキャッシュアップデートに使用する」を参照してください。

次のトピックでは，プロバイダを操作できるクライアントデータセットの追加プロパティとメソッドについて説明します。

プロバイダを指定する

データアクセスメカニズムと関連するクライアントデータセットとは異なり，TClientDataSet には，データをパッケージ化したり更新を適用する内部プロバイダコンポーネントがあります。したがって，ソースデータセットまたは XML ドキュメントからのデータを表す場合は，クライアントデータセットを外部プロバイダコンポーネントに関連付ける必要があります。

TClientDataSet をプロバイダと関連付ける方法は，プロバイダがクライアントデータセットと同じアプリケーション内にあるか，それとも別のシステムで実行されているリモートアプリケーションサーバー上にあるかによります。

- プロバイダがクライアントデータセットと同じアプリケーション内にある場合，オブジェクトインスペクタ内の ProviderName プロパティのドロップダウンリストからプロバイダを選択することにより，プロバイダと関連付けられます。これは，プロバイダがクライアントデータセットと同じ Owner である限り有効です（クライアントデータセットおよびプロバイダは，同じフォームやデータモジュールに配置されていれば，同じ Owner です）。別の Owner を持つローカルプロバイダを使用するには，クライアントデータセットの SetProvider メソッドを使用して，実行時に関連付けなければなりません。

  最終的にリモートプロバイダにスケールアップできると考えている場合，または IAppServer インターフェースを直接呼び出したい場合には，RemoteServer プロパティを TLocalConnection コンポーネントに設定するようにします。TLocalConnection を使用すると，TLocalConnection インスタンスにより，アプリケーションに対してローカルであるすべてのプロバイダのリストを管理し，クライアントの IAppServer 呼び出しを処理することができます。TLocalConnection を使用しない場合は，アプリケーションにより，クライアントデータセットからの IAppServer 呼び出しを処理する隠れたオブジェクトが作成されます。
- プロバイダがリモートアプリケーション サーバーにある場合，ProviderName プロパティに加えて，クライアントデータセットをアプリケーションサーバーに接続するコンポーネントを指定す

る必要があります。このタスクを処理できるプロパティには，プロバイダのリストが取り出される接続コンポーネントの名前を指定する RemoteServer またはクライアントデータセットと接続コンポーネントの間の間接性の追加レベルを与える中央ブローカーを指定する ConnectionBroker の 2 つのプロパティがあります。接続コンポーネントおよび接続ブローカ（使用されている場合）は，クライアントデータセットと同じデータモジュールにあります。接続コンポーネントは，「データブローカ」とも呼ばれるアプリケーションサーバーへの接続を確立し，維持します。詳細については，29-4 ページの「クライアントアプリケーションの構造」を参照してください。

設計時に RemoteServer または ConnectionBroker を指定した後，オブジェクトインスペクタで ProviderName プロパティのドロップダウンリストからプロバイダを選択できます。このリストには，ローカルプロバイダ（同じフォームまたはデータモジュール内）および，接続コンポーネントを通してアクセス可能なリモートプロバイダが含まれています。

メモ 接続コンポーネントが TDCOMConnection のインスタンスである場合，アプリケーションサーバーはクライアントマシンに登録されていなければなりません。

実行時にはコード内に ProviderName を設定することで，使用可能な（ローカルおよびリモートの）プロバイダに切り替えられます。

ソースデータセットまたはドキュメントにデータを要求する

クライアントデータセットにより，プロバイダからデータパケットを取得する方法を指定できます。デフォルトでは，すべてのレコードがソースデータセットから取り出されます。これは，ソースデータセットおよびプロバイダが内部コンポーネント（TBDEClientDataSet，TSQLClientDataSet，および TIBClientDataSet の場合のように）であるか，または TClientDataSet のデータを与える別のコンポーネントであるかに関係なくいえることです。

クライアントデータセットでレコードを取得する方法を変更するには，PacketRecords および FetchOnDemand プロパティを使用します。

インクリメンタルフェッチ

PacketRecords プロパティを変更することにより，クライアントデータセットで取得するデータチャンクを小さくするよう指定できます。PacketRecords は一度に取得するレコードの数か，返すレコードの種類を指定します。デフォルトでは PacketRecords は -1 に設定されます。これは，クライアントデータセットが最初に開かれたとき，またはアプリケーションが明示的に GetNextPacket を呼び出したときに，使用可能な全レコードが一度に取得されることを意味します。PacketRecords が -1 の場合，クライアントデータセットが最初にデータを取得すると，使用可能なレコードをすべて取得したことになるので，それ以上データを取得する必要はなくなります。

レコードを少ない単位で取得するには，PacketRecords に取得するレコードの数を設定します。たとえば次の文は，各データパケットのサイズを 10 レコードに設定します。

```
ClientDataSet1->PacketRecords = 10;
```

このようにレコードを何個かずつ取得するプロセスを「インクリメンタルフェッチ」と呼びます。PacketRecords がゼロより大きいとき，クライアントデータセットはインクリメンタルフェッチを使います。

レコードの各集まりを取得するには，クライアントデータセットで GetNextPacket を呼び出します。新たに取得されたパケットは，クライアントデータセット内にあるデータの最後に追加されます。GetNextPacket は，取得したレコードの数を返します。戻り値が PacketRecords と同じ場合，使用可能なレコードの終わりに到達しなかったことを意味します。戻り値が 0 より大きく PacketRecords より小さい場合，取得操作の間に最終レコードに到達したことを意味します。GetNextPacket が 0 を返した場合，取得するレコードはそれ以上存在しません。

注意 ステートレスアプリケーションサーバー上のリモートプロバイダサーバーからデータを取得する場合，インクリメンタルフェッチは機能しません。ステートレスのリモートデータモジュールでインクリメンタルフェッチを使う方法についての詳細は，29-20 ページの「リモートデータモジュールでのステート情報のサポート」を参照してください。

メモ PacketRecords を使用しても，ソースデータセットのメタデータ情報を取り出すことができます。メタデータ情報を取り出すには，PacketRecords を 0 に設定します。

フェッチオンデマンド

レコードの自動取得は，FetchOnDemand プロパティによって制御されます。FetchOnDemand が **true**（デフォルト）の場合，必要に応じてクライアントデータセットによりレコードが自動的に取得されます。レコードの自動取得を禁止するには，FetchOnDemand を **false** に設定します。FetchOnDemand が **false** の場合，アプリケーションは GetNextPacket を明示的に呼び出してレコードを取得しなければなりません。

たとえば，非常に大規模な読み取り専用データセットを表す必要のあるアプリケーションでは，FetchOnDemand をオフにすると，クライアントデータセットがメモリに入らないデータを読み込まないようにできます。取得してから次の取得までの間，クライアントデータセットは EmptyDataSet メソッドを使用してキャッシュを解放します。ただしこのアプローチは，クライアントが更新内容をサーバーに登録しなければならないときには，うまく機能しません。

プロバイダは，データパケット内のレコードが BLOB データとネストされた詳細データセットを含むかどうかを制御します。プロバイダによってこの情報がレコードから除外された場合，FetchOnDemand プロパティによりクライアントデータセットは，必要に応じて自動的に BLOB データと詳細データセットを取り出します。FetchOnDemand が **false** で，プロバイダが BLOB データおよび詳細データセットをレコードとともに含まない場合，明示的に FetchBlobs メソッドか FetchDetails メソッドを呼び出してこの情報を取得する必要があります。

ソースデータセットからパラメータを取得する

クライアントデータセットがパラメータ値を取得する必要があるのは，次の 2 つの場合です。

- アプリケーション側で，ストアドプロシージャの出力パラメータ値を知る必要がある
- アプリケーションが問い合わせやストアドプロシージャの入力パラメータを，ソースデータセットの現在値に初期化する

クライアントデータセットは，パラメータ値を自身の Params プロパティに格納します。この値は，クライアントデータセットがソースデータセットからデータを取り込むときに，出力パラメータによって更新されます。ただし，クライアントデータセットがデータを取得していないときに，クライ

アントアプリケーション内の TClientDataSet コンポーネントは出力パラメータを必要とする場合があります。

レコードを読み込んでいないときに出力パラメータを取り出す場合や，入力パラメータを初期化する場合には，クライアントデータセットは FetchParams メソッドを呼び出してソースデータセットからパラメータ値を要求できます。パラメータ値はプロバイダからデータパケットに入れて返され，クライアントデータセットの Params プロパティに割り当てられます。

設計時は，クライアントデータセットを右クリックして［パラメータの読み込み］を選択すると Params プロパティを初期化できます。

メモ クライアントデータセットで内部プロバイダとソースデータセットを使っている場合は，Params プロパティは常時，内部ソースデータセットのパラメータを反映しているので，FetchParams を呼び出す必要はありません。TClientDataSet では FetchParams メソッド（または［パラメータの読み込み］コマンド）は，クライアントデータセットがパラメータを指定する機能を備えているプロバイダに接続されていなければ動作しません。たとえば，ソースデータセットがテーブル型データセットである場合，取得するパラメータはありません。

プロバイダがステートレスアプリケーションサーバーの一部として別のシステム上にある場合，FetchParams を使って出力パラメータを取り出すことはできません。ステートレスアプリケーションサーバーでは，他のクライアントが問い合わせまたはストアドプロシージャを変更して再実行できるので，FetchParams を呼び出す前に出力パラメータが変更されることがあります。ステートレスアプリケーションサーバーから出力パラメータを取り出すには，Execute メソッドを使用します。プロバイダが問い合わせまたはストアドプロシージャに関連付けられている場合，Execute はプロバイダに対して，問い合わせかストアドプロシージャを実行して出力パラメータを返すように指示します。これによって返されたパラメータは，Params プロパティを自動的に更新するために使用されます。

ソースデータセットにパラメータを渡す

クライアントデータセットは，パラメータをソースデータセットに渡して，データパケットで送信されるデータの内容を指定できます。これらのパラメータでは次のことを指定できます。

- アプリケーションサーバーで実行される問い合わせまたはストアドプロシージャの入力パラメータ値
- データパケットにして送信されるレコードを制限する項目値

クライアントデータセットがソースデータセットに送信するパラメータ値は，設計時または実行時に指定できます。設計時には，クライアントデータセットを選択して，オブジェクトインスペクタで Params プロパティをダブルクリックします。これによりコレクションエディタが呼び出され，パラメータを追加，削除，再配置できます。コレクションエディタでパラメータを選択することにより，オブジェクトインスペクタでそのパラメータのプロパティを編集できます。

実行時は，Params プロパティの CreateParam メソッドを使用してクライアントデータセットにパラメータを追加します。CreateParam は，指定された名前，パラメータタイプ，およびデータタイプのパラメータオブジェクトを返します。これで，そのパラメータオブジェクトのプロパティを使用してパラメータに値を割り当てることができます。

たとえば，次のコードでは，CustNo という名前の入力パラメータを値 605 に加えます。

```
TParam *pParam = ClientDataSet1->Params->CreateParam(ftInteger, "CustNo", ptInput);
pParam->AsInteger = 605;
```

クライアントデータセットがアクティブでなければ，Active プロパティを **true** に設定するだけで，パラメータをアプリケーションサーバーに送信してそれらのパラメータ値を反映したデータパケットを得ることができます。

問い合わせまたはストアドプロシージャパラメータを送信する

クライアントデータセットの CommandType プロパティが ctQuery または ctStoredProc である場合，またはクライアントデータセットが TClientDataSet インスタンスである場合，関連するプロバイダが問い合わせまたはストアドプロシージャの結果を表すときに，Params プロパティでパラメータ値を指定します。クライアントデータセットが，ソースデータセットからデータを要求する場合，あるいは Execute メソッドを使用してデータセットを返さない問い合わせまたはストアドプロシージャを実行する場合，データの要求または実行コマンドとともにパラメータ値を渡します。プロバイダがパラメータを受信すると，関連付けられている問い合わせまたはストアドプロシージャに割り当てます。次にプロバイダは，このパラメータ値を使用した問い合わせまたはストアドプロシージャの実行をデータセットに指示し，クライアントデータセットがデータを要求している場合には，結果セットの最初のレコードからデータの送信を開始します。

メモ パラメータ名は，ソースデータセット上の対応するパラメータ名と一致する必要があります。

パラメータでレコードを制限する

クライアントデータセットが次の場合

- 関連するプロバイダが TTable または TSQLTable コンポーネントを表す TClientDataSet インスタンス
- CommandType プロパティが ctTable である TSQLClientDataSet または TBDEClientDataSet インスタンス

以上の場合に，Params プロパティを使って，メモリ内にキャッシュされるレコードを制限できます。それぞれのパラメータは，レコードをクライアントデータセットのデータに入れる前に一致していなければならない項目値を表します。動作はフィルタと似ていますが，ただし，フィルタの場合，レコードはメモリ内にそのままキャッシュされますが，使用することはできません。

それぞれのパラメータ名は項目の名前と一致している必要があります。TClientDataSet を使用する場合，これらは，プロバイダに関連付けられた TTable または TSQLTable コンポーネントの項目名です。TSQLClientDataSet または TBDEClientDataSet を使用する場合，これらは，データベース上のテーブル内の項目の名前です。クライアントデータセット内のデータは，対応する項目の値がパラメータの指定値と一致するレコードだけを含みます。

たとえば，1 人の顧客から受けた注文を表示するアプリケーションを考えてみましょう。ユーザーが顧客を指定すると，クライアントデータセットは Params プロパティに，注文を表示する顧客を識別する値を示す CustID という 1 つのパラメータ（またはサーバーテーブル内の呼び出される項目）を設定します。クライアントデータセットがソースデータセットからデータを要求する場合は，このパラメータ値を渡します。するとプロバイダは，指定された顧客のレコードだけを送信します。これ

は，プロバイダからすべての注文レコードをクライアントアプリケーションに送信させてから，クライアントデータセットを使用してレコードにフィルタをかけるよりも効率的です。

サーバーからの制約の処理

データベースサーバーで，どのようなデータが有効かについて制約を定義しているとき，クライアントデータセットでそのことについて認識している場合に使用できます。このようにして，クライアントデータセットにより，ユーザーの編集で絶対にサーバー制約の違反が生じないことを保証できます。そのため，このような違反は，拒絶される場合に，データベースサーバーに決して渡されることはありません。このことは，わずかな更新でも更新プロセス中にエラー条件が発生することを意味します。

データのソースに関係なく，明示的にクライアントデータセットに追加することにより，このようなサーバー制約を複製できます。この手順については，27-7 ページの「カスタム制約の指定」で述べます。

ただし，サーバー制約が自動的にデータパケットに含まれれば，さらに便利です。その後，デフォルトの式と制約を明示的に指定する必要がなくなりますが，クライアントデータセットでは，サーバー制約が変わったときにその値を変更し有効になるようにします。これがデフォルトです。ソースデータセットでサーバー制約を認識できる場合，サーバー制約はプロバイダによって自動的にデータパケットに入れられ，ユーザーが編集結果を変更ログに登録したときにクライアントデータセットにより有効になります。

メモ BDE を使用するデータセットのみが，サーバーから制約をインポートできます。つまり，サーバー制約は，TBDEClientDataSet または TClientDataSet を BDE ベースのデータセットを表すプロバイダとともに使用したときに限り，データパケットに入るということです。サーバー制約をインポートする方法と，プロバイダがデータパケットにサーバー制約を入れるのを禁止する方法の詳細は，28-12 ページの「サーバー制約の処理」を参照してください。

メモ 制約がインポートされた後の制約の処理については，23-20 ページの「サーバー制約の使い方」を参照してください。

サーバーの制約と式をインポートすると開発者がアプリケーションのデータ整合性を維持できるため非常に便利ですが，一時的に制約を解除しなければならないことがあります。たとえば，サーバーの制約が項目の現在の最大値に基づいている場合，クライアントデータセットがインクリメンタルフェッチを使用すると，クライアント側の項目の現在の最大値はデータベースサーバーの最大値と相違して，制約の適用方法が異なってくることがあります。また，制約が有効な場合にクライアントデータセットがレコードにフィルタを適用すると，フィルタが制約条件と競合することがあります。どちらの場合も，アプリケーションで制約チェックを無効にできます。

制約を一時的に解除するには，DisableConstraints メソッドを呼び出します。DisableConstraints が呼び出されるたびに，参照カウントが増えていきます。参照カウントがゼロより大きければ，クライアントデートセットに対する制約は適用されません。

クライアントデータセットに対する制約を再度有効にするには，データセットの EnableConstraints メソッドを呼び出します。EnableConstraints を呼び出すたびに，参照カウントが減っていきます。参照カウントがゼロになると，制約が再度有効になります。

ヒント DisableConstraints と EnableConstraints は必ずペアにして呼び出し，必要に応じて制約を適用できるようにします。

レコードのリフレッシュ

クライアントデータセットで扱うデータは，ソースデータセットからのメモリ上のデータのスナップショットです。ソースデータセットがサーバーデータを表す場合は，時間の経過と共にほかのユーザーがそのデータを変更することもあります。クライアントデータセット内のデータは，ますます元データから離れて行きます。

ほかのデータセットと同様クライアントデータセットには，サーバー上の現在値と一致するようにレコードを更新する Refresh メソッドがあります。ただし Refresh の呼び出しが機能するのは，変更ログ内に編集内容がないときに限ります。未適用の編集データがあるときに Refresh を呼び出すと，例外になります。

クライアントデータセットは，変更ログをそのままにしてデータを更新することもできます。これを行うには，RefreshRecord メソッドを呼び出します。Refresh メソッドとは違って RefreshRecord は，クライアントデータセット内の現在レコードだけを更新します。RefreshRecord は，プロバイダから取得した元のレコード値を変更しますが，変更ログ内にある変更内容はそのままにします。

注意 RefreshRecord の呼び出しがいつも適切とは限りません。ユーザーの編集内容が，元となるデータセットにほかのユーザーが加えた変更と競合すると，RefreshRecord の呼び出しによりその競合が隠されてしまうからです。クライアントデータセットが更新を適用するとき，調停エラーが発生せず，アプリケーションはその衝突を解決できません。

更新エラーが隠されることを避けるために，RefreshRecord を呼び出す前に未処理の更新内容がないかチェックするとよいでしょう。たとえば，次の AfterScroll では，ユーザーが新しいレコードに移動するごとに現在のレコードがリフレッシュされますが（最新の値になるようにする），そうしても安全な場合に限ります。

```
void __fastcall TForm1::ClientDataSet1AfterScroll(TDataSet *DataSet)
{
  if (ClientDataSet1->UpdateStatus == usUnModified)
    ClientDataSet1->RefreshRecord();
}
```

カスタムイベントによるプロバイダとの通信

クライアントデータセットは，IAppServer と呼ばれる特別なインターフェースを通してプロバイダコンポーネントと通信します。プロバイダがローカルの場合，IAppServer は自動的に生成されるオブジェクトへのインターフェースで，クライアントデータセットとプロバイダの間の全通信を処理します。プロバイダがリモートの場合，IAppServer はアプリケーションサーバー上にあるリモートデータモジュールの関連 COM オブジェクトへのインターフェイスまたは（SOAP サーバーの場合）接続コンポーネントによって生成されたインターフェイスです。

TClientDataSet は，IAppServer インターフェースを使用する通信をカスタマイズする多数の方法を提供します。クライアントデータセットのプロバイダで指定される IAppServer メソッドの呼び出しの前後で，TClientDataSet はプロバイダとの通信を可能にする特別なイベントを受け取ります。これら

のイベントは，プロバイダ上の同様なイベントと対応しています。それでたとえばクライアントデータセットが ApplyUpdates メソッドを呼び出すとき，以下のイベントが発生します。

1. クライアントデータセットは BeforeApplyUpdates イベントを受け取ります。その場合このイベントでは，OwnerData という名前の OleVariant に任意のカスタム情報を指定できます。
2. プロバイダは BeforeApplyUpdates イベントを受け取ります。このイベントでは，クライアントデータセットからの OwnerData に応答して，OwnerData の値を新しい情報に更新することができます。
3. プロバイダは，データパケット（これに付随するイベントも含まれる）を作成する通常の処理を行います。
4. プロバイダは AfterApplyUpdates イベントを受け取ります。このイベントでは OwnerData の現在値に応答して，クライアント向けの新しい値に更新できます。
5. クライアントデータセットは AfterApplyUpdates イベントを受け取ります。このイベントでは，OwnerData に返された値に応答できます。

IAppServer のほかのメソッド呼び出しにも Before*XXX* イベントと After*XXX* イベントのセットが伴い，これらのイベントはクライアントデータセットとプロバイダ間の通信をカスタマイズできます。

また，クライアントデータセットには特別なメソッドとして DataRequest があります。このメソッドの目的は，プロバイダとのアプリケーション独自の交信を可能にすることです。クライアントデータセットが DataRequest を呼び出す場合，任意の情報の入ったパラメータとして OleVariant を渡します。これによってプロバイダ上で OnDataRequest イベントが生成されます。このイベントでは，任意のアプリケーション定義の方法で応答し，クライアントデータセットに値を返すことができます。

ソースデータセットのオーバーライド

特定のデータアクセスメカニズムと関連付けられているクライアントデータセットでは，CommandText と CommandType プロパティを使って，クライアントデータセットが表すデータを指定します。ただし，TClientDataSet を使用する場合，データはクライアントデータセットではなく，ソースデータセットによって指定されます。通常，このソースデータセットには，データを生成する SQL 文またはデータベーステーブルやストアドプロシージャの名前を指定するプロパティがあります。

TClientDataSet は，プロバイダが許せば，データセットがどのデータを表すかを示すデータセットに関するプロパティを上書きできます。つまり，プロバイダが許せば，クライアントデータセットの CommandText プロパティで，どのようなデータを表すかを指定するプロバイダのデータセットのプロパティを置き換えることができるということです。これによって，TClientDataSet は，処理対象データを動的に指定することができます。

デフォルトでは，外部プロバイダコンポーネントは，クライアントデータセットがこのように CommandText 値を使用することを許しません。TClientDataSet に CommandText プロパティの使用を許すには，プロバイダの Options プロパティに poAllowCommandText を追加する必要があります。そうでない場合，CommandText は無視されます。

メモ TSQLClientDataSet，TBDEClientDataSet，または TIBClientDataSet の Options プロパティから poAllowCommandText を除去しないでください。クライアントデータセットの Options プロパティ

は，内部プロバイダに転送されるため，poAllowCommandText を削除すると，クライアントデータセットはアクセスするデータを指定できなくなります。

クライアントデータセットは，CommandText 文字列を次の 2 通りの状況で送信します。

- クライアントデータセットが最初に開かれるとき。プロバイダから最初のデータパケットを取得した後は，クライアントデータセットは以降のデータパケットを取得するときに CommandText を送信しません。
- クライアントデータセットがプロバイダに Execute コマンドを送信するとき

上記以外のときに SQL コマンドを送信したり，テーブル名やストアドプロシージャ名を変更するには，AppServer プロパティとして利用可能な IAppServer インターフェースを明示的に使用しなければなりません。このプロパティは，プクライアントデータセットがロバイダとの通信に使用するインターフェースを表します。

クライアントデータセットでファイルデータを使用する

クライアントデータセットは，ディスク上の専用ファイルだけでなく，サーバーデータも取り扱えます。そのため，ファイルベースのデータベースアプリケーションと「ブリーフケースモデル」アプリケーションで使用できます。クライアントデータセットが自分のデータ用に使用する特殊ファイルを「MyBase」と呼びます。

ヒント すべてのクライアントデータセットはブリーフケースモデルのアプリケーションに適していますが，純粋な MyBase アプリケーション（プロバイダを使用していないもの）については，オーバーヘッドの少ない TClientDataSet を使用するのが好ましいと思われます。

純粋な MyBase アプリケーションでは，クライアントアプリケーションはサーバーからテーブル定義およびテーブルデータを取得できず，どんなサーバーにも更新内容を適用できません。クライアントデータセットは単独で次のことを実行しなければなりません。

- テーブルを定義し，作成する
- 保存されたデータを読み込む
- データに編集内容をマージする
- データを保存する

新しいデータセットを作成する

サーバーデータを表さないクライアントデータセットを定義，作成するのに，3 つの方法があります。

- 持続的な項目とインデックス定義を使用して新規のクライアントデータセットを定義および作成できます。これは，テーブル型データセットを作成する場合と同じ方式に従います。詳しくは，22-37 ページの「テーブルの作成と削除」を参照してください。

- 既存のデータセットをコピーできます（設計時または実行時）。既存のデータセットからのコピーについての詳細は，27-13 ページの「データを別のデータセットからコピーする」を参照してください。
- 任意の XML ドキュメントからクライアントデータセットを作成できます。詳しくは，30-6 ページの「データパケットへの XML ドキュメントの変換」を参照してください。

データセットを作成したら，それをファイルに保存できます。その後は，テーブルを再び作成する必要はなく，保存したファイルから読み込みできます。ファイルデータベースアプリケーションに着手するときは，アプリケーション自体を書く前に，まず，データセット用の空ファイルを作成して保存します。このようにして，すでに定義されているクライアントデータセットのメタデータから始めると，ユーザーインターフェースを簡単に設定できます。

ファイルまたはストリームからデータを読み込む

ファイルからデータを読み込むには，クライアントデータセットの LoadFromFile メソッドを呼び出します。LoadFromFile は 1 つのパラメータを取り，データを読み出すファイルを指定する文字列です。ファイル名には，絶対パス名を指定できます。クライアントデータセットのデータを常に同じファイルから読み込む場合は，かわりに FileName プロパティを使用できます。FileName が既存ファイルの名前ならば，クライアントデータセットが開かれたときに自動的にデータが読み込まれます。

ストリームからデータを読み込むには，クライアントデータセットの LoadFromStream メソッドを呼び出します。LoadFromStream は 1 つのパラメータをとります。データ供給元となるストリームオブジェクトです。

LoadFromFile（LoadFromStream）から読み込まれたデータは，クライアントデータセットのデータ形式に従って，そのクライアントデータセットか別のクライアントデータセットが SaveToFile (SaveToStream) メソッドを使用して保存したデータか，または XML ドキュメントから生成されたデータでなければなりません。ファイルまたはストリームへのデータの保存については，27-34 ページの「ファイルまたはストリームにデータを保存する」を参照してください。任意の XML ドキュメントからクライアントデータセットを作成する方法についての詳細は，第 30 章「データベースアプリケーションでの XML の使用」を参照してください。

LoadFromFile または LoadFromStream を呼び出すと，ファイル内のすべてのデータが Data プロパティに読み込まれます。データが保存されたときに変更ログにあった編集内容は，Delta プロパティに読み込まれます。ただし，ファイルから読み出されたインデックスはデータセット作成時のインデックスのみです。

変更内容をデータにマージする

クライアントデータセットでデータを編集する場合，データのすべての編集結果は，メモリ内の変更ログにしか存在しません。このログは，データ自体とは別に保守できます。ただし，クライアントデータセットを使うオブジェクトには完全に透過です。つまり，クライアントデータセット内を移動したりそのデータを表示するコントロールには，変更内容を含めたデータビューが認識されます。しかし，変更内容を元に戻す必要がない場合は，MergeChangeLog メソッドを呼び出すことによって変

更ログをクライアントデータセットのデータにマージしなければなりません。MergeChangeLog は，変更ログ内のすべての変更項目値で Data 内のレコードを上書きします。

MergeChangeLog を実行した後，Data には変更ログに存在していたすべての変更内容と既存のデータを合成したデータが含まれます。この合成データは Data の新たな基礎となり，これに対してそれ以降の変更が可能になります。MergeChangeLog によって全レコードの変更ログがクリアされ，ChangeCount プロパティは 0 にリセットされます。

注意 プロバイダを使用するクライアントデータセットについては，MergeChangeLog を呼び出さないでください。この場合は，ApplyUpdates を呼び出して変更内容をデータベースに書き込みます。詳細については，27-20 ページの「更新を適用する」を参照してください。

メモ 変更内容を別のクライアントデータセットのデータにマージすることも可能です。ただし，そのデータセットが元々 Data プロパティでデータを供給した場合に限ります。これを行うには，データセットプロバイダを使用しなければなりません。この方法の例についての詳細は，27-13 ページの「データを直接割り当てる」を参照してください。

変更ログの拡張された元に戻す機能を使用したくない場合は，クライアントデータセットの LogChanges プロパティを **false** に設定します。LogChanges が **false** の場合，編集結果は，レコードを登録し，MergeChangeLog を呼び出す必要がなければ自動的にマージされます。

ファイルまたはストリームにデータを保存する

変更内容をクライアントデータセットのデータにマージした後も，このデータはメモリ上にのみ存在します。アプリケーションでクライアントデータセットを閉じてからまた開いても変更内容は保持されますが，アプリケーションをシャットダウンすると消えます。データを確定するには，ディスクに書き込まなければなりません。ディスクに変更内容を書き込むには，SaveToFile メソッドを使います。

SaveToFile は 1 つのパラメータを取ります。データを書き込むファイルを指定する文字列です。ファイル名には，絶対パス名を指定できます。指定したファイルがすでに存在する場合，そのファイルの現在の内容はすべて上書きされます。

メモ SaveToFile は，実行時にクライアントデータセットに追加されたインデックスを保存せず，クライアントデータセットを作成したときに追加されたインデックスのみを保存します。

データを常に同じファイルに保存する場合は，かわりに FileName プロパティを使用できます。FileName が設定されると，クライアントデータセットが閉じられるときにデータは指定されたファイルに自動的に保存されます。

SaveToStream メソッドを使用して，データをストリームに保存することもできます。SaveToStream は 1 つのパラメータをとります。データを受け取るストリームオブジェクトです。

メモ 変更ログ内に変更内容が存在する場合にクライアントデータセットを保存すると，それらはデータにマージされません。LoadFromFile メソッドまたは LoadFromStream メソッドを使用してデータを再読み込みする場合，変更ログにはマージされていない変更内容が残っています。これは，ブリーフケースモデルをサポートするアプリケーションでは重要です。そのようなアプリケーションでは，変更内容を最終的にアプリケーションサーバーのプロバイダコンポーネントに適用する必要があるからです。

第28章

プロバイダコンポーネントの使い方

プロバイダコンポーネント（TDataSetProvider および TXMLTransformProvider）は，クライアントデータセットがデータを取得するための最も一般的なメカニズムを備えています。プロバイダは次のことを行います。

- クライアントデータセット（または XML ブローカ）からデータリクエストを受信して，要求されたデータを取り出して，送信可能なデータパケットにパッケージ化し，それをクライアントデータセット（または XML ブローカ）に返します。こうすることを「解決する」と言います。
- 更新されたデータをクライアントデータセット（または XML ブローカ）から受け取り，データベースサーバー，ソースデータセット，またはソース XML ドキュメントに更新内容を適用して更新エラーを記録し，調停のため未解決の更新をクライアントデータセットに返します。こうすることを「解決する」と言います。

プロバイダコンポーネントのほとんどの機能は，自動的に実行されます。プロバイダ上でデータセットまたは XML ドキュメントのデータからデータパケットを作成したり，更新内容を適用するコードを書く必要はありません。ただし，プロバイダコンポーネントには，クライアントに渡すデータとしてどんな情報をパッケージ化するか，またクライアントリクエストにアプリケーションがどう応答するかについて，アプリケーションからより直接的に制御するための，多数のイベントとプロパティが用意されています。

TBDEClientDataSet，TSQLClientDataSet，または TIBClientDataSet を使用する場合，プロバイダはクライアントデータセットの内部に組み込まれており，アプリケーションからは直接アクセスしません。しかし，TClientDataSet または TXMLBroker を使用する場合にはプロバイダは独立したコンポーネントであり，クライアントのためそしてプロバイドおよび解決プロセスで発生するイベントに応答するために，パッケージ化する情報を制御するのにプロバイダを使用できます。内部プロバイダを持つクライアントデータセットは，プロバイダのプロパティおよびイベントの一部を，自らのプロパティおよびイベントとして表示しますが，ほとんどのコントロールに対して別のプロバイダコンポーネントを持つ TClientDataSet を使用することもできます。

別のプロバイダコンポーネントを使用する場合，これはクライアントデータセット（または XML ドキュメント）と同じアプリケーション内に配置したり，多層アプリケーションの一部としてアプリケーションサーバー上に配置できます。

この章では，クライアントデータセットまたは XML ブローカとの対話を制御するためのプロバイダコンポーネントの使用方法を説明します。

データソースの決定

プロバイダコンポーネントを使う場合は，データの取得に使用されるソースを指定しなければなりません。このように取得したデータを，プロバイダはデータパケットの形に組み立てます。C++Builder のご使用バージョンによっては，ソースを次のうちの 1 つとして指定することができます。

- データセットからデータをプロバイドするには，TDataSetProvider を使用します。
- XML ドキュメントからデータをプロバイドするには，TXMLTransformProvider を使用します。

データセットをデータのソースとして使用する

プロバイダがデータセットプロバイダ（TDataSetProvider）の場合，ソースデータセットを示すようにプロバイダの DataSet プロパティを設定します。設計時には，オブジェクトインスペクタの DataSet プロパティ内のドロップダウンリストのデータセットから選択します。

TDataSetProvider は，IProviderSupport インターフェイスを使用して，ソースデータセットと対話します。このインターフェースは TDataSet によって導入されるので，すべてのデータセットに使用できます。ただし TDataSet に実装されている IProviderSupport メソッドは，たいていの場合，何もしないか例外を生成するスタブです。

C++Builder と一緒に配布されているデータセットクラス（BDE 対応データセット，ADO 対応データセット，dbExpress データセット，および InterBase Express データセット）は，IProviderSupport インターフェースをもっと有用な形で実装するためにこれらのメソッドをオーバーライドしています。クライアントデータセットは，継承した IProviderSupport 実装に何も追加しませんが，プロバイダの ResolveToDataSet プロパティが **true** の場合，ソースデータセットとして使用できます。

コンポーネント開発者が TDataSet から独自のカスタム下位クラスを作成する場合，データセットがデータをプロバイダに提供する時には，該当するすべての IProviderSupport メソッドを上書きする必要があります。プロバイダがデータパケットを読み取り専用に提供するだけの場合（つまり，更新を適用しない場合）は，TDataSet に実装されている IProviderSupport メソッドで十分です。

XML ドキュメントをデータのソースとして使用する

プロバイダが XML プロバイダの場合，ソースドキュメントを示すようにプロバイダの XMLDataFile プロパティを設定します。

XML プロバイダは，ソースドキュメントをデータパケットに変換する必要があります。そして，ソースドキュメントを示すだけでなく，そのドキュメントをデータパケットに変換する方法も指定し

なければなりません。変換は，プロバイダの TransformRead プロパティによって取り扱われます。TransformRead は TXMLTransform オブジェクトを表します。そのプロパティを設定することにより，使用する変換を指定し，そのイベントを使って変換への入力を指定します。XML プロバイダの使い方についての詳細は，30-8 ページの「XML ドキュメントをプロバイダのソースとして使う」を参照してください。

クライアントデータセットとの通信

プロバイダとクライアントデータセットまたは XML ブローカ間のすべての通信は，IAppServer インターフェースを通して行われます。プロバイダがクライアントと同じアプリケーションの場合，このインターフェースは自動的に生成された隠れたオブジェクトにより実装されるか，または TLocalConnection コンポーネントによって実装されます。プロバイダが多層アプリケーションの一部の場合，このインターフェイスはアプリケーションサーバー用インターフェイスまたは（SOAP サーバーの場合）接続コンポーネントによって生成されたインターフェイスです。

ほとんどのクライアントアプリケーションは直接 IAppServer を使用しないで，クライアントデータセットまたは XML ブローカのプロパティとメソッドを通して間接的に呼び出します。ただし必要に応じて，クライアントデータセットの AppServer プロパティを使用して，IAppServer インターフェースを直接呼び出すことができます。

表 28.1 は，IAppServer インターフェースのメソッドと，プロバイダコンポーネントとクライアントデータセット上で対応するメソッドとイベントの一覧です。この IAppServer メソッドには，Provider パラメータが含まれています。多層アプリケーションではこのパラメータは，クライアントデータセットの通信相手であるアプリケーションサーバー上のプロバイダを指示します。またほとんどのメソッドには OwnerData という OleVariant パラメータがあり，これによってクライアントデータセットとプロバイダは相互にカスタム情報を受け渡すことができます。OwnerData はデフォルトでは使用されませんが，すべてのイベントハンドラに渡されるので，クライアントデータセットからの呼び出しの前後に，プロバイダがアプリケーションで定義された情報を調整するコードを書くことができます。

表 28.1　AppServer インターフェースのメンバー

IAppServer	プロバイダコンポーネント	TClientDataSet
AS_ApplyUpdates メソッド	ApplyUpdates メソッド， BeforeApplyUpdates イベント， AfterApplyUpdates イベント	ApplyUpdates メソッド， BeforeApplyUpdates イベント， AfterApplyUpdates イベント
AS_DataRequest メソッド	DataRequest メソッド， OnDataRequest イベント	DataRequest メソッド
AS_Execute メソッド	Execute メソッド， BeforeExecute イベント， AfterExecute イベント	Execute メソッド， BeforeExecute イベント， AfterExecute イベント
AS_GetParams メソッド	GetParams メソッド， BeforeGetParams イベント， AfterGetParams イベント	FetchParams メソッド， BeforeGetParams イベント， AfterGetParams イベント
AS_GetProviderNames メソッド	使用可能なすべてのプロバイダを識別するために使用される	ProviderName プロパティの設計時リストを作成するために使用される

表 28.1 AppServer インターフェースのメンバー（つづき）

IAppServer	プロバイダコンポーネント	TClientDataSet
AS_GetRecords メソッド	GetRecords メソッド， BeforeGetRecords イベント， AfterGetRecords イベント	GetNextPacket メソッド， Data プロパティ， BeforeGetRecords イベント， AfterGetRecords イベント
AS_RowRequest メソッド	RowRequest メソッド， BeforeRowRequest イベント， AfterRowRequest イベント	FetchBlobs メソッド， FetchDetails メソッド， RefreshRecord メソッド， BeforeRowRequest イベント， AfterRowRequest イベント

データセットプロバイダを使用した更新の適用方法の選択

TXMLTransformProvider コンポーネントは常に関連する XML ドキュメントに更新を適用します。ただし，TDataSetProvider を使用する場合，更新の適用方法を選択できます。デフォルトでは TDataSetProvider コンポーネントが更新を適用したり，更新エラーを解決する場合，動的に生成される SQL 文を使用してデータベースサーバーと直接通信します。このアプローチは，サーバーアプリケーションが更新を 2 回（最初はデータセットに，次はリモートサーバーに）マージする必要がないという利点があります。

ただし，いつもこのアプローチが適切とは限りません。たとえば，データセットのイベントを使いたいときがあります。あるいは，使用するデータセットが SQL 文の使用をサポートしないこともあります（たとえば TClientDataSet コンポーネントからプロバイドする場合）。

TDataSetProvider を使用すると，SQL 文を使ってデータベースサーバーに更新を適用するか，それとも ResolveToDataSet プロパティの設定によってソースデータセットに更新を適用するかを，選択できます。このプロパティが **true** なら，更新はデータセットに適用されます。**false** なら，更新は，基のデータベースサーバーに直接適用されます。

データパケットに加える情報を制御する

データセットプロバイダを操作する場合，クライアントとの間で送受信されるデータパケットにどのような情報を含めるかを制御するのに以下のようないくつかの方法があります。

- データパケットに加える項目を指定する
- データパケットに影響するオプションを設定する
- データパケットにカスタム情報を追加する

メモ データパケットの内容を制御するこのような手法は，データセットプロバイダでしか使用できません。TXMLTransformProvider を使用する場合は，プロバイダで使用する変換ファイルを制御することによりデータパケットの内容を制御することしかできません。

データパケットに加える項目を指定する

データセットプロバイダを使用する場合，データパケットに加える項目を制御するには，プロバイダがパケットの構築に使うデータセットに持続的項目を作成します。するとプロバイダは，その項目だけを追加します。ソースデータセットによって値が動的に生成される項目（計算項目や参照項目など）の追加は可能ですが，受信側のクライアントデータセットでは静的な読み取り専用の項目になります。持続的項目についての詳細は，23-3 ページの「持続的項目コンポーネント」を参照してください。

クライアントデータセット側でデータを編集して更新を適用するようにしたい場合，データパケット内でレコードが重複することのないように，十分多くの項目を含めなければなりません。そうでないと更新が適用されるとき，どのレコードを更新してよいのか判断できません。重複させないためだけの目的で設けられた余分な項目をクライアントデータセットが表示したり使ったりするのを避けたい場合は，それらの項目の ProviderFlags プロパティに pfHidden を設定します。

メモ 重複レコードを避けるために十分多くの項目を加えることは，プロバイダのソースデータセットが問い合わせを表すときにも考慮すべき点です。アプリケーション側ですべての項目を使用していないとしても，すべてのレコードの一意性を保証できるだけの項目を含めるように，問い合わせを指定しなければなりません。

データパケットに影響するオプションを設定する

データセットプロバイダの Options プロパティを使うと，いつ BLOB やネストされた詳細テーブルを送信するか，項目表示プロパティを含めるか，どのような種類の更新を受容するかなどを指定できます。次の表では，Options に設定可能な値を一覧表示します。

表 28.2　プロバイダのオプション

値	説明
poAutoRefresh	プロバイダが更新を適用する場合，プロバイダはクライアントデータセットを現在のレコード値で更新する
poReadOnly	クライアントデータセットはプロバイダに更新を適用できない
poDisableEdits	クライアントデータセットは既存の値を変更できない。項目を編集しようとすると，クライアントデータセットは例外を生成する（これは，クライアントデータセットがレコードを挿入，削除する機能には影響しない）
poDisableInserts	クライアントデータセットは新規レコードを挿入できない。新規レコードを挿入しようとすると，クライアントデータセットは例外を生成する（これは，クライアントデータセットがレコードを削除，変更する機能には影響しない）
poDisableDeletes	クライアントデータセットはレコードを削除できない。レコードを削除しようとすると，クライアントデータセットは例外を生成する（これは，クライアントデータセットがレコードを挿入または変更する機能には影響しない）
poFetchBlobsOnDemand	BLOB 項目の値は，データパケットに含まれていない。そのかわりに，クライアントアプリケーションはこれらの値を必要に応じて要求しなければならない。クライアントデータセットの FetchOnDemand プロパティが **true** の場合，クライアントデータセットはその値を自動的に要求する。そうでない場合アプリケーションは，クライアントデータセットの FetchBlobs メソッドを呼び出して BLOB データを取得しなければならない

表 28.2 プロバイダのオプション（つづき）

値	説明
poFetchDetailsOnDemand	プロバイダのデータセットがマスター / 詳細関係のマスターを表す場合，ネストされた詳細値はデータパケットに含まれていない。そのかわりに，クライアントデータセットはこれらの値を必要に応じて要求しなければならない。クライアントデータセットの FetchOnDemand プロパティが **true** の場合，クライアントデータセットはその値を自動的に要求する。そうでない場合アプリケーションは，クライアントデータセットの FetchDetails メソッドを使用してネストされた詳細データを取得しなければならない
poIncFieldProps	データパケットは，次のような項目プロパティを含む（これらのうち該当するもの）。Alignment, DisplayLabel, DisplayWidth, Visible, DisplayFormat, EditFormat, MaxValue, MinValue, Currency, EditMask, DisplayValues
poCascadeDeletes	プロバイダのデータセットがマスター / 詳細関係のマスターを表す場合，マスターレコードが削除されると詳細レコードも自動的にサーバーによって削除される。このオプションを使用するには，データベースサーバーが，参照の整合性の一部としてカスケード削除を行うように設定されている必要がある
poCascadeUpdates	プロバイダのデータセットがマスター / 詳細関係のマスターを表す場合，詳細テーブルのキー値はマスターレコードの対応する値が変わると自動的に更新される。このオプションを使用するには，データベースサーバーが，参照の整合性の一部としてカスケード更新を行うように設定されている必要がある
poAllowMultiRecordUpdates	1 回の更新で，元になるデータベーステーブルの複数のレコードを変更できる。これはトリガー，参照の整合性，ソースデータセット上の SQL 文などの結果でもよい。エラーが発生した場合，イベントハンドラを使用して更新されたレコードにアクセスできるが，結果として変更されたほかのレコードにはアクセスできない
poNoReset	クライアントデータセットは，データをプロバイドする前にプロバイダがカーソルを先頭レコードに再配置するように指定できない
poPropogateChanges	更新プロセスの一環としてサーバーが更新レコードに加える変更はクライアントに送り戻され，クライアントデータセットにマージされる
poAllowCommandText	クライアントは，関連付けられたデータセットの SQL テキスト，あるいはそれによって表わされるテーブルまたはストアドプロシージャの名前をオーバーライドできる
poRetainServerOrder	クライアントデータセットは，データセット内のレコードを再ソートせずにデフォルトの順序を有効にする

データパケットにカスタム情報を追加する

データセットプロバイダは，アプリケーションによって定義された情報を OnGetDataSetProperties イベントを使用してデータパケットに追加できます。この情報は OleVariant としてコード化され，指定した名前で保存されます。その後クライアントデータセットは，GetOptionalParam メソッドを使用して情報を取り出します。その情報を，クライアントデータセットがレコード更新時に送信するデルタパケットの中に含めるように指定できます。この場合クライアントデータセットは，この情報をまったく認識しないかもしれませんが，プロバイダは自分自身への往復メッセージを送信できます。

カスタム情報を OnGetDataSetProperties イベントに追加するときは，個々の属性（「オプションパラメータ」とも呼ぶ）について 3 つの要素から成るバリアント配列を使って指定します。その要素とは，名前（文字列），値（Variant 型），および，クライアントが更新を適用するときにその情報をデ

ルタパケットに含めるかどうかを示す論理フラグです。複数の属性を追加するときは，バリアント配列のバリアント配列を作成します。たとえば次の OnGetDataSetProperties イベントハンドラは，データがプロバイドされた時刻とソースデータセットのレコード総数という 2 つの値を送信します。データがプロバイドされた時刻だけは，クライアントデータセットが更新を適用するときに返されます。

```
void __fastcall TMyDataModule1::Provider1GetDataSetProperties(TObject *Sender,
  TDataSet *DataSet, out OleVariant Properties)
{
  int ArrayBounds[2];
  ArrayBounds[0] = 0;
  ArrayBounds[1] = 1;
  Properties = VarArrayCreate(ArrayBounds, 1, varVariant);
  Variant values[3];
  values[0] = Variant("TimeProvided");
  values[1] = Variant(Now());
  values[2] = Variant(true);
  Properties[0] = VarArrayOf(values,2);
  values[0] = Variant("TableSize");
  values[1] = Variant(DataSet->RecordCount);
  values[2] = Variant(false);
  Properties[1] = VarArrayOf(values,2);
}
```

クライアントデータセットが更新を適用するとき，元のレコードがプロバイドされた時刻は，プロバイダの OnUpdateData イベントで読み取ることができます。

```
void __fastcall TMyDataModule1::Provider1UpdateData(TObject *Sender,
  TCustomClientDataSet *DataSet)
{
  Variant WhenProvided = DataSet->GetOptionalParam("TimeProvided");
  ...
}
```

クライアントのデータリクエストに応答する

通常，クライアントのデータリクエストは自動的に処理されます。クライアントデータセットまたは XML ブローカは，GetRecords を呼び出すことによって（IAppServer インターフェースを介して間接的に）データパケットを要求します。プロバイダは，関連付けられたデータセットまたは XML ドキュメントからデータを取り出し，データパケットを作成し，そのパケットをクライアントに送信することによって自動的に応答します。

プロバイダは，データがデータパケットに入れられた後，パケットがクライアントに送信される前に，そのデータを編集することもできます。たとえばユーザーのアクセスレベルなどの規準に基づいてレコードをパケットから削除したり，多層アプリケーションでは機密データを送信前に暗号化する場合があります。

データパケットをクライアントに送信する前にデータを編集するには，OnGetData イベントハンドラを書きます。OnGetData イベントハンドラは，データパケットをクライアントデータセットの形でパラメータとしてプロバイドします。このクライアントデータセットのメソッドを使用すると，クライアントに送信される前にデータを編集できます。

IAppServer インターフェースを介して行われるすべてのメソッド呼び出しと同様に，プロバイダは GetRecords 呼び出しの前後に，クライアントデータセットとの間で持続的ステート情報を通信できます。この通信は，BeforeGetRecords と AfterGetRecords イベントハンドラを介して行われます。アプリケーションサーバーの持続的ステート情報についての詳細は，29-20 ページの「リモートデータモジュールでのステート情報のサポート」を参照してください。

クライアントの更新リクエストに応答する

プロバイダは，クライアントデータセットまたは XML ブローカから受け取った Delta データパケットに基づいて，データベースレコードに更新を適用します。クライアントは，ApplyUpdates メソッドを呼び出すことによって（IAppServer インターフェースを介して間接的に）更新を要求します。

IAppServer インターフェースを介して行われるすべてのメソッド呼び出しと同様に，プロバイダは ApplyUpdates 呼び出しの前後に，クライアントデータセットとの間で持続的ステート情報を通信できます。この通信は，BeforeApplyUpdates と AfterApplyUpdates イベントハンドラを介して行われます。アプリケーションサーバーの持続的ステート情報についての詳細は，29-20 ページの「リモートデータモジュールでのステート情報のサポート」を参照してください。

データセットプロバイダを使用している場合，追加イベントが多数あれば制御できる範囲が広がります。

データセットプロバイダは更新リクエストを受け取ると OnUpdateData イベントを発生させます。このイベントでは，Delta パケットがデータセットに書き込まれたり，更新の適用方法に反映される前に，Delta パケットを編集できます。OnUpdateData イベント後，プロバイダは変更をデータベースまたはソースデータセットに書き込みます。

プロバイダは，更新をレコード単位で行います。プロバイダは各レコードに適用する前に，BeforeUpdateRecord イベントを発生させます。これを使用すると，更新が適用される前に更新内容を選別できます。レコードの更新時にエラーが起きると，プロバイダは OnUpdateError イベントハンドラを受信して，エラーを解決します。通常エラー発生の原因は，変更がサーバーの制約に違反したり，プロバイダがレコードを取り出した後クライアントデータセットが更新適用を要求する前に，別のアプリケーションがデータベースレコードを変更することです。

更新エラーは，データセットプロバイダまたはクライアントデータセットによって処理できます。プロバイダが多層アプリケーションの一部の場合，プロバイダは，解決のためにユーザーとの対話を必要としないすべての更新エラーを処理する必要があります。プロバイダがエラー条件を解決できない場合，プロバイダは違反レコードのコピーを一時的に格納します。レコードの処理が完了すると，プロバイダは検出したエラー数をクライアントデータセットに返し，未解決のレコードを結果データパケットにコピーして，調停のためクライアントデータセットに返します。

すべてのプロバイダイベントのイベントハンドラには，クライアントデータセットとして更新内容のセットが渡されます。イベントハンドラが一部の更新のみを処理する場合，レコードの更新ステータスに基づいてデータセットにフィルタをかけることができます。レコードにフィルタをかけると，イベントハンドラは使用しないレコードまでソートする必要がなくなります。レコードの更新ステータスに基づいてクライアントデータセットにフィルタをかけるには，StatusFilter プロパティを設定します。

メモ 更新が 1 つのテーブルを表していないデータセットに送られる場合，アプリケーションは特別なサポートを提供する必要があります。その手順についての詳細は，28-11 ページの「1 つのテーブルを表していないデータセットに更新を適用する」を参照してください。

データベースを更新する前にデルタパケットを編集する

プロバイダがデータベースに更新を適用する前に，プロバイダは OnUpdateData イベントを発生させます。OnUpdateData イベントハンドラは，パラメータとして Delta パケットのコピーを受け取ります。これはクライアントデータセットです。

OnUpdateData イベントハンドラ内では，データセットに書き込まれる前に Delta パケットを編集するため，クライアントデータセットの任意のプロパティとメソッドを使用できます。特に便利なプロパティの 1 つとして，UpdateStatus プロパティがあります。UpdateStatus は，デルタパケット内の現在レコードが表す変更の種類を示します。これが取りうる値を表 28.3 に示します。

表 28.3　UpdateStatus の値

値	説明
usUnmodified	レコード内容は変更されていない
usModified	レコード内容は変更された
usInserted	レコードが挿入された
usDeleted	レコードが削除された

たとえば次の OnUpdateData イベントハンドラは，データベースに挿入される全新規レコードに現在の日付を挿入します。

```
void __fastcall TMyDataModule1::Provider1UpdateData(TObject *Sender,
  TCustomClientDataSet *DataSet)
{
  DataSet->First();
  while (!DataSet->Eof)
  {
    if (DataSet->UpdateStatus == usInserted)
    {
      DataSet->Edit();
      DataSet->FieldByName("DateCreated")->AsDateTime = Date();
      DataSet->Post();
    }
    DataSet->Next();
  }
}
```

更新の適用方法に影響を与える

また OnUpdateData イベントを使用すると，データセットプロバイダはデルタパケット内のレコードがデータベースに適用される方法を指示できます。

デフォルトではデルタパケットの変更は，以下のように自動的に生成される SQL の UPDATE，INSERT，DELETE 文を使用して，データベースに書き込まれます。

```
UPDATE EMPLOYEES
  set EMPNO = 748, NAME = 'Smith', TITLE = 'Programmer 1', DEPT = 52
WHERE
  EMPNO = 748 and NAME = 'Smith' and TITLE = 'Programmer 1' and DEPT = 47
```

特に指定しない限り，デルタパケットレコードの全項目は，UPDATE 節と WHERE 節で指定されます。しかし，一部の項目を除外する場合もあります。その方法の 1 つは，プロバイダの UpdateMode プロパティを設定することです。UpdateMode には，次の値を割り当てます。

表 28.4　UpdateMode の値

値	説明
upWhereAll	全項目を項目の特定に使用する（WHERE 節）
upWhereChanged	キー項目と変更された項目だけをレコードの特定に使用する
upWhereKeyOnly	キー項目だけをレコードの特定に使用する

しかし，より高い制御性が必要なこともあります。たとえば前の文で，EMPNO 項目を UPDATE 節から外して EMPNO 項目が変更されないよう保護し，TITLE と DEPT 項目を WHERE 節から外してほかのアプリケーションによって変更された場合の矛盾を避けたいとします。特定の項目が表れる節を指定するには，ProviderFlags プロパティを使います。ProviderFlags は，表 28.5 に示す任意の値を含むセットです。

表 28.5　ProviderFlags の値

値	説明
pfInWhere	UpdateMode が upWhereAll または upWhereChanged のときに，項目は生成される INSERT，DELETE，UPDATE 文の WHERE 節に含まれている
pfInUpdate	項目は，生成される UPDATE 文の UPDATE 節で指定する
pfInKey	項目は，UpdateMode が upWhereKeyOnly のときに生成される文の WHERE 節に使用されている
pfHidden	項目は一意性を確保するためレコードに追加されているが，クライアント側では表示したり使用できない

次の OnUpdateData イベントハンドラでは，TITLE 項目の更新が可能で，レコードの検索には EMPNO と DEPT 項目を使用します。エラーが発生すると，キーに基づいてレコードを特定する第 2 の試みが行われ，生成された SQL は EMPNO 項目だけを検索します。

```
void __fastcall TMyDataModule1::Provider1UpdateData(TObject *Sender,
  TCustomClientDataSet *DataSet)
{
  DataSet->FieldByName("EMPNO")->ProviderFlags.Clear();
  DataSet->FieldByName("EMPNO")->ProviderFlags << pfInWHere << pfInKey;
  DataSet->FieldByName("TITLE")->ProviderFlags.Clear();
  DataSet->FieldByName("TITLE")->ProviderFlags << pfInUpdate;
  DataSet->FieldByName("DEPT")->ProviderFlags.Clear();
  DataSet->FieldByName("DEPT")->ProviderFlags << pfInWhere;
}
```

メモ　UpdateFlags プロパティを使うと，データセットを更新するときに動的に生成される SQL を使用しない場合でも，更新の適用方法を指定できます。これらのフラグはまた，レコードの特定に使う項目と更新される項目を指定します。

個々の更新を選別する

更新が適用される直前に，データセットプロバイダは BeforeUpdateRecord イベントを受信します。このイベントを使用して更新が適用される前にレコードを編集できますが，それは OnUpdateData イベントを使ってデルタパケット全体を編集するのに似ています。たとえばプロバイダは，更新による競合をチェックする場合，BLOB 項目（メモなど）を比較しません。BLOB 項目などの更新エラーをチェックするため，BeforeUpdateRecord イベントを使用できます。

また，このイベントを使用して自ら更新を適用したり，更新を選別して排除することができます。BeforeUpdateRecord イベントハンドラでは，更新内容がすでに処理されているので適用する必要がないことを指示できます。するとプロバイダはそのレコードをスキップしますが，更新エラーとはみなしません。たとえばこのイベントは，更新をストアドプロシージャ（自動的に更新できない）に適用するメカニズムを提供し，イベントハンドラ内でレコードが更新されるとプロバイダが自動処理をスキップするようにできます。

プロバイダ上の更新エラーを解決する

データセットプロバイダがデルタパケットにレコードを登録しようとしてエラー条件が発生すると，OnUpdateError イベントが発生します。プロバイダが更新エラーを解決できない場合，プロバイダは違反レコードのコピーを一時的に保存します。レコードの処理が完了すると，プロバイダは検出したエラー数を返し，未解決のレコードを結果データパケットにコピーして，調停のためクライアントに返します。

多層アプリケーションではこのメカニズムにより，アプリケーションサーバー上で機械的に解決できるどんな更新エラーも処理可能になり，しかもクライアントアプリケーション側でのユーザーとの対話によるエラー状態の修正も可能です。

OnUpdateError ハンドラは，変更できなかったレコードのコピー，データベースからのエラーコード，プロバイダがレコードを挿入 / 削除 / 更新しようとしたかを示す情報を取得します。問題レコードは，クライアントデータセットに戻されます。このデータセットには，データナビゲーションメソッドを使用してはいけません。ただし，データセットの各項目には，NewValue，OldValue，および CurValue プロパティを使用でき，それによって問題の原因を特定したり，更新エラーを解決するための変更を加えたりできます。OnUpdateError イベントハンドラが問題を修正できたら，修正されたレコードを適用するため Response パラメータを設定します。

1 つのテーブルを表していないデータセットに更新を適用する

データセットプロバイダが更新を直接データベースサーバーに適用する SQL 文を生成する場合，レコードを保存しているデータベーステーブルの名前が必要です。これは，テーブルタイプデータセットや「ライブ」TQuery コンポーネントのような多くのデータセットの場合，自動的に処理できます。ただし，結果セットを伴うまたは複数テーブルへの問い合わせを伴うストアドプロシージャの基になるデータに対してプロバイダが更新を適用しなければならない場合は，自動更新をするのに問題があります。更新の適用先のテーブル名を取得する簡単な方法はありません。

問い合わせまたはストアドプロシージャが，BDE 対応データセットであり（TQuery または TStoredProc），更新オブジェクトが関連付けられているなら，プロバイダはその更新オブジェクトを使用できます。一方，更新オブジェクトがなければ，プログラムによって OnGetTableName イベントハンドラ内でテーブル名を指定することができます。イベントハンドラがテーブル名を指定すれば，プロバイダは更新を適用する適切な SQL 文を生成できます。

テーブル名の指定は，更新対象のテーブルが 1 つのデータベーステーブルである（つまり 1 つのテーブル内のレコードだけが更新を必要としている）場合にのみ有効です。更新では元になっている複数のデータベーステーブルを変更する必要のある場合は，プロバイダの BeforeUpdateRecord イベントを使用して，コード内で明示的に更新を適用しなければなりません。このイベントハンドラが更新を適用したら，プロバイダでエラーが発生しないようにイベントハンドラの Applied パラメータを **true** にします。

メモ プロバイダを BDE 対応データセットに関連付けている場合，BeforeUpdateRecord イベントハンドラでアップデートオブジェクトを使い，カスタマイズした SQL 文を使用して更新を適用できます。詳細は，24-39 ページの「アップデートオブジェクトを使ったデータセットの更新」を参照してください。

クライアントが生成するイベントへの応答

プロバイダコンポーネントは，クライアントデータセットから直接プロバイダを呼び出す呼び出しを独自に作成できる汎用イベントを実装しています。これは OnDataRequest イベントです。

OnDataRequest は，プロバイダの通常の機能の一部ではありません。これは単に，クライアントデータセットがプロバイダと直接通信するためのフックです。イベントハンドラは，入力パラメータとして OleVariant をとり，OleVariant を返します。OleVariant を使用することにより，インターフェースとしてほとんどどんな情報でもプロバイダと受け渡しできるほどの汎用性が得られます。

OnDataRequest イベントを発生させるには，クライアントアプリケーションはクライアントデータセットの DataRequest メソッドを呼び出します。

サーバー制約の処理

ほとんどのリレーショナルデータベース管理システムでは，データの整合性を実装するためにテーブルに制約を設けています。制約は，テーブルと列のデータ値についての規則，または各種テーブル内の列相互のデータ関係についての規則です。たとえば，SQL-92 準拠のほとんどのリレーショナルデータベースは，以下の制約をサポートしています。

- NOT NULL。列に与える値がヌルにならないようにする
- NOT NULL UNIQUE。列値がヌルにならないようにし，別のレコードでその列にすでにあるほかのどの値とも重複しないようにする
- CHECK。列に与える値が一定の範囲内になるか，一定数の可能な値のどれかになるようにする
- CONSTRAINT。テーブル全体を検査する制約で，複数の列に適用される

- PRIMARY KEY。インデックス付けのために 1 つまたは複数の列をテーブルの一次キーと指定する
- FOREIGN KEY。あるテーブル内の 1 つまたは複数の列がほかのテーブルを参照することを指定する

メモ 制約は上記に示したものに限られるわけではありません。データベースサーバーはこれらの制約の一部またはすべてを部分的にでも全体的にでもサポートできます。これらのほかにも制約をサポートすることがあります。サポートされている制約についての詳細は，各サーバーのマニュアルを参照してください。

データベースサーバーの制約は，従来のデスクトップデータベースアプリケーションで管理しているきたさまざまなデータ検査と同じことを行います。多層データベースアプリケーションでは，アプリケーションサーバーやクライアントアプリケーションのコードでサーバーの制約と同じことを記述しなくても，その制約を利用できます。

プロバイダが BDE 対応データセットと一緒に動作する場合，プロバイダの Constraints プロパティを使うと，クライアントデータセットとの間で受け渡しされるデータに，サーバーの制約を複製して適用することができます。Constraints が **true**（デフォルト）の場合，ソースデータセットに保存されているサーバーの制約はデータパケットに含まれており，クライアントによるデータ更新の試みに影響を与えます。

重要 プロバイダが制約情報をクライアントデータセットに渡すためには，制約をデータベースサーバーから取り出せなければなりません。データベースの制約をサーバーからインポートするには，SQL エクスプローラを使ってデータベースサーバーの制約とデフォルト式をデータディクショナリにインポートします。データディクショナリ内の制約とデフォルト式は自動的に，BDE 対応データセットでの使用が可能になります。

クライアントデータセットに送るデータにサーバーの制約を適用したくない場合があります。たとえば，パケットでデータを受け取り，さらにレコードを取得する前にレコードのローカル更新が可能なクライアントデータセットの場合，更新対象の一連のデータが一時的に不完全になるため，適用される可能性のある一部のサーバー制約を解除しなければならないことがあります。プロバイダからクライアントデータセットへの制約の複製を防ぐには，Constraints を **false** に設定します。また，クライアントデータセットは，DisableConstraints メソッドと EnableConstraints メソッドを使って制約を無効にしたり有効にしたりできます。クライアントデータセットからの制約の有効化と無効化についての詳細は，27-29 ページの「サーバーからの制約の処理」を参照してください。

第29章

多層アプリケーションの作成

この章では，多層のクライアント / サーバーデータベースアプリケーションの作成のしかたについて説明します。多層クライアント / サーバーアプリケーションは，「層」と呼ばれる論理ユニットに分割され，各ユニットは別々のマシン上で連係して実行されます。多層アプリケーションは，ローカルエリアネットワーク上，さらにはインターネット上で，相互にデータを共用し通信します。多層アプリケーションには，ビジネスロジックの集中管理，軽量化されたクライアントアプリケーションなどの数多くの利点があります。

「3 層モデル」と呼ばれるもっとも単純なモデルでは，多層アプリケーションを以下の 3 つに分割します。

- **クライアントアプリケーション**：ユーザーのマシン上でユーザーインターフェースを提供する
- **アプリケーションサーバー**：すべてのクライアントからアクセスできる中央のネットワーク位置にあり，共通のデータサービスを提供する
- **リモートデータベースサーバー**：リレーショナルデータベース管理システム（RDBMS）を提供する

この 3 層モデルでは，アプリケーションサーバーがクライアントとリモートデータベースサーバー間のデータの流れを管理するので，アプリケーションサーバーを「データブローカ」と呼ぶことがあります。データベースバックエンドを独自に作成することもできますが，通常作成するのはアプリケーションサーバーとそのクライアントだけです。

もっと複雑な多層アプリケーションでは，クライアントとリモートデータベースサーバーの間に追加サービスが存在します。たとえば，安全なインターネットトランザクションを処理するセキュリティサービスブローカや，他のプラットフォーム上のデータベースとデータを共有するためのブリッジサービスなどがあります。

VCL と CLX では，多層アプリケーションの開発をサポートしていますが，このサポートは，クライアントデータセットが送信可能なデータパケットを使用してプロバイダコンポーネントと通信する方法を拡張したものです。この章では，3 層データベースアプリケーションの作成に焦点をあてます。3 層アプリケーションの作成と管理の方法を理解すれば，必要に応じて追加サービス層を作成および追加できます。

多層データベースモデルの利点

多層データベースモデルでは，データベースアプリケーションを論理的な要素に分割します。クライアントアプリケーションはデータの表示とユーザーとの対話に集中できます。理想的には，データがどう保存されどう保守されるかまったく知りません。アプリケーションサーバー（中間層）は，複数のクライアントからのリクエストと更新内容を調整して処理します。アプリケーションサーバーは，データセットの定義，およびデータベースサーバーとの対話の詳細をすべて処理します。

この多層モデルには，次のような利点があります。

- **ビジネスロジックの共有中間層へのカプセル化：**いろいろなクライアントアプリケーションがすべて同じ中間層にアクセスします。これにより，個別のクライアントアプリケーションごとにビジネスルールを繰り返すことの冗長性（および管理コスト）を避けることができます。
- **軽量クライアントアプリケーション：**クライアントアプリケーションは，処理の多くを中間層に委ねることによってファイルサイズが小さくなるように書けます。クライアントアプリケーションは小さいだけでなく配布も楽です。というのは，インストール，環境設定，およびデータベース接続ソフトウェア（ボーランドデータベースエンジンおよびデータベースサーバーのクライアント側ソフトウェアなど）の保守を考える必要がないからです。軽量クライアントアプリケーションは，インターネットを通して配布できる柔軟性があります。
- **分散データ処理：**何台かのマシンにアプリケーションの作業を分散させると，負荷が分散されるのでパフォーマンスが向上する上，サーバーがダウンした場合はバックアップシステムに引き継がせることができます。
- **広がるセキュリティの可能性：**機密に関連する機能を切り離して，異なるアクセス制限の層に置くことができます。これによって，柔軟で，レベル設定のできるセキュリティが提供されます。中間層は機密データへのエントリポイントを制限できるので，アクセスを制御しやすくなります。HTTP または MTS を使用する場合は，それらでサポートされているセキュリティモデルを利用できます。

プロバイダベースの多層アプリケーションを理解する

多層アプリケーションでは，コンポーネントパレットの［DataSnap］ページ，［Data Access］ページ，場合によっては［WebServices］ページにあるコンポーネント，［新規作成］ダイアログの［多層サポート］ページまたは［WebServices］ページのウィザードによって作成されるリモートデータモジュールを使用します。これらは，プロバイダコンポーネントがデータを送信可能なデータパケットにパッケージ化し，送信可能なデルタパケットとして受信した更新内容を処理できるという特徴をもとにしています。

多層データベースアプリケーションに必要なコンポーネントを次の表 29.1 にまとめます。

表 29.1　多層アプリケーションで使用するコンポーネント

コンポーネント	説明
リモートデータモジュール	COM オートメーションサーバーまたは Web サービス アプリケーションとともに使用して，クライアントアプリケーションからその中に含まれるプロバイダへのアクセスを可能にする特別なデータモジュール。アプリケーションサーバー上で使用される
プロバイダコンポーネント	データパケットを作成し，クライアント更新内容を解決することによってデータを提供するデータブローカ。アプリケーションサーバー上で使用される
クライアントデータセットコンポーネント	midas.dll または midaslib.dcu を使ってデータパケット内のデータを管理する特別なデータセット。クライアントデータセットは，クライアントアプリケーションで使用される。ローカルでキャッシュアップデートし，デルタパケットの形でアプリケーションサーバーに適用する
接続コンポーネント	サーバーの位置を特定し，接続を確立し，クライアントデータセットが IAppServer インターフェースを利用できるようにするコンポーネントファミリー。各接続コンポーネントは，特定の通信プロトコル専用になっている

プロバイダおよびクライアントデータセットコンポーネントは，midas.dll または midaslib.dcu を必要とします。これにより，データパケットとして格納されているデータセットを管理します（プロバイダはアプリケーションサーバーで使用され，クライアントデータセットはクライアントアプリケーションで使用されるため，midas.dll を使用する場合は，アプリケーションサーバーとクライアントアプリケーションの両方に配布しなければならないことに注意してください）。

BDE 対応データセットを使用する場合，アプリケーションサーバーに SQL エクスプローラも必要でしょう。SQL エクスプローラは，データベース管理を助けるとともに，データディクショナリにサーバーの制約をインポートすることによって多層アプリケーションの任意のレベルでサーバーの制約をチェックできるようにします。

メモ　アプリケーションサーバーを配布するにはサーバーライセンスを購入する必要があります。

これらのコンポーネントが適合するアーキテクチャの概要は，18-13 ページの「多層アーキテクチャを使う」で説明しています。

3 層アプリケーションの概要

プロバイダ ベースの 3 層アプリケーションでの処理の通常の順序は次のとおりです。

1. ユーザーがクライアントアプリケーションを起動します。クライアントはアプリケーションサーバーに接続します（設計時または実行時にアプリケーションサーバーを指定できます）。アプリケーションサーバーは，実行中でなければ起動されます。クライアントはアプリケーションサーバーと通信するための IAppServer インターフェースを受け取ります。
2. クライアントがアプリケーションサーバーにデータを要求します。クライアントはすべてのデータを一度に要求することもあれば，セッション全体を通じて連続的なデータチャンクを要求すること（フェッチオンデマンド）もあります。
3. アプリケーションサーバーが，必要であれば最初にデータベース接続を確立してデータを取り出し，クライアント用にデータをパッケージ化して，データパケットをクライアントに返します。追加の情報（たとえば，項目表示特性）を，データパケットのメタデータに含めることができま

す。このようにデータをパッケージ化してデータパケットにする過程を「プロバイドする」と言います。

4. クライアントがデータパケットをデコードして，ユーザーにデータを表示します。
5. ユーザーがクライアントアプリケーションと対話すると，データは更新されます（レコードの追加，削除，または変更）。この変更はクライアントによって変更ログに格納されます。
6. 一般にはユーザーアクションに応じて，最終的にクライアントが更新をアプリケーションサーバーに適用します。更新を適用するために，クライアントは変更ログをパッケージ化してデータパケットとしてサーバーに送ります。
7. アプリケーションサーバーは，（該当する場合に，トランザクションのコンテキスト内で）パッケージをデコードし，更新内容を登録します。レコードが登録できない場合（たとえば，クライアントがレコードを要求した後，クライアントがそのレコードの更新を適用する前に，別のアプリケーションがそのレコードを変更したため），アプリケーションサーバーは，そのクライアントの変更を現在のデータで調停するよう試みるか，登録できなかったレコードを保存します。レコードを登録し，問題のあるレコードをキャッシングするこのプロセスを「解決」といいます。
8. 解決処理を終了したアプリケーションサーバーが，他の解決処理を行うために未登録のレコードをクライアントに返します。
9. クライアントが未解決のレコードを調停します。クライアントはさまざまな方法で未解決のレコードを調停できます。通常，クライアントは，レコードの登録を妨げた状況の修正を試みるかまたは変更を破棄します。問題の状況が修正できない場合，クライアントは再び更新を適用します。
10. クライアントはサーバーからのデータを更新します。

クライアントアプリケーションの構造

エンドユーザーにとって，多層アプリケーションにおけるクライアントアプリケーションの外観と動作は，キャッシュアップデートを使用する 2 層アプリケーションと変わりません。ユーザーとの対話は，TClientDataSet コンポーネントからのデータを表示する標準のデータベース対応コントロールを介して行われます。クライアントデータセットのプロパティ，イベント，およびメソッドの使い方についての詳細は，第 27 章「クライアントデータセットの使い方」を参照してください。

TClientDataSet は，プロバイダからデータを取得し，プロバイダに更新内容を適用します。外部プロバイダでクライアントデータセットを使用する 2 層アプリケーションとまったく同様です。プロバイダについての詳細は，第 28 章「プロバイダコンポーネントの使い方」を参照してください。プロバイダとの通信を手助けするクライアントデータセット機能についての詳細は，27-23 ページの「クライアントデータセットでデータプロバイダを使用する」を参照してください。

クライアントデータセットは，IAppServer インターフェースを介してプロバイダと通信します。クライアントデータセットは，このインターフェースを接続コンポーネントから得ます。接続コンポーネントは，アプリケーションサーバーへの接続を確立します。各種通信プロトコル用の接続コンポーネントが利用できます。これらの接続コンポーネントを次の表にまとめます。

表 29.2 接続コンポーネント

コンポーネント	プロトコル
TDCOMConnection	DCOM
TSocketConnection	Windows ソケット（TCP/IP）SocketConnection
TWebConnection	HTTP
TSOAPConnection	SOAP（HTTP および XML）

メモ コンポーネントパレットの［DataSnap］ページでは，アプリケーションサーバーにまったく接続せず，その代わりに，同じアプリケーションでプロバイダと通信するときに使用するクライアントデータセット用に IAppServer インターフェースを備えている接続コンポーネントを用意しています。この TLocalConnection コンポーネントは必須というわけではありませんが，後から多層アプリケーションに簡単に拡張することができます。

接続コンポーネントの使い方についての詳細は，29-23 ページの「アプリケーションサーバーへの接続」を参照してください。

アプリケーションサーバーの構造

アプリケーションサーバーを設定して実行しても，サーバーはクライアントアプリケーションとの接続を確立しません。かわりに，接続はクライアントアプリケーションによって開始，維持されます。クライアントアプリケーションは接続コンポーネントを使ってアプリケーションサーバーに接続します。アプリケーションサーバーのインターフェースを使って選択したプロバイダと交信します。入ってくるリクエストを管理したりインターフェースを供給するためのコードを書かなくても，これがすべて自動的に行われます。

アプリケーションサーバーの基本は，IAppServer インターフェースをサポートする専用データモジュールであるリモートデータモジュールです（Web サーバーとしても機能するアプリケーションサーバーでは，リモートデータモジュールが IAppServerSOAP インターフェースもサポートしており，IAppServer よりもむしろこちらを使用します）。クライアントアプリケーションは，リモートデータモジュールのインターフェースを使用して，アプリケーションサーバー上のプロバイダと通信します。リモートデータモジュールが IAppServerSOAP を使用する場合，接続コンポーネントはこのインターフェースをクライアントデータセットが使用できる IAppServer インターフェースに適合させます。

リモートデータモジュールには 2 つのタイプがあります。

- COM ベースのリモートデータモジュールは，関連付けられたオブジェクトを使用するデータモジュールです。このオブジェクトは実装オブジェクトと呼ばれ，REMOTEDATAMODULE_IMPL() から派生します。REMOTEDATAMODULE_IMPL は Atlvcl.h で定義されているマクロで，実装オブジェクトの下位オブジェクトをリストしています。具体的には，ATL クラスの CComObjectRootEx および CComCoClass のほか，IAppServer インターフェースなどが下位オブジェクトです。MTS または COM+ の提供する分散アプリケーションサービスを利用できるアプリケーションサーバーを作成する場合は，REMOTEDATAMODULE_IMPL にも IObjectControl インターフェースが含まれます。IObjectControl は，すべてのトランザクションオブジェクトで要求されるインターフェースです。

プログラマにかわってウィザードが実装クラスを作成し，この実装クラスが IObjectContext インターフェースにアクセスします。実装クラスにかわってシステムが IObjectContext インターフェースを提供します。実装クラスは，IObjectContext を使ってトランザクションを管理し，リソースを解放し，セキュリティサポートを利用します。

- SOAP データモジュールは，IAppServerSOAP インターフェースを呼び出し可能インターフェースとして実装するデータモジュールです。これらのデータモジュールを Web サービス アプリケーションに追加すると，クライアントは Web サービスとしてデータにアクセスできます。Web サービスアプリケーションについての詳細は，第 36 章「Web サービスの使い方」を参照してください。

メモ MTS または COM+ のもとでアプリケーションサーバーを配布する場合，リモートデータモジュールには，アプリケーションサーバーがアクティブ，または非アクティブになるときのイベントが含まれます。これにより，アプリケーションサーバーがアクティブになったときにデータベースに接続され，非アクティブになったときに切断されるようにできます。

リモートデータモジュールの内容

リモートデータモジュールには，ほかのデータモジュールと同様に，非ビジュアルコンポーネントを含められます。ただし，含める必要のあるコンポーネントが次のようにいくつかあります。

- リモートデータモジュールがデータベースサーバーから情報を公開する場合，そのデータサーバーからのレコードを表すためにデータセットを含める必要があります。データセットとデータベースサーバーとの対話に，ある種のデータベース接続コンポーネントなど，他のコンポーネントが必要になることがあります。データセットについての詳細は，第 22 章「データセットについて」を参照してください。データベース接続コンポーネントについての詳細は，第 21 章「データベースへの接続」を参照してください。

  リモートデータモジュールがクライアントに公開するすべてのデータセットについて，データセットプロバイダを含めなければなりません。データセットプロバイダは，データをクライアントデータセットに送れるようにデータパケットの形にパッケージ化し，クライアントデータセットから受信した更新をソースデータセットまたはデータベースサーバーに戻します。データセットの状態についての詳細は，第 28 章「プロバイダコンポーネントの使い方」を参照してください。

- リモートデータモジュールがクライアントに公開するすべての XML ドキュメントについて，XML プロバイダを含めなければなりません。XML プロバイダは，データベースサーバーではなく XML ドキュメントからデータを取得し，更新を XML ドキュメントに適用することを除き，データセットプロバイダに似た動作をします。XML プロバイダについての詳細は，30-8 ページの「XML ドキュメントをプロバイダのソースとして使う」を参照してください。

メモ データセットをデータベースサーバーに接続するデータベース接続コンポーネントと多層アプリケーション内のクライアントアプリケーションによって使用される接続コンポーネントとを混同しないでください。多層アプリケーションの接続コンポーネントは，コンポーネントパレットの［DataSnap］ページまたは［WebServices］ページにあります。

トランザクションデータモジュールを使う

MTS（Windows 2000 より前）または COM+（Windows 2000 以降）の提供する分散アプリケーション用の特別なサービスを利用したアプリケーションサーバーを作成することができます。そのためには，通常のリモートデータモジュールでなく，トランザクションデータモジュールを作成します。

トランザクションデータモジュールを使ってアプリケーションを作成すると，以下の特別なサービスを利用できます。

- **セキュリティ。**MTS と COM+ は，アプリケーションサーバーにロール（役割）ベースのセキュリティを提供します。クライアントに役割が割り当てられ，それによってクライアントがアプリケーションサーバーのインターフェースにアクセスできるかどうか決まります。IObjectContext を使うと，これらのセキュリティサービスにアクセスできます（作成した実装クラスを通じて IObjectContext にアクセスする）。MTS セキュリティと COM+ セキュリティについての詳細は，44-16 ページの「ロールベースのセキュリティ」を参照してください。
- **データベースハンドルのプーリング。**トランザクションデータモジュールにより自動的にデータベース接続がプールされ，ADO または（MTS を使用中で，MTS POOLING をオンにする場合は）BDE を介してデータベース接続が行われます。あるクライアントがデータベース接続を終了すると，別のクライアントがそれを再使用できます。これにより，中間層がリモートデータベースサーバーからログオフして再びログオンする必要がなくなるので，ネットワークトラフィックが削減されます。データベースハンドルをプールする場合，アプリケーションが接続を最大限に共有できるように，データベースコンポーネント接続コンポーネントの KeepConnection プロパティを **false** に設定しなければなりません。データベースハンドルのプーリングについての詳細は，44-6 ページの「データベースリソースディスペンサ」を参照してください。
- **トランザクション。**トランザクションデータモジュールを使うと，単一のデータベース接続より高度なトランザクションサポートを提供できます。トランザクションデータモジュールは，複数のデータベースにまたがるトランザクションに参加することも，データベースをまったく扱わない機能を持つことも可能です。トランザクションデータモジュールなどのトランザクションオブジェクトが提供するトランザクションサポートについての詳細は，29-18 ページの「多層アプリケーションでのトランザクション管理」を参照してください。
- **即時アクティブ化と即時非アクティブ化。**インスタンスが，必要に応じてアクティブになったり非アクティブになったりするようにサーバーを書くことができます。即時アクティブ化（just-in-time activation）と即時非アクティブ化を使用する場合，アプリケーションサーバーは，クライアントリクエストを処理する必要があるときだけインスタンス化されます。これにより，リソース（データベースハンドルなど）が使用されないときにアプリケーションサーバーが拘束されるのを防ぎます。

  即時アクティブ化と即時非アクティブ化の使用は，すべてのクライアントをリモートデータモジュールの 1 つのインスタンスを介してルーティングする場合と，クライアント接続ごとに個別のインスタンスを作成する場合の中間に位置します。リモートデータモジュールのインスタンスが 1 つの場合，アプリケーションサーバーは，すべてのデータベース呼び出しを 1 つのデータベース接続を介して処理しなければなりません。これがネックになり，クライアント数が多い場合はパフォーマンスに影響します。リモートデータモジュールのインスタンスが複数ならば，各インスタンスは別個のデータベース接続を保持できるので，データベースアクセスを 1 本にシリアル化する必要がありません。ただし，これでは，データベース接続が別のクライアントのリモートデータモジュールに関連付けられている間ほかのクライアントがデータベース接続を使えないので，リソースを独占してしまいます。

トランザクション，即時アクティブ化，および即時非アクティブ化を利用するには，リモートデータモジュールのインスタンスはステートレスでなければなりません。これは，クライアントがステート

情報に依存している場合には追加サポートを提供しなければならないことを意味します。たとえばクライアントは，インクリメンタルフェッチを行うときには，現在のレコードに関する情報を渡さなければなりません。多層アプリケーションにおけるステート情報とリモートデータモジュールについての詳細は，29-20 ページの「リモートデータモジュールでのステート情報のサポート」を参照してください。

デフォルトでは，トランザクションデータモジュールへの自動的に生成された呼び出しはすべて，トランザクション的です（つまり，そのような呼び出しの終了時に，データモジュールを非アクティブ化し，現在のトランザクションをコミットまたはロールバックするとみなされます）。AutoComplete プロパティを **false** に設定することに持続的ステート情報に依存するトランザクションデータモジュールを書くこともできますが，カスタムインターフェースを使わない限り，そのようなモジュールは，トランザクション，即時アクティブ化，即時非アクティブ化をサポートしません。

注意 アプリケーションサーバーにトランザクションデータモジュールが含まれている場合は，トランザクションデータモジュールがアクティブになるまでデータベース接続を開いてはいけません。アプリケーション開発中は，アプリケーションを実行する前にすべてのデータセットがアクティブでデータベースが接続されていないことを確認してください。アプリケーション自体については，データモジュールがアクティブになったときデータベース接続を開き，データモジュールが非アクティブになったとき閉じるコードを追加しなければなりません。

リモートデータモジュールのプーリング

オブジェクトをプールすると，クライアント間で共有するアプリケーションサーバーのキャッシュが作成されるので，リソースを節約できます。どのように動作するかは，リモートデータモジュールの種類と接続プロトコルによって異なります。

トランザクションデータモジュールを作成して COM+ にインストールする場合には，COM+ Component Manager を使うと，アプリケーションサーバーをプールオブジェクトとしてインストールできます。詳細は，44-9 ページの「オブジェクトのプール」を参照してください。

トランザクションデータモジュールを使わない場合でも，TWebConnection を使って接続を形成すれば，オブジェクトのプールを利用できます。HTTP を使ったオブジェクトプーリングでは，作成するアプリケーションサーバーのインスタンス数を制限します。これにより，保持しなければならないデータベース接続数，ならびにアプリケーションサーバーが使用するその他のリソースを制限することができます。

Web サーバーアプリケーション（アプリケーションサーバーに呼び出しを渡すアプリケーション）は，クライアントリクエストを受け取ると，プール内にあるアプリケーションサーバーの最初のサーバーにクライアントリクエストを渡します。使用できるアプリケーションサーバーがない場合は，（指定されている最大数の範囲内で）新規のモジュールを作成します。これにより，すべてのクライアントを単一のアプリケーションサーバーインスタンスを経由させること（これがボトルネックになることがある）と，各クライアント接続ごとに別々のインスタンスを作成すること（多くのリソースを消費する）の中間のものを提供します。

プール内のアプリケーションサーバーインスタンスがしばらくクライアントリクエストを受け取らないと，自動的に解放されます。そのため，プールが使用されていないのにリソースを独占するという状況はなくなります。

Web 接続（HTTP）の使用時にオブジェクトプーリングをセットアップするには，次のようにします。

1. 実装クラスの UpdateRegistry メソッドを配置します。UpdateRegistry メソッドは，実装ユニットのヘッダーファイルに現れます。

```
static HRESULT WINAPI UpdateRegistry(BOOL bRegister)
{
  TRemoteDataModuleRegistrar regObj(GetObjectCLSID(), GetProgID(), GetDescription());
  regObj.Singleton = false;
  regObj.EnableWeb = true;
  regObj.EnableSocket = true;
  return regObj.UpdateRegistry(bRegister);
}
```

2. TRemoteDataModuleRegistrar のインスタンスである regObj 変数の RegisterPooled フラグを，**true** に設定します。リモートデータモジュールのキャッシュの管理方法を指示するには，regObj の他のプロパティを設定します。たとえば以下のコードは，リモートデータモジュールの最大インスタンス数を 10 にし，15 分以上アイドルになるとキャッシュから解放します。

```
static HRESULT WINAPI UpdateRegistry(BOOL bRegister)
{
  TRemoteDataModuleRegistrar regObj(GetObjectCLSID(), GetProgID(), GetDescription());
  regObj.Singleton = false;
  regObj.EnableWeb = true;
  regObj.EnableSocket = true;
  regObj.RegisterPooled = true;
  regObj.Timeout = 15;
  regObj.Max = 10;
  return regObj.UpdateRegistry(bRegister);
}
```

どちらの方法でオブジェクトプーリングを行うにせよ，作成するアプリケーションサーバーがステートレスでなければなりません。その理由は，単一のインスタンスが，複数クライアントからのリクエストを処理する可能性があるからです。単一インスタンスが持続的ステート情報に依存していると，クライアントが互いに干渉するおそれがあります。リモートデータモジュールがステートレスであることを確認する方法については，29-20 ページの「リモートデータモジュールでのステート情報のサポート」を参照してください。

接続プロトコルの選択

クライアントアプリケーションをアプリケーションサーバーに接続するために使う各種通信プロトコルには，それぞれに特有な利点があります。プロトコルを選択する前に，クライアント数はどれだけ見込まれるか，アプリケーションをどう配布するか，そして将来の開発の見通しを考えなければなりません。

DCOM 接続を使う

DCOM は，通信への最も直接的なアプローチを提供し，サーバーにほかの実行時アプリケーションを必要としません。ただし，DCOM は Windows 95 には付属していないため，古いクライアントマシンに DCOM がインストールされていないことがあります。

DCOM は，トランザクションデータモジュールの作成時に，セキュリティサービスを使用可能にする唯一のアプローチを提供します。このセキュリティサービスは，トランザクションオブジェクトの

呼び出し元にロール（役割）を割り当てることをベースにしています。DCOM を使うとき，DCOM は，アプリケーションサーバー（MTS または COM+）を呼び出すシステムへの呼び出し元を特定します。したがって，呼び出し元のロールを正確に特定することが可能です。ただし，ほかのプロトコルを使うときは，アプリケーションサーバーとは別に実行時実行ファイルがあって，そのファイルがクライアント呼び出しを受信します。この実行時実行ファイルは，クライアントにかわってアプリケーションサーバーへの COM 呼び出しを行います。このため，別々のクライアントにロールを割り当てることはできません。事実上，この実行時実行ファイルが唯一のクライアントです。セキュリティおよびトランザクションオブジェクトについての詳細は，44-16 ページの「ロールベースのセキュリティ」を参照してください。

ソケット接続を使う

TCP/IP ソケットを使うと，軽量なクライアントを作成できます。たとえば，Web ベースのクライアントアプリケーションを作成する場合，クライアントシステムが DCOM をサポートしているとは限りません。ソケットは，アプリケーションサーバーへの接続に利用できるとわかっている最小共通標準となります。ソケットについての詳細は，第 37 章「ソケットの操作」を参照してください。

クライアントから直接リモートデータモジュールをインスタンス化する（DCOM の場合）かわりに，ソケットはサーバーにある別のアプリケーション（ScktSrver.exe）を使います。このアプリケーションは，クライアントリクエストを受け取り，COM を使ってアプリケーションサーバーをインスタンス化します。クライアント上の接続コンポーネントとサーバー上の ScktSrvr.exe は，IAppServer 呼び出しをマーシャルする役割を持ちます。

メモ ScktSrvr.exe は，NT サービスアプリケーションとして実行できます。ScktSrvr.exe をサービス一覧に登録するには，-install コマンドラインオプションを指定して実行します。登録解除するには，**-uninstall** コマンドラインオプションを指定して実行します。

ソケット接続が使用できるためには，アプリケーションサーバーが，ソケット接続によって自身がクライアントから利用可能なことを登録しておく必要があります。デフォルトでは，新規のリモートデータモジュールはすべて，実装オブジェクトの UpdateRegistry メソッドにおいて，TRemoteDataModuleRegistrar オブジェクトを介して自動的に登録されます。この自動登録を不可にするには，オブジェクトの EnableSocket プロパティを **false** に設定します。

メモ 以前のサーバーはこの登録を追加していないので，ScktSrvr.exe の［接続｜登録されたオブジェクトのみ表示］メニュー項目のチェックマークをはずすことによって，アプリケーションサーバーが登録されているかどうかのチェックを使用不可にすることができます。

ソケットの使用においては，クライアントシステムがアプリケーションサーバー上のインターフェースへの参照を解放する前にクライアントシステムに障害が発生する場合に対して，サーバーは保護されていません。このため，（周期的に keep-alive メッセージを送信する）DCOM を使う場合よりもメッセージトラフィックは少なくなりますが，アプリケーションサーバーがクライアントの障害に気付かないためにリソースを解放できない事態が発生します。

Web 接続を使う

HTTP を使うと，ファイアウォールによって保護されたサーバーと交信するクライアントを作成できます。HTTP メッセージは内部アプリケーションへの制御付きアクセスを提供するので，クライアントアプリケーションを広範囲に安全に配布することができます。ソケット接続と同様に，HTTP メッ

セージはアプリケーションサーバーへの接続に使用できる最も一般的な手段を提供します。HTTP メッセージについての詳細は，第 32 章「インターネットサーバーアプリケーションの作成」を参照してください。

DCOM の場合のようにクライアントからリモートデータモジュールを直接インスタンス化する代わりに，HTTP 接続はサーバー上の Web サーバーアプリケーション（httpsrvr.dll）を使います。この DLL は，クライアントのリクエストを受け付け，COM を使ってアプリケーションサーバーをインスタンス化します。このため，これらは Web 接続と呼ばれています。クライアント上の接続コンポーネントとサーバー上の httpsrvr.dll は，IAppServer 呼び出しをマーシャルする役割を持ちます。

Web 接続では，wininet.dll（クライアントシステム上で実行されるインターネットユーティリティのライブラリ）によって提供される SSL セキュリティを利用できます。サーバーシステム上の Web サーバーが認証を要求するように設定した後，Web 接続コンポーネントのプロパティを使ってユーザー名とパスワードを指定することができます。

セキュリティを強化する手段として，アプリケーションサーバーは，Web 接続によって自身がクライアントから利用可能なことを登録しなければなりません。デフォルトでは，新規のリモートデータモジュールはすべて，実装オブジェクトの UpdateRegistry メソッドにおいて，TRemoteDataModuleRegistrar オブジェクトを介して自動的に登録されます。この自動登録を不可にするには，オブジェクトの EnableWeb プロパティを **false** に設定します。

Web 接続では，オブジェクトのプールを利用できます。これによって，サーバーは，クライアントからのリクエストで利用できるアプリケーションサーバーインスタンスの，制限されたプールを作成することができます。アプリケーションサーバーをプールすると，データモジュールとそのデータベース接続が必要ないときでもサーバーのリソースが消費されるということはなくなります。オブジェクトのプールの詳細については，29-8 ページの「リモートデータモジュールのプーリング」を参照してください。

ほかのほとんどの接続コンポーネントとは異なり，接続が HTTP 経由で確立されているときはコールバックを使用できません。

SOAP 接続を使う

SOAP は，VCL または CLX で Web サービスアプリケーションをサポートする基盤となっているプロトコルです。SOAP は，XML エンコーディングを使用してメソッド呼び出しをマーシャルします。SOAP 接続では，HTTP をトランスポートプロトコルとして使用しています。

Windows と Linux の両方をサポートしているため，SOAP 接続にはクロスプラットフォームアプリケーションで動作するという利点があります。SOAP 接続は HTTP を使用するので，Web 接続と同じ利点があります。つまり，HTTP はすべてのクライアントで利用できるとわかっている最小共通標準であり，クライアントは「ファイアウォール」で保護されたアプリケーションサーバーと通信できるということです。SOAP を使ってアプリケーションを配布する方法についての詳細は，第 36 章「Web サービスの使い方」を参照してください。

HTTP 接続と同様，接続が SOAP 経由で確立されているときはコールバックを使用できません。SOAP 接続でも，アプリケーションサーバー内のリモートデータモジュールは 1 つに制限されています。

多層アプリケーションの構築

多層データベースアプリケーションを作成する一般的な手順は次のとおりです。

1. アプリケーションサーバーを作成します。
2. アプリケーションサーバーを登録またはインストールします。
3. クライアントアプリケーションを作成します。

作成する順序は重要です。アプリケーションサーバーを先に作成して実行してから，クライアントを作成しなければなりません。設計時には，クライアントをテストするためにアプリケーションサーバーに接続できます。もちろん，設計時にアプリケーションサーバーを指定しないでクライアントを作成し，実行時にサーバー名だけ指定することも可能です。ただし，そうすると，設計時のコーディングでアプリケーションが意図どおりに動作するか確かめられないし，オブジェクトインスペクタでサーバーとプロバイダを選択できません。

メモ クライアントアプリケーションをサーバーと異なるシステム上で作成する場合，DCOM 接続を使用するときは，アプリケーションサーバーをクライアントシステムに登録するとよいでしょう。これにより，接続コンポーネントは設計時にアプリケーションサーバーを認識できるので，オブジェクトインスペクタのドロップダウンリストでサーバー名とプロバイダ名を選択できます（Web 接続，SOAP 接続，またはソケット接続を使用するときは，接続コンポーネントが登録されたプロバイダ名をサーバーマシンから取得します）。

アプリケーションサーバーの作成

アプリケーションサーバーの作成は，ほとんどのデータベースアプリケーションの作成とよく似ています。主な違いは，アプリケーションサーバーにはリモートデータモジュールが含まれることです。

アプリケーションサーバーを作成する手順は次のとおりです。

1. 新規プロジェクトを開始します。
   - SOAP をトランスポートプロトコルとして使用している場合は，新しい Web サービスアプリケーションでなければなりません。［ファイル｜新規作成｜その他］を選択し，［新規作成］ダイアログの［Web Services］ページで，［SOAP サーバーアプリケーション］を選択します。
   - 他のトランスポートプロトコルについては，［ファイル｜新規作成｜アプリケーション］を選択するだけです。

   新規プロジェクトを保存します。
2. プロジェクトに新しいリモートデータモジュールを追加します。メインメニューの［ファイル｜新規作成｜その他］を選択し，［新規作成］ダイアログの［多層サポート］ページまたはを［Web サービス］ページ選択し，次のいずれかを選択します。
   - **リモートデータモジュール**。クライアントが DCOM，HTTP，またはソケットを使ってアクセスする COM オートメーションサーバーを作成する場合

- **トランザクションデータモジュール。**MTS または COM+ のもとで動作するリモートデータモジュールを作成する場合。接続は，DCOM，HTTP，またはソケットを使って形成される。ただし，DCOM 以外はセキュリティサービスをサポートしない
- **SOAP サーバーデータモジュール**。Web サービスアプリケーションで SOAP サーバーを作成する場合

リモートデータモジュールのセットアップについての詳細は，29-14 ページの「リモートデータモジュールをセットアップする」を参照してください。

メモ リモートデータモジュールまたはトランザクションデータモジュールを選択すると，ウィザードが，リモートデータモジュールへの参照を含み，その参照を使用してプロバイダを探す特別な COM オートメーションオブジェクトも作成します。このオブジェクトを「実装オブジェクト」と呼びます。SOAP データモジュールでは，IAppServerSOAP インターフェース自体を実装しているので，個別の実装オブジェクトは必要ありません。

3. データモジュールに適切なデータセットコンポーネントを配置し，それらをデータベースサーバーにアクセスするようセットアップします。
4. データモジュールのデータセットごとに TDataSetProvider コンポーネントを配置します。プロバイダは，クライアントリクエストの解決とデータのパッケージ化のために必要です。プロバイダコンポーネントの DataSet プロパティを，アクセスするデータセットの名前に設定します。プロバイダ用に追加のプロパティを設定できます。プロバイダのセットアップについての詳細は，第 28 章「プロバイダコンポーネントの使い方」を参照してください。

   XML ドキュメントのデータを操作する場合は，データセットと TDataSetProvider コンポーネントの代わりに TXMLTransformProvider コンポーネントを使用できます。TXMLTransformProvider を使用する場合，XMLDataFile プロパティの設定で，データの取り出し元および更新の適用先である XML ドキュメントを指定します。
5. イベント，共有ビジネスルール，共有データ検証，共有セキュリティを実装するアプリケーションサーバーコードを記述します。このコードを作成するときに，次のようにできます。
   - アプリケーションサーバーのインターフェースを拡張して，クライアントアプリケーションがサーバーを呼び出す方法を追加します。アプリケーションサーバーのインターフェースを拡張する方法の詳細は，29-17 ページの「アプリケーションサーバーのインターフェースを拡張する」で説明しています。
   - 更新を適用するときに自動的に作成されるトランザクションを超えるトランザクションサポートを行います。多層データベースアプリケーションのトランザクションについては，29-18 ページの「多層アプリケーションでのトランザクション管理」で説明しています。
   - アプリケーションサーバーでデータセット間のマスター / 詳細関係を作成します。マスター / 詳細関係については，29-19 ページの「マスター / 詳細関係のサポート」で述べています。
   - アプリケーションサーバーがステートレスであることを確認します。ステート情報の取り扱いについては，29-20 ページの「リモートデータモジュールでのステート情報のサポート」で説明します。

- アプリケーションサーバーを複数のリモートデータモジュールに分割します。複数のリモートデータモジュールの使い方については，29-21 ページの「複数のリモートデータモジュールを使う」で説明します。

6. アプリケーションサーバーを保存およびコンパイルして，登録またはインストールします。アプリケーションサーバーの登録については，29-22 ページの「アプリケーションサーバーを登録する」で説明します。
7. サーバーアプリケーションが DCOM または SOAP を使わない場合は，クライアントメッセージを受信し，リモートデータモジュールをインスタンス化し，インターフェース呼び出しをマーシャルする実行時ソフトウェアをインストールしなければなりません。
   - TCP/IP ソケットの場合，これはソケットディスパッチャアプリケーション，Scktsrvr.exe です。
   - HTTP 接続の場合，これは httpsrvr.dll です。httpsrvr.dll は，Web サーバーと一緒にインストールされなければならない ISAPI/NSAPI DLL です。

リモートデータモジュールをセットアップする

リモートデータモジュールを作成する場合，それがクライアントリクエストにどう応答するかを示す何らかの情報をプロバイドしなければなりません。この情報は，リモートデータモジュールの種類によって異なります。

トランザクション属性のないリモートデータモジュールの設定

トランザクション的属性を含めずにアプリケーションに COM ベースのリモートデータモジュールを追加するには，［ファイル｜新規作成｜その他］を選択し，［新規作成］ダイアログの［多層サポート］ページで［リモートデータモジュール］を選択します。リモートデータモジュールウィザードが表示されます。

リモートデータモジュールのクラス名を指定しなければなりません。これは，アプリケーションが作成する TCRemoteDataModule の派生クラスの基本名になります。また，アプリケーションサーバーのインターフェースにもこの基本名が使われます。たとえば，クラス名 MyDataServer を指定すると，ウィザードは TCRemoteDataModule の下位オブジェクトである TMyDataServer を宣言する新しいユニットを作成します。ウィザードは，ユニットのヘッダーファイルに IAppServer の下位の IMyDataServer を実装する実装クラス（TMyDataServerImpl）も宣言します。

メモ 新規のインターフェースには，自分でプロパティやメソッドを追加できます。詳細については，29-17 ページの「アプリケーションサーバーのインターフェースを拡張する」を参照してください。

リモートデータモジュールウィザードでスレッドモデルを指定しなければなりません。［シングルスレッドモデル］，［アパートメントスレッドモデル］，［フリースレッドモデル］，［フリー / アパートメント両用］，または［ニュートラル］を選択できます。

- ［シングルスレッドモデル］を選択すると，COM は一度に 1 つのクライアントリクエストだけがサービスされるようにします。クライアントリクエストが相互に衝突することを心配する必要はありません。
- ［アパートメントスレッドモデル］を選択すると，COM は，リモートデータモジュールのどのインスタンスも一度に 1 つのリクエストだけをサービスするようにします。アパートメントスレッ

ドライブラリでコーディングする場合，グローバル変数や，リモートデータモジュール内にないオブジェクトを使うならばスレッドの衝突に対する対策を考えなければなりません。これは，BDE 対応のデータセットを使う場合に推奨されるモデルです（ただし，BDE 対応データセットに対するスレッド関連事項を処理するために，AutoSessionName プロパティが **true** に設定されているセッションコンポーネントが必要です）。

- ［フリースレッドモデル］を選択すると，アプリケーションはいくつかのスレッドのクライアントリクエストを同時に受信できます。アプリケーションがスレッドを安全に処理できるよう，作成者が考えなければなりません。複数のクライアントがリモートデータモジュールに同時にアクセスできるので，インスタンスデータ（プロパティ，含まれるオブジェクトなど），ならびにグローバル変数を保護しなければなりません。これは ADO データセットを使う場合に推奨されるモデルです。
- ［フリー / アパートメント両用］を選択すると，ライブラリは［フリースレッドモデル］オプションを選択したときと同じように動作しますが，1 点だけ例外があります。すべてのコールバック（クライアントインターフェースへの呼び出し）はシリアル化されます。
- ［ニュートラル］を選択すると，フリースレッドモデルと同様に，リモートデータモジュールは複数の呼び出しを別々のスレッドで同時に受信できます。ただし COM は，複数のスレッドが同じメソッドに同時にアクセスするのを禁止します。

トランザクションリモートデータモジュールの設定

MTS または COM+ を使用する場合にアプリケーションにリモートデータモジュールを追加するには，［ファイル｜新規作成｜その他］を選択し，［新規作成］ダイアログの［多層サポート］ページで［トランザクションデータモジュール］を選択します。トランザクションデータモジュールウィザードが表示されます。

リモートデータモジュールのクラス名を指定しなければなりません。これは，アプリケーションが作成する TCRemoteDataModule の派生クラスの基本名になります。また，アプリケーションサーバーのインターフェースにもこの基本名が使われます。たとえば，クラス名 MyDataServer を指定すると，ウィザードは TCRemoteDataModule の下位オブジェクトである TMyDataServer を宣言する新しいユニットを作成します。ウィザードは，ユニットのヘッダーファイルに（IAppServer から派生した）IMyDataServer および（全トランザクションオブジェクトで要求される）IObjectControl を実装する実装クラス（TMyDataServerImpl）も宣言します。TMyDataServerImpl には，トランザクションの管理や，セキュリティのチェックなどに使用できる IObjectContext インターフェース用のデータメンバーが含まれます。

メモ 新規のインターフェースに自分でプロパティやメソッドを追加できます。詳細については，29-17 ページの「アプリケーションサーバーのインターフェースを拡張する」を参照してください。

トランザクションデータモジュールウィザードでスレッドモデルを指定しなければなりません。［シングルスレッドモデル］，［アパートメントスレッドモデル］，または［フリー / アパートメント両用］を選択します。

- ［シングルスレッドモデル］を選択すると，クライアントリクエストがシリアル化され，一度に 1 つのクライアントリクエストだけがサービスされます。クライアントリクエストが相互に衝突することを心配する必要はありません。

- ［アパートメントスレッドモデル］を選択すると，システムは，リモートデータモジュールのどのインスタンスも一度に 1 つのリクエストに対してだけサービスするようにします。また，呼び出しはいつも同じスレッドを使います。グローバル変数，またはそのリモートデータモジュールに含まれていないオブジェクトを使う場合，スレッドの衝突に対して保護策を講じなければなりません。グローバル変数を使うかわりに，共有プロパティマネージャを使用できます。共有プロパティマネージャについては，44-6 ページの「共有プロパティマネージャ」を参照してください。
- ［フリー / アパートメント両用］を選択すると，［アパートメントスレッドモデル］を選択したときと同じように，MTS はアプリケーションサーバーのインターフェースを呼び出します。ただし，クライアントアプリケーションに行うコールバックはすべてシリアル化されるので，それらが相互に衝突するおそれは心配ありません。

メモ MTS での［アパートメントスレッドモデル］は，DCOM の同じ名前のモデルとは異なります。

リモートデータモジュールのトランザクション属性も指定しなければなりません。次のオプションから選択できます。

- ［トランザクションが必要］。このオプションを選択すると，クライアントがアプリケーションサーバーのインターフェースを使用するたびに，その呼び出しはトランザクションのコンテキスト内で実行されます。呼び出し元がトランザクションを指定した場合，新規トランザクションを作成する必要はありません。
- ［新しいトランザクションが必要］。このオプションを選択すると，クライアントがアプリケーションサーバーのインターフェースを使用するたびに，その呼び出し用に新規のトランザクションが自動的に作成されます。
- ［トランザクションのサポート］。このオプションを選択すると，アプリケーションサーバーはトランザクションのコンテキスト内で使用できますが，呼び出し元がインターフェースを呼び出すときは呼び出し元がトランザクションを指定しなければなりません。
- ［サポートしない］。このオプションを選択すると，アプリケーションサーバーをトランザクションのコンテキスト内では使用できません。

TSoapDataModule の設定

アプリケーションに TSoapDataModule を追加するには，［ファイル｜新規作成｜その他］を選択して，［新規作成］ダイアログの［WebServices］ページで［SOAP データモジュール］を選択します。SOAP データモジュールウィザードが表示されます。

SOAP データモジュールのクラス名を指定しなければなりません。これは，アプリケーションが作成する TSoapDataModule の派生クラスの基本名になります。たとえば，クラス名 MyDataServer を指定すると，ウィザードは TSoapDataModule の下位オブジェクトである TMyDataServer を宣言する新しいユニットを作成します。この新しいクラスは，TSoapDataModule から IAppServer と IAppServerSOAP の実装を継承します。

SOAP データモジュールは，他のリモートデータモジュールとは異なり，IAppServer または IAppServerSOAP から派生するインターフェースを実装しません。これは，Web サービスアプリケーションが入ってくるインターフェース呼び出しをディスパッチする方法が違うためです。かわりに，Web サービスの追加ウィザードを使用して新しいインターフェースを追加できます。

メモ TSoapDataModule を使用するには，新しいデータモジュールを Web サービスアプリケーションに追加しなければなりません。IAppServerSOAP インターフェースは，呼び出し可能なインターフェースであり，新しいユニットのスタートアップコードに登録されます。このため，メイン Web モジュールの呼び出し側コンポーネントは受け取った呼び出しをすべてデータモジュール側へ転送することができます。

メモ アプリケーションサーバーで，Kylix 2 または Delphi 6（アップデートパッチ 2 より前のバージョン）で作成したクライアントに応答させる場合には，IAppServer インターフェースを登録するコードを追加する必要があります。IAppServerSOAP を登録しているスタートアップコードを見つけ，その登録呼び出しの直後に，RegDefIAppServerInvClass グローバル関数の呼び出しを追加します。この関数は，データモジュールを IAppServer の実装として登録します。

アプリケーションサーバーのインターフェースを拡張する

COM ベースのサーバーでは，クライアントアプリケーションは，データモジュールウィザードによって作成された実装クラスを作成または接続することによって，アプリケーションサーバーとやりとりを行います。アプリケーションサーバーとのすべての通信にインスタンスのインターフェースを使用します。

COM ベースのアプリケーションを使用する場合は，実装クラスのインターフェースに追加することにより，クライアントアプリケーションのサポートを追加できます。このインターフェースは IAppServer の下位クラスで，リモートデータモジュールを作成するときにウィザードによって自動的に作成されます。

実装クラスのインターフェースに追加するには，タイプライブラリエディタを使用してください。タイプライブラリエディタの使い方については，第 39 章「タイプライブラリの操作」を参照してください。

COM インターフェースに追加する場合，変更内容はユニットソースコードとタイプライブラリファイル（.TLB）に追加されます。

メモ タイプライブラリエディタで［更新］を選択し，IDE から変更を保存することによって，TLB ファイルを明示的に保存しなければなりません。

実装クラスのインターフェースへの追加が済んだら，実装クラスに追加されたプロパティとメソッドを指定してください。コードを追加してこの実装を完成させます。具体的には，新しいメソッドの本文を記述します。

クライアントアプリケーションは，接続コンポーネントの AppServer プロパティを使ってインターフェース拡張機能を呼び出します。この方法についての詳細は，29-28 ページの「サーバーインターフェースの呼び出し」を参照してください。

アプリケーションサーバーのインターフェースにコールバックを追加する

コールバックを導入すると，アプリケーションサーバーがクライアントアプリケーションを呼び出すようにできます。これを行うには，クライアントアプリケーションがアプリケーションサーバーのメソッドの 1 つにインターフェースを渡し，後でアプリケーションサーバーがこのメソッドを必要に応じて呼び出します。ただし，実装クラスのインターフェースの拡張機能にコールバックが含まれてい

ると，HTTP または SOAP ベースの接続を使用することができません。TWebConnection と TSoapConnection は，コールバックをサポートしていません。ソケットベースの接続を使う場合，クライアントアプリケーションは，コールバックを使用するかどうかを SupportCallbacks プロパティの設定によって示さなければなりません。それ以外のタイプの接続は，自動的にコールバックをサポートします。

トランザクションアプリケーションサーバーのインターフェースを拡張する

トランザクションまたは即時アクティブ化を使っている場合は，すべての新規メソッドは終了時に IObjectContext の SetComplete メソッドを呼び出して，終了を示さなければなりません。これにより，トランザクションが完了し，アプリケーションサーバーが非アクティブ化できるようになります。

さらに，セーフ参照が出される場合を除き，クライアントがアプリケーションサーバー上のオブジェクトまたはインターフェースと直接交信することを許可する新規メソッドから値を返すことはできません。ステートレスの MTS データモジュールを使っている場合は，セーフ参照の使用を無視することによって，リモートデータモジュールがアクティブであることが保証できないためにクラッシュを引き起こす可能性があります。セーフ参照についての詳細は，44-25 ページの「オブジェクト参照を渡す」を参照してください。

多層アプリケーションでのトランザクション管理

クライアントアプリケーションがアプリケーションサーバーに更新を適用するとき，プロバイダコンポーネントは，更新を適用してエラーを解決する過程を自動的に 1 つのトランザクションにラップします。このトランザクションは，問題レコードの件数が ApplyUpdates メソッドの引数として指定される MaxErrors の値を超えなければコミットされます。そうでなければ，ロールバックされます。

また，データベース接続コンポーネントを追加したり，SQL をデータベースサーバーに送信して直接トランザクションを管理したりすることにより，サーバーアプリケーションにトランザクションのサポートを追加できます。これは，2 層アプリケーションにおけるトランザクションの管理と同じ方法で行えます。この種類のトランザクション制御についての詳細は，21-6 ページの「トランザクションの管理」を参照してください。

トランザクションデータモジュールを使う場合は，MTS トランザクションまたは COM+ トランザクションを使うことによってトランザクションサポートを拡張できます。このトランザクションには，アプリケーションサーバー上の任意のビジネスロジックを含められます。また，2 相コミットをサポートしているので，トランザクションが複数のデータベースにまたがってもかまいません。

BDE ベースおよび ADO ベースのデータアクセスコンポーネント以外は，2 相コミットをサポートしません。複数のデータベースにまたがるトランザクションにする場合は，Interbase Express または dbExpress コンポーネントを使ってはいけません。

注意 BDE を使うとき，2 相コミットは Oracle7 および MS-SQL データベースでのみ完全に実装されます。トランザクションが複数のデータベースにかかわっていて，それらのデータベース中に Oracle7 または MS-SQL 以外のリモートサーバーがあるならば，作成するトランザクションは部分的にしか成功しないという危険性が多少あります。ただし，個々のデータベースに限って言えば，常にトランザクションはサポートされています。

デフォルトでは，トランザクションデータモジュールの IAppServer 呼び出しはすべてトランザクション的です。プログラマがすべきことは，データモジュールのトランザクション属性を設定するとき，そのモジュールがトランザクションに参加しなければならないことを指示することだけです。また，定義したトランザクションをカプセル化するメソッド呼び出しを含めることにより，アプリケーションサーバーのインターフェースを拡張することができます。

トランザクション属性を設定するとき，アプリケーションサーバーがトランザクションを必要とすることを指示しておけば，クライアントがインターフェースのメソッドを呼び出すたびに，それは自動的にトランザクションの中にラップされます。それ以降，クライアントからアプリケーションサーバーへのすべての呼び出しは，トランザクションが完了したことを記述するまでそのトランザクション内にリストされます。これらの呼び出しは，全体として成功するかロールバックされます。

メモ MTS または COM+ トランザクションを，データベース接続コンポーネントによって作成された明示的トランザクションや，明示的 SQL コマンドを使って作成された明示的トランザクションと組み合わせないでください。トランザクションにトランザクションデータモジュールが含められると，データベース呼び出しもすべてそのトランザクションに含められます。

MTS トランザクションおよび COM+ トランザクションについての詳細は，44-10 ページの「MTS および COM+ のトランザクションサポート」を参照してください。

マスター / 詳細関係のサポート

クライアントアプリケーションのクライアントデータセット間のマスター / 詳細関係は，テーブル型データセットを使用したセットアップと同じように作成できます。この方法でのマスター / 詳細関係のセットアップについての詳細は，22-33 ページの「マスター / 詳細関係の作成」を参照してください。

ただし，このアプローチには 2 つの短所があります。

- 詳細テーブルは，たとえ一度に 1 つの詳細セットしか使わなくても，アプリケーションサーバーからすべてのレコードを取り出して格納しなければならない（この問題は，パラメータの使用によって軽減できる。詳細については，27-28 ページの「パラメータでレコードを制限する」を参照してください。）
- クライアントデータセットが更新をデータセットレベルで適用することと，マスター / 詳細の更新は複数のデータセットにわたるので，更新を適用することが非常に難しい。データベース接続コンポーネントを使用して単一のトランザクションで複数のテーブルに更新を適用できる 2 層環境でさえ，マスター / 詳細フォームで更新を適用するには慎重を要する。

多層アプリケーションでは，ネストされたテーブルを使ってマスター / 詳細関係を表すことによって，これらの問題を回避できます。データセットから提供するときにこれを行うには，アプリケーションサーバー上のデータセット間にマスター / 詳細関係をセットアップします。次に，プロバイダコンポーネントの DataSet プロパティをマスターテーブルに設定します。XML ドキュメントから提供するときにネストされたテーブルを使用してマスター / 詳細関係を表すには，ネストされた詳細セットを定義している変換ファイルを使用します。

クライアントがプロバイダの GetRecords メソッドを呼び出すと，プロバイダは自動的に詳細データセットをデータパケットのレコード内の DataSet 項目として含めます。クライアントがプロバイダの ApplyUpdates メソッドを呼び出すと，プロバイダは自動的に更新の適用を適切な順序で処理します。

リモートデータモジュールでのステート情報のサポート

IAppServer インターフェースは，クライアントデータセットがアプリケーションサーバー上のプロバイダと通信するために使用しますが，ほとんどはステートレスです。ステートレスなアプリケーションは，クライアントからの前回の呼び出しで起こったことを「記憶」していません。ステートレスのこの性質は，トランザクションデータモジュールにデータベース接続をプールする場合には便利です。なぜなら，アプリケーションサーバーとしては，レコードの最新性などのような持続的情報に関してデータベース接続を区別する必要がないからです。同様に，ステートレスのこの品質は，即時アクティブ化やオブジェクトプーリングを使用する場合のように，リモートデータモジュールのインスタンスを共有する場合には重要です。SOAP データモジュールはステートレスでなければなりません。

しかし，アプリケーションサーバーへの呼び出しの後でもステート情報を維持したい場合があります。たとえば，インクリメンタルフェッチを使ってデータを要求するときに，アプリケーションサーバー上のプロバイダは前回の呼び出し時から情報（現在のレコード）を「記憶」していなければなりません。

クライアントデータセットは，IAppServer インターフェースへの呼び出し（AS_ApplyUpdates, AS_Execute，AS_GetParams，AS_GetRecords，または AS_RowRequest）を実行する前後に，カスタムステート情報の送信と取り出しが可能なイベントを受信します。同様に，プロバイダは，クライアントによって生成されるこれらの呼び出しに応答する前後に，カスタムステート情報の送信と取り出しが可能なイベントを受信します。この機構を使えば，アプリケーションサーバーがステートレスであっても，クライアントアプリケーションとアプリケーションサーバーとの間で持続的なステート情報をやりとりすることができます。

たとえば，次のパラメータ付きの問い合わせを表すデータセットを考えてみましょう。

```
SELECT * from CUSTOMER WHERE CUST_NO > :MinVal ORDER BY CUST_NO
```

ステートレスなアプリケーションサーバーでインクリメンタルフェッチを可能にするには，次のような処理を行うことができます。

- プロバイダが一組のレコードをデータパケットにパッケージ化するときに，パケット内の最後のレコードの CUST_NO の値に注目します。

```
TRemoteDataModule1::DataSetProvider1GetData(TObject *Sender, TCustomClientDataSet *DataSet)
{
  DataSet->Last(); // 最後のレコードに移動する
  TComponent *pProvider = dynamic_cast<TComponent *>(Sender);
  pProvider->Tag = DataSet->FieldValues["CUST_NO"];
}
```

- プロバイダは，データパケットを送信した後，この最後の CUST_NO の値をクライアントに送信します。

```
TRemoteDataModule1::DataSetProvider1AfterGetRecords(TObject *Sender, OleVariant &OwnerData)
{
  TComponent *pProvider = dynamic_cast<TComponent *>(Sender);
  OwnerData = pProvider->Tag;
}
```

- クライアントでは，クライアントデータセットで CUST_NO のこの最後の値を保存します。

```
TDataModule1::ClientDataSet1AfterGetRecords(TObject *Sender, OleVariant &OwnerData)
{
  TComponent *pDS = dynamic_cast<TComponent *>(Sender);
  pDS->Tag = OwnerData;
}
```

- データパケットを取得する前に，クライアントは受信した CUST_NO の最後の値を送信します。

```
TDataModule1::ClientDataSet1BeforeGetRecords(TObject *Sender, OleVariant &OwnerData)
{
  TClientDataSet *pDS = dynamic_cast<TClientDataSet *>(Sender);
  if (!pDS->Active)
    return;
  OwnerData = pDS->Tag;
}
```

- 最後に，サーバー上では，プロバイダが問い合わせで最小値として送信された最後の CUST_NO を使用します。

```
TRemoteDataModule1::DataSetProvider1BeforeGetRecords(TObject *Sender, OleVariant &OwnerData)
{
  if (!VarIsEmpty(OwnerData))
  {
    TDataSetProvider *pProv = dynamic_cast<TDataSetProvider *>(Sender);
    TSQLDataSet *pDS = (dynamic_cast<TSQLDataSet *>(pProv->DataSet);
    pDS->Params->ParamValues["MinVal"] = OwnerData;
    pDS->Refresh(); // 問い合わせを強制再実行する
  }
}
```

複数のリモートデータモジュールを使う

アプリケーションサーバーを構造化し，複数のリモートデータモジュールを使用するようにするとよいでしょう。複数のリモートデータモジュールを使用することにより，コードを分割し，大規模なアプリケーションサーバーを各ユニットが相対的に自己包含型になっている複数のユニットに編成できます。

独立して機能するアプリケーションサーバーには複数のリモートデータモジュールをいつでも作成できますが，コンポーネントパレットの［DataSnap］ページの特殊な接続コンポーネントでは，接続をクライアントから他の「子」リモートデータモジュールにディスパッチするメイン「親」リモートデータモジュールを 1 つ持つモデルをサポートしています。このモジュールでは，COM ベースのアプリケーションサーバーを使用する必要があります。

親リモートデータモジュールを作成するには，その IAppServer インターフェースを拡張し，子リモートデータモジュールのインターフェースを公開するプロパティを追加する必要があります。つまり，子リモートデータモジュールごとに，値が子データモジュールの IAppServer インターフェースである親データモジュールのインターフェースに追加するということです。

親リモートデータモジュールのインターフェースの拡張についての詳細は，29-17 ページの「アプリケーションサーバーのインターフェースを拡張する」を参照してください。

ヒント また，子データモジュールごとにインターフェースを拡張し，親データモジュールのインターフェースや他の子データモジュールのインターフェースを公開することもできます。したがって，アプリケーションサーバー内のさまざまなデータモジュール同士が自由に通信し合うことができます。

子リモートデータモジュールを表すプロパティをメインリモートデータモジュールに追加しておくと，クライアントアプリケーションはアプリケーションサーバー上のそれぞれのリモートデータモジュールに対する接続を別々に構成する必要がありません。その代わりに，親リモートデータモジュールとの単一の接続を共有し，メッセージを「子」データモジュールにディスパッチします。それぞれのクライアントアプリケーションはすべてのリモートデータモジュールに対し同じ接続を使用するため，リモートデータモジュールは単一のデータベース接続を共有することにより，リソースを節約できます。子アプリケーションで単一の接続を共有する方法についての詳細は，29-29 ページの「複数のデータモジュールを使用するアプリケーションサーバーに接続する」を参照してください。

アプリケーションサーバーを登録する

クライアントアプリケーションがアプリケーションサーバーを特定して使用するには，まず，登録するか，インストールしておかなければなりません。

- アプリケーションサーバーは，通信プロトコルとして DCOM，HTTP，またはソケットを使用する場合，オートメーションサーバーとして機能し，ほかの COM サーバーと同様に登録が必要です。COM サーバーの登録についての詳細は，41-17 ページの「COM オブジェクトを登録する」を参照してください。
- トランザクションデータモジュールを使う場合，アプリケーションサーバーの登録は行いません。かわりに，MTS または COM+ にインストールします。トランザクションオブジェクトのインストールについての詳細は，44-27 ページの「トランザクションオブジェクトのインストール」を参照してください。
- アプリケーションサーバーで SOAP を使用するときは，アプリケーションは Web サービスアプリケーションでなければなりません。したがって，Web サーバーに登録して，入ってくる HTTP メッセージを受信するようにしなければなりません。さらに，アプリケーション内の呼び出し可能インターフェースを記述している WSDL ドキュメントを公開する必要があります。Web サービスアプリケーションの WSDL ドキュメントの書き出しについての詳細は，36-15 ページの「Web サービスアプリケーション用の WSDL ドキュメントの生成」を参照してください。

クライアントアプリケーションの作成

ほとんどの点に関して，多層クライアントアプリケーションの作成は，クライアントデータセットを使用してキャッシュアップデートを実行する 2 層クライアントの作成と似ています。主な違いは，多層クライアントではアプリケーションサーバーへの仲介を確立するための接続コンポーネントを使用するという点です。

多層クライアントアプリケーションを作成するには，まず新規プロジェクトを開始します。その後の手順は次のとおりです。

1. プロジェクトに新しいデータモジュールを追加します。
2. 接続コンポーネントをデータモジュールに入れます。追加する接続コンポーネントのタイプは，使用したい通信プロトコルによって決まります。詳しくは，29-4 ページの「クライアントアプリケーションの構造」を参照してください。
3. 接続コンポーネントのプロパティの設定で，接続先のアプリケーションサーバーを指定します。接続コンポーネントのセットアップについての詳細は，29-23 ページの「アプリケーションサーバーへの接続」を参照してください。
4. アプリケーションで必要であれば，その他の接続コンポーネントのプロパティを設定します。たとえば，接続コンポーネントが ObjectBroker プロパティをいくつかのサーバーの中から動的に選択することもできます。接続コンポーネントの使い方についての詳細は，29-27 ページの「サーバー接続の管理」を参照してください。
5. TClientDataSet コンポーネントをデータモジュールに必要なだけ入れ，各コンポーネントの RemoteServer プロパティを，手順 2 で配置した接続コンポーネントの名前に設定します。クライアントデータセットの詳細については，第 27 章「クライアントデータセットの使い方」を参照してください。
6. 各 TClientDataSet コンポーネントの ProviderName プロパティを設定します。設計時に接続コンポーネントをアプリケーションサーバーに接続する場合は，ProviderName プロパティのドロップダウンリストから使用可能なアプリケーションサーバーのプロバイダを選択できます。
7. ほかのデータベースアプリケーションの作成と同じように続けます。多層アプリケーションのクライアントで使用できる追加機能がいくつかあります。
   - アプリケーション側で，アプリケーションサーバーの直接呼び出しを実行できます。29-28 ページの「サーバーインターフェースの呼び出し」でこの方法について説明しています。
   - プロバイダコンポーネントとの対話をサポートするクライアントデータセットの特別な機能を使用することもできます。これらについては，27-23 ページの「クライアントデータセットでデータプロバイダを使用する」で説明しています。

アプリケーションサーバーへの接続

アプリケーションサーバーへの接続を確立して維持するために，クライアントアプリケーションは 1 つまたは複数の接続コンポーネントを使います。これらのコンポーネントは，コンポーネントパレットの［DataSnap］ページまたは［WebServices］ページにあります。

接続コンポーネントは，次の目的で使います。

- アプリケーションサーバーと通信するためのプロトコルを識別する。各接続コンポーネントは，異なる通信プロトコルを表します。使用可能なプロトコルの利点や制限については，29-9 ページの「接続プロトコルの選択」を参照してください。
- サーバーマシンの特定のしかたを示す。サーバーマシンの識別のしかたの詳細は，プロトコルによって異なります。
- サーバーマシン上のアプリケーションサーバーを識別する。

- SOAP を使わない場合，ServerName または ServerGUID プロパティを使ってサーバーを識別します。ServerName は，アプリケーションサーバー側のリモートデータモジュールを作成するときに指定するクラスの基本名です。サーバー側でこの値をどう指定するかについての詳細は，29-14 ページの「リモートデータモジュールをセットアップする」を参照してください。サーバーがクライアントマシン上に登録またはインストールされていたり，接続コンポーネントがサーバーマシンに接続されている場合は，設計時にオブジェクトインスペクタのドロップダウンリストから選択することによって ServerName プロパティを設定できます。ServerGUID は，リモートデータモジュールのインターフェースの GUID を指定します。この値は，タイプライブラリエディタを使って参照できます。

  SOAP を使用する場合，サーバーはサーバーマシンの位置を特定するための URL で識別されます。29-26 ページの「SOAP を使った接続の指定」の手順に従います。
- サーバー接続を管理する。接続コンポーネントは，接続の作成または切断，およびアプリケーションサーバーインターフェースの呼び出しに使用できます。

通常，アプリケーションサーバーはクライアントアプリケーションと別のマシンにありますが，サーバーがクライアントアプリケーションと同じマシンに常駐している場合でも（たとえば，多層アプリケーション全体の構築およびテスト中），接続コンポーネントを使って名前でアプリケーションサーバーを識別し，サーバーマシンを指定し，サーバーインターフェースを使用できます。

DCOM を使った接続の指定

DCOM を使ってアプリケーションサーバーと通信するときは，クライアントアプリケーションは，アプリケーションサーバーに接続するための TDCOMConnection コンポーネントを含みます。TDCOMConnection は，ComputerName プロパティを使ってサーバーのあるマシンを識別します。

ComputerName が空白であれば，DCOM 接続コンポーネントはアプリケーションサーバーがクライアントマシン上に常駐しているとみなし，アプリケーションサーバーがシステムレジストリエントリにあるとみなします。DCOM を使っている場合にクライアント上にアプリケーションサーバー用のシステムレジストリエントリを用意せず，しかもサーバーがクライアントと異なるマシンに常駐しているならば，ComputerName を指定しなければなりません。

メモ たとえシステムレジストリにアプリケーションサーバーのエントリがあっても，ComputerName を指定してこのエントリをオーバーライドできます。これは，開発，テスト，デバッグのときに特に便利です。

クライアントアプリケーションが選択できるサーバーが複数ある場合，ComputerName の値を指定するかわりに ObjectBroker プロパティを使えます。詳細については，29-27 ページの「接続ブローカ」を参照してください。

存在しないホストコンピュータまたはサーバーの名前を指定した場合，接続を開こうとすると DCOM 接続コンポーネントは例外を発生させます。

ソケットを使った接続の指定

マシンが TCP/IP アドレスを有する場合，ソケットを使ってアプリケーションサーバーとの接続を確立できます。この方法は，比較的多数のマシンに適用できるという利点を持ちますが，セキュリティ

プロトコルの利用については規定されません。ソケットを使う場合，アプリケーションサーバーに接続するために TSocketConnection コンポーネントを含めます。

TSocketConnection は，サーバーシステムの IP アドレスまたはホスト名と，サーバーマシン上で実行されているソケットディスパッチャプログラム（Scktsrver.exe）のポート番号を使ってサーバーマシンを識別します。IP アドレスとポート値についての詳細は，37-3 ページの「ソケットの記述」を参照してください。

TSocketConnection の 3 つのプロパティがこの情報を指定します。

- Address は，サーバーの IP アドレスを指定します。
- Host は，サーバーのホスト名を指定します。
- Port は，アプリケーションサーバー上のソケットディスパッチャプログラムのポート番号を指定します。

Address と Host は，互いに排他関係にあります。一方の値を設定すると，もう一方の値は設定解除されます。どちらを使うかについては，37-4 ページの「ホストの記述」を参照してください。

クライアントアプリケーションが選択できるサーバーが複数ある場合，Address または Host を指定する変わりに ObjectBroker プロパティを使用できます。詳細については，29-27 ページの「接続ブローカ」を参照してください。

デフォルトでは，Port の値は 211 です。これはアプリケーションサーバーに入ってくるメッセージを転送するソケットディスパッチャプログラムのデフォルトポート番号です。ソケットディスパッチャが異なるポートを使うように設定されているならば，Port プロパティもその値と一致するように設定します。

メモ ソケットディスパッチャのポートは，ポートの動作中に Borland Socket Server のトレーアイコンを右クリックして［プロパティ］を選択すれば設定できます。

ソケット接続はセキュリティプロトコルの使用を提供していませんが，ソケット接続をカスタマイズして独自の暗号化を追加することががができます。手順は次のとおりです。

1. IDataIntercept インターフェースをサポートする COM オブジェクトを作成します。これは，データの暗号化と暗号解除のためのインターフェースです。
2. 新しい COM サーバーをクライアントマシンに登録します。
3. ソケット接続コンポーネントの InterceptName または InterceptGUID プロパティの設定により，この COM オブジェクトを指定します。
4. 最後に，Borland Socket Server のトレーアイコンを右クリックして［プロパティ］を選択し，プロパティタブで［インターセプト名］または［インターセプト GUID］にインターセプタの ProgId または GUID を設定します。

この機構は，データの圧縮と解凍についても使用できます。

HTTP を使った接続の指定

TCP/IP アドレスを持つ任意のマシンから HTTP を使ってアプリケーションサーバーへの接続を確立することができます。ただし，ソケットとは異なり，HTTP では SSL セキュリティを使用することが

でき，ファイアウォールで保護されたサーバーとも通信できます。HTTP を使うときには，アプリケーションサーバーと接続するために TWebConnection コンポーネントを含めます。

Web 接続コンポーネントは，Web サーバーアプリケーション（httpsrvr.dll）への接続を確立し，これによってサーバーアプリケーションと通信します。TWebConnection は，Uniform Resource Locator（URL）を使って httpsrvr.dll の場所を見つけます。URL は，プロトコル（http，または，SSL セキュリティを使用している場合は https），Web サーバー httpsrvr.dll を実行するマシンのホスト名，および Web サーバーアプリケーション（httpsrvr.dll）へのパスを示します。この値を，URL プロパティを使って指定します。

メモ TWebConnection 接続を使うときは，wininet.dll がクライアントマシンにインストールされていなければなりません。IE3 またはそれ以降のバージョンがインストールされていれば，Windows システムディレクトリ内に wininet.dll があります。

Web サーバーに認証が必要な場合や，使用しているプロクシーサーバーに認証が必要な場合は，接続コンポーネントがログオンできるように UserName プロパティと Password プロパティに値を設定しなければなりません。

クライアントアプリケーションが選択できるサーバーが複数ある場合，URL の値を指定するかわりに ObjectBroker プロパティを使えます。詳細については，29-27 ページの「接続ブローカ」を参照してください。

SOAP を使った接続の指定

SOAP アプリケーションサーバーへの接続を確立するには，TSoapConnection コンポーネントを使用します。TSoapConnection は TWebConnection に非常によく似ています。これもまた，HTTP をトランスポート プロトコルとして使用しているからです。そのため，TCP/IP アドレスを持つ任意のマシンから TSoapConnection を使用することができ，SSL セキュリティを利用して，ファイアウォールで保護されたサーバーと通信できます。

SOAP 接続コンポーネントは，IAppServerSOAP インターフェースまたは IAppServer インターフェースを実装する Web サービスプロバイダへの接続を確立します（UseSOAPAdapter プロパティは，サーバーがサポートすると想定されるインターフェースを指定します）。サーバーが IAppServerSOAP インターフェースを実装すると，TSoapConnection はこのインターフェースをクライアントデータセット用の IAppServer インターフェースに変換します。TSoapConnection は，Uniform Resource Locator（URL）を使って Web サーバーアプリケーションの場所を見つけます。URL は，プロトコル（http，または，SSL セキュリティを使用している場合は https），Web サーバーを実行するマシンのホスト名，Web サービスアプリケーションの名前，および アプリケーションサーバー上の THTTPSoapDispatcher のパス名と一致するパスを示します。この値を，URL プロパティを使って指定します。

メモ TSoapConnection 接続を使うときは，wininet.dll がクライアントマシンにインストールされていなければなりません。IE3 またはそれ以降のバージョンがインストールされていれば，Windows システムディレクトリ内に wininet.dll があります。

Web サーバーに認証が必要な場合や，使用しているプロクシーサーバーに認証が必要な場合は，接続コンポーネントがログオンできるように UserName プロパティと Password プロパティに値を設定しなければなりません。

接続ブローカ

クライアントアプリケーションが選択できる COM ベースのサーバーが複数ある場合，オブジェクトブローカを使って使用可能なサーバーシステムを特定することができます。オブジェクトブローカは，接続コンポーネントが選択可能なサーバーのリストを保守しています。接続コンポーネントは，アプリケーションサーバーに接続する必要のあるとき，オブジェクトブローカにコンピュータ名（つまり IP アドレス，ホスト名，または URL）を要求します。ブローカはコンピュータ名を指定し，接続コンポーネントが接続を確立します。指定した名前が機能しなかったら（サーバーがダウンしているなど），ブローカは接続が確立できるまで別の名前を指定していきます。

接続コンポーネントは，ブローカから指定される名前で接続を確立したら，その名前を適切なプロパティ（ComputerName，Address，Host，RemoteHost，または URL）の値として保存します。接続コンポーネントがその後接続を閉じてからまた接続を開く必要が生じたら，このプロパティ値を使ってみて，接続に失敗した場合だけ新しい名前をブローカに要求します。

オブジェクトブローカを使用するには，接続コンポーネントの ObjectBroker プロパティを指定します。ObjectBroker プロパティが設定されているとき，接続コンポーネントは ComputerName，Address，Host，RemoteHost，または URL の値をディスクに保存しません。

サーバー接続の管理

接続コンポーネントの主な目的は，アプリケーションサーバーの位置を特定し，それと接続することです。接続コンポーネントはサーバー接続を管理するので，接続コンポーネントを使ってアプリケーションサーバーのインターフェースのメソッドも呼び出せます。

サーバーに接続する

アプリケーションサーバーの位置を特定してそれに接続するには，まず，接続コンポーネントのプロパティの設定によってアプリケーションサーバーを識別します。この手順については，29-23 ページの「アプリケーションサーバーへの接続」で述べます。接続を開く前に，接続コンポーネントを使用してアプリケーションサーバーと通信するどのクライアントデータセットも，そのことを示すために自分の RemoteServer プロパティにその接続コンポーネントを指定しなければなりません。

接続は，クライアントデータセットがアプリケーションサーバーにアクセスしようとするときに自動的に開きます。たとえば，クライアントデータセットの Active プロパティを **true** に設定すると，RemoteServer プロパティが設定されていれば接続が開きます。

どのクライアントデータセットも接続コンポーネントにリンクしていない場合，接続コンポーネントの Connected プロパティを **true** に設定することによって接続を開くことができます。

接続コンポーネントは，アプリケーションサーバーとの接続を確立する前に，BeforeConnect イベントを発生させます。BeforeConnect ハンドラ内にコードを書くことによって，接続に先だって特別なアクションを実行できます。接続を確立したら，接続コンポーネントは AfterConnect イベントを発生させるので，そこで何らかのアクションを実行できます。

サーバー接続を切断または変更する

接続コンポーネントがアプリケーションサーバーへの接続を切断するのは次の場合です。

- Connected プロパティを **false** に設定する
- 接続コンポーネントを解放する。ユーザーがクライアントアプリケーションを閉じると，接続オブジェクトは自動的に解放される
- アプリケーションサーバーを識別するどれかのプロパティを変更する（ServerName，ServerGUID，ComputerName など）これらのプロパティを変更すると，実行時に，使用可能なアプリケーションサーバー間で切り替えることができる。接続コンポーネントは現在の接続を切断し，新規接続を確立する

メモ 利用可能なアプリケーションサーバー間の切り替えに 1 つの接続コンポーネントを使う代わりに，クライアントアプリケーションは，それぞれ異なるアプリケーションサーバーに接続された複数の接続コンポーネントを使うこともできます。

BeforeDisconnect イベントハンドラが用意されている場合，接続コンポーネントは接続を切断する前に自動的にその BeforeDisconnect イベントハンドラを呼び出します。切断の前に特別な処理を実行するには，BeforeDisconnect ハンドラを記述します。同様に，接続の切断後には AfterDisconnect イベントハンドラが呼び出されます。切断後に特別な処理を実行したい場合は，AfterDisconnect ハンドラを記述します。

サーバーインターフェースの呼び出し

アプリケーションは，IAppServer インターフェースを直接呼び出す必要はありません。クライアントデータセットのプロパティとメソッドを使うときに自動的に適切な呼び出しが行われるからです。ただし，SOAP を使用しない場合，IAppServer インターフェースと直接やりとりする必要がなければ，アプリケーションサーバーのインターフェースに独自の拡張機能を追加することもできます。アプリケーションサーバーのインターフェースを拡張する場合，接続コンポーネントによって作成された接続を使ってこれらの拡張機能を呼び出す方法が必要です。これを行うには，接続コンポーネントの AppServer プロパティを使います。アプリケーションサーバーインターフェースの拡張についての詳細は，29-17 ページの「アプリケーションサーバーのインターフェースを拡張する」を参照してください。

AppServer は Variant 型で，アプリケーションサーバーのインターフェースを表します。このインターフェースを呼び出すには，この Variant 型からディスパッチインターフェースを取得する必要があります。ディスパッチインターフェースの名前は，リモートデータモジュールを作成したときに作成されたインターフェースと同じ名前で，末尾に Disp が付加されます。したがって，リモートデータモジュールの名前が MyAppServer の場合，アプリケーションサーバーのインターフェースを呼び出すコードは，AppServer を使用して次のようになります。

```
IDispatch* disp = (IDispatch*)(MyConnection->AppServer)
IMyAppServerDisp TempInterface( (IMyAppServer*)disp);
TempInterface.SpecialMethod(x,y);
```

メモ タイプライブラリエディタが生成する _TLB.h ファイルの中で，ディスパッチインターフェースが宣言されます。

SOAP を使用している場合は，AppServer プロパティを使用できません。その代わりに，リモートインターフェースオブジェクト（THTTPRio）を使用して，アーリーバインディング呼び出しを実行する必要があります。すべてのアーリーバインディング呼び出しの場合と同様，クライアントアプリ

ケーションはアプリケーションサーバーのインターフェース宣言をコンパイル時に知っていなければなりません。呼び出したいインターフェースを記述している WSDL ドキュメントの参照によって，クライアント アプリケーションにインターフェースを追加できます。ただし，SOAP サーバーの場合，このインターフェースは SOAP データモジュールのインターフェースとはまったく別のものです。インターフェースを記述している WSDL ドキュメントの読み込みについての詳細は，36-16 ページの「WSDL ドキュメントのインポート」を参照してください。

メモ サーバーインターフェースを宣言しているユニットも，呼び出しレジストリに登録する必要があります。呼び出し可能インターフェースの登録方法についての詳細は，36-2 ページの「起動可能インターフェースについて」を参照してください。

インターフェースを宣言し登録しているユニットを生成する WSDL ドキュメントを読み込んだ後，目的のインターフェース用の THTTPRio のインスタンスを作成します。

```
THTTPRio *X = new THTTPRio(NULL);
```

次に，接続コンポーネントで使用する URL をリモートインターフェースオブジェクトに割り当て，呼び出すインターフェースの名前を追加します。

```
X->URL = SoapConnection1.URL + "IMyInterface";
```

これで，QueryInterface メソッドを使用してサーバーのメソッドを呼び出すインターフェースを取得できます。

```
InterfaceVariable = X->QueryInterface(IMyInterfaceIntf);
if (InterfaceVariable)
{
  InterfaceVariable->SpecialMethod(a,b);
}
```

ただし，QueryInterface の呼び出しには，引数として呼び出し可能なインターフェース自体でなく，呼び出し可能なインターフェースの DelphiInterface ラッパーが必要です。

複数のデータモジュールを使用するアプリケーションサーバーに接続する

29-21 ページの「複数のリモートデータモジュールを使う」で説明しているように，COM ベースのアプリケーションサーバーでメイン「親」リモートデータモジュールといくつかの子リモートデータモジュールを使用する場合，アプリケーションサーバー上のすべてのリモートデータモジュールについて別々の接続コンポーネントが必要です。それぞれの接続コンポーネントは，単一のリモートデータモジュールへの接続を表します。

クライアントアプリケーションで，アプリケーションサーバー上のそれぞれのリモートデータモジュールに対し独立の接続を行うことができますが，すべての接続コンポーネントが共有するアプリケーションサーバーに単一の接続を使用するほうが効率的です。つまり，アプリケーションサーバーに「メイン」リモートデータモジュールに接続する単一接続コンポーネントを追加し，その後，それぞれの「子」リモートデータモジュールについて，メインリモートデータモジュールへの接続を共有するコンポーネントを追加します。

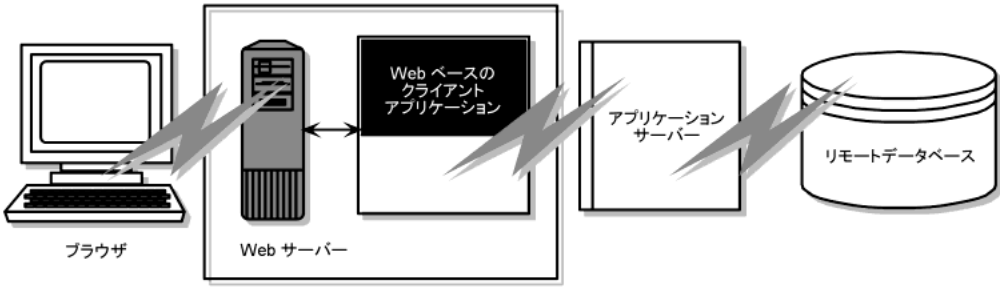
1. メインリモートデータモジュールに接続するには，29-23 ページの「アプリケーションサーバーへの接続」で説明されているように，接続コンポーネントを追加して設定します。唯一の制限は，SOAP 接続を使用できないという点です。
2. 子リモートデータモジュールごとに，TSharedConnection コンポーネントを使用します。
   - ParentConnection プロパティを，手順 1 で追加した接続コンポーネントに設定します。TSharedConnection コンポーネントは，このメインの接続で確立した接続を共有します。
   - ChildName プロパティを，目的の子リモートデータモジュールのインターフェースを公開するメインリモートデータモジュールのインターフェース上のプロパティの名前に設定します。

手順 2 で 配置した TSharedConnection コンポーネントをクライアントデータセットの RemoteServer プロパティの値として割り当てると，子リモートデータモジュールへのまったく独立した接続を使用しているかのように動作します。ただし，TSharedConnection コンポーネントでは，手順 1 で配置したコンポーネントによって確立された接続を使用します。

Web ベースのクライアントアプリケーションの作成

多層データベースアプリケーション用に Web ベースのクライアントを作成したい場合，クライアント層を特別の Web アプリケーションで置換しなければなりません。特別の Web アプリケーションは，アプリケーションサーバーに対してはクライアントの役割をし，同時に，同じマシン上の Web サーバーと一緒にインストールされた Web サーバーアプリケーションとしても機能するものです。このアーキテクチャを図 29.1 に示します。

図 29.1　Web ベースの多層データベースアプリケーション



Web アプリケーションの構築には，2 つのアプローチがあります。

- 多層データベースアーキテクチャを ActiveX 形式と組み合わせて，クライアントアプリケーションを ActiveX コントロールとして配布する。この方法では，ActiveX をサポートしているブラウザならどれでも，クライアントアプリケーションをインプロセスサーバーとして実行できる
- XML データパケットを使って InternetExpress アプリケーションを構築する。この方法では，Java スクリプトをサポートするブラウザなら HTML ページを介してクライアントアプリケーションと対話することができる

これら 2 つのアプローチは大きく異なっています。どちらを選ぶかは，次の事項を考慮して決めます。

- それぞれのアプローチは，異なる技術に依存しています（一方は ActiveX，もう一方は Java スクリプトと XML）。エンドユーザーがどちらのシステムを使用するかを考慮します。1 つ目のアプローチでは，ブラウザが ActiveX をサポートしている必要があります（つまり，クライアントは Windows プラットフォームに限定されます）。2 つ目のアプローチでは，ブラウザが Java スクリプトと DHTML 機能をサポートしている必要があります。これは Netscape 4 と Internet Explorer 4 で導入されています。
- ActiveX コントロールは，ブラウザにダウンロードされてインプロセスサーバーとして動作しなければなりません。その結果，ActiveX アプローチを使うクライアントは，HTML ベースのアプリケーションでのクライアントよりも大量のメモリを必要とします。
- InternetExpress のアプローチは，ほかの HTML ページと統合できます。ActiveX クライアントは，別個のウィンドウで実行されなければなりません。
- InternetExpress のアプローチでは，標準の HTTP を使うので，ActiveX アプリケーションで発生するようなファイアウォール問題を回避することができます。
- ActiveX のアプローチでは，アプリケーションのプログラミングでより大きな自由度が得られます。Java スクリプトライブラリの機能に制限されることはありません。ActiveX アプローチで使用されるクライアントデータベースは，InternetExpress アプローチで使用される XML ブローカよりも多くの機能を提供します（フィルタ，範囲，集約，オプションパラメータ，BLOB やネストされた詳細の遅延取り出しなど）。

注意 Web クライアントアプリケーションは，それを表示するブラウザによって，表示内容や動作が異なることがあります。アプリケーションのテストには，エンドユーザーが使用すると予想されるブラウザを使ってください。

ActiveX コントロールとしてクライアントアプリケーションを配布する

多層データベースアーキテクチャは，ActiveX 機能と組み合わせて ActiveX コントロールとしてクライアントアプリケーションを配布できます。

クライアントアプリケーションを ActiveX コントロールとして配布する場合，アプリケーションサーバーを，ほかの多層アプリケーションと同じように作成します。アプリケーションサーバーの作成のしかたについては，29-12 ページの「アプリケーションサーバーの作成」を参照してください。

クライアントアプリケーションを作成するときには，普通のフォームでなくアクティブフォームを使わなければなりません。詳しくは，「クライアントアプリケーションのアクティブフォームの作成」を参照してください。

クライアントアプリケーションの構築と配布が完了したら，別のマシンの ActiveX 対応 Web ブラウザから，クライアントアプリケーションにアクセスできます。Web ブラウザがクライアントアプリケーションを正常に起動するには，同じマシン上で Web サーバーが動作している必要があります。

クライアントアプリケーションとアプリケーションサーバーとの間の通信に DCOM を利用する場合，Web ブラウザが動作するマシンに，DCOM を使って作業できる環境が必要になります。Windows 95 マシンの場合は，Microsoft が提供する DCOM 95 をインストールしなければなりません。

クライアントアプリケーションのアクティブフォームの作成

1. クライアントアプリケーションを ActiveX コントロールとして配布するために，クライアントアプリケーションと同じシステム上で動作する Web サーバーを用意しなければなりません。Microsoft の Personal Web サーバーなどの既製のサーバーを使用してもよいし，第 37 章「ソケットの操作」で述べられているようにソケットコンポーネントを使って自分で Web サーバーを書くこともできます。
2. 29-22 ページの「クライアントアプリケーションの作成」に記載されている手順に従って，クライアントアプリケーションを作成します。ただし，プロジェクトを開始する際は，通常のクライアントプロジェクトと違って，［ファイル｜新規作成｜その他｜ ActiveX ｜ Active フォーム］を選択します。
3. クライアントアプリケーションがデータモジュールを使う場合は，アクティブフォームの初期化に呼び出しを追加して，データモジュールを明示的に作成します。
4. クライアントアプリケーションが完成したら，プロジェクトをコンパイルし，［プロジェクト｜ Web 配布オプション］を選択します。［Web 配布オプション］ダイアログボックスでは，以下のことを行わなければなりません。
   1. ［プロジェクト］ページで，配布先ディレクトリ，配布先ディレクトリの URL，および HTML ディレクトリを指定します。通常，配布先ディレクトリと HTML ディレクトリは，Web サーバーのプロジェクトディレクトリと同一です。配布先の URL は，サーバーマシンの名前であるのが一般的です。
   2. ［追加ファイル］ページで，クライアントアプリケーションに midas.dll を含めます。
5. 最後に，［プロジェクト｜ Web 配布］を選択して，アクティブフォームとしてクライアントアプリケーションを配布します。

アクティブフォームを実行できるすべての Web ブラウザから，クライアントアプリケーションを配布する際に作成した .HTM ファイルを指定して，クライアントアプリケーションを実行できます。この .HTM ファイルは，クライアントアプリケーションを作成したプロジェクトと同じ名前を持ち，ターゲットディレクトリとして指定されたディレクトリに格納されています。

InternetExpress 使用による Web アプリケーションの構築

クライアントアプリケーションは，OleVariant ではなく XML でコーディングされたデータパケットをアプリケーションサーバーが供給するように要求することができます。XML でコーディングされたデータパケット，データベース関数の特別な Java スクリプトライブラリ，および Web サーバーアプリケーションサポートを組み合わせることにより，Java スクリプトをサポートする Web ブラウザでアクセスできる軽量クライアントアプリケーションを作成することができます。この機能の組み合わせを InternetExpress と呼びます。

InternetExpress アプリケーションを構築する前に，Web サーバーアプリケーションのアーキテクチャについて理解しておく必要があります。これについては，第 32 章「インターネットサーバーアプリケーションの作成」で説明しています。

InternetExpress アプリケーションは，基本 Web アプリケーションアーキテクチャを拡張したもので，アプリケーションサーバーのクライアントとして動作します。InternetExpress アプリケーションは，

HTML，XML，および Java スクリプトを混在させた HTML ページを生成します。HTML は，ユーザーがブラウザで表示したときのページのレイアウトと外観を制御します。XML は，データベース情報を表すデータパケットとデルタパケットをエンコードします。Java スクリプトは，クライアントマシン上で，HTML コントロールによるこれらの XML データパケット内のデータの解釈および操作を可能にします。

InternetExpress アプリケーションが DCOM を使ってアプリケーションサーバーに接続する場合は，さらに追加の手順によって，アプリケーションサーバーがクライアントにアクセス権と起動権を許可するようにしなければなりません。詳細は，29-35 ページの「アプリケーションサーバーに対するアクセス権と起動権を許可する」を参照してください。

ヒント InternetExpress アプリケーションを作成し，アプリケーションサーバーがない場合でも，「ライブ」データを扱う Web サーバーアプリケーションを提供することができます。これを行うには，Web モジュールにプロバイダとそのデータセットを追加するだけです。

InternetExpress アプリケーションの構築

次の手順は，InternetExpress を使用して Web アプリケーションを構築する 1 つの方法です。その結果が，Java スクリプト対応 Web ブラウザを介してアプリケーションサーバーからのデータをユーザー側で対話操作できる HTML ページを作成するアプリケーションです。さらに，InternetExpress ページプロデューサ（TInetXPageProducer）を使用することにより Site Express アーキテクチャを採用する InternetExpress アプリケーションを構築することもできます。

1. ［ファイル｜新規作成｜その他］を選択して［新規作成］ダイアログボックスを表示し，［新規作成］ページで［Web サーバーアプリケーション］を選択します。この手順については，33-1 ページの「WebBroker による Web サーバーアプリケーションの作成」で述べます。
2. コンポーネントパレットの［DataSnap］ページで，新規 Web サーバーアプリケーションを作成するときに表示される Web モジュールに，接続コンポーネントを追加します。追加する接続コンポーネントのタイプは，使用したい通信プロトコルによって決まります。詳しくは，29-9 ページの「接続プロトコルの選択」を参照してください。
3. 接続コンポーネントのプロパティの設定で，接続先のアプリケーションサーバーを指定します。接続コンポーネントのセットアップについての詳細は，29-23 ページの「アプリケーションサーバーへの接続」を参照してください。
4. クライアントデータセットのかわりに，コンポーネントパレットの［InternetExpress］ページにある XML ブローカを，Web モジュールに追加します。TClientDataSet と同様に，TXMLBroker は，アプリケーションサーバー上のプロバイダのデータを表し，IAppServer インターフェースを介してアプリケーションサーバーと対話します。ただし，クライアントデータセットとは異なり，XML ブローカは，データパケットを OleVariant としてではなく XML として要求し，データコントロールではなく InternetExpress コンポーネントと対話します。
5. XML ブローカの RemoteServer プロパティに，手順 2 で追加した接続コンポーネントを指定します。ProviderName プロパティには，データを提供して更新を適用するアプリケーションサーバー上のプロバイダを指定します。XML ブローカの設定については，29-35 ページの「XML ブローカの使い方」を参照してください。

6. ユーザーのブラウザに表示される個々のページについて，InternetExpress ページプロデューサ（TInetXPageProducer）を Web モジュールに追加します。各ページプロデューサの IncludePathURL プロパティに，生成された HTML コントロールにデータ管理機能を追加するための Java スクリプトライブラリがどこにあるかを指定します。
7. Web ページを右クリックしてから［アクションの設定］を選択し，アクションエディタを表示します。ブラウザから処理したい各メッセージに対してアクション項目を追加します。Producer プロパティを設定するか，OnAction イベントハンドラにコードを書いて，手順 6 で追加したページプロデューサをこれらのアクションと関連付けます。アクションエディタでのアクション項目の追加のしかたについては，33-5 ページの「ディスパッチャへのアクションの追加」を参照してください。
8. 各 Web ページをダブルクリックして Web ページエディタを開きます（このエディタは，オブジェクトインスペクタ内で WebPageItems プロパティの横の省略記号ボタンをクリックしても表示できます）。このエディタで，ユーザーのブラウザに表示されるページを設計するために Web 項目を追加することができます。InternetExpress アプリケーション用の Web ページの設計については，29-38 ページの「InternetExpress ページプロデューサの使用による Web ページの作成」を参照してください。
9. Web アプリケーションを構築します。このアプリケーションを Web サーバーにインストールすると，ブラウザは，URL のスクリプト名部分にそのアプリケーションの名前を，そしてパス情報部分に Web ページコンポーネントの名前を指定することによって，アプリケーションを呼び出せます。

Java スクリプトライブラリの使い方

InternetExpress コンポーネントによって生成された HTML ページと，それらのページに含まれる Web 項目は，Source \ Webmidas ディレクトリに入っている数個の Java スクリプトライブラリを使用します。

表 29.3　Java スクリプトライブラリ

ライブラリ	説明
xmldom.js	このライブラリは，Java スクリプトで書かれた，DOM 互換の XML パーサー。これによって，XML をサポートしていないパーサーが XML データパケットを使用することができる。ただし，IE5 以降でサポートされる XML Islands のサポートは含まれない
xmldb.js	このライブラリは，XML データパケットと XML デルタパケットを管理するデータアクセスクラスを定義する
xmldisp.js	このライブラリは，xmldb 内のデータアクセスクラスを HTML ページ内の HTML コントロールに関連付けるクラスを定義する
xmlerrdisp.js	このライブラリは，更新エラーを調停するときに使用できるクラスを定義する。組み込みの InternetExpress コンポーネントがこのクラスを使うことはないが，調停プロデューサを記述するときに便利である
xmlshow.js	このライブラリには，形式設定された XML データパケットおよび XML デルタパケットを表示する関数が含まれる。InternetExpress コンポーネントがこのライブラリを使うことはないが，デバッグのときに便利である

これらのライブラリをインストールしたら，すべての InternetExpress ページプロデューサの IncludePathURL プロパティに，ライブラリのある場所を指定しなければなりません。

Web 項目を使って Web ページを生成するかわりに，これらのライブラリ内で提供される Java スクリプトクラスを使って自分の HTML ページを記述することができます。ただし，これらのクラスには

最低限のエラーチェック機能しかないので（生成される Web ページのサイズを抑えるため），自分の書くコードに不正な点がないよう注意しなければなりません。

アプリケーションサーバーに対するアクセス権と起動権を許可する

InternetExpress アプリケーションからのリクエストは，アプリケーションサーバーには，IUSR_*computername* という名前のゲストアカウントから送信されているように見えます（ここで，*computername* は Web アプリケーションを実行しているシステムの名前）。デフォルトでは，このアカウントには，アプリケーションサーバーに対してアクセス権も起動権もありません。これらの許可なしに Web アプリケーションを使用しようとすると，要求されたページを Web ブラウザがロードしようとしてタイムアウトになり，EOLE_ACCESS_ERROR エラーが発生します。

メモ アプリケーションサーバーはこのゲストアカウントのもとで動作しているので，ほかのアカウントからはシャットダウンできません。

Web アプリケーションに対するアクセス権と起動権を許可するには，DCOMCnfg.exe を実行します。このファイルは，アプリケーションサーバーを実行するマシンの System32 ディレクトリにあります。以下に，アプリケーションサーバーの設定方法を述べます。

1. DCOMCnfg を実行して，Applications ページのアプリケーション一覧からアプリケーションサーバーを選択します。
2. ［プロパティ］ボタンをクリックします。ダイアログが変わったら，［セキュリティ］ページを選択します。
3. ［独自のアクセス権を使う］を選択し，［編集］ボタンを押します。IUSR_*computername* という名前を，アクセス権を持つアカウントの一覧に追加します。ここで，*computername* は，Web アプリケーションを実行するマシンの名前です。
4. ［独自の起動アクセス権を使う］を選択してから［編集］ボタンを押します。この一覧にも IUSR_*computername* を追加します。
5. ［OK］ボタンをクリックします。

XML ブローカの使い方

XML ブローカには 2 つの主な機能があります。

- XML データパケットをアプリケーションサーバーから取り出し，それを，InternetExpress アプリケーションのために HTML を生成する Web 項目から使用できるようにする
- ブラウザからの更新を XML デルタパケットの形で受け取り，それをアプリケーションサーバーに適用する。

XML データパケットの取り出し

XML ブローカが，HTML ページを生成するコンポーネントに XML データパケットを供給するには，その前にアプリケーションサーバーからそのデータパケットを取り出しておかなければなりません。これを行うには，XML ブローカは，アプリケーションサーバーの IAppServer インターフェースを使います。このインターフェースは，接続コンポーネントから取得します。

メモ SOAP を使っても，アプリケーションサーバーが IAppServerSOAP をサポートする場合は，接続コンポーネントが 2 つのインターフェースの間のアダプタとして機能するので，XML ブローカは IAppServer を使用します。

XML プロデューサが IAppServer インターフェースを使用できるように，次のプロパティを設定します。

- RemoteServer プロパティに，アプリケーションサーバーへの接続を確立して IAppServer インターフェースを取得する接続コンポーネントを設定します。設計時には，オブジェクトインスペクタのドロップダウンリストでこの値を選択できます。
- ProviderName プロパティに，XML データパケットを使用したいデータセットを表す，アプリケーションサーバー上のプロバイダコンポーネントの名前を設定します。このプロバイダは，XML データパケットを提供すると共に，XML デルタパケットからの更新を適用します。設計時には，RemoteServer プロパティが設定されていて接続コンポーネントにアクティブな接続が存在すれば，使用可能なプロバイダの一覧がオブジェクトインスペクタに表示されます（DCOM 接続を使用する場合，アプリケーションサーバーもクライアントマシンに登録されていなければなりません）。

データパケットに何を含めたいか指定するために，2 つのプロパティを使用できます。

- データパケットに追加されるレコード数を制限するには，MaxRecords プロパティを設定します。大きなデータセットでは特に，MaxRecords プロパティを設定することが重要です。なぜなら，InternetExpress アプリケーションは，データパケット全体をクライアント Web ブラウザに送信するからです。データパケットが大きすぎると，ダウンロード時間が極端に長くなることがあります。
- アプリケーションサーバー上のプロバイダが問い合わせかストアドプロシージャを表している場合は，XML データパケットを取得する前にパラメータ値を指定しなければなりません。これらのパラメータ値は，Params プロパティを使って指定します。

InternetExpress アプリケーション用の HTML および Java スクリプトを生成するコンポーネントは，XMLBroker プロパティが設定されていれば，自動的に XML ブローカの XML データパケットを使用します。コード内で直接 XML データパケットを取得するには，RequestRecords メソッドを使います。

メモ XML ブローカは，データパケットを別のコンポーネントに供給するとき（または，RequestRecords を呼び出すとき），OnRequestRecords イベントを受け取ります。このイベントを使って，アプリケーションサーバーからのデータパケットのかわりに，ユーザーが指定した XML 文字列を供給することができます。たとえば，GetXMLRecords を使ってアプリケーションサーバーから XML データパケットを取り出し，それを編集してから Web ページに供給することができます。

XML デルタパケットからの更新の適用

XML ブローカを Web モジュール（または，TWebDispatcher を含むデータモジュール）に追加すると , ブローカは自動的に自分自身を自動ディスパッチオブジェクトとして Web ディスパッチャに登録します。このことは，ほかのコンポーネントと異なり，XML ブローカコンポーネント用にアクション項目を作成しなくても XML ブローカコンポーネントが Web ブラウザからの更新メッセージに応答できることを意味します。これらのメッセージには，アプリケーションサーバーに適用されるべき XML デルタパケットが含まれます。これらは一般に，Web クライアントアプリケーションによって作成される HTML ページの 1 つに作成するボタンから発生します。

ディスパッチャが XML ブローカに対するメッセージを認識できるようにするためには，それらのメッセージについて WebDispatch プロパティを使って記述しておかなければなりません。PathInfo プロパティに，XML ブローカに対するメッセージの送り先 URL のパス部分を設定します。MethodType に，その URL（通常 mtPost）宛ての更新メッセージのメソッドヘッダの値を設定します。指定したパスを持つすべてのメッセージに応答したい場合は，MethodType に mtAny を設定します。XML ブローカが更新メッセージに直接は応答しないようにしたい場合（たとえば，アクション項目を使って明示的に処理させたい場合）は，Enabled プロパティを **false** に設定します。Web ブラウザからのメッセージをどのコンポーネントに処理させるかについて Web ディスパッチャがどう決定するかの詳細は，33-5 ページの「リクエストメッセージのディスパッチ」を参照してください。

ディスパッチャが更新メッセージを XML ブローカに渡すと，XML ブローカはその更新をアプリケーションサーバーに渡し，もしも更新エラーがあればすべての更新エラーを記述する XML デルタパケットを受け取ります。最終的に，XML ブローカはブラウザにレスポンスメッセージを送信し，XML デルタパケットが生成された同じページをブラウザに再び表示させるか，または新しいコンテンツを送信します。

この更新プロセスの始めから終わりまでの間，カスタム処理を挿入できるイベントが多数あります。

1. ディスパッチャが最初に更新メッセージを XML ブローカに渡すとき，BeforeDispatch イベントを受け取ります。このとき，リクエストを前処理したり，リクエストを完全に処理したりさえもできます。このイベントによって，XML ブローカが更新メッセージ以外のメッセージを処理することが可能になります。
2. BeforeDispatch イベントハンドラがメッセージを処理しなければ，XML ブローカは OnRequestUpdate イベントを受け取ります。ここで，デフォルトの処理ではなくユーザーによる更新の適用を行うことができます。
3. OnRequestUpdate イベントハンドラがリクエストを処理しなければ，XML ブローカは，更新を適用して，更新エラーがあればそれを含むデルタパケットを受信します。
4. 更新エラーがなければ，XML ブローカは OnGetResponse イベントを受信します。このイベントを利用して，更新の適用に成功した旨のレスポンスメッセージを作成したり，更新されたデータをブラウザに送信することができます。OnGetResponse イベントハンドラが応答を完了しない（Handled パラメータを **true** に設定しない）場合，XML ブローカは，ブラウザにデルタパケットが生成されたドキュメントを再表示させるレスポンスを送信します。
5. 更新エラーがあった場合は，XML ブローカはかわりに OnGetErrorResponse イベントを受け取ります。このイベントを使って，更新エラーの解決を試行したり，エラーがあったことをエンドユーザーに伝える Web ページを生成したりできます。OnGetErrorResponse イベントハンドラがレスポンスを完了しない（Handled パラメータを **true** に設定しない）場合は，XML ブローカは，レスポンスメッセージのコンテンツを生成する ReconcileProducer という特別なコンテンツプロデューサを呼び出します。
6. 最後に，XML ブローカは，AfterDispatch イベントを受け取ります。このイベントで，Web ブラウザにレスポンスを送り返す前の最終アクションを実行できます。

InternetExpress ページプロデューサの使用による Web ページの作成

個々の InternetExpress ページプロデューサは，アプリケーションクライアントのブラウザに表示される 1 つの HTML ドキュメントを生成します。アプリケーションにいくつかの別個の Web ドキュメントが含まれている場合は，各ドキュメントに 1 つずつページプロデューサを使用します。

InternetExpress ページプロデューサ（TInetXPageProducer）は，特別のページプロデューサコンポーネントです。ほかのページプロデューサと同様に，アクション項目の Producer プロパティに割り当てたり，OnAction イベントハンドラから明示的に呼び出したりできます。コンテンツプロデューサをアクション項目と一緒に利用する方法についての詳細は，33-8 ページの「アクション項目によるリクエストメッセージへのレスポンス」を参照してください。ページプロデューサについての詳細は，33-14 ページの「ページプロデューサコンポーネントの使い方」を参照してください。

InternetExpress ページプロデューサには，HTMLDoc プロパティの値としてデフォルトのテンプレートがあります。このテンプレートには，InternetExpress が HTML ドキュメントを組み立てる（埋め込み Java スクリプトと XML を使って）ために使用する HTML 透過のタグ一式が入っています。InternetExpress がすべての HTML 透過のタグを翻訳してドキュメントを組み立てるには，そのページに埋め込まれている Java スクリプトが使う Java スクリプトライブラリの場所を指定しておかなければなりません。この場所は，IncludePathURL プロパティに指定します。

Web ページの各部を生成するそれぞれのコンポーネントは，Web ページエディタを使って指定できます。Web ページエディタを表示するには，Web ページコンポーネントをダブルクリックするか，オブジェクトインスペクタで WebPageItems プロパティの横の省略記号ボタンをクリックします。

Web ページエディタでコンポーネントを追加すると，そのコンポーネントは，InternetExpress ページプロデューサのデフォルトテンプレート内にある HTML 透過タグを置き換える HTML を生成します。これらのコンポーネントは，WebPageItems プロパティの値になります。コンポーネントを使用したい順に追加した後，テンプレートに独自の HTML を追加したりデフォルトタグを変更したりしてテンプレートをカスタマイズすることができます。

Web ページエディタの使い方

Web ページエディタを使って，Web 項目を InternetExpress ページプロデューサに追加し，その結果作成される HTML ページを表示することができます。Web ページエディタを表示するには，InternetExpress ページプロデューサコンポーネントをダブルクリックします。

メモ Web ページエディタを使用するには，Internet Explorer 4 またはそれ以降のバージョンが必要です。

Web ページエディタの最上部には，HTML ドキュメントを生成する Web 項目が表示されます。これらの Web 項目はネストされ，各タイプの Web 項目は，それぞれのサブ項目によって生成される HTML を組み立てます。項目のタイプによって，異なるサブ項目が入ります。左側には，すべての Web 項目が，ネスト関係がわかるようにツリー表示されます。右側には，現在選択されている項目に含められる Web 項目が表示されます。Web ページエディタの上部でコンポーネントを選択すると，そのプロパティをオブジェクトインスペクタで設定できます。

現在選択されている項目にサブ項目を追加するには，［追加］ボタンをクリックします。［Web コンポーネントの追加］ダイアログに，現在選択されている項目に追加できる項目だけが一覧表示されます。

InternetExpress ページプロデューサは，次の 2 つのタイプの項目を含むことができます。どちらも HTML フォームを生成します。

- TDataForm。これは，データ，ならびにそのデータの操作または更新のサブミットを制御する各種コントロールを表示する HTML フォームを生成します。

  TDataForm に追加された項目は，データをマルチレコードグリッド（TDataGrid）で表示，または単一レコード内の単一フィールドを表す各コントロールのセット（TFieldGroup）によって表示します。さらに，データ間を移動したり更新を書き込んだりするためのボタン一式（TDataNavigator），または更新を Web クライアントに返して適用するボタン（TApplyUpdatesButton）を追加できます。これらの各項目には，個々のフィールドまたはボタンを表すサブ項目が含まれます。さらには，たいていの Web 項目と同様に，それに含まれる任意の項目のレイアウトをカスタマイズするためのレイアウトグリッド（TLayoutGroup）も追加できます。

- TQueryForm。これは，アプリケーションによって定義される値の表示または読み取りのための HTML フォームを生成します。たとえば，このフォームを使って，パラメータ値を表示およびサブミットできます。

  TQueryForm に追加した項目は，アプリケーションによって定義される値（TQueryFieldGroup）を表示，あるいはこれらの値をサブミットまたはリセットするためのボタン一式（TQueryButtons）を表示します。これらの各項目には，個々の値やボタンを表すサブ項目が含まれます。また，データフォームのように，問い合わせフォームにレイアウトグリッドを追加することもできます。

Web ページエディタの下部には，生成された HTML コードが表示され，ブラウザ（IE4）でどのような外観になるか確認できます。

Web 項目プロパティの設定

Web ページエディタを使って追加する Web 項目は，HTML を生成するための専用コンポーネントです。各 Web 項目クラスは，最終的な HTML ドキュメントの特定のコントロールまたはセクションを生成するよう意図されていますが，共通のプロパティのセットが最終的な HTML の外観に影響を与えます。

Web 項目が XML データパケットからの情報を表す場合（たとえば，一組のフィールド表示コントロールまたはパラメータ表示コントロール，あるいはデータを操作するためのボタンを生成する場合），XMLBroker プロパティが，データパケットを管理する XML ブローカにその Web 項目を関連付けます。さらに，XMLDataSetField プロパティを使って，そのデータパケットのデータセットフィールドに格納されるデータセットを指定することができます。特定のフィールドまたはパラメータ値を表す Web 項目には，FieldName または ParamName プロパティがあります。

任意の Web 項目にスタイル属性を適用し，それによって生成されるすべての HTML の全体の外観に影響を与えることができます。スタイルとスタイルシートは，HTML 4 標準の一部です。これらの使用によって，HTML ドキュメントで，タグおよびそのスコープ内のすべてに適用される表示属性を定義できます。Web 項目は，幅広い使い方が可能です。

- スタイルの最も単純な使い方は，スタイル属性を直接 Web 項目上で定義することです。これには Style プロパティを使用します。Style の値は，単に標準 HTML スタイル定義の属性定義部分であり，次のようなものです。
  ```
  color: red
  ```
- また，スタイルシート（一連のスタイル定義を定義）を定義することもできます。各スタイルシートを定義するには，スタイルセレクタ（そのスタイルが常に適用されるタグの名前，またはユーザー定義スタイル名）と属性定義を中カッコで囲みます。
  ```
  H2 B  {color: red}
  .MyStyle  {font-family: arial; font-weight: bold; font-size: 18px }
  ```
  このような定義全体を，InternetExpress ページプロデューサは自分の Styles プロパティとして保持します。各 Web 項目は，StyleRule プロパティの設定によってユーザー定義名のスタイルを参照します。
- スタイルシートをほかのアプリケーションと共用する場合は，そのスタイル定義を InternetExpress ページプロデューサの Styles プロパティではなく StylesFile プロパティの値として指定できます。その場合も，個々の Web 項目は StyleRule プロパティを使ってスタイルを参照します。

Web 項目のもうひとつのよく使用されるプロパティは，Custom プロパティです。Custom には，生成される HTML タグに追加するオプションのセットが含まれます。HTML は，各種のタグについて異なるセットのオプションを定義します。VCL リファレンスでは，ほとんどの Web 項目の Custom プロパティについて考えられるオプションの例が提供されています。考えられるオプションの詳細については，HTML リファレンスを参照してください。

InternetExpress ページプロデューサのテンプレートをカスタマイズする方法

InternetExpress ページプロデューサのテンプレートは，HTML ドキュメントに特別なタグが埋め込まれているもので，このタグをアプリケーションが動的に翻訳します。最初，ページプロデューサは，デフォルトのテンプレートを HTMLDoc プロパティの値として生成します。このデフォルトテンプレートは次の形式です。

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<#INCLUDES> <#STYLES> <#WARNINGS> <#FORMS> <#SCRIPT>
</BODY>
</HTML>
```

デフォルトテンプレート内の HTML 透過のタグは，次のように翻訳されます。

<#INCLUDES> は，Java スクリプトライブラリをインクルードするステートメントを生成します。これらのステートメントは次の形式になります。

```
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmldom.js"> </SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmldb.js"> </SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmlbind.js"> </SCRIPT>
```

<#STYLES> は，InternetExpress ページプロデューサの Styles または StylesFile プロパティにリストされた定義を基に，スタイルシートを定義したステートメントを生成します。

<#WARNINGS> は，実行時には何も生成しません。設計時には，HTML ドキュメントの生成時に検出された問題について警告メッセージを追加します。これらのメッセージは，Web ページエディタで表示されます。

<#FORMS> は，Web ページエディタで追加したコンポーネントによって作成される HTML を生成します。各コンポーネントからの HTML は，WebPageItems 内に設定された順序で生成されます。

<#SCRIPT> は，Java スクリプト宣言のブロックを生成します。これらの宣言は，Web ページエディタで追加したコンポーネントによって生成される HTML で使用されます。

デフォルトテンプレートは，HTMLDoc の値の変更か HTMLFile プロパティの設定によって置換することができます。カスタム HTML テンプレートには，デフォルトテンプレートに含まれる HTML 透過のタグをどれでも使用できます。InternetExpress ページプロデューサは，Content メソッドが呼び出されたときに自動的にこれらのタグを翻訳します。InternetExpress ページプロデューサは，そのほかにも次の 3 つのタグを自動的に翻訳します。

<#BODYELEMENTS> は，デフォルトテンプレート内の 5 つのタグの結果と同じ HTML によって置換されます。デフォルトのレイアウトを使いたいが，HTML エディタを使ってエレメントを追加したいときに HTML エディタ内でテンプレートを生成するときに便利です。

<#COMPONENT Name=WebComponentName> は，WebComponentName という名前のコンポーネントが生成する HTML によって置換されます。このコンポーネントに指定できるのは，Web ページエディタで追加されたコンポーネントのうちの 1 つ，または，IWebContent インターフェースをサポートしていて InternetExpress ページプロデューサと同じ Owner を持つ任意のコンポーネントです。

<#DATAPACKET XMLBroker=BrokerName> は，BrokerName で指定された XML ブローカ から取得できる XML データパケットで置換されます。Web ページエディタで InternetExpress ページプロデューサによって生成された HTML を表示するとき，実際の XML データパケットのかわりにこのタグが表示されます。

さらに，カスタムテンプレートには，ほかにも自分で定義した任意の HTML 透過タグを挿入することができます。InternetExpress ページプロデューサは，自動的に翻訳する 7 つのタイプ以外のタグがあると，OnHTMLTag イベントを発生させるので，ここで自分の翻訳を実行するコードを記述できます。HTML テンプレートの一般的な情報は，33-14 ページの「HTML テンプレート」を参照してください。

ヒント Web ページエディタに表示されるコンポーネントは，静的コードを生成します。つまり，アプリケーションサーバーがデータパケットに出現するメタデータを変更しない限り，HTML はいつ生成されても常に同じです。実行時にリクエストメッセージがあるたびにそれに応答してこのコードを動的に生成するオーバーヘッドを避けるために，Web ページエディタで生成された HTML をコピーしてそれをテンプレートとして使用することができます。このようにしても，Web ページエディタは実際の XML のかわりに <#DATAPACKET> タグを表示するので，これをテンプレートとして使用すれば，アプリケーションがアプリケーションサーバーからのデータパケットを動的に取り出すことが可能です。

第30章

データベースアプリケーションでの XML の使用

C++Builder では，データベースサーバーへの接続をサポートするだけでなく，XML ドキュメントをデータベースサーバーであるかのように取り扱うことができます。XML（Extensible Markup Language）は，構造化されたデータを記述するためのマークアップ言語です。XML ドキュメントは，Web アプリケーション，ビジネス間通信などで使用するデータ用に，標準の送信可能形式を採用しています。XML ドキュメントの直接の取り扱いに対する C++Builder でのサポートについての詳細は，第 35 章「XML ドキュメントの操作」を参照してください。

データベースアプリケーションで XML ドキュメントを操作するため C++Builder でサポートしている機能は，データパケット（クライアントデータセットの Data プロパティ）を XML ドキュメントに変換し，XML ドキュメントをデータパケットに変換できる一組のコンポーネントに基づいています。これらのコンポーネントを使用するには，まず，XML ドキュメントとデータパケットの間の変換を定義する必要があります。変換を定義したら，特別なコンポーネントを使用して以下を実行できます。

- XML ドキュメントをデータパケットに変換する
- XML ドキュメントからデータを供給し，XML ドキュメントの更新を解決する
- XML ドキュメントをプロバイダのクライアントとして使用する

変換の定義

データパケットと XML ドキュメントとの間の変換を行う前に，データパケット内のメタデータと対応する XML ドキュメントのノードとの関係を定義しなければなりません。この関係の記述は，変換と呼ばれる特別な XML ドキュメント内に保存されます。

それぞれの変換ファイルには，XML スキーマ内のノードとデータパケット内の項目との間のマッピング，および変換の結果の構造を表す XML ドキュメントの骨格の 2 つが格納されます。変換は，一方向マッピングです。つまり，XML スキーマまたはドキュメントからデータパケットへ，または

データパケット内のメタデータから XML スキーマへの方向です。変換ファイルをペアで作成することもよくあります。1 つは，XML からデータパケットへのマッピングで，もう 1 つはデータパケットから XML へのマッピングです。

マッピング用の変換ファイルを作成するには，bin ディレクトリに置かれている付属の XMLMapper ユーティリティを使用してください。

XML ノードとデータパケット項目の間のマッピング

XML では，構造化されたデータをテキストベースで格納または記述しています。データセットは，構造化されたデータを格納または記述するもう 1 つの手段です。したがって，XML ドキュメントをデータセットに変換するには，XML ドキュメント内のノードとデータセット内の項目との対応関係を指定する必要があります。

たとえば，一組の電子メールメッセージを表す XML ドキュメントを考えましょう。これは次のように，記述できます（単一メッセージを含む）。

```
<?xml version="1.0" standalone='yes' ?>
<email>
   <head>
      <from>
         <name>Dave Boss</name>
         <address>dboss@MyCo.com</address>
      </from>
      <to>
         <name>Joe Engineer</name>
         <address>jengineer@MyCo.com</address>
      </to>
      <cc>
         <name>Robin Smith/name>
         <address>rsmith@MyCo.com</address>
      </cc>
      <cc>
         <name>Leonard Devon</name>
         <address>ldevon@MyCo.com</address>
      </cc>
   </head>
   <body>
      <subject>XML components</subject>
      <content>
        Joe,
        Attached is the specification for the new XML component support in C++Builder.
        This looks like a good solution to our buisness-to-buisness application!
        Also attached, please find the project schedule.Do you think its reasonable?
           Dave.
      </content>
      <attachment attachfile="XMLSpec.txt"/>
      <attachment attachfile="Schedule.txt"/>
   </body>
</email>
```

このドキュメントとデータセットとの間の 1 つの自然なマッピングにより，それぞれの電子メールメッセージが単一のレコードにマッピングされます。レコードには，送信側の名前とアドレス用の項目があります。電子メールメッセージの受信者は複数あってよいため，受信者（<to>）は，ネストさ

れたデータセットにマッピングされます。同様に，cc のリストはネストされたデータセットにマッピングされます。件名行は，文字列項目にマッピングされますが，メッセージ自体（<content>）はおそらくメモ型項目になるでしょう。添付ファイルの名前は，ネストされたデータセットにマッピングされますが，それは，1 つのメッセージに複数の添付ファイルを付けることができるからです。そのため，上の電子メールは次のようなデータセットにマッピングされます。

送信者名（Name）	送信者アドレス（Address）	受信者（To）	CC	件名（Subject）	内容（Content）	添付（Attach）
Dave Boss	dboss@MyCo.Com	（データセット）	（データセット）	XML コンポーネント	（メモ型）	（データセット）

「To」項目内のネストされたデータが次の場合，

Name	Address
Joe Engineer	jengineer@MyCo.Com

「CC」項目内のネストされたデータセットは次のようになります。

Name	Address
Robin Smith	rsmith@MyCo.Com
Leonard Devon	ldevon@MyCo.Com

そして「Attach」項目内のネストされたデータセットは次のようになります。

Attachfile
XMLSpec.txt
Schedule.txt

このようなマッピングを定義するには，繰り返し可能な XML ドキュメントのノードを識別し，それらのノードをネストされているデータセットにマッピングする必要があります。値を持ち，一度しか出現しないタグ付き要素（<content>...</content> など）は，値としてみなせるデータの型を反映するデータタイプの項目にマッピングされます。タグの属性（添付タグの AttachFile 属性など）も項目にマッピングされます。

XML ドキュメント内のすべてのタグが対応するデータセット内に記述されているわけではないことに注意してください。たとえば，<head>...</head> 要素は，得られるデータセット内に対応する要素がありません。通常，値を持つ要素のみ，繰り返せる要素，またはタグの属性がデータセットの項目（ネストされたデータセット項目を含む）にマッピングされます。この規則の例外は，XML ドキュメント内の親ノードが子ノードの値から構築される値が入っている項目にマッピングされる場合です。たとえば，XML ドキュメントに次のような一組のタグが含まれているとします。

```
<FullName>
   <Title> Mr. </Title>
   <FirstName> John </FirstName>
   <LastName> Smith </LastName>
</FullName>
```

これは，次の値を持つ単一のデータセット項目にマッピングできます。

```
Mr. John Smith
```

XMLMapper を使用する

XML Mapper ユーティリティ xmlmapper.exe を使用し，以下の方法でマッピングを定義できます。

- 既存の XML スキーマ（またはドキュメント）から定義するクライアントデータセットへ。これは，すでに XML スキーマが用意されているデータを操作するためにデータベースアプリケーションを作成するときに使用します。
- 既存のデータパケットから定義している新しい XML スキーマへ。これは，XML で既存のデータベース情報を公開するときに使用します。たとえば，新しいビジネス間通信システムを作成する場合です。
- 既存の XML スキーマと既存のデータパケットの間で。これは，XML スキーマとデータベースがあり，両方とも同じ情報を記述しており，同時に機能させたい場合に使用します。

マッピングを定義した後，XML ドキュメントをデータパケットに変換し，データパケットを XML ドキュメントに変換するための変換ファイルを生成できます。変換ファイルだけが双方向であることに注意してください。つまり，単一のマッピングを使用して，XML からデータパケットへ，データパケットから XML への両方の変換を生成できるということです。

メモ XML マッパーを使用するには，2 つの .DLL（midas.dll および msxml.dll）が正常に動作している必要があります。xmlmapper.exe を使用する前に，これらの .DLL が両方ともインストールされていることを確認してください。さらに，msxml.dll を COM サーバーとして登録しなければなりません。登録するには，Regsvr32.exe を使用します。

XML スキーマまたはデータパケットをロードする

マッピングを定義し，変換ファイルを生成する前に，まず，XML ドキュメントの記述と，マッピングするデータパケットをロードする必要があります。

XML ドキュメントまたはスキーマをロードするには，［ファイル｜開く］を選択して，表示されるダイアログでドキュメントまたはスキーマを選択します。

データパケットをロードするには，［ファイル｜開く］を選択して，表示されるダイアログでデータパケットファイルを選択します（データパケットは，クライアントデータセットの SaveToFile メソッドを呼び出すときに生成される単なるファイルにすぎません）。データパケットをディスクに保存していない場合，データパケットペインで右クリックし，［リモートサーバーに接続］を選択して，多層アプリケーションのアプリケーションサーバーから直接，データパケットを取得できます。

XML ドキュメントまたはスキーマだけをロードしたり，データパケットだけをロードしたり，またはその両方をロードしたりできます。マッピングの片側のみをロードする場合，XML マッパーは他方の側の自然なマッピングを生成できます。

マッピングを定義する

XML ドキュメントとデータパケットとの間のマッピングに，データパケット内のすべての項目または XML ドキュメント内のすべてのタグ付き要素が含まれている必要はありません。したがって，ま

ず，マッピングされる要素を指定しなければなりません。これらの要素を指定するには，最初に，ダイアログの中央のペイン内の［マッピング］ページを選択してください。

データパケット内の項目にマッピングする XML ドキュメントまたはスキーマの要素を指定するには，XML ドキュメントペインの［ドキュメント表示］タブまたは［スキーマ表示］タブを選択して，データパケット項目にマッピングされる要素のノードをダブルクリックします。

XML ドキュメント内のタグ付き要素または属性にマッピングされるデータパケットの項目を指定するには，データパケットペイン内の項目のノードをダブルクリックします。

マッピングの片側のみをロードした場合（XML ドキュメントまたはデータパケット），マッピングされるノードを選択した後，他方の側を生成できます。

- XML ドキュメントからデータパケットを生成する場合，まずデータパケット内で対応する項目の種類を決定する選択されているノードの属性を定義します。中央のペインで，［ノードプロパティ］ペインを選択します。マッピングに加わっているそれぞれのノードを選択し，対応する項目の属性を示します。マッピングが簡単でない場合（たとえば，ある項目に対応するサブノードを持つノードで，その項目の値がサブノードから構築される場合），［ユーザー定義の変換］チェックボックスにチェックマークを付けます。ユーザー定義ノードに対し変換を実行するため，後でイベントハンドラを作成する必要があります。

  ノードをマッピングする方法を指定した後，［作成｜データパケットから XML］を選択します。対応するデータパケットが自動的に生成され，データパケットペインに表示されます。

- データパケットから XML ドキュメントを生成する場合には，［作成｜ XML からデータパケット］を選択します。データパケット内の項目，レコード，およびデータセットに対応する XML ドキュメント内のタグおよび属性の名前を指定するためのダイアログが表示されます。項目値については，値でタグ付き要素にマッピングするのか，それとも名前を付ける方法で属性にマッピングするのかを指定します。@ 記号で始まる名前はレコードに対応するタグの属性にマッピングされますが，@ 記号で始まらない名前は値を持ち，レコードの要素内でネストされているタグ付き要素にマッピングされます。

- XML ドキュメントとデータパケット（クライアントデータセットファイル）の両方をロードしている場合，同じ順序で対応するノードを選択するようにしてください。対応するノードが［マッピング］ページの最上段のテーブル内に隣り合って表示されます。

XML ドキュメントとデータパケットの両方をロードまたは生成し，マッピング内に現れるノードを選択した後，［マッピング］ページの最上段のテーブルに定義済みのマッピングが表示されます。

変換ファイルを生成する

変換ファイルを生成するには，次の手順に従ってください。

1. まず，変換の作成内容を示すラジオボタンを選択します。
   - データパケットから XML ドキュメントへのマッピングの場合，［データパケットから XML］ボタンを選択します。
   - XML ドキュメントからデータパケットへのマッピングの場合，［XML からデータパケット］ボタンを選択します。

2. データパケットを生成する場合は，［形式を指定してデータパケットを作成］セクションのラジオボタンも使用することになります。これらのボタンで，データパケットを使用する方法（データセットとして，更新を適用するデルタパケットとして，またはデータを取得する前にプロバイダに渡すパラメータとして）を指定します。
3. ［変換の作成とテスト］をクリックして，変換のメモリ内バージョンを生成します。XML マッパーは，データパケットペイン内のデータパケットに対し生成する XML ドキュメント，または XML ドキュメントペイン内で XML ドキュメントに対して生成するデータパケットを表示します。
4. 最後に，［ファイル｜保存｜変換］を選択して，変換ファイルを保存します。変換ファイルは，すでに定義されている変換を記述した特別な XML ファイル（拡張子は .xtr）です。

データパケットへの XML ドキュメントの変換

XML ドキュメントをデータパケットに変換する方法を記述した変換ファイルを作成した後，変換で使用されるスキーマに準拠する XML ドキュメント用のデータパケットを作成できます。その後，これらのデータパケットをクライアントデータセットに割り当てて，ファイルに保存し，ファイルベースのデータベースアプリケーションの基本とします。

TXMLTransform コンポーネントは，変換ファイルに記述されたマッピングに従って，XML ドキュメントをデータパケットに変換します。

メモ また TXMLTransform を使用して，XML 形式で表示されるデータパケットを任意の XML ドキュメントに変換することもできます。

ソース XML ドキュメントを指定する

ソース XML ドキュメントを指定する方法は 3 通りあります。

- ソースドキュメントがディスク上の .xml ファイルであれば，SourceXmlFile プロパティを使用できます。
- ソースドキュメントが XML のメモリ内文字列であれば，SourceXml プロパティを使用できます。
- ソースドキュメントの IDOMDocument インターフェースがある場合は，SourceXmlDocument プロパティを使用できます。

TXMLTransform では，上記の順序でこれらのプロパティをチェックします。つまり，まず SourceXmlFile プロパティ内のファイル名をチェックします。SourceXmlFile が空文字列である場合のみ，SourceXml プロパティをチェックします。SourceXml が空文字列である場合のみ，SourceXmlDocument プロパティをチェックします。

変換を指定する

XML ドキュメントをデータパケットに変換する変換を指定する方法は 2 通りあります。

- TransformationFile プロパティの設定で，xmlmapper.exe を使用して作成された変換ファイルを示す

- 変換用の IDOMDocument インターフェースがある場合は，TransformationDocument プロパティを設定する

TXMLTransform では，上記の順序でこれらのプロパティをチェックします。つまり，まず TransformationFile プロパティ内のファイル名をチェックします。TransformationFile が空文字列である場合のみ，TransformationDocument プロパティをチェックします。

結果のデータパケットを取得する

TXMLTransform で変換を実行し，データパケットを生成するには，Data プロパティを読み込むだけでかまいません。たとえば，次のコードでは，XML ドキュメントと変換ファイルを使用して，データパケットを生成します。その後，クライアントデータセットに割り当てます。

```
XMLTransform1->SourceXMLFile = "CustomerDocument.xml";
XMLTransform1->TransformationFile = "CustXMLToCustTable.xtr";
ClientDataSet1->XMLData = XMLTransform1->Data;
```

ユーザー定義ノードを変換する

xmlmapper.exe を使用して変換を定義するときに，XML ドキュメント内のノードの一部が「ユーザー定義」になるように指定できます。ユーザー定義ノードは，ノード値から項目値への直接的な変換に依存しない，コードで変換を行うノードです。

OnTranslate イベントを使用してユーザー定義ノードを変換するコードを作成できます。OnTranslate は，TXMLTransform コンポーネントで XML ドキュメント内のユーザー定義ノードを見つけるごとに呼び出されます。OnTranslate イベントハンドラでは，ソースドキュメントを読み込んで，データパケット内の項目に対する結果の値を指定できます。

たとえば次の OnTranslate イベントハンドラは，XML ドキュメント内のノードを次の形式で

```
<FullName>
   <Title> </Title>
   <FirstName> </FirstName>
   <LastName> </LastName>
</FullName>
```

単一の項目値に変換します。

```
void __fastcall TForm1::XMLTransform1Translate(TObject *Sender, AnsiString Id,
  _di_IDOMNode SrcNode, AnsiString &Value, _di_IDOMNode DestNode)
{
  if (Id == "FullName")
  {
    Value = '';
    if (SrcNode.hasChildNodes)
    {
      _di_IXMLDOMNode CurNode = SrcNode.firstChild;
      Value = SrcValue + AnsiString(CurNode.nodeValue);
      while (CurNode != SrcNode.lastChild)
      {
        CurNode = CurNode.nextSibling;
        Value = Value + AnsiString(" ");
        Value = Value + AnsiString(CurNode.nodeValue);
```

```
        }
      }
    }
  }
```

XML ドキュメントをプロバイダのソースとして使う

TXMLTransformProvider コンポーネントでは，XML ドキュメントをデータベーステーブルと同様に使用できます。TXMLTransformProvider により，XML ドキュメントからのデータがパッケージ化され，クライアントからの更新がその XML ドキュメントに適用されます。クライアントデータベースや XML ブローカなどのクライアントには，他のプロバイダコンポーネントのように見えます。プロバイダコンポーネントについての詳細は，第 28 章「プロバイダコンポーネントの使い方」を参照してください。クライアントデータベースとともにプロバイダコンポーネントを使用する方法についての詳細は，27-23 ページの「クライアントデータセットでデータプロバイダを使用する」を参照してください。

XML プロバイダのデータの供給元および更新の適用先である XML ドキュメントを指定できます。それには，XMLDataFile プロパティを使用します。

TXMLTransformProvider コンポーネントは，内部の TXMLTransform コンポーネントを使用して，データパケットとソース XML ドキュメントとの間の変換を実行します。XML ドキュメントをデータパケットに変換するコンポーネントと，更新を適用した後データパケットを XML 形式のソースドキュメントに変換して戻すコンポーネントの 2 つがあります。これら 2 つの TXMLTransform コンポーネントにアクセスするには，それぞれ TransformRead プロパティと TransformWrite プロパティを使用します。

TXMLTransformProvider を使用する場合は，2 つの TXMLTransform コンポーネントでデータパケットと XML ドキュメントの間の変換に使用する変換を指定しなければなりません。そのためには，TXMLTransform コンポーネントの TransformationFile または TransformationDocument プロパティを使用します。スタンドアロンの TXMLTransform コンポーネントを使用する場合とまったく同様です。

さらに，変換にユーザー定義ノードが含まれる場合には，OnTranslate イベントハンドラを内部 TXMLTransform コンポーネントに渡す必要があります。

TransformRead と TransformWrite の値である TXMLTransform コンポーネントでソースドキュメントを指定する必要はありません。TransformRead については，ソースはプロバイダの XMLDataFile プロパティで指定したファイルです（が，XMLDataFile を空文字列に設定した場合，TransformRead->XmlSource または TransformRead->XmlSourceDocument を使用してソースドキュメントを指定することができます）。TransformWrite については，ソースは更新の適用時にプロバイダによって内部的に生成されます。

XML ドキュメントをプロバイダのクライアントとして使う

TXMLTransformClient コンポーネントは，XML ドキュメント（またはドキュメントの集まり）をアプリケーションサーバーのクライアント（または，単に，TDataSetProvider コンポーネントを介した接続先のデータセットのクライアント）として使用するためのアダプタとして機能します。つまり，TXMLTransform クライアントを使用することで，データベースデータを XML ドキュメントとして公開し，外部アプリケーションから XML ドキュメント形式で渡された更新要求（挿入または削除）を使用できます。

TXMLTransformClient オブジェクトによるデータの取得元および更新の適用先であるプロバイダを指定するには，ProviderName プロパティを設定します。クライアントデータセットの ProviderName プロパティの場合のように，ProviderName はリモートアプリケーションサーバー上のプロバイダの名前とすることもできますし，また TXMLTransformClient オブジェクトと同じフォームまたはデータモジュール内のローカルプロバイダとすることもできます。プロバイダについての詳細は，第 28 章「プロバイダコンポーネントの使い方」を参照してください。

プロバイダがリモートアプリケーションサーバー上にある場合は，DataSnap 接続コンポーネントを使用して，アプリケーションサーバーに接続する必要があります。接続コンポーネントを指定するには，RemoteServer プロパティを使用します。DataSnap 接続コンポーネントについての詳細は，29-23 ページの「アプリケーションサーバーへの接続」を参照してください。

プロバイダから XML ドキュメントを取得する

TXMLTransformClient は，内部 TXMLTransform コンポーネントを使用して，プロバイダからのデータパケットを XML ドキュメントに変換します。この TXMLTransform コンポーネントには，TransformGetData プロパティの値としてアクセスできます。

プロバイダからのデータを表す XML ドキュメントを作成する前に，TransformGetData でデータパケットを適切な XML 形式に変換するのに使用する変換ファイルを指定しなければなりません。そのためには，TXMLTransform コンポーネントの TransformationFile または TransformationDocument プロパティを使用します。スタンドアロンの TXMLTransform コンポーネントを使用する場合とまったく同様です。変換にユーザー定義ノードが含まれる場合，TransformGetData を OnTranslate イベントハンドラともども渡すようにします。

TransformGetData のソースドキュメントを指定する必要はありません。TXMLTransformClient によってプロバイダから取得されるからです。ただし，プロバイダ側で入力パラメータを想定している場合，データ取得の前に設定しておくとよいでしょう。プロバイダからデータを取得する前に，SetParams メソッドを使用して，これらの入力パラメータを渡します。SetParams は，パラメータ値の抽出元である XML の文字列と，XML をデータパケットに変換する変換ファイルの名前を引数として取ります。SetParams は，変換ファイルを使用して，XML の文字列をデータパケットに変換し，そのデータパケットからパラメータ値を抽出します。

メモ パラメータドキュメントまたは変換を別の方法で指定する場合は，これらの引数のいずれかを指定変更できます。単に，TransformSetParams プロパティでプロパティの 1 つを，パラメータまたは変換時

に使用する変換を含むドキュメントを指すように設定し，SetParams の呼び出し時に指定変更する引数を空文字列に設定するだけです。使用可能なプロパティの詳細は，30-6 ページの「データパケットへの XML ドキュメントの変換」を参照してください。

TransformGetData を設定し，入力パラメータを指定した後，GetDataAsXml メソッドを呼び出して，XML を取得できます。GetDataAsXml は，現在のパラメータ値をプロバイダに送信し，データパケットを取得し，XML ドキュメントに変換し，そのドキュメントを文字列として返します。この文字列は，次のようにしてファイルに保存できます。

```
XMLTransformClient1->ProviderName = "Provider1";
XMLTransformClient1->TransformGetData->TransformationFile = "CustTableToCustXML.xtr";
XMLTransformClient1->TransFormSetParams->SourceXmlFile = "InputParams.xml";
XMLTransformClient1->SetParams("", "InputParamsToDP.xtr");
AnsiString XML = XMLTransformClient1->GetDataAsXml();
TFileStream pXMLDoc = new TFileStream("Customers.xml", fmCreate || fmOpenWrite);
__try
{
  pXMLDoc->Write(XML.c_str(), XML.Length());
}
__finally
{
  delete pXMLDoc;
}
```

XML ドキュメントからプロバイダに更新を適用する

TXMLTransformClient ではさらに，XML ドキュメントのすべてのデータをプロバイダのデータセットに送信したり，XML ドキュメント内のすべてのレコードをプロバイダのデータセットから削除したりすることもできます。これらの更新を実行するには，ApplyUpdates メソッドを呼び出します。その際に渡すパラメータは次のとおりです。

- 文字列。この値は，挿入または削除するデータを含む XML ドキュメントの内容です。
- XML データを挿入または削除デルタパケットに変換するための変換ファイルの名前（XML マッパーユーティリティを使用して変換ファイルを定義するときに，変換が挿入デルタパケットなのか，削除デルタパケットなのかを指定します）。
- 更新操作を中止するまで許容できる更新エラー数。指定された数よりも少ないレコードを挿入または削除できない場合，ApplyUpdates は実際に発生したエラーの数を返します。指定された数よりも多いレコードを挿入または削除できない場合，更新操作全体がロールバックされ，更新は実行されません。

次の呼び出しでは，XML ドキュメント Customers.xml がデルタパケットに変換され，エラーの数に関係なくすべての更新が適用されます。

```
StringList1->LoadFromFile("Customers.xml");
nErrors = ApplyUpdates(StringList1->Text, "CustXMLToInsert.xtr", -1);
```

第 III 部

インターネットアプリケーションの作成

第 III 部「インターネットアプリケーションの作成」の各章では，インターネットを介した分散アプリケーションの作成に必要な概念およびテクニックを説明します。

メモ　ここで取り上げるコンポーネントは，C++Builder の一部の版では使用できません。

第31章

CORBA アプリケーションの作成

CORBA（Common Object Request Broker Architecture）は，分散オブジェクトアプリケーションの開発に対する複雑さを軽減するためにオブジェクト管理グループ（OMG: Object Management Group）によって提唱された仕様です。

名前が示すとおり，CORBA は分散アプリケーションを作成するためのオブジェクト指向のアプローチを提供します。これは，第 32 章「インターネットサーバーアプリケーションの作成」に記述されているような HTTP アプリケーション用のメッセージ指向のアプローチと対照を成すものです。CORBA では定義されたインターフェースを使って，クライアントアプリケーションがリモート使用できるオブジェクトをサーバーアプリケーションに実装します。

メモ COM は，分散アプリケーションに別のオブジェクト指向のアプローチを提供します。COM についての詳細は，第 38 章「COM テクノロジーの概要」を参照してください。COM とは異なり，CORBA は Windows 以外のプラットフォームにも適用できる規格です。つまり，C++Builder を使用すると，別のプラットフォーム上で作動する CORBA を利用したアプリケーションと通信できるクライアントまたはサーバーアプリケーションを作成できます。

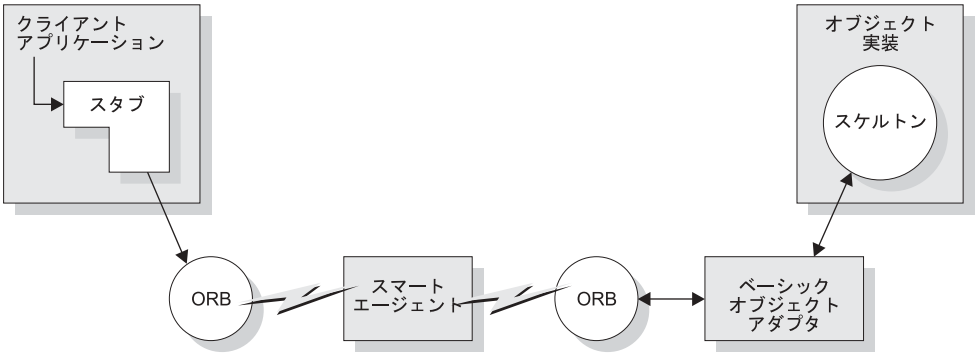
CORBA の仕様は，サーバー上に実装されているオブジェクトとクライアントアプリケーションがどのように通信するかを定義しています。この通信は，ORB（Object Request Broker）によって処理されます。C++Builder は Borland の VisiBroker ORB（バージョン 4.5）と統合されているので，容易に CORBA 開発が行えます。

クライアントがサーバーマシン上のオブジェクトと通信できる基本 ORB 技術に加えて，CORBA 規格はいくつかの標準サービスを定義します。これらのサービスは定義されたインターフェースを使用しているため，サーバーが別のベンダーで作成されている場合でも，開発者はこれらのサービスを利用するクライアントを作成できます。

CORBA アプリケーションの概要

すでにオブジェクト指向プログラミングの経験がある方は，CORBA を使うとリモートオブジェクトをほとんどローカルオブジェクトと同じように使用できるので，分散アプリケーションを楽に構築できます。これは，CORBA アプリケーションが，ほかのオブジェクト指向アプリケーションと同じように設計されているからです。違いは，オブジェクトが別のマシンにあるときにネットワーク通信を処理するための層が追加されていることだけです。この追加層は，スタブおよびスケルトンと呼ばれる特殊なオブジェクトで処理します。

図 31.1　CORBA アプリケーションの構造



CORBA クライアント上で，スタブは同じプロセス，別のプロセス，または別の（サーバー）マシンによって実装されるオブジェクトのプロクシーとして動作します。クライアントは，インターフェースを実装するほかのオブジェクトであるかのようにスタブと対話します。

ただし，インターフェースを実装するほかのオブジェクトとは異なり，スタブはクライアントマシンにインストールされた ORB ソフトウェア内からの呼び出しで，インターフェース呼び出しを処理します。VisiBroker ORB は，ローカルエリアネットワーク上のどこかで実行されている Smart Agent（osagent）を利用します。Smart Agent は，動的な分散ディレクトリサービスで，実際のオブジェクト実装を提供する利用可能なサーバーを見つけます。

CORBA サーバー上で，ORB ソフトウェアはインターフェース呼び出しを自動生成スケルトンに渡します。スケルトンは，BOA（Basic Object Adaptor）を使って ORB ソフトウェアと通信します。BOA を使用して，スケルトンはオブジェクトを Smart Agent に登録し，オブジェクトの範囲（リモートマシンで使用できるかどうか）を示して，いつオブジェクトがインスタンス化され，クライアントに応答する用意ができるかを示します。

スタブとスケルトンを理解する

CORBA の分散オブジェクトの基盤は，そのインターフェースです。インターフェースは実装情報をもたない点を除けば，クラス定義と似ています。インターフェースを定義するには，CORBA インターフェース定義言語（IDL）を使います。インターフェース定義を IDL ファイルに記述してプロジェクトに追加できます。IDL ファイル，スタブ，スケルトンについての詳細は，『VisiBroker

Programmer's Guide』を参照してください。C++Builder で IDL ファイルを使う方法についての詳細は，31-5 ページの「オブジェクトインターフェースを定義する」を参照してください。

C++Builder で IDL ファイルをコンパイルすると 2 つの .cpp ファイルがビルドされます。1 つはクライアント用のファイルで，スタブクラスの実装が記述されています。もう 1 つはサーバー用のファイルで，スケルトンクラスの実装が記述されています。スタブとスケルトンは，CORBA アプリケーションにインターフェース呼び出しのマーシャリングを可能にするメカニズムを提供します。マーシャリングによって次のことが行われます。

- サーバープロセス内にオブジェクトインスタンスを置き，クライアントプロセス内のコードでインスタンスを利用できるようにする
- インターフェース呼び出しの引数をクライアントから渡されたとおりに転送し，リモートオブジェクトのプロセス空間に入れる

クライアントアプリケーションは，CORBA オブジェクトのメソッドを呼び出すと，引数をスタック上に置いて，スタブオブジェクトを呼び出します。スタブは，引数をマーシャルバッファ内に書き込み，構造化された呼び出しをリモートオブジェクトに転送します。サーバースケルトンは，この構造を取り出して引数をスタック上に置いて，オブジェクトの実装を呼び出します。実質的には，スケルトンはクライアントの呼び出しを自身のアドレス領域に再作成します。

Smart Agent の使い方

Smart Agent（osagent）は，動的な分散ディレクトリサービスで，オブジェクトを実装しているサーバーを見つけます。選択できるサーバーが複数ある場合には，Smart Agent は負荷分散を提供します。また，接続が切断されたときにサーバーを再起動したり，必要であれば別のホスト上のサーバーを見つけて，サーバーの故障から保護します。

Smart Agent は，ローカルネットワーク内の少なくとも 1 つのホスト上で起動されていなければなりません。ここでのローカルネットワークとは，同報通信メッセージを送ることができるネットワークのことです。ORB は，同報通信メッセージを使用して Smart Agent を見つけます。ネットワークに複数の Smart Agent がある場合は，最初に応答したものを使用します。Smart Agent が見つかると，ORB はポイントツーポイントの UDP プロトコルを使用して Smart Agent と通信します。UDP プロトコルは，TCP 接続よりも消費するネットワークリソースが少なくてすみます。

ネットワークに複数の Smart Agent が含まれている場合，各 Smart Agent は利用可能なオブジェクトのサブセットを認識し，ほかの Smart Agent と通信して直接認識できないオブジェクトを見つけます。1 つの Smart Agent が不意に終了した場合，この Smart Agent が追跡していたオブジェクトは別の利用可能な Smart Agent に自動的に再登録されます。

ユーザーのローカルネットワーク上での複数の Smart Agent の構成および使用方法の詳細については，『VisiBroker Installation and Administration Guide』を参照してください。

サーバーアプリケーションを起動する

サーバーアプリケーションを起動すると，クライアント呼び出しを受け入れることができるインターフェースが BOA（Basic Object Adaptor）を介して ORB に通知されます。ORB を初期化し，サーバー

が待機状態であることを ORB に知らせるコードは，CORBA サーバーアプリケーションを起動するために使用するウィザードによって自動的にアプリケーションに追加されます。

一般的に，CORBA サーバーアプリケーションは手動で起動します。しかし，オブジェクト起動デーモン（OAD: Object Activation Daemon）を使用することで，サーバーを起動させたり，クライアントが必要なときだけオブジェクトをインスタンス化することもできます。

OAD を利用するには，オブジェクトを登録しなければなりません。オブジェクトを OAD に登録すると，そのオブジェクトを実装するサーバーアプリケーションとそのオブジェクトの間の関連付けがインプリメンテーションリポジトリというデータベースに記録されます。

インプリメンテーションリポジトリ内にオブジェクト用のエントリが作成されると，OAD はそのアプリケーションの動作を ORB に対してシミュレートします。クライアントがオブジェクトを要求すると，ORB はサーバーアプリケーションであるかのように OAD に接触します。その後 OAD は，クライアント要求を実際のサーバーに転送し，必要であればアプリケーションを起動します。

OAD へのオブジェクトの登録についての詳細は，『VisiBroker Programmer's Guide』を参照してください。

インターフェース呼び出しを動的にバインドする

一般的に，CORBA クライアントは，静的バインディングを使用してサーバー上のオブジェクトのインターフェースを呼び出します。このアプローチには，高いパフォーマンスとコンパイル時型チェックを含む多くの利点があります。ただし，実行時までどのインターフェースを使用すればよいのかわからない場合があります。このような場合には，C++Builder では実行時に動的にインターフェースをバインドすることができます。

動的バインディングを使用すると，インターフェースをインターフェースリポジトリに登録できます。

CORBA クライアントアプリケーションで動的バインディングを利用する方法についての詳細は，『Visibroker Programmer's Guide』の「Dynamic Invocation Interface (DII)」を参照してください。

CORBA サーバーの作成

C++Builder には，CORBA サーバーの開発を補助するためのウィザードがいくつか用意されています。C++Builder を使って CORBA サーバーを作成する手順を次に説明します。

1. オブジェクトインターフェースを定義します。これらのインターフェースには，クライアントアプリケーションのサーバーとの対話方法が定義されます。また C++Builder は，これらのインターフェースからスタブとスケルトンの実装を作成します。
2. CORBA サーバーウィザードを使って，スタートアップ時に CORBA BOA と ORB を初期化するコードを含む新規 CORBA サーバーアプリケーションを作成します。
3. インターフェース定義が記述された IDL ファイルをコンパイルしスケルトンクラス（およびスタブクラス）を生成します。
4. CORBA オブジェクトウィザードを使って，実装クラスを定義（およびインスタンス化）します。

5. 手順 5 で作成したクラスを完成し，CORBA オブジェクトを実装します。
6. 必要ならば，CORBA インターフェースを変更し，実装クラスにその変更を反映させます。

上記の手順に加えて，必要であれば IDL ファイルをインターフェースリポジトリとオブジェクト起動デーモン（OAD: Object Activation Daemon）に登録します。

オブジェクトインターフェースを定義する

CORBA オブジェクトインターフェースの定義には，CORBA インターフェース定義言語（IDL）を使います。IDL の構文は C++ と似ているので，IDL ファイルは C++ のヘッダーファイルと同じように見えます。IDL ファイルの働きもヘッダーファイルと似ています。つまり，ヘッダーファイルで共有可能なクラスを宣言するように，IDL ファイルでも共有可能なインターフェースを宣言します。ただし，ヘッダーファイルは同じアプリケーション内の他のモジュールとのみ共有可能なのに対し，CORBA では，インターフェース（クラス）を他のアプリケーションと共有でき，さらに同じアプリケーションの他のモジュールとも共有できる点が異なります。

メモ IDL という用語は，異なる複数のインターフェース定義言語に対して使用されます。IDL には，CORBA IDL（OMG で定義），Microsoft IDL（COM で使用），DCE IDL があります。CORBA IDL の詳細については，ヘルプを参照してください。

C++Builder 内から新規 IDL ファイルを定義するには，［ファイル｜新規作成｜その他］を選択し，［新規作成］ダイアログの［多層サポート］ページから［CORBA IDL ファイル］を選択します。コードエディタが開いて空白の IDL ファイルが表示されます。また，この IDL ファイルは現在のプロジェクトに追加されます。

すでにオブジェクトを定義した IDL ファイルがある場合は，［プロジェクト｜プロジェクトの追加］を選択し，［ファイルのタイプ］から IDL ファイルを選び，追加する IDL ファイルを選択して，このファイルをプロジェクトに追加します。

CORBA サーバーウィザードの使い方

新規 CORBA サーバープロジェクトを作成するには，［ファイル｜新規作成｜その他］を選択し，［新規作成］ダイアログの［多層サポート］ページから［CORBA サーバー］を選択します。CORBA サーバーウィザードが，コンソールアプリケーションと Windows アプリケーションのどちらを作成するかをたずねます。

コンソールアプリケーションを作成する場合は，サーバーが VCL クラスを使用するかどうかを指定できます。VCL チェックボックスをチェックしないと，生成したすべてのコードを他のプラットフォームに移植できます。

アプリケーションの種類を選択するほかに，既存の IDL ファイルをプロジェクトにインクルードするか，新規の空白の IDL ファイルをインクルードするかを指定できます。サーバープロジェクトには，最終的に 1 つ以上の IDL ファイルが含まれます。IDL ファイルに，クライアントがサーバーと対話するために使用するインターフェースが定義されます。

メモ CORBA サーバープロジェクトの作成時に IDL ファイルを追加しなくても，［プロジェクト｜プロジェクトの追加］を選択（既存の IDL ファイルがある場合）するか，［新規作成］ダイアログの［多

層サポート］ページから［CORBA IDL ファイル］を選択（新規 IDL ファイルを定義する場合）すれば，後でいつでも追加できます。

使用するサーバーの種類を指定し［OK］を選択すると，CORBA サーバーウィザードは，指定した種類の新規サーバープロジェクトを作成します。また，プロジェクトファイルおよび ORB（Object Request Broker）と BOA（Basic Object Adaptor）を初期化する開始コードに CORBA ライブラリが追加されます。

自動的に生成されたコードは，ORB と BOA との対話に使用する orb と boa の 2 つの変数を宣言します。

```
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
CORBA::BOA_var boa = orb->BOA_init(argc, argv);
```

コンソールアプリケーションを指定した場合は，次の行も追加されます。

```
boa->impl_is_ready();
```

この呼び出しにより，Windows メッセージループが Windows アプリケーションで Windows メッセージを受信できるのと同じように，CORBA アプリケーションで BOA からのメッセージを受信できるようになります (Windows アプリケーションは，CORBA メッセージでも Windows メッセージループを使います)。

ORB と BOA の詳細については，VisiBroker のマニュアルを参照してください。

IDL ファイルからスタブとスケルトンを生成する

CORBA インターフェースを定義した IDL ファイルを CORBA プロジェクトに追加したら，サーバーとクライアントアプリケーション間の対話を処理するスタブクラスとスケルトンクラスを生成できます。

IDL ファイルをコンパイルするだけで，スタブクラスとスケルトンクラスが生成されます。IDL ファイルをコンパイルするには，IDL ファイルがコードエディタに表示されているときに［プロジェクト｜ユニットのコンパイル］を選択するか，プロジェクトマネージャで IDL ファイルを右クリックし，［コンパイル］を選択します。IDL コンパイラにより生成された次の 2 つの新規ファイルが，コードエディタとプロジェクトマネージャに表示されます。

- サーバーファイル（xxx_s.cpp）。このファイルには，スケルトンクラスの実装が入っています。
- クライアントファイル（xxx_c.cpp）。このファイルには，スタブクラスの実装が入っています。

［プロジェクトオプション］ダイアログの［CORBA］ページでは，IDL ファイルからスタブとスケルトンクラスを生成する方法を設定します。たとえば，プロジェクトに生成したサーバーユニットのみを含めるように指定したり，サーバーが委任モデル（31-8 ページの「委任モデルの使い方」参照）を使用しているときは Tie クラスを含めるように指定できます。これらのオプションを指定すると，現在のプロジェクトに含まれるすべての IDL ファイルのコンパイルに反映されます。

CORBA オブジェクト実装ウィザードの使い方

IDL ファイルで定義した CORBA オブジェクトを実装するには，すべてのスケルトンクラスに対して実装クラスを作成する必要があります。実装するには，［ファイル｜新規作成｜その他］を選択し，［新規作成］ダイアログの［多層サポート］ページから［CORBA オブジェクトの実装］を選択します。すると，CORBA オブジェクト実装ウィザードが起動します。

CORBA オブジェクト実装ウィザードには，プロジェクトに含まれる IDL ファイルで定義されているすべてのインターフェースが表示されています。IDL ファイルを選択し，その IDL ファイルから実装するインターフェースを作成します。実装クラスに名前を付け，オブジェクトを定義する .h ファイルとオブジェクトを実装する .cpp ファイルのユニット名を指定します。

ウィザードで，実装クラスを，自動的に生成されるスケルトンクラスからの派生クラスとするか，委任モデル（Tie クラス）とするかを指定します。委任モデルを使用する（およびサーバーアプリケーションが VCL 対応）の場合は，実装クラスをデータモジュールにするようにウィザードで指定できます。これにより，実装のプログラミングに使用するコンポーネントを，実装データモジュールに追加できます。

ウィザードでは，アプリケーションの起動時に CORBA オブジェクトをインスタンス化するコードを追加することもできます。こうすると，オブジェクトはクライアントの要求を受信できるようになります。起動時にオブジェクトをインスタンス化する場合は，オブジェクトに名前を付け，クライアントが識別できるようにする必要があります。この名前はコントラクタにパラメータとして渡されますが，インスタンスを参照する変数名ではありません。クライアントアプリケーションが CORBA オブジェクトにバインドされるとき，アプリケーションはこの名前を使って目的のオブジェクトを指定します。

［OK］を選択すると，CORBA オブジェクト実装ウィザードは実装クラスの定義を生成し，すべてのメソッドの中身が記述されていない実装を生成します。また，オブジェクトをインスタンス化するコードを追加するように指定することもできます。

これらの変更がプロジェクトに追加される前に，変更内容を表示して確認し，編集することができます。この編集を行うには，［OK］を選択する前に，［更新部分の表示］ボックスをチェックします。変更を表示後編集すると，それらがプロジェクトに追加されます。ウィザードを終了するとき，変更の表示と編集を選択しなくても，コードエディタを使えば後で編集できます。

複数のインターフェースを実装するには，インターフェースの数だけ CORBA オブジェクト実装ウィザードを呼び出す必要があります。ただし，ウィザードで複数の名前を指定すれば，1 つの実装クラスの複数インスタンスを生成することができます。

CORBA オブジェクトをインスタンス化する

個々の CORBA オブジェクトについて，インスタンス化する方法を指定する必要があります。次の 2 種類の方法から選択します。

- サーバーアプリケーションの起動時にオブジェクトをインスタンス化する。この場合，サーバーアプリケーションが使用可能な状態であれば，クライアントアプリケーションでオブジェクトを使用できます。

- クライアントの要求に応じてオブジェクトをインスタンス化する（既存のオブジェクトのメソッド呼び出し）。この場合，サーバーアプリケーションはインターフェースメソッドの実装内でオブジェクトをインスタンス化し，そのメソッドからオブジェクトへの参照をクライアントに返します。

アプリケーションの起動時にオブジェクトをインスタンス化する場合は，CORBA オブジェクト実装ウィザードのメイン画面にある［インスタンス化］ボックスをチェックします。ウィザードは，プロジェクトのソースファイルに直接実装クラスをインスタンス化するためのコードを追加します。このコードを表示するには，［プロジェクト｜ソースの表示］を選択します。CORBA サーバーウィザードは ORB_init と BOA_init を呼び出す行を挿入します。その後，impl_is_ready() への呼び出しの前に，実装クラスをインスタンス化して，boa にオブジェクトが要求を受信できることを通知するコードをサーバーウィザードが追加します。

```
MyObjImpl MyObject_TheObject("TheObject"); // オブジェクトのインスタンスを作成
boa->obj_is_ready(&MyObject_TheObject);    // boa に要求の受け付けが可能なことを通知
```

コンストラクタと obj_is_ready の呼び出しの間に，初期化コードを追加できます。

```
MyObjImpl MyObject_TheObject("TheObject"); // オブジェクトのインスタンスを作成
MyObject_TheObject.Initialize(50000);
boa->obj_is_ready(&MyObject_TheObject);    // boa に要求の受け付けが可能なことを通知
```

メモ 委任モデル（下記参照）を使う場合は，Tie クラスをインスタンス化する呼び出しも追加されます。

起動時にオブジェクトをインスタンス化しない場合は，メソッドからコンストラクタ（必要ならば Tie クラスのコンストラクタも）を呼び出し，オブジェクトインスタンスに参照を返します。BOA の obj_is_ready メソッドの呼び出しも追加することをお勧めします。

CORBA クライアントからの呼び出しを受けるオブジェクトには，名前を付けておく必要があります。名前がないと，クライアントはそのオブジェクトを見つけられません。一時的なオブジェクト（クライアントの呼び出しによって作成され，参照として返されるオブジェクト）には名前を付ける必要はありません。

委任モデルの使い方

デフォルトでは，CORBA オブジェクトは IDL ファイルをコンパイルしたときに作成されるスケルトンクラスの派生クラスとなっています。つまり，オブジェクトクラスが，インターフェースの呼び出しをマーシャリングし BOA と対話するすべてのコードをスケルトンクラスから継承します。しかし，（多重継承を許可しない）VCL クラスから派生したプログラミング済みオブジェクトを CORBA サーバーでエクスポーズする場合，または他の非 CORBA アプリケーションと CORBA サーバーを共有する場合は，CORBA または生成されたサーバーファイルに依存しないサーバークラスを使うことも可能です。

スケルトンクラスの派生でないオブジェクトをエクスポーズする場合，アプリケーションは委任モデルを使用する必要があります。委任モデルでは，スケルトンクラスのインスタンスが CORBA オブジェクトを直接実装せず，インターフェースクラスを独立した実装クラスに渡します。つまり，実装クラスはスケルトンから呼び出されるだけで，スケルトンの派生クラスではありません。

委任モデルを使うには，IDL ファイルのコンパイル時に Tie クラスを生成する必要があります。IDL ファイルをコンパイルする前に，［プロジェクト｜オプション］を選択し，［プロジェクトオプション］ダイアログにある［CORBA］ページで Tie クラスを生成するボックスをチェックします。

IDL ファイルをコンパイルするとき，サーバーファイルにはスケルトンクラスの他に特別な Tie クラスが含まれています。この Tie クラスは，実装クラスのプロクシーの機能を果たし，そのメソッド実装を実装クラスに委任します（呼び出しを渡します）。

CORBA オブジェクトウィザードを使って実装クラスを定義するには，［委任（Tie）］ボタンを選択し，実装モデルがスケルトンクラスの派生クラスとして作成されないようにします。CORBA オブジェクトウィザードは，クラス定義の隣にコメントを追加します。このコメントを削除してはなりません。C++Builder はこのコメントによって，そのクラスを CORBA 実装クラスとして識別します。

［Tie］ボタンを選択すると，CORBA オブジェクトウィザードは実装クラスの生成方法を変更し，さらにオブジェクトをインスタンス化する自動生成されたコードも変更します。

サーバーアプリケーションが実装クラスをインスタンス化する場合は，実装クラスのインスタンスを引数としてコンストラクタに渡して，サーバーアプリケーションは関連する Tie クラスをインスタンス化する必要があります。起動時に実装クラスをインスタンス化する場合は，そのためのコードが CORBA オブジェクトウィザードによって自動的に生成されます。しかし，CORBA オブジェクトを実行時に動的にインスタンス化する場合は，手動でこのコードを追加する必要があります。次に例を示します。

```
MyObjImpl myobj(); // 実装オブジェクトのインスタンス化
_tie_MyObj<MyObjImpl> tieobj(myobj, "InstanceName");
```

変更の表示と編集

プロジェクトに変更を加える前に，その変更をプレビューし編集するように心がけてください。これには 2 つの利点があります。

- プロジェクトに対して行われるすべての変更を確認できます。C++Builder が内部的に使用するので開発者は関知しなくてもよい，自動的に生成されるコードは表示されません。
- これらの変更は，実装ファイル内を検索しなくてもカスタマイズできます。これは，コード量が多いファイルに変更を加える場合には特に効果的です。

変更を表示し編集するには，［プロジェクトの更新］ダイアログボックスを使います。このダイアログボックスを表示するには，いずれかの CORBA ウィザードから［更新の表示］を選択するか，［編集｜CORBA インプリメンテーションの更新］を選択します。

［プロジェクトの更新］ダイアログの左側のペインに変更が表示されます。変更を取り消すには，その項目の左にあるボックスのチェックを外します。変更を取り消した場合，それはプロジェクトに反映されないので，手動で同じコードを書く必要があります。

メモ 変更が行われた順に一覧表示されます。1 つの変更を削除すると，それに依存する別の変更も削除される場合があります。たとえば，新規ファイルの作成を削除すると，そのファイルへコードを追加する変更も削除されます（ボックスのチェックが外れます）。

変更リストを上下にスクロールすると，コードペインで追加されたコードが表示されます。そのコードを追加したファイルが，コードペインの一番下にリストされます。

変更をカスタマイズしたい場合は，コードペインを編集できます。たとえば，既存のクラスの動作を継承するように，実装クラスに上位クラスを追加できます。また，自動的に生成された未実装のメソッドを記述して，メソッドを実装するコードを追加することもできます。

自動的に生成された変更の表示と編集が終わったら，［OK］を選択します。これにより，編集内容を含むすべての変更がプロジェクトに追加されます。変更を破棄する場合は，ダイアログボックスの［キャンセル］を選択します。

CORBA オブジェクトを実装する

委任モデルを使用しない場合，実装クラスは IDL ファイルをコンパイルしたときに自動的に生成されたスケルトンクラスからの派生クラスとなります。委任モデルを使用した場合は，実装クラスに明示的な上位クラスはありません。このクラス定義をサーバーアプリケーションの他のクラスから継承するように変更することは可能ですが，既存の継承をスケルトンクラスから削除するのは避けてください。

IDL ファイル account.idl から，次の宣言を行う場合を考えてみてください。

```
interface Account {
  float balance();
};
```

IDL ファイルを（Tie クラスを使わずに）コンパイルすると，生成されたサーバーヘッダー（account_s.hh）には，インターフェースの各メソッドに対して純粋仮想メソッドを持つスケルトンクラスの宣言が含まれます。

CORBA オブジェクトウィザードにより，スケルトンクラス（_sk_Account) から派生した実装ユニットが作成されます。そのヘッダーファイルは次のようになります。

```
#ifndef Interface1ServerH
#define Interface1ServerH
#include "account_s.hh"
//---------------------------------------------------------------------------
class AccountImpl: public _sk_Account
{
  protected:
  public:
    AccountImpl(const char *object_name=NULL);
    CORBA::Float balance();
};

#endif
```

データメンバーまたはメソッドを，実装に必要なクラス定義に追加する場合があります。次に例を示します。

```
#ifndef Interface1ServerH
#define Interface1ServerH
#include "account_s.hh"
//---------------------------------------------------------------------------
class AccountImpl: public _sk_Account
{
  protected:
    CORBA::Float _bal;
  public:
    void Initialize(CORBA::Float startbal); // クライアントでは使用不可
    AccountImpl(const char *object_name=NULL);
    CORBA::Float balance();
};

#endif
```

これらの追加されたメソッドとプロパティは，サーバーアプリケーションからは使用可能ですが，クライアントアプリケーションにはエクスポーズされません。

生成された .cpp ファイルで，CORBA オブジェクトが動作するように，実装クラスの本文を記述します。

```
AccountImpl::AccountImpl(const char *object_name):
    _sk_Account(object_name)
{
}

CORBA::Float AccountImpl::balance()
{
  return _bal;
};

void Initialize(CORBA::Float startbal) // クライアントでは使用不可
{
  _bal = startbal;
}
```

メモ CORBA サーバーから BOA と対話するコードを記述することができます。たとえば，BOA を使うと，サーバーオブジェクトを一時的に隠したり非アクティブにし，後で再度アクティブにすることができます。BOA と対話するためのコードを記述する方法の詳細については，『VisiBroker Programmer's Guide』を参照してください。

スレッドの衝突に対する保護

デフォルトでは，CORBA アプリケーションはマルチスレッドです。このため，CORBA オブジェクトを実装するときには，スレッド衝突から保護する必要があります。

明示的に指定しない限り，BOA はクライアントスレッドをプールします。つまり，クライアント要求は存在するすべてのスレッドを使用できます。ただし，セッションごとのスレッド規則を付けて BOA を起動すれば，各クライアントが常に同じスレッドを使用するように指定することは可能です。各クライアントが常に同じスレッドを使用する場合は，スレッド変数に持続的なクライアント情報を格納できます。BOA 起動時にスレッド規則を設定する方法については，『VisiBroker Programmer's Guide』を参照してください。

VisiBroker ライブラリには，スレッド衝突からコードを保護するクラスが含まれています。これらは，ヘッダーファイル vthread.h で定義されています。このファイルは，VisiBroker インクルードディレクトリにインストールされています。VisiBroker スレッドサポートクラスには，ミューテックス（VisMutex），条件変数（VISCondition），および読み取り / 書き込みロックがあります。読み取り / 書き込みロックは，VCL の複数読み取り時の排他書き込みシンクロナイザ（VISRWLock）と同じような働きをします。これらのクラスを使用する利点は，VisiBroker をサポートするすべてのプラットフォームに移植可能なことです。

たとえば，インスタンスデータを保護するには，実装クラスに VisMutex を追加します。

```
class A {
  VISMutex _mtx;
  ...
}
```

こうすると，インスタンスデータと同期アクセスするどのメソッド内でも，ミューテックスをロックできるようになります。

```
void A::AccessSharedMemory(...)
{
  VISMutex_var lock(_mtx);  // 終了時に解放したロックを取得
  // ここにインスタンスデータにアクセスするためのコードを追加する
}
```

ミューテックスは AccessSharedMemory が実行を完了した時点で解放されますが，これは AccessSharedMemory 内で例外が発生した場合でも変わらない点に注意してください。

サーバーアプリケーションで VCL ライブラリを使用する場合，C++Builder にはスレッド衝突から保護するためのクラスがいくつか用意されています。詳細については，第 11 章「マルチスレッドアプリケーションの作成」を参照してください。

CORBA インターフェースを変更する

CORBA オブジェクトウィザードを使って実装クラスを生成した後で，その IDL ファイルのインターフェースに変更を加える場合，実装クラスの記述に開発者が費やした作業を無駄にすることなく C++Builder は自動的にサーバープロジェクトを更新し，開発者が加えた変更を反映させます。

IDL ファイルを変更した後，［編集｜ CORBA インプリメンテーションの更新］を選択します。このコマンドは，IDL ファイルを再コンパイルして，自動的に生成されたクライアントとサーバーのファイルに，変更されたインターフェース定義と現在のコンパイラオプションを反映させます。IDL ファイルのコンパイルが正常に完了すると，［プロジェクトの更新］ダイアログを使って，C++Builder が行った変更をプレビューし，編集できるようになります。

メモ C++Builder が既存の実装クラスに対応するインターフェースを見つけられなかった場合は，開発者がインターフェースの名前を変更したかどうかを確認するメッセージが表示されます。［はい］を選択すると，実装クラスとインターフェースを一致させるため，変更したインターフェース名を入力するよう促されます。

［CORBA インプリメンテーションの更新］コマンドを使うと，新規メソッドが実装クラスに追加され，既存のメソッドと属性に加えた変更が反映されるように宣言が更新されます。ただし，変更の種類によっては更新されない場合もあるので注意が必要です。たとえば，メソッドや属性を削除しても，そのコードは削除されず，クラス内に残ってしまいますが，CORBA クライアントからは削除されたメソッドを使うことができません。同様に，メソッドや属性の名前を変更すると，削除と追加を行ったとみなされ，旧メソッドまたは属性が残ったまま，新しいメソッドまたは属性のコードが生成されます。

サーバーインターフェースを登録する

サーバーオブジェクトへのクライアント呼び出しが静的バインディングだけを使用する場合は，サーバーインターフェースの登録は必ずしも必要ありませんが，登録することを推奨します。インターフェースを登録できるユーティリティには，以下の 2 つがあります。

- **インターフェースリポジトリ。**インターフェースリポジトリを使って登録すると，クライアントは動的起動インターフェース（DII）使用時に，インターフェースに関する情報を取得するようなプログラムを書くことができます。DII の利用についての詳細は，31-15 ページの「動的起動インターフェースの使い方」を参照してください。インターフェースリポジトリを利用した登録は，ほかの開発者がクライアントアプリケーションを作成するときにインターフェースを参照することができる便利な手段です。

  インターフェースリポジトリにインターフェースを登録するには，［ツール｜ IDL リポジトリ］を選択します。

- **オブジェクト起動デーモン。**オブジェクト起動デーモン（OAD）に登録すると，サーバーを起動する必要がなくなり，またクライアントが必要とするまでオブジェクトをインスタンス化する必要がなくなります。これはユーザーのサーバーシステム上のリソースを節約します。

CORBA クライアントの作成

CORBA クライアントを作成する場合，まずはじめにクライアントアプリケーションがクライアントマシン上の ORB ソフトウェアと確実に対話できるようにする必要があります。そのためには，CORBA クライアントウィザードを使います。［ファイル｜新規作成｜その他］を選択し，［新規作成］ダイアログの［多層サポート］ページから［CORBA クライアント］を選択します。CORBA クライアントウィザードでは，コンソールアプリケーションと Windows アプリケーションのどちらを作成するかを指定できます。

ここでは，作成する CORBA クライアントアプリケーションが VCL クラスを使用するかどうかを指定できます。VCL チェックボックスをチェックしないと，生成したすべてのコードを他のプラットフォームに移植できます。

CORBA クライアントウィザードで，使用するサーバーオブジェクトのインターフェースを定義した既存の IDL ファイルをすべてインクルードします。生成したクライアントユニットをプロジェクトに明示的に追加すれば，IDL ファイルをインクルードしなくても CORBA クライアントアプリケーションは作成できますが，この方法は推奨しません。プロジェクトにサーバーインターフェースの IDL ファイルがインクルードされている場合は，CORBA オブジェクトウィザードを使ってオブジェクトをサーバーにバインドできます。

メモ CORBA クライアントプロジェクトの作成時に IDL ファイルを追加しなくても，［プロジェクト｜プロジェクトに追加］を選択すれば後で IDL ファイルを追加できます。

CORBA クライアントウィザードは，常に指定した種類の新規クライアントプロジェクトを作成し，CORBA ライブラリをプロジェクトファイルに追加し，次の起動コードを追加して ORB（Object Request Broker）を初期化します。

```
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
```

コールバックインターフェースを CORBA サーバーに渡す場合は，クライアントアプリケーションの BOA（Basic Object Adaptor）も初期化する必要があります。そのためには，ウィザードの該当するボックスをチェックします。

次に，C++Builder のほかのアプリケーションを作成したときと同じ方法でアプリケーションの作成を進めます。ただし，サーバーアプリケーション内に定義されているオブジェクトを使用したい場合には，オブジェクトインスタンスを直接操作しません。その代わりに，CORBA オブジェクトへの参照を取得して操作します。CORBA オブジェクトの参照を取得するには，静的バインディング（アーリーバインディング）を使用する方法と動的バインディングを使用する方法の 2 つがあります。

静的バインディングを使用するには，［編集｜ CORBA オブジェクトを使う］を選択して CORBA オブジェクトウィザードを起動します。静的バインディングは動的バインディングを使用するよりも実行が速く，コンパイル時型チェックやコード補完などの利点があります。

ただし，実行時までどのオブジェクトまたはインターフェースを使用するのかわからない場合があります。このような場合には，動的バインディングを利用できます。動的バインディングは，Any という CORBA 固有の型を使って要求をサーバーに渡す汎用 CORBA オブジェクトを使います。

スタブの使い方

スタブクラスは，IDL ファイルをコンパイルすると自動的に生成され，同時に生成される BaseName_c.cpp と BaseName_c.hh というフォーム名の付いたクライアントファイルで定義されます。

メモ C++Builder では，［プロジェクトオプション］ダイアログの［CORBA］ページを使って，サーバーファイルを作成せずにクライアント（スタブ）ファイルのみをビルドするように指定することができます。

CORBA クライアントを作成するときには，生成したクライアントファイル内のコードを編集しません。その代わり，スタブクラスを使うときにインスタンス化します。これを行うには，［編集｜ CORBA オブジェクトを使う］を選択して CORBA オブジェクトウィザードを起動します。

CORBA オブジェクトウィザードで，インターフェースが記述された IDL ファイルを指定し，使用するインターフェースを選択します。CORBA オブジェクトの特定の名前の付いたインスタンスのみをバインドしたい場合は，［オブジェクト名］ボックスにその名前を入力します。

CORBA オブジェクトウィザードでは，サーバーオブジェクトにバインドする方法を選択できます。

- クライアントアプリケーションが VCL 対応の Windows アプリケーションの場合は，CORBA オブジェクトのスタブクラスのインスタンスを保持するプロパティをアプリケーションのフォームに作成できます。このプロパティは，CORBA サーバーオブジェクトのインスタンスと同じように使用できます。
- コンソールアプリケーションを作成する場合は，ウィザードによりアプリケーションの main() 関数でスタブクラスを変数としてインスタンス化できます。同様に，Windows アプリケーションを作成する場合は，WinMain() 関数でスタブクラスを変数としてインスタンス化できます。
- どちらの種類のアプリケーションを作成する場合でも，ウィザードを使うと，指定したユニットにある既存のクラスにプロパティを追加することも，スタブのインスタンスをプロパティとして持つ新しいクラスを作成することもできます。

どの方法を使っても，ウィザードは，必要なヘッダーファイルを追加し，スタブ変数またはプロパティを CORBA サーバーオブジェクトにバインドするコードを生成します。たとえば，次のコード

は，コンソールアプリケーションの main() 関数の中で MyServerObj という名前のサーバーインターフェース用スタブをインスタンス化します。

```
#include <corba.h>
#include <condefs.h>
#include "MyServerObj_c.hh"
#pragma argsused
int main(int argc, char* argv[])
{
  try
  {
    // ORB を初期化
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    MyServerObj_var TheObject = MyServerObj::_bind("InstanceName");
  }
  catch(const CORBA::Exception& e)
  {
    cerr << e << endl;
    return(1);
  }
  return 0;
}
```

注意 サーバーオブジェクトをアプリケーションの他の部分にバインドするコードをコピーする場合は，スタブ変数がローカル変数またはクラスのデータメンバーであることを確認してください。グローバルスタブ変数は，orb 変数が解放される前に CORBA 参照カウントを解放しないことがあるので注意が必要です。グローバルスタブ変数を使わなければならない場合は，orb が解放される前に，必ずグローバルスタブ変数を NULL に設定してバインドを解除します。

生成されたクライアントユニット（BaseName_c）しかクライアントアプリケーションに含まれていない場合は，CORBA オブジェクトウィザードを使ってサーバーオブジェクトをバインドすることはできません。その場合は，手動でコードを記述してください。VisiBroker のバインドオプションを使って，独自のバインドコードを記述したい場合があります。たとえば，特定のサーバーを指定し，CORBA 接続が解除されたときに自動的にサーバーへの再バインドを行う機能を使用不可にする場合などです。サーバーオブジェクトにバインドするためのコードの記述についての詳細は，VisiBroker のドキュメントを参照してください。

動的起動インターフェースの使い方

動的起動インターフェース（DII）を使用すると，インターフェース呼び出しを明示的にマーシャリングするスタブクラスを使用せずにクライアントアプリケーションがサーバーオブジェクトを呼び出せます。DII は，クライアントが要求を送る前にすべての型情報をエンコードし，その情報をサーバーでデコードする必要があるので，スタブクラスを使う場合よりも実行速度が遅くなります。

クライアントアプリケーションで DII を使うための手順を以下に示します。

1. 汎用 CORBA オブジェクトを作成します。このオブジェクトは，要求をマーシャリングするためのコードが実装に含まれない点以外は，スタブオブジェクトと同じような役割を果たします。オブジェクトのリポジトリ ID を指定し，どのオブジェクトが DII 要求を受け取るかを示す必要があります。

```
CORBA::Object_var diiObj;
try
{
  diiObj = orb->bind("IDL:ServerObj:1.0");
} catch (const CORBA::Exception& E)
{
  // バインドの例外処理
}
```

2. 次に，汎用 CORBA オブジェクトを使って，汎用 CORBA オブジェクトの特定のメソッドに対する要求オブジェクトを作成します。

```
CORBA::Request_var req = diiObj->_request("methodName");
```

3. 次に，メソッドが取る引数をすべて追加します。引数は，CORBA::NVList_ptr 型の要求の引数リストに追加されます。引数リストの各引数は Variant と類似した CORBA::Any 型を取ります。引数は必ず CORBA::Any 型でなければなりません。これは，名前付きの値リスト（CORBA::NVList）は，どのような型の値を含むかわからない，引数リストを処理する必要があるからです。

```
CORBA::Any arg;
arg <<= (const char *)"argvalue";
CORBA::NVList_ptr arguments = req->arguments();
arguments->add_value("arg1", arg, CORBA::ARG_IN);
```

4. 要求を生成します。

```
req->invoke();
```

5. 最後にエラーをチェックし，結果を取得します。

```
if (req->env()->exception())
  // 例外処理
else
{
  CORBA::Any_ptr pRetVal = req->result()->value();
  CORBA::float val;
  Any_ptr >>= val;
}
```

動的起動インターフェースの使い方についての詳細は，『VisiBroker Programmer's Guide』を参照してください。

CORBA サーバーのテスト

C++Builder には CORBATest デモプログラムも入っています。これをビルドすると，サーバーのインターフェースをテストに使用できるユニバーサルクライアントが生成されます。CORBATest は，CORBA サーバーアプリケーションのオブジェクトをテストするスクリプトを記録し実行します。

メモ CORBATest を使って多数のサーバーをテストする場合は，［ツール］メニューにインストールして使用してください。インストールするには，［ツール｜ツールの設定］を選択し，［ツールオプション］ダイアログボックスで CorbaTest.exe を追加します。

テストツールのセットアップ

CORBATest を使って CORBA アプリケーションをテストするには，テストするサーバーのインターフェースを，実行中のインターフェースリポジトリに登録する必要があります。登録する手順は次のとおりです。

1. ［ツール｜ Visibroker SmartAgent］にチェックマークが付いていない場合は，このメニュー項目を選択して Visibroker SmartAgent を実行します。
2. テストするサーバーアプリケーションを実行します。
3. ［ツール｜ IDL リポジトリ］を選択して，テストするサーバーアプリケーションの .idl ファイルをインターフェースリポジトリに追加します。実行中のインターフェースリポジトリならばどれでもテストに使用できます。しかし，専用のテスト用リポジトリを使用して，テストが他のインターフェースリポジトリに影響を与えないようにすることもできます。専用のテストリポジトリを起動するには，［ツール｜ IDL リポジトリ］を選択して，［IDL リポジトリの更新］ダイアログの［IREP の追加］ボタンを選択します。
4. テストツールを起動するには，corbatest ディレクトリ（.. \ examples \ corba \ corbatest）にあるデモコードから CORBATest.exe をビルドして実行します。

テストスクリプトの記録と実行

テストスクリプトとは，呼び出される一連の CORBA オブジェクトメソッドとメソッドに渡されるパラメータを記述したものです。1 つのスクリプトに同一のメソッドの複数の呼び出しを記述することができます。たとえば，異なるパラメータ値を代入してメソッドをテストする場合などです。

スクリプトをビルドするには，テストツールの上部中央にあるコマンドペインにコマンドを追加します。コマンドを追加する手順は，次のとおりです。

1. メソッドを呼び出すオブジェクトのオブジェクトペイン（上部左）にタブ付きのページがない場合は，［Edit ｜ Add Object］を選択します。［New Object］ダイアログボックスが表示されます。ここで，リポジトリ ID によりオブジェクトを指定します。オブジェクトのリポジトリ ID は「IDL:MyInterface1:1.0」の形式の文字列です。「MyInterface」には，.idl ファイルで宣言したインターフェース名が入ります。オブジェクトを選択し，名前を付けます。
2. 上部左側のペインでは，メソッドを追加したいオブジェクトのタブを選択します。各タブには，オブジェクトを追加したとき（手順 1）に付けたオブジェクト名のラベルが付いています。各タブページには，テスト可能なオブジェクトの動作（メソッド）が表示されています。
3. テストツールの上部中央にあるコマンドペインにオブジェクトペインから動作をドラッグし，コマンドをスクリプトに追加します。
4. コマンドペインのメソッドを選択すると，上部右側の詳細ペインに入力パラメータが表示されます。入力パラメータに値を入力します。ステータスバーには，現在の入力パラメータに必要な値の型が表示されます。

上記の手順を繰り返し，オブジェクトにコマンドを必要なだけ追加します。スクリプトに必要なコマンドがすべて入ったら，［File ｜ Save Script As］を選択するか［Save Script］ボタンを選択して，ス

クリプトに名前を付けて保存します。新規スクリプトを開始するには，［File ｜ New Script］を選択するか［New Script］ボタンを選択します。

メモ スクリプトに加えたオブジェクトまたはコマンドを削除するには，そのオブジェクトまたはコマンドを選択し，［Edit ｜ Remove Command］を選択します。

スクリプトを実行するには，［Run ｜ Run Script］を選択するか，［Run Script］ボタンを選択します。スクリプトを実行すると，テストツールは動的起動インターフェースを使って CORBA サーバーを呼び出し，指定されたパラメータ値を渡します。すべてのメソッドについて，戻り値（存在する場合）と出力パラメータ値がテストツールの結果セクションに表示されます。自動テストの場合は，結果ファイルに保存されます。

結果セクションには，結果タブにあるスクリプトのすべてのコマンドの実行結果が表示されます。この結果は，結果セクションの別のページにも，必要な情報を見分けやすいようにカテゴリ別に分類されて表示されます。これらのページには，戻り値，入力パラメータ値，出力パラメータ値，およびスクリプトの実行中に発生したエラーが表示されます。

第32章

インターネットサーバーアプリケーションの作成

Web サーバーアプリケーションは既存の Web サーバーの機能を拡張します。Web サーバーアプリケーションは Web サーバーから HTTP リクエストメッセージを受信し，メッセージで要求されたアクションを実行し，レスポンスを作成して Web サーバーに返します。C++Builder アプリケーションで実行できる操作の多くは Web サーバーアプリケーションに組み込むことができます。

C++Builder では，Web サーバーアプリケーションを開発するためのアーキテクチャは，WebBroker と WebSnap の 2 種類が用意されています。WebSnap と WebBroker は異なるアーキテクチャですが，共通する要素が多数あります。WebSnap アーキテクチャは，WebBroker のスーパーセットとして機能します。WebSnap アーキテクチャは追加のコンポーネントを提供し，開発者がアプリケーションを実行せずにページのコンテンツを表示できる［プレビュー］タブなどの新機能も提供します。WebSnap を使用して開発されたアプリケーションには WebBroker コンポーネントを含めることができますが，WebBroker を使用して開発されたアプリケーションには WebSnap コンポーネントを含めることができません。

この章では，WebBroker と WebSnap テクノロジーの機能について説明し，インターネットベースのクライアント / サーバーアプリケーションについての一般的な情報を提供します。

WebBroker と WebSnap とは

アプリケーションの機能の 1 つは，データをユーザーからアクセスできるようにすることです。標準の C++Builder アプリケーションでは，ダイアログやスクロールするウィンドウなどの従来のフロントエンド要素を作成することによってこれを実現します。開発者は，使い慣れた C++Builder のフォーム設計ツールを使って，これらのオブジェクトの正確な配置を指定することができます。しかし，Web サーバーアプリケーションはこれとは異なる方法で設計しなければなりません。ユーザーに渡る情報はすべて，HTTP を介して転送される HTML ページの形式になっていなければなりません。一般的にページは，クライアントマシン上の Web ブラウザアプリケーションによって解釈され，Web ブラウザはそのときの状態としてユーザーの特定のシステムに適切な形式でページを表示します。

Web サーバーアプリケーションを構築する際にまず行うことは，使用するアーキテクチャとして WebBroker か WebSnap を選択することです。どちらのアーキテクチャを使用しても，以下のような同じ機能が多数提供されます。

- CGI および Apache DSO の 2 種類の Web サーバーアプリケーションのサポート。これらについては，32-6 ページの「Web サーバーアプリケーションの種類」で説明しています。
- 受信したクライアントリクエストが別個のスレッドで処理されるようにするためのマルチスレッドのサポート。
- レスポンスの高速化のための Web モジュールのキャッシング。
- クロスプラットフォーム開発。作成した Web サーバーアプリケーションを，Windows および Linux オペレーティングシステム間で簡単に移植できます。ソースコードはどちらのプラットフォームでもコンパイルされます。

WebBroker および WebSnap コンポーネントは，ページ転送のすべての処理を行います。WebSnap は WebBroker を土台として使用するので，WebSnap には WebBroker のアーキテクチャの機能がすべて組み込まれています。ただし，WebSnap はより強力なページ生成用のツールを備えています。また WebSnap アプリケーションでは，サーバー側スクリプティングを使って，実行時にページ生成を操作することができます。WebBroker には，このスクリプティングの機能はありません。WebBroker が提供するツールは，WebSnap ほど完備されていず，WebSnap ほど直感的でもありません。新規に Web サーバーアプリケーションを開発する場合は，WebBroker ではなく WebSnap を選択した方がよいでしょう。

次の表に両者の主な違いを示します。

表 32.1　WebBroker と WebSnap の比較

WebBroker	WebSnap
下位互換性あり	WebSnap アプリケーションは，コンテンツを作成する WebBroker コンポーネントを使用できるが，新たに Web モジュールとディスパッチャも WebBroker コンポーネントを使用できるようになった
1 つのアプリケーションでは 1 つの Web モジュールのみが使用できる	複数の Web モジュールによってアプリケーションをユニットに区分できるので，複数の開発者が同じプロジェクトの作業をしたときの衝突が減る
アプリケーションでは 1 つの Web ディスパッチャのみが使用できる	複数の特定用途向けディスパッチャが，異なる種類のリクエストを処理する
コンテンツを作成する専用のコンポーネントには，ページプロデューサ，InternetExpress コンポーネント，Web サービスコンポーネントなどがある	WebBroker アプリケーションに表示できるすべてのコンテンツプロデューサ，および複雑なデータ駆動型 Web ページをすばやく構築するために設計されたその他多数の機能をサポートする
スクリプトはサポートしない	サーバー側スクリプトをサポートするので，HTML 生成ロジックをビジネスロジックから分離できる
名前付きページの組み込みサポートがない	名前付きページは，ページディスパッチャによって自動的に取得でき，サーバー側スクリプトから処理できる
セッションをサポートしない	短時間のあいだ必要なエンドユーザーについての情報をセッションが格納する。これはログイン / ログアウトのサポートなどのタスクに使用できる

表 32.1　WebBroker と WebSnap の比較（つづき）

WebBroker	WebSnap
アクション項目または自動ディスパッチコンポーネントを使用して，あらゆるリクエストを明示的に処理しなければならない	ディスパッチコンポーネントがさまざまなリクエストに自動的に応答する
作成するコンテンツのプレビューを提供するコンポーネントは，いくつかの専用コンポーネントだけである。大部分の開発は視覚的でない	WebSnap を使用すると，Web ページを視覚的に構築でき，設計時に結果を表示できる。あらゆるコンポーネントでプレビューが使用できる

WebBroker についての詳細は，第 33 章「WebBroker の使い方」を参照してください。WebSnap についての詳細は，第 34 章「WebSnap を使用しての Web サーバーアプリケーションの作成」を参照してください。

用語と標準

インターネット上の動作を制御するプロトコルの多くは，RFC（Request for Comment）ドキュメントで定義されています。このドキュメントは，インターネットプロトコルのエンジニアリングと開発の部門である IETF（Internet Engineering Task Force）が作成，更新，管理を行っています。インターネットアプリケーションを作成するときに役立つ主な RFC には以下のものがあります。

- RFC822『Standard for the format of ARPA Internet text messages』。メッセージヘッダーの構造，およびその内容について説明しています。
- RFC1521『MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies』。マルチパートやマルチフォーマットのメッセージをカプセル化して転送するための方法について説明しています。
- RFC1945『Hypertext Transfer Protocol - HTTP/1.0』。ハイパードキュメントを配布するために使う転送プロトコルについて説明しています。

IETF の Web サイト `www.ietf.cnri.reston.va.us` には RFC のライブラリがあります。

URL の構成要素

URL（Uniform Resource Locator）は，ネット上で利用できるリソースの位置を完全な形式で記述したものです。複数の要素から構成されており，そのそれぞれにアプリケーションがアクセスできます。図 32.1 にこれらの要素を示します。

図 32.1　URL の構成要素

最初の部分（厳密には URL の一部ではない）は，プロトコル（http）を識別します。http だけでなく，https（secure http），ftp などの別のプロトコルも指定できます。

ホスト部は，Web サーバーおよび Web サーバーアプリケーションを実行するマシンを識別します。上の図には示していませんが，ホスト部はメッセージ受信ポートをオーバーライドすることができます。ポート番号はプロトコルによって暗黙に定義されるので，通常はポートを指定する必要はありません。

スクリプト名は，Web サーバーアプリケーションの名前を指定します。ここで指定するアプリケーションは，Web サーバーがメッセージを渡す相手のアプリケーションです。

スクリプト名の後にパス情報が続きます。パス情報は，Web サーバーアプリケーション内でのメッセージの送り先を識別します。パス情報の値が指すものは，ホストマシン上のディレクトリ，特定メッセージに応答するコンポーネントの名前，あるいは，Web サーバーアプリケーションが受信メッセージ処理の分割に使用する何らかの別のメカニズムです。

問い合わせの部分には，名前の付いた値一式が入ります。この名前と値は Web サーバーアプリケーションで定義されます。

URI と URL

URL は HTTP 標準 RFC1945 で定義されている URI（Uniform Resource Identifier）のサブセットです。Web サーバーアプリケーションは，いくつものソースからコンテンツを生成することがよくあります。コンテンツの最終結果は特定の位置にあらかじめ配置されているわけではなく，必要に応じて作成されます。URI は，このような特定の位置にないリソースを記述できます。

HTTP リクエストヘッダー情報

HTTP リクエストメッセージには，クライアント，リクエストのターゲット，リクエストの処理方法，およびリクエストとともに送られるコンテンツに関する情報を示すヘッダーが含まれています。各ヘッダーは，「Host」などの文字列によって識別されます。たとえば，次の HTTP リクエストを考えてみましょう。

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

最初の行はリクエストを GET として識別します。GET リクエストメッセージは，単語 GET に続く URI（この場合は，/art/gallery.dll/animals?animal=dog&color=black）と関連付けられているコンテンツを返すように，Web サーバーアプリケーションに要求します。1 行目の最後の部分は，クライアントが HTTP 1.0 標準を使用していることを示しています。

2 行目は，Connection ヘッダーで，一度リクエストに対応したら接続を終了してはならないことを示しています。3 行目は，User-Agent ヘッダーで，リクエストを生成するプログラムに関する情報を提供します。次の行は，Host ヘッダーで，Host 名およびこの接続を確立するための接点となるサーバー上のポートを提供します。最後の行は，Accept ヘッダーで，有効なレスポンスとしてクライアントが受け取ることができる媒体の種類をリストします。

HTTP サーバー動作

Web ブラウザのクライアント / サーバーの特徴は見た目は単純です。ほとんどのユーザーは，簡単な手順で World Wide Web から情報を検索できます。リンクをクリックすると情報が画面に表示されます。より詳細な知識があるユーザーならば HTML 構文のほか，実際に使うプロトコルのクライアント / サーバーの特徴をある程度理解しています。ページ単位の単純な Web サイトコンテンツを作成する場合，通常はそれで十分です。もっと複雑な Web ページを作成する場合は，HTML を使って情報を自動的に収集して表示するために，さまざまな方法を利用できます。

Web サーバーアプリケーションを構築する前に，クライアントがどのようにリクエストを発行するか，サーバーがどのようにクライアントリクエストに応答するかを理解すると役立ちます。

クライアントリクエストの作成

HTML ハイパーテキストリンクを選択するかユーザーが URL を指定すると，ブラウザはプロトコル，指定されたドメイン，情報へのパス，日時，動作環境，ブラウザ自身などのコンテンツ情報を収集します。次にリクエストを作成します。

たとえば，フォームのボタンをクリックして選択された基準に基づいてイメージページを表示するために，クライアントは次のような URL を作成します。

```
http://www.TSite.com/art/gallery.dll/animals?animal=dog&color=black
```

この URL は，www.TSite.com ドメインの HTTP サーバーを指定します。クライアントは www.TSite.com の HTTP サーバーに接続し，リクエストをそのサーバーに渡します。次にリクエストの例を示します。

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

クライアントリクエストへの対処

Web サーバーはクライアントリクエストを受信すると，サーバーの環境設定に基づいてアクションをいくつでも実行できます。サーバーがリクエストの /gallery.dll 部をプログラムとして認識するように環境設定されている場合，リクエストに関する情報をプログラムに送ります。リクエストに関する情報をプログラムに渡す方法は，Web サーバーアプリケーションの種類によって異なります。

- プログラムが CGI（Common Gateway Interface）プログラムの場合，サーバーはリクエストの情報を CGI プログラムに直接渡す。サーバーはプログラムが終了するまで待機している。CGI プログラムは終了すると，コンテンツをサーバーに直接返す
- プログラムが WinCGI の場合，サーバーはファイルを開いてリクエスト情報を書き込む。次に WinCGI プログラムを実行し，その際に WinCGI プログラムに対しクライアント情報が入っているファイルの位置と WinCGI プログラムがコンテンツを作成するために使うファイルの位置を渡

す。サーバーはプログラムが終了するまで待機している。WinCGI プログラムが終了すると，このプログラムが書き込んだコンテンツファイルからサーバーがデータを読み出す

- プログラムがダイナミックリンクライブラリ（DLL）の場合，サーバーは必要であれば DLL をロードしてからリクエストの情報を構造体として DLL に渡す。サーバーはプログラムが終了するまで待機している。DLL は，終了するとコンテンツをサーバーに直接返す

どの場合も，プログラムはリクエストに基づいて動作し，プログラマが指定したアクションを実行します。アクションには，データベースへのアクセス，テーブル参照または計算，HTML ドキュメントの作成または選択などがあります。

クライアントリクエストへのレスポンス

Web サーバーアプリケーションはクライアントリクエストの処理が完了すると，HTML コードやその他の MIME コンテンツのページを作成し，サーバーを経由してそのページを表示用にクライアントに返します。レスポンスを返す方法もプログラムの種類によって異なります。

- WinCGI スクリプトは終了時に，HTML ページを作成し，それをファイルに書き込み，レスポンス情報を別のファイルに書き込み，この 2 つのファイルの位置をサーバーに返す。サーバーは両方のファイルを開き，HTML ページをクライアントに返す
- DLL は終了時に，HTML ページとすべてのレスポンス情報をサーバーに直接返す。サーバーはこれをクライアントに返す

Web サーバーアプリケーションを DLL として作成すると，個々のリクエストに対処するために必要な処理の数やディスクへのアクセス回数が減るので，システムの負荷やリソースの使用量を軽減できます。

Web サーバーアプリケーションの種類

WebBroker または WebSnap のどちらを使用する場合でも，5 種類の標準的な Web サーバーアプリケーションを作成できます。さらに，Web アプリケーションデバッガの実行形式ファイルを作成できます。このファイルでは，作成したアプリケーションに Web サーバーが統合されるのでアプリケーションロジックをデバッグできます。Web アプリケーションデバッガの実行形式ファイルは，デバッグのみを目的としています。アプリケーションを配布するときは，その他の 5 種類の 1 つに移行する必要があります。

ISAPI と NSAPI

ISAPI または NSAPI の Web サーバーアプリケーションは Web サーバーがロードする DLL です。クライアントリクエスト情報は構造体として DLL に渡されます。ISAPI/NSAPI アプリケーションはこの情報を評価して，適切なリクエストオブジェクトとレスポンスオブジェクトを作成します。それぞれのリクエストメッセージは自動的に別個の実行スレッドで処理されます。

CGI 実行形式

CGI 実行形式の Web サーバーアプリケーションは，標準入力でクライアントリクエスト情報を受信し，標準出力で結果をサーバーに返すコンソールアプリケーションです。CGI アプリケーションはこのデータを評価して，適切なリクエストオブジェクトとレスポンスオブジェクトを作成します。それぞれのリクエストメッセージは別個のアプリケーションインスタンスで処理されます。

WinCGI 実行形式

WinCGI 実行形式の Web サーバーアプリケーションは，サーバーが書き込んだ環境設定（INI）ファイルからクライアントリクエスト情報を受信し，サーバーがクライアントに返すファイルに結果を書き込む Windows アプリケーションです。Web サーバーアプリケーションは INI ファイルを評価して，適切なリクエストオブジェクトとレスポンスオブジェクトを作成します。それぞれのリクエストメッセージは別個のアプリケーションインスタンスで処理されます。

Apache

Apache の Web サーバーアプリケーションは Web サーバーがロードする DLL です。クライアントリクエスト情報は構造体として DLL に渡されます。Apache Web サーバーアプリケーションはこの情報を評価して，適切なリクエストオブジェクトとレスポンスオブジェクトを作成します。それぞれのリクエストメッセージは自動的に別個の実行スレッドで処理されます。

Apache Web サーバーアプリケーションを配布する場合は，アプリケーション固有の情報を Apache 環境設定ファイル内で指定する必要があります。デフォルトのモジュール名は，プロジェクト名の末尾に `_module` を付けたものになります。たとえば，プロジェクトの名前が Project1 ならば，モジュール名は `Project1_module` になります。同様に，デフォルトのコンテンツ型は，プロジェクト名に `-content` を付けたものになり，デフォルトのハンドラ型はプロジェクト名に `-handler` を付けたものになります。

これらの定義は，必要に応じてプロジェクトファイル（.bpr ファイル）内で変更できます。たとえば，プロジェクトを作成すると，デフォルトのモジュール名がプロジェクトファイルに保存されます。次に例を示します。

```
extern "C"
{
  Httpd::module __declspec(dllexport) Project1_module;
}
```

メモ 保存プロセス中にプロジェクトの名前を変更すると，その名前は自動的には変更されません。プロジェクトの名前を変更した場合は常に，プロジェクトファイル内のモジュール名を変更してプロジェクト名と一致させなければなりません。モジュール名を変更すると，コンテンツとハンドラの定義が自動的に変更されます。

Apache 環境設定ファイル内でのモジュール，コンテンツ，およびハンドラの定義の使い方については，Apache Web サイトの httpd.apache.org に掲載されているドキュメントを参照してください。

Web アプリケーションデバッガ

上に挙げたサーバーの種類は，実際の環境での長所と短所がありますが，デバッグに適したものはありません。アプリケーションの配布とデバッガの構成を行うと，Web サーバーアプリケーションのデバッグがほかのアプリケーションのデバッグよりもはるかに退屈な作業になりかねません。

幸い，Web サーバーアプリケーションのデバッグをそこまで複雑にする必要はありません。C++Builder には，デバッグを簡単にする Web アプリケーションデバッガが含まれています。Web アプリケーションデバッガは，開発用のマシン上で Web サーバーのように機能します。Web サーバーアプリケーションを Web アプリケーションデバッガの実行形式ファイルとして構築する場合は，構築プロセス中に自動的に配布が行われます。アプリケーションのデバッグを開始するには，［実行｜実行］を使用します。次に，［ツール｜ Web アプリケーションデバッガ］を選択し，デフォルトの URL をクリックし，表示された Web ブラウザ内でアプリケーションを選択します。そのアプリケーションがブラウザウィンドウ内で起動され，開発者は IDE を使用してブレークポイントを設定でき，デバッグ情報を取得できます。

アプリケーションをテストする準備ができたら，またはアプリケーションを実際の環境で配布する準備ができたら，次に説明する手順で Web アプリケーションデバッガプロジェクトを変換して，その他のターゲットタイプのいずれかにすることができます。

メモ Web アプリケーションデバッガプロジェクトを作成するときに，そのプロジェクトに CoClass 名を指定する必要があります。これは単に Web アプリケーションデバッガがそのアプリケーションを参照するために使用する名前です。大部分の開発者は，アプリケーションの名前を CoClass 名として使用します。

Web サーバーアプリケーションのターゲットタイプの変換

WebBroker と WebSnap の強力な機能の 1 つは，数種類のターゲットサーバーを提供することです。C++Builder では，ターゲットタイプ間での変換が容易にできます。

WebBroker と WebSnap は設計方針が多少異なるので，それぞれのアーキテクチャに対して異なる変換方法を使用しなければなりません。WebBroker アプリケーションのターゲットタイプを変換する手順は次のとおりです。

1. Web モジュールを右クリックして，［リポジトリに追加］を選択します。
2. ［リポジトリに追加］ダイアログボックスで，自分の Web モジュールに，タイトル，コメント，リポジトリページ（データモジュールが適切でしょう），作者名，およびアイコンを設定します。
3. ［OK］を選択して，自分の Web モジュールをテンプレートとして保存します。
4. メインメニューの［ファイル｜新規作成］を選択し，［新規作成］タブで［Web サーバーアプリケーション］を選びます。［Web サーバーアプリケーションの新規作成］で，適切なターゲットタイプを選びます。
5. 自動生成された Web モジュールを削除します。
6. メインメニューの［ファイル｜新規作成］を選択し，手順 3 で保存したテンプレートを選びます。テンプレートが手順 2 で指定したページに表示されます。

WebSnap アプリケーションのターゲットタイプを変換する手順は次のとおりです。

1. IDE 内でプロジェクトを開きます。
2. ［表示｜プロジェクトマネージャ］を選択してプロジェクトマネージャを表示します。プロジェクトを展開して，すべてのユニットを表示します。

3. プロジェクトマネージャで，［新規作成］ボタンをクリックして新しい Web サーバーアプリケーションプロジェクトを作成します。［WebSnap］タブで［WebSnap アプリケーション］をダブルクリックします。使用するサーバーの種類など，プロジェクトに必要なオプションを選択して［OK］をクリックします。
4. プロジェクトマネージャで，新しく作成したプロジェクトを展開します。表示されているファイルをすべて選択して削除します。
5. プロジェクトの各ファイル（Web アプリケーションデバッガプロジェクトのフォームファイルを除く）を，新しいプロジェクトに 1 つずつドラッグします。新しいプロジェクトへのファイルの追加を確認するダイアログが表示されたら，［はい］をクリックします。

サーバーアプリケーションのデバッグ

Web サーバーアプリケーションのデバッグについては，固有の問題があります。Web サーバーアプリケーションが Web サーバーからのメッセージに応じて実行されるからです。IDE からはアプリケーションを起動することはできません。こうすると Web サーバーがループから除外され，アプリケーションは待ち受けているリクエストメッセージを見つけることができなくなるからです。

この後の各トピックでは，Web サーバーアプリケーションのデバッグに使用できる方法について説明します。

Web アプリケーションデバッガの使い方

Web アプリケーションデバッガは，HTTP リクエスト，レスポンス，およびレスポンス時間を簡単に監視する方法を提供します。Web アプリケーションデバッガは，Web サーバーに取ってかわります。いったんデバッグしたアプリケーションは，サポートされる Web アプリケーションの種類の 1 つに変換し，市販の Web サーバーによってインストールできるようになります。

Web アプリケーションデバッガを使用するには，まず Web アプリケーションを Web アプリケーションデバッガの実行形式ファイルとして作成しなければなりません。WebBroker または WebSnap のどちらを使用する場合でも，Web サーバーアプリケーションを作成するウィザードでは，アプリケーションの作成を最初に開始するときにオプションとしてこれが含まれています。これによって COM サーバーでもある Web サーバーアプリケーションが作成されます。

WebBroker を使用してこの Web サーバーアプリケーションを作成する方法についての詳細は，第 33 章「WebBroker の使い方」を参照してください。WebSnap の使い方についての詳細は，第 34 章「WebSnap を使用しての Web サーバーアプリケーションの作成」を参照してください。

Web アプリケーションデバッガを使用してのアプリケーションの起動

開発した Web サーバーアプリケーションは，次の手順で実行およびデバッグできます。

1. 作成したプロジェクトを IDE にロードして，ほかの実行形式ファイルと同様にアプリケーションをデバッグできるようにブレークポイントを設定します。

2. ［実行｜実行］を選択します。作成した Web サーバーアプリケーションである COM サーバーのコンソールウィンドウが表示されます。初めてアプリケーションを実行したときに，COM サーバーが登録され，Web アプリケーションデバッガがアクセスできるようになります。
3. ［ツール｜ Web アプリケーションデバッガ］を選択します。
4. ［起動］ボタンをクリックします。デフォルトのブラウザに［ServerInfo］ページが表示されます。
5. ［ServerInfo］ページは，登録されたすべての Web アプリケーションデバッガ実行形式ファイルのドロップダウンリストを表示します。作成したアプリケーションをドロップダウンリストから選択します。該当するアプリケーションがこのドロップダウンリストに見つからない場合は，アプリケーションを実行形式ファイルとして実行してみてください。アプリケーションを登録するには，一度実行しなければなりません。それでもアプリケーションがドロップダウンリストに表示されない場合は，Web ページを更新してみてください（Web ブラウザがこのページをキャッシュしている場合，最新の変更内容が表示されない場合があります）。
6. 作成したアプリケーションをドロップダウンリストから選択したら，［実行］ボタンを押します。作成したアプリケーションが Web アプリケーションデバッガ内で起動され，このアプリケーションと Web アプリケーションデバッガとの間で渡されるリクエストとレスポンスのメッセージについての詳細が提供されます。

別の種類の Web サーバーアプリケーションへの変換

Web アプリケーションデバッガを使用して Web サーバーアプリケーションのデバッグを終えたら，市販の Web サーバーにインストールできる種類のアプリケーションに変換する必要があります。アプリケーションの変換方法についての詳細については，32-8 ページの「Web サーバーアプリケーションのターゲットタイプの変換」を参照してください。

DLL である Web アプリケーションのデバッグ

ISAPI アプリケーション，NSAPI アプリケーション，および Apache アプリケーションは，実際にはエントリポイントが定義済みの DLL です。Web サーバーは，これらのエントリポイントを呼び出して，リクエストメッセージをアプリケーションに渡します。これらのアプリケーションは DLL なので，サーバーを起動するアプリケーションの実行パラメータを設定することでデバッグできます。

アプリケーションの実行時引数を設定するには，［実行｜実行時引数］を選択し，［ホストアプリケーション］と［実行時の引数］を設定して，Web サーバー用の実行形式ファイルの指定，およびこれを起動するときに必要な引数を指定します。Web サーバー上でのこれらの値についての詳細は，Web サーバーベンダーが提供するマニュアルを参照してください。

メモ Web サーバーによっては，この方法でホストアプリケーションを起動する権限を得るにはさらに変更が必要な場合があります。詳細については，ご使用の Web サーバーのベンダーにお問い合わせください。

ヒント Windows 2000 と IIS 5 を使用する場合，適切に権限を設定するための変更についての詳細は，次の Web サイトで説明されています。

```
http://community.borland.com/article/0,1410,23024,00.html
```

［ホストアプリケーション］と［実行時の引数］を設定したら，ブレークポイントを設定できます。そうすれば，サーバーがリクエストメッセージを DLL に渡したときに，ブレークポイントの 1 つに達し，通常どおりにデバッグできるようになります。

メモ アプリケーションの実行時引数を使って Web サーバーを起動するときには，そのサーバーがまだ実行されていないことを確認してください。

DLL のデバッグに必要なユーザー権利

Windows で DLL をデバッグするためには，適切なユーザー権利を持っていなければなりません。このユーザー権利を取得する手順は次のとおりです。

1. コントロールパネルの［管理ツール］で，［ローカルセキュリティポリシー］をクリックします。［ローカルポリシー］を展開し，［ユーザー権利の割り当て］をダブルクリックします。右側のパネルでオペレーティングシステムの一部として［機能］をダブルクリックします。
2. ［追加］を選択してユーザーをリストに追加します。現在のユーザーを追加します。
3. 変更内容を有効にするために再起動します。

第33章

WebBroker の使い方

（コンポーネントパレットの［Internet］タブにある）WebBroker コンポーネントを使うと，特定の URI（Uniform Resource Identifier）に関連付けられたイベントハンドラを作成できます。プログラム上で HTML または XML ドキュメントを作成し，処理が完了したときにクライアントに転送することもできます。WebBroker コンポーネントはクロスプラットフォームアプリケーションの開発に使用できます。

Web ページのコンテンツはデータベースから取り出されることがよくあります。インターネットコンポーネントを使うとデータベースへの接続を自動的に管理できるので，1 つの DLL で複数のスレッドセーフなデータベース接続を同時に処理できます。

この章の以降の節では，WebBroker コンポーネントを使用して Web サーバーアプリケーションを作成する方法を説明します。

WebBroker による Web サーバーアプリケーションの作成

WebBroker アーキテクチャを使用して新しい Web サーバーアプリケーションを作成する手順は次のとおりです。

1. ［ファイル｜新規作成｜その他］を選択します。
2. ［新規作成］ダイアログボックスで，［新規作成］タブを選択し，［Web サーバーアプリケーション］を選択します。
3. このダイアログボックスでは，次の種類の Web サーバーアプリケーションを選択できます。
   - ISAPI/NSAPI：この種類のアプリケーションを選択すると，Web サーバーで必要なエクスポートされたメソッドを持つ DLL としてプロジェクトが設定される。また，アプリケーションを生成する適切なヘッダーファイルも含まれる

- Apache：この種類のアプリケーションを選択すると，Apache Web サーバーで必要なエクスポートされたメソッドを持つ DLL としてプロジェクトが設定される。また，アプリケーションを生成する適切なヘッダーファイルも含まれる
- CGI 実行形式：この種類のアプリケーションを選択すると，プロジェクトがコンソールアプリケーションとして設定され，適切なヘッダーファイルが含まれる
- WinCGI 実行形式：この種類のアプリケーションを選択すると，プロジェクトが Windows アプリケーションとして設定され，適切なヘッダーファイルが含まれる
- Web アプリケーションデバッガ実行形式ファイル：この種類のアプリケーションを選択すると，Web サーバーアプリケーションの開発およびテストのための環境が設定される。この種類のアプリケーションは配布を目的としていない

Web サーバーアプリケーションを選択するときは，開発者のアプリケーションが使う Web サーバーの種類と通信できる種類を選択します。これにより，インターネットコンポーネントを使い，空の Web モジュールを持つように設定された新しいプロジェクトが作成されます。

Web モジュール

Web モジュール（TWebModule）は，TDataModule の下位オブジェクトであり，TDataModule と同じように使用されます。Web アプリケーション内で，ビジネスルールおよび非ビジュアルコンポーネントを集中管理できます。

Web モジュールには，アプリケーションがレスポンスメッセージを作成するときに使うコンテンツプロデューサを追加してください。このようなコンテンツプロデューサは，組み込みのコンテンツプロデューサ（TPageProducer，TDataSetPageProducer，TDataSetTableProducer，TQueryTableProducer，TInetXPageProducer など）でも，開発者が自分で作成した TCustomContentProducer の下位オブジェクトでもかまいません。データベースのデータが入っているレスポンスメッセージをアプリケーションで生成する場合，データアクセスコンポーネントまたは特殊コンポーネントを追加することにより，多層データベースアプリケーションでクライアントとして機能する Web サーバーを作成することも可能です。

Web モジュールは非ビジュアルコンポーネントやビジネスルールを格納するだけでなく，ディスパッチャとして機能します。つまり，受信した HTTP リクエストメッセージを，そのリクエストへのレスポンスを生成するアクション項目に対応付けます。

開発者は Web アプリケーションで使う非ビジュアルコンポーネントやビジネスルールがいくつも入っているデータモジュールをすでに用意しているかもしれません。その場合は，Web モジュールを自分の既存のデータモジュールで置き換えられます。これには，自動生成された Web モジュールを削除し，かわりに自分のデータモジュールを追加します。次に，そのデータモジュールに TWebDispatcher コンポーネントを追加して，データモジュールが Web モジュールと同様にアクション項目にリクエストメッセージをディスパッチできるようにします。受信した HTTP リクエストメッセージに応答するアクション項目の選択方法を変更するには，TCustomWebDispatcher から新しいディスパッチャコンポーネントを派生させて，データモジュールに追加します。

プロジェクトに入れられるディスパッチャは 1 つだけです。プロジェクトを作成すると自動的に生成される Web モジュールか，Web モジュールのかわりとしてデータモジュールに追加する

TWebDispatcher コンポーネントのどちらか 1 つです。ディスパッチャが入っているデータモジュールを実行時にもう 1 つ作成すると，Web サーバーアプリケーションは実行時エラーを生成します。

メモ 設計時に設定する Web モジュールは実際にはテンプレートです。ISAPI アプリケーションと NSAPI アプリケーションでは，リクエストメッセージごとに別々のスレッドが生成され，スレッドごとに Web モジュールの別々のインスタンスとリクエストのコンテンツが動的に作成されます。

注意 DLL ベースで使用される Web サーバーアプリケーションの Web モジュールは，レスポンス時間を速くするために後で再使用できるようにキャッシングされています。ディスパッチャの状態とアクションリストは，リクエストごとには再初期化されません。実行時にアクション項目の Enabled プロパティや Default プロパティを変更すると，それ以降のクライアントリクエストでこのモジュールを使ったときに予想外の結果が生じることがあります。

Web Application オブジェクト

Web アプリケーション用に設定するプロジェクトには，Application というグローバル変数が入っています。Application は TWebApplication の下位オブジェクトで，作成するアプリケーションの種類に合わせて TISAPIApplication，TApacheApplication，TCGIApplication のいずれかになります。Web サーバーが HTTP リクエストメッセージを受信すると，それに応答して実行されます。

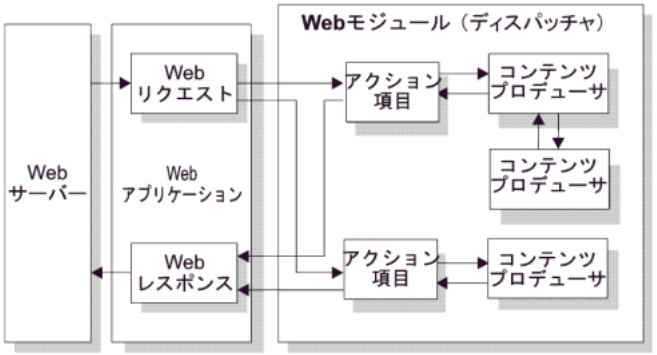
注意 プロジェクトファイルの CGIApp.hpp または ISAPIApp.hpp の include 文の後に Forms.hpp をインクルードしないでください。Forms.hpp もグローバル変数 Application を宣言します。CGIApp.hpp または ISAPIApp.hpp の後で Application が宣言されると，Application は誤った型のオブジェクトに初期化されます。

WebBroker アプリケーションの構造

Web アプリケーションは，HTTP リクエストメッセージを受信すると，TWebRequest オブジェクトと TWebResponse オブジェクトを作成します。TWebRequest は HTTP リクエストメッセージを表し，TWebResponse はリクエストメッセージに対して返される応答を表します。これらのオブジェクトは Web ディスパッチャ（Web モジュールまたは TWebDispatcher コンポーネント）に渡されます。

Web ディスパッチャは Web サーバーアプリケーションの処理の流れを制御します。ディスパッチャは，HTTP リクエストメッセージの種類ごとに処理方法を規定しているアクション項目（TWebActionItem）の集合を保持しています。ディスパッチャは，HTTP リクエストメッセージの処理に適したアクション項目または自動ディスパッチコンポーネントを特定し，リクエストオブジェクトとレスポンスオブジェクトを所定のイベントハンドラに渡します。これにより，要求されたアクションの実行やレスポンスメッセージの作成ができるようになります。詳細については，33-4 ページの「Web ディスパッチャ」を参照してください。

図 33.1 サーバーアプリケーションの構造



アクション項目は，リクエストを読み出して，それに対するレスポンスメッセージを作成しなければなりません。専用のコンテンツプロデューサコンポーネントを使うと，アクション項目は HTML カスタムコードやその他の MIME コンテンツを保持できるレスポンスメッセージのコンテンツを動的に作成できます。コンテンツプロデューサは，ほかのコンテンツプロデューサや THTMLTagAttributes の下位オブジェクトを使って，アクション項目が簡単にレスポンスメッセージのコンテンツを作成できるようにします。コンテンツプロデューサについての詳細は，33-13 ページの「レスポンスメッセージのコンテンツの生成」を参照してください。

多層データベースアプリケーションの Web クライアントを作成する場合，Web サーバーアプリケーションには，XML でエンコードされたデータベース情報と Java スクリプトでエンコードされたデータベース操作クラスを表す追加の自動ディスパッチコンポーネントを含めることができます。Web サービスを実装するサーバーを作成する場合は，Web サーバーアプリケーションに自動ディスパッチコンポーネントを追加することで，SOAP に基づいたメッセージがインボーカに渡り，解釈され実行されます。ディスパッチャは，自身のアクション項目をすべて試行した後，上記の自動ディスパッチコンポーネントを呼び出してリクエストメッセージを処理します。

すべてのアクション項目（または自動ディスパッチコンポーネント）が TWebResponse オブジェクトを完全なものにしてレスポンスを完成させると，ディスパッチャはこの結果を Web アプリケーションに返します。アプリケーションはこのレスポンスを Web サーバーを経由してクライアントに送信します。

Web ディスパッチャ

Web モジュールを使う場合，Web モジュールは Web ディスパッチャとして機能します。既存のデータモジュールを使う場合は，そのデータモジュールにディスパッチャコンポーネント TWebDispatcher を 1 つ追加しなければなりません。ディスパッチャは，リクエストメッセージの種類ごとに処理方法を規定しているアクション項目の集合を保持しています。ディスパッチャは，Web アプリケーションからリクエストオブジェクトとレスポンスオブジェクトを受け取ると，そのリクエストに応答するため 1 つまたは複数のアクション項目を選択します。

ディスパッチャへのアクションの追加

ディスパッチャの Actions プロパティで省略記号（...）をクリックすると，オブジェクトインスペクタからアクションエディタを開くことができます。［追加］ボタンをクリックすることにより，アクションエディタでアクション項目を追加できます。

異なるリクエストメソッドまたはターゲット URI に応答するには，ディスパッチャにアクションを追加します。アクション項目を設定するにはいくつもの方法があります。リクエストを前処理するアクション項目を最初に設定し，最後にデフォルトアクション（レスポンスが完全かどうか確認してからレスポンスを送信するかエラーコードを返します）を設定できます。リクエストの種類ごとに別々のアクション項目を追加し，アクション項目ごとにリクエストを完全に処理することもできます。

アクション項目についての詳細は，33-6 ページの「アクション項目」を参照してください。

リクエストメッセージのディスパッチ

ディスパッチャは，クライアントリクエストを受信すると，BeforeDispatch イベントを生成します。これにより，アプリケーションはリクエストメッセージを前処理してから，アクション項目を使って処理できます。

BeforeDispatch イベントを生成した後，ディスパッチャはアクション項目のリストの繰り返し処理を行い，リクエストメッセージのターゲット URL の PathInfo 部分に対応した，リクエストメッセージのメソッドとして指定されたサービスも提供するエントリを探します。これは，TWebRequest オブジェクトの PathInfo プロパティと MethodType プロパティを，アクション項目の同名のプロパティと比較すればできます。

ディスパッチャは，適切なアクション項目を見つけると，そのアクション項目を起動します。起動されたアクション項目は次のいずれかを行います。

- レスポンスの内容を埋め，そのレスポンスを送信するか，リクエストの処理が完了したことを通知する
- レスポンスに追加を行い，他のアクション項目によって処理が完了できるようにする
- リクエストを他のアクション項目に任せる

アクション項目をすべてチェックした後もメッセージが処理されていなければ，ディスパッチャは，アクション項目を使用しない自動ディスパッチコンポーネントが特別に登録されていないかチェックします。この自動ディスパッチコンポーネントは多層データベースアプリケーションに固有のコンポーネントです。詳細については，29-32 ページの「InternetExpress 使用による Web アプリケーションの構築」を参照してください。

すべてのアクション項目と，特別に登録された自動ディスパッチコンポーネントをチェックした後も，なおリクエストメッセージの処理が完了していない場合，ディスパッチャはデフォルトのアクション項目を呼び出します。デフォルトアクション項目はターゲット URL にもリクエストのメソッドにも対応している必要はありません。

ディスパッチャがアクション項目リストの（デフォルトアクション項目があればそれも含めて）最後まで到達しても起動されるアクションがないと，サーバーには何も返されません。サーバーはクライアントへの接続を切断するだけです。

アクション項目がリクエストを処理すると，ディスパッチャは AfterDispatch イベントを生成します。これにより，アプリケーションは生成されたレスポンスを確認して最後の変更ができます。

アクション項目

アクション項目（TWebActionItem）はそれぞれ，特定の種類のリクエストメッセージに応じて決まった処理を実行します。

アクション項目はリクエストに完全に応答することも，部分的に応答して残りをほかのアクション項目に任せることもできます。アクション項目はリクエストに対する HTTP レスポンスメッセージを送信することも，レスポンスを部分的に設定するだけにして残りの処理をほかのアクション項目に任せることもできます。アクション項目が処理したレスポンスを送信しない場合には，Web サーバーアプリケーションがそのレスポンスメッセージを送信します。

アクション項目を起動するタイミング

アクション項目のプロパティによって，ディスパッチャが HTTP リクエストメッセージを処理するためにアクションをいつ選択するかが決定されます。アクション項目のプロパティを設定するには，アクションエディタを起動する必要があります。それには，オブジェクトインスペクタでディスパッチャの Actions プロパティを選択し，省略記号（...）を選択します。アクションエディタでアクションを選択すると，オブジェクトインスペクタでそのプロパティを変更できます。

ターゲット URL

ディスパッチャはアクション項目の PathInfo プロパティと，リクエストメッセージの PathInfo を比較します。このプロパティの値は，アクション項目が処理できるすべてのリクエストに対応した URL のパス情報部分でなければなりません。たとえば，次の URL があるとします。

```
http://www.TSite.com/art/gallery.dll/mammals?animal=dog&color=black
```

`/gallery.dll` 部分が Web サーバーアプリケーションであると仮定すると，パス情報部分は次のようになります。

```
/mammals
```

リクエストの応答時に Web アプリケーションが情報を検索する場所を指定したり，1 つの Web アプリケーションで複数のサービスを提供するために論理的なサブサービスを定義する場合などに，パス情報を使います。

リクエストメソッド型

アクション項目の MethodType プロパティは，そのアクション項目が処理できるリクエストメッセージの種類を示します。ディスパッチャはアクション項目の MethodType プロパティと，リクエストメッセージの MethodType を比較します。次の表に MethodType の値を示します。

表 33.1 MethodType の値

値	意味
mtGet	ターゲット URI に関連した情報をレスポンスメッセージに入れて返すことを要求している
mtHead	mtGet リクエストを処理するときと同様にレスポンスのヘッダープロパティを要求しているが，レスポンスのコンテンツは省略する
mtPost	Web アプリケーションに登録すべき情報を提供している
mtPut	ターゲット URI に関連付けられたリソースを，リクエストメッセージのコンテンツによって置き換えることを要求する
mtAny	アクション項目がどのリクエストメソッド型（mtGet，mtHead，mtPut，mtPost）にも対応している

アクション項目を使用可能 / 使用不可にする

アクション項目には，そのアクション項目を使用可能または使用不可にする Enabled プロパティがあります。Enabled を **false** に設定するとアクション項目は使用不可になり，ディスパッチャはリクエストを処理するアクション項目を探すときにこのアクション項目を対象外とみなします。

ディスパッチャの BeforeDispatch イベントハンドラは，ディスパッチャがリクエストメッセージに適したアクション項目を探す前に，アクション項目の Enabled プロパティを変更してどのアクション項目がリクエストを処理すべきかを制御できます。

注意 実行時にアクションの Enabled プロパティを変更すると，それ以降のリクエストに対して予想外の結果が生じることがあります。Web サーバーアプリケーションが Web モジュールをキャッシュする DLL の場合，次のリクエストに対する初期状態は再初期化されません。この場合 BeforeDispatch イベントを使用して，すべてのアクション項目を初期状態に初期化させることができます。

デフォルトアクション項目を選択する

デフォルトアクション項目に指定できるアクション項目は 1 つだけです。アクション項目の Default プロパティを **true** に設定すると，そのアクション項目がデフォルトアクション項目として選択されます。あるアクション項目の Default プロパティを **true** に設定すると，それまでにデフォルトとして選択されていたアクション項目があれば，そのアクション項目の Default プロパティは **false** に設定されます。

ディスパッチャは，リクエストを処理するアクション項目をアクション項目リストから検索するとき，デフォルトアクション項目の名前を格納します。ディスパッチャがアクション項目リストの最後まで到達してもリクエストの処理が完了していない場合，ディスパッチャはデフォルトアクション項目を実行します。

ディスパッチャは，デフォルトアクション項目の PathInfo または MethodType を確認しません。ディスパッチャは，デフォルトアクション項目の Enabled プロパティも確認しません。このため，デフォルトアクション項目の Enabled プロパティを **false** にして，最後にならないとデフォルトアクション項目が呼び出されないようにすることができます。

デフォルトアクション項目は，URI または MethodType が無効であることを示すエラーコードを返すだけの場合も含めて，あらゆるリクエストを処理する準備をしておかなければなりません。デフォルトアクション項目がリクエストを処理しないと，Web クライアントにレスポンスが送信されません。

注意 実行時にアクションの Default プロパティを変更すると，現在のリクエストに対して予想外の結果が生じることがあります。すでに起動済みのアクション項目の Default プロパティを **true** に設定しても，そのアクションは再評価されず，アクション項目リストの最後まで到達してもディスパッチャはそのアクションを起動しません。

アクション項目によるリクエストメッセージへのレスポンス

Web サーバーアプリケーションの実際の仕事は，実行されるアクションによって行われます。Web ディスパッチャがアクション項目を起動すると，そのアクション項目は 2 通りの方法で現在のリクエストメッセージに応答することができます。

- アクション項目の Producer プロパティとしてプロデューサコンポーネントが関連付けられている場合，そのプロデューサは自動的に Content メソッドを使ってレスポンスメッセージの Content を割り当てます。コンポーネントパレットの［Internet］ページには，各種のコンテンツプロデューサコンポーネントがあり，レスポンスメッセージのコンテンツとなる HTML ページを作成するのに役立ちます。
- プロデューサがレスポンスのコンテンツを割り当てた後で（関連付けられたプロデューサがある場合），アクション項目は OnAction イベントを受け取ります。OnAction イベントハンドラには，HTTP リクエストメッセージを表す TWebRequest オブジェクトと，すべてのレスポンス情報を書き込むための TWebResponse オブジェクトが渡されます。

アクション項目のコンテンツが 1 つのコンテンツプロデューサによって生成可能な場合は，そのコンテンツプロデューサをアクション項目の Producer プロパティとして割り当てるのが最も簡単です。その場合でも，OnAction イベントハンドラから任意のコンテンツプロデューサにいつでもアクセスすることができます。OnAction イベントハンドラには高い柔軟性があるので，複数のコンテンツプロデューサを使ってレスポンスメッセージプロパティの割り当てなどを行うことができます。

コンテンツプロデューサコンポーネントと OnAction イベントハンドラは，どちらも任意のオブジェクトまたはランタイムライブラリのメソッドを使ってリクエストメッセージに応答することができます。どちらからでも，データベースへのアクセス，計算の実行，HTML ドキュメントの作成や選択などが行えます。コンテンツプロデューサコンポーネントによるレスポンスコンテンツの生成については，33-13 ページの「レスポンスメッセージのコンテンツの生成」を参照してください。

レスポンスの送信

OnAction イベントハンドラは，TWebResponse オブジェクトのメソッドを使って Web クライアントにレスポンスを送り返すことができます。ただし，アクション項目がクライアントにレスポンスを送信しない場合は，最後にリクエストに対処したアクション項目でそのリクエストが処理されたことが示されていれば，Web サーバーアプリケーションがレスポンスを送信します。

複数のアクション項目を使う

リクエストに対して 1 つのアクション項目から応答することも，複数のアクション項目で処理を分担して応答することもできます。アクション項目がレスポンスメッセージのセットアップを完了していない場合，アクション項目は OnAction イベントハンドラの Handled パラメータを **false** に設定して，その状態を知らせなければなりません。

複数のアクション項目でリクエストメッセージへのレスポンス処理を分担し，アクション項目ごとに Handled パラメータを **false** に設定してほかのアクション項目が処理を継続できるようにする場合，デフォルトアクション項目の Handled パラメータは **true** のままにしておいてください。そうしないと，レスポンスが Web クライアントに送信されません。

複数のアクション項目で処理を分担する場合，デフォルトアクション項目の OnAction イベントハンドラまたはディスパッチャの AfterDispatch イベントハンドラが，すべての処理が実行されたかどうか確認し，実行されていなければ該当するエラーコードを返さなければなりません。

クライアントリクエスト情報へのアクセス

Web サーバーアプリケーションが HTTP リクエストメッセージを受信すると，クライアントリクエストのヘッダーが TWebRequest オブジェクトのプロパティにロードされます。NSAPI アプリケーションと ISAPI アプリケーションの場合，リクエストメッセージは TISAPIRequest オブジェクトによってカプセル化されます。コンソール CGI アプリケーションは TCGIRequest オブジェクトを使い，Windows CGI アプリケーションは TWinCGIRequest オブジェクトを使います。

リクエストオブジェクトのプロパティは読み出し専用です。これらのプロパティを使うと，クライアントリクエストで使えるすべての情報を収集できます。

リクエストヘッダー情報の入ったプロパティ

リクエストオブジェクトのほとんどのプロパティには，HTTP リクエストヘッダーから受け取ったリクエストに関する情報が入っています。すべてのリクエストがこれらのどのプロパティにも値を提供するわけではありません。また，リクエストによっては，リクエストオブジェクトのプロパティで指定されていないヘッダー項目が含まれていることがあります。特に，HTTP 標準の継続的な改訂に伴ってこのようなことが起こります。リクエストオブジェクトのプロパティとして表示されていないリクエストヘッダー項目の値を取得するには，GetFieldByName メソッドを使います。

ターゲットを識別するプロパティ

リクエストメッセージ全体は URL プロパティに格納されています。通常はこれが URL であり，プロトコル（HTTP），Host（ホスト名），ScriptName（サーバーアプリケーション），PathInfo（サーバーアプリケーションの位置），Query（クライアントの要求）に分割することができます。

これらの要素は，それぞれに相当するプロパティに格納されています。プロトコルは HTTP から始まる文字列となります。Host プロパティと ScriptName プロパティは，Web サーバーアプリケーションを識別します。PathInfo は，ディスパッチャがアクション項目とリクエストメッセージを対応付けるときに使用します。Host，ScriptName，PathInfo の 3 つのプロパティは，Web サーバーによって異なる扱われ方をするため，注意が必要です。Query は，必要な情報の詳細を指定するのにリクエストで使われます。Query の値を項目ごとに分解された形でアクセスするには，QueryFields プロパティを使います。

Web クライアントを識別するプロパティ

リクエストには，リクエストの送信元情報を示すプロパティもいくつか含まれています。たとえば，送信側の電子メールアドレス（From プロパティ），メッセージの送信元 URI（Referer プロパティまたは RemoteHost プロパティ）などがあります。リクエストにコンテンツが含まれており，そのコンテンツがリクエストと同じ URI からのものでない場合，コンテンツのソースが DerivedFrom プロパティによって示されます。クライアントの IP アドレス（RemoteAddr プロパティ）やリクエストを送信したアプリケーションの名前やバージョン（UserAgent プロパティ）も確認できます。

リクエストの目的を示すプロパティ

Method プロパティは，リクエストメッセージがサーバーアプリケーションに何を要求しているかを説明する文字列です。次の表に HTTP 1.1 標準で定義しているメソッドを示します。

値	メッセージのリクエスト
OPTIONS	利用可能な通信オプションについての情報
GET	URL プロパティで指定された情報
HEAD	GET メッセージからのヘッダー情報と同等。レスポンスのコンテンツは不要
POST	該当する場合，Content プロパティのデータをサーバーアプリケーションが登録する
PUT	サーバーアプリケーションが，URL プロパティで指定されたリソースを Content プロパティのデータで置換する
DELETE	URL プロパティで指定されたリソースをサーバーアプリケーションが削除または非表示にする
TRACE	リクエストの受信を確認するためループバックをサーバーアプリケーションが送信する

Method プロパティは，Web クライアントがサーバーに要求する他のメソッドを示すことができます。

Web サーバーアプリケーションは Method のすべての値に応答する必要はありません。HTTP 標準では，Web サーバーアプリケーションが GET リクエストと HEAD リクエストの両方を処理することは要求していません。

MethodType プロパティは，Method の値が GET（mtGet），HEAD（mtHead），POST（mtPost），Put（mtPut），その他の文字列（mtAny）のどれであるかを示します。ディスパッチャは MethodType プロパティの値と，各アクション項目の MethodType を照合します。

予想されるレスポンスを示すプロパティ

Accept プロパティは，Web クライアントがレスポンスメッセージのコンテンツとして受け付けるメディアの種類を指定します。IfModifiedSince プロパティは，最近変更された情報だけをクライアントが必要としているかどうかを示します。Cookie プロパティには，レスポンスを変更できる状態情報（通常，アプリケーションが以前書き込んだもの）があります。

コンテンツを示すプロパティ

ほとんどのリクエストは，情報を求めるものであるため，コンテンツは含まれていません。ただし，POST リクエストのように，Web サーバーアプリケーションが使うと思われるコンテンツを提供するものもあります。コンテンツのメディアの種類は ContentType プロパティで指定され，長さは ContentLength プロパティで指定されます。データ圧縮などのためにメッセージのコンテンツがエンコードされている場合，この情報は ContentEncoding プロパティにあります。コンテンツを作成した

アプリケーションの名前とバージョン番号は ContentVersion プロパティで指定されています。Title プロパティもコンテンツに関する情報を提供することがあります。

HTTP リクエストメッセージのコンテンツ

リクエストメッセージによっては，ヘッダー項目のほかに，Web サーバーアプリケーションがなんらかの方法で処理しなければならないコンテンツ部分が含まれていることがあります。たとえば，POST リクエストには，Web サーバーアプリケーションがアクセスするデータベースに追加しなければならない情報が入っています。

コンテンツの未処理の値は Content プロパティに格納されます。コンテンツを解析して項目ごとにアンド記号（&）で区切ることができる場合，これを解析したものが ContentFields プロパティに格納されています。

HTTP レスポンスメッセージの作成

Web サーバーアプリケーションが受信した HTTP リクエストメッセージ用に TWebRequest オブジェクトを作成するときは，返信するレスポンスメッセージを表す TWebResponse オブジェクトも作成します。NSAPI アプリケーションと ISAPI アプリケーションでは，レスポンスメッセージが TISAPIResponse オブジェクトによってカプセル化されています。コンソール CGI アプリケーションは TCGIResponse オブジェクトを使い，Windows CGI アプリケーションは TWinCGIResponse オブジェクトを使います。

OnAction イベントハンドラは Web クライアントリクエストへのレスポンスを生成する場合，レスポンスオブジェクトのプロパティを指定します。単にエラーコードを返すか，リクエストをほかの URI にリダイレクトするだけのこともあります。イベントハンドラがほかのソースから情報を取得し，それを完全な形式に組み立てなければならない複雑な計算が含まれていることもあります。要求されたアクションが実行されたことを伝えるだけの場合も含めて，ほとんどのリクエストメッセージではレスポンスが必要です。

レスポンスヘッダーの指定

TWebResponse オブジェクトのほとんどのプロパティは，Web クライアントに返信する HTTP レスポンスメッセージのヘッダー情報を表します。アクション項目は，その OnAction イベントハンドラからこれらのプロパティを設定します。

すべてのレスポンスメッセージがどのヘッダープロパティにも値を指定する必要があるわけではありません。設定が必要なプロパティは，リクエストの性質やレスポンスの状態によって異なります。

レスポンス状態を表示する

レスポンスメッセージには，レスポンスの状態を示すステータスコードが入っていなければなりません。ステータスコードを指定するには，StatusCode プロパティを設定します。HTTP 標準では，既定の意味を持つ標準のステータスコードがいくつも定義されています。未使用の有効な値を使えば自分でステータスコードを定義することもできます。

ステータスコードは以下のような 3 桁の数字であり，最上位の桁がレスポンスクラスを示します。

- 1xx：Informational（リクエストは受信されたが処理は完了していない）
- 2xx：Success（リクエストが受信され，認識され，受け付けられた）
- 3xx：Redirection（リクエストを完了するにはクライアントのアクションがさらに必要である）
- 4xx：Client Error（リクエストを認識できない，または処理できない）
- 5xx：Server Error（リクエストは有効だが，サーバーが処理できなかった）

ステータスコードと一緒に表示される文字列はステータスコードの意味を示しています。これは，ReasonString プロパティによって指定されます。定義済みのステータスコードの場合は ReasonString プロパティを設定する必要はありません。自分でステータスコードを定義した場合は，ReasonString プロパティも設定する必要があります。

クライアントのアクションを要求する

ステータスコードが 300 ～ 399 の場合，クライアントがさらにアクションを実行しなければ Web サーバーアプリケーションはリクエストの処理を完了できません。クライアントを別の URI にリダイレクトしなければならないか，リクエストを処理する新しい URI が作成されたことを示さなければならない場合は，Location プロパティを設定します。クライアントがパスワードを入力しないと処理を続けられない場合は，WWWAuthenticate プロパティを設定します。

サーバーアプリケーションを記述する

レスポンスヘッダーのプロパティの中には，Web サーバーアプリケーションの機能を記述するものがあります。Allow プロパティはアプリケーションが応答できるメソッドを指定します。Server プロパティは，レスポンスの生成に使うアプリケーションの名前とバージョン番号を指定します。Cookies プロパティは，後続のリクエストメッセージにサーバーアプリケーションに関する任意の情報を含ませることができ，クライアントに保持させることができます。

コンテンツを記述する

プロパティの中にはレスポンスのコンテンツを記述するものがあります。ContentType はレスポンスのメディアタイプを指定し，ContentVersion はそのメディアタイプに関連付けられたバージョン番号を指定します。ContentLength はコンテンツの長さを指定します。データ圧縮などのためにコンテンツがエンコードされている場合，そのメカニズムは ContentEncoding プロパティで指定されます。コンテンツが別の URI のものである場合，このコンテンツは DerivedFrom プロパティで指定されます。コンテンツの値に期間の制限がある場合，LastModified プロパティと Expires プロパティが，値がまだ有効かどうかを示します。Title プロパティもコンテンツに関する情報を提供することがあります。

レスポンスのコンテンツの設定

リクエストによっては，リクエストメッセージへのレスポンスがレスポンスのヘッダープロパティに完全に含まれている場合があります。しかし，ほとんどの場合は，OnAction イベントハンドラが最終的なアクションとしてレスポンスメッセージになんらかのコンテンツを割り当てなければなりません。このコンテンツは，ファイルに格納された静的情報の場合も，アクション項目またはそのコンテンツプロデューサが動的に作成した情報の場合もあります。

レスポンスメッセージのコンテンツは，Content プロパティまたは ContentStream プロパティを使って設定できます。

Content プロパティは AnsiString です。AnsiString はテキスト値に限られるわけではありません。したがって，Content プロパティの値は，HTML コマンド文字列，グラフィックコンテンツ（ビットストリームなど），その他の種類の MIME コンテンツのどれでもかまいません。

レスポンスメッセージのコンテンツをストリームから読み出せる場合は ContentStream プロパティを使います。たとえば，レスポンスメッセージがファイルの内容を送信しなければならない場合は，ContentStream プロパティに TFileStream オブジェクトを使います。Content プロパティの場合と同様に，ContentStream は HTML コマンド文字列かその他の種類の MIME コンテンツを示すことができます。ContentStream プロパティを使う場合は，自分でストリームを解放しないでください。Web レスポンスオブジェクトが自動的に解放します。

メモ ContentStream プロパティの値が NULL 以外の場合，Content プロパティは無視されます。

レスポンスの送信

リクエストメッセージに対してこれ以上処理が必要ないと判断したら，OnAction イベントハンドラからレスポンスを直接送信できます。レスポンスオブジェクトには，レスポンスの送信用に SendResponse と SendRedirect の 2 つのメソッドがあります。指定したコンテンツと TWebResponse オブジェクトのすべてのヘッダープロパティを使ってレスポンスを送信するには，SendResponse を呼び出します。Web クライアントを別の URI にリダイレクトするだけなら，SendRedirect メソッドを使う方が効率的です。

いずれのイベントハンドラもレスポンスを送信しないときには，ディスパッチャの終了後に Web アプリケーションオブジェクトがレスポンスを送信します。ただし，レスポンスを処理したことを示しているアクション項目がない場合，アプリケーションはレスポンスを送信せずに Web クライアントへの接続を切断します。

レスポンスメッセージのコンテンツの生成

C++Builder には，アクション項目で HTTP レスポンスメッセージ用のコンテンツを作成するのに役立つオブジェクトがいくつも用意されています。これらのオブジェクトを使うと，ファイル内に保存するか Web クライアントに直接返送するための HTML コマンド文字列を生成できます。TCustomContentProducer またはその下位オブジェクトから自分でコンテンツプロデューサを作成することもできます。

TCustomContentProducer にはどの種類の MIME でも HTTP レスポンスメッセージのコンテンツとして作成できる汎用インターフェースがあります。この下位オブジェクトには，ページプロデューサとテーブルプロデューサが含まれます。

- ページプロデューサは HTML ドキュメントをスキャンし，カスタマイズした HTML コードに置換する特殊なタグを探します。これについては以下のセクションで説明します。
- テーブルプロデューサはデータセットの情報を使って HTML コマンドを作成します。これについては 33-17 ページの「レスポンスでのデータベース情報の使い方」で説明します。

ページプロデューサコンポーネントの使い方

ページプロデューサ（TPageProducer とその下位オブジェクト）は HTML テンプレートを取り出し，特殊な HTML 透過タグをカスタマイズした HTML コードに置換してそのテンプレートを変換します。HTTP リクエストメッセージへの標準のレスポンスを生成しなければならない場合，ページプロデューサ置換用の標準のレスポンステンプレートセットを格納できます。ページプロデューサを連鎖させて，HTML 透過タグの適切な置換を連続的に行うことにより，1 つの HTML ドキュメントを構築することができます。

HTML テンプレート

HTML テンプレートは一連の HTML コマンドと HTML 透過タグで構成されます。次に HTML 透過タグの形式を示します。

```
<#TagName Param1=Value1 Param2=Value2 ...>
```

不等号カッコ（< と >）はタグのスコープ全体を定義します。左不等号カッコ（<）の直後にはスペースなしでシャープ（#）が続きます。ページプロデューサはシャープの後ろの文字列を HTML 透過タグとして認識します。タグ名はシャープの直後にスペースなしで続きます。タグ名は任意の有効な識別子であり，タグが表す変換型を識別します。

HTML 透過タグでは，タグ名の後ろに変換の詳細を指定するパラメータを入れることもできます。パラメータの形式は ParamName=Value です。パラメータ名，等号（=），値の間にはスペースを入れません。パラメータどうしはスペースで区切ります。

不等号カッコ（< と >）は，#TagName 構造を認識しない HTML ブラウザにタグを無視させます。

独自の HTML 透過タグを作成すると，ページプロデューサが処理する情報のすべての種類を表すことができますが，TTag データ型の値と関連付けられている定義済みのタグ名も用意されています。これらの定義済みタグ名は，レスポンスメッセージに変更を加える HTML コマンドに対応します。次の表に定義済みのタグ名を示します。

タグ名	TTag 値	タグの変換先
Link	tgLink	ハイパーテキストリンク。<A> タグと </A> タグで囲まれた HTML シーケンスに変換される
Image	tgImage	グラフィックイメージ。HTML <IMG> タグに変換される
Table	tgTable	HTML テーブル。<TABLE> タグと </TABLE> タグで囲まれた HTML シーケンスに変換される
ImageMap	tgImageMap	ホットゾーンが関連付けられたグラフィックイメージ。<MAP> タグと </MAP> タグで囲まれた HTML シーケンスに変換される
Object	tgObject	埋め込まれた ActiveX オブジェクト。<OBJECT> タグと </OBJECT> タグで囲まれた HTML シーケンスに変換される
Embed	tgEmbed	Netscape 準拠のアドイン DLL。<EMBED> タグと </EMBED> タグで囲まれた HTML シーケンスに変換される

その他のタグ名はすべて tgCustom に関連付けられます。ページプロデューサは，定義済みのタグ名に対する組み込みの処理を供給していません。定義済みのタグは，単にアプリケーションが変換処理をより一般的なタスク構成に組み入れるのを支援するために提供されています。

メモ　定義済みのタグ名は大文字と小文字を区別しません。

HTML テンプレートを指定する

TPageProducer には，HTML テンプレートを指定する方法がいくつも用意されています。HTMLFile プロパティには，HTML テンプレートが入っているファイルの名前を設定できます。HTMLDoc プロパティには，HTML テンプレートが入っている TStrings オブジェクトを設定できます。HTMLFile プロパティまたは HTMLDoc プロパティを使ってテンプレートを指定する場合は，Content メソッドを呼び出して変換済みの HTML コマンドを生成できます。

ContentFromString メソッドを呼び出すと，パラメータとして渡される単一の文字列（AnsiString）で構成されている HTML テンプレートを直接変換できます。また，ContentFromStream メソッドを呼び出すことにより，ストリームから HTML テンプレートを読み出して，変換することもできます。したがって，たとえばすべての HTML テンプレートをデータベースの 1 つのメモ型項目に格納し，TBlobStream オブジェクトから直接テンプレートを読み出して，ContentFromStream メソッドを使って変換済みの HTML コマンドを取得することができます。

HTML 透過タグを変換する

ページプロデューサのいずれかの Content メソッドを呼び出すと，そのページプロデューサは HTML テンプレートを変換します。Content メソッドは HTML 透過タグを検出すると，OnHTMLTag イベントを起動します。HTML 透過タグを使ってレスポンスメッセージを動的に変更したい場合，それぞれの HTML 透過タグを置換できるように，イベントハンドラを作成しなければなりません。

ページプロデューサの OnHTMLTag イベントハンドラを作成していない場合，HTML 透過タグは空文字列に置換されます。

アクション項目からページプロデューサを使う

ページプロデューサコンポーネントは一般的に，HTMLFile プロパティを使用して HTML テンプレートが入っているファイルを指定するために使われます。OnAction イベントハンドラは，Content メソッドを呼び出してテンプレートを最終的な HTML シーケンスに変換します。

```
void __fastcall WebModule1::MyActionEventHandler(TObject *Sender,
   TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
  PageProducer1->HTMLFile = "Greeting.html";
  Response->Content = PageProducer1->Content();
}
```

Greeting.html は以下の HTML テンプレートとします。

```
<HTML>
<HEAD><TITLE>Our Brand New Web Site</TITLE></HEAD>
<BODY>
Hello <#UserName>!Welcome to our Web site.
</BODY>
</HTML>
```

OnHTMLTag イベントハンドラは実行時に HTML 内のカスタムタグ（<#UserName>）を置換します。

```
void __fastcall WebModule1::HTMLTagHandler(TObject *Sender, TTag Tag,
   const AnsiString TagString, TStrings *TagParams, AnsiString &ReplaceText)
{
```

```
    if (CompareText(TagString,"UserName") == 0)
      ReplaceText = ((TPageProducer *)Sender)->Dispatcher->Request->Content;
}
```

リクエストメッセージのコンテンツが文字列 Mr. Ed である場合，Response->Content の値は次のようになります。

```
<HTML>
<HEAD><TITLE>Our Brand New Web Site</TITLE></HEAD>
<BODY>
Hello Mr. Ed!Welcome to our Web site.
</BODY>
</HTML>
```

メモ この例では，OnAction イベントハンドラを使ってコンテンツプロデューサを呼び出し，レスポンスメッセージのコンテンツを割り当てています。設計時にページプロデューサの HTMLFile プロパティを割り当てている場合は，OnAction イベントハンドラを記述する必要はありません。その場合は，アクション項目の Producer プロパティの値として PageProducer1 を割り当てるだけで，上記の OnAction イベントハンドラと同じ処理が行えます。

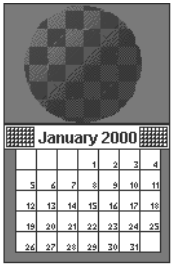
ページプロデューサの連鎖

OnHTMLTag イベントハンドラの置換用文字列は，HTTP レスポンスメッセージで使いたい最終的な HTML シーケンスである必要はありません。複数のページプロデューサを使うと，あるページプロデューサの出力を次のページプロデューサの入力にできます。

複数のページプロデューサを連鎖させる一番簡単な方法は，各ページプロデューサを個別のアクション項目と関連付けることです。ただし，どのアクション項目も PathInfo と MethodType が同じでなければなりません。1 番目のアクション項目は，自身のコンテンツプロデューサから Web レスポンスメッセージのコンテンツを設定しますが，このアクション項目の OnAction イベントハンドラは，Web レスポンスメッセージが処理されたと判断されないように（Handled パラメータを **false** に）します。2 番目以降のアクション項目では，おのおのの関連プロデューサの ContentFromString メソッドを使って，Web レスポンスメッセージのコンテンツを操作します。2 番目以降のアクション項目は，以下の例のように OnAction イベントハンドラを使用します。

```
void __fastcall WebModule1::Action2Action(TObject *Sender,
   TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
  Response->Content = PageProducer2->ContentFromString(Response->Content);
}
```

たとえば，目的のページの月と年を指定するリクエストメッセージに応じて，カレンダーページを返すアプリケーションがあるとします。各カレンダーページにはイメージがあり，その後ろには，前月と翌月の小さなイメージの間に何年の何月かが示されます。最後に実際のカレンダーが続きます。この結果，次の図に示すようになります。



カレンダーの一般的な形式がテンプレートファイルに格納されます。これは次のようになります。

```
<HTML>
<Head></HEAD>
<BODY>
<#MonthlyImage> <#TitleLine><#MainBody>
</BODY>
</HTML>
```

最初のページプロデューサの OnHTMLTag イベントハンドラは，リクエストメッセージから月と年を探します。この情報とテンプレートファイルを使って，ページプロデューサは次の処理を実行します。

- <#MonthlyImage> を <#Image Month=January Year=2000> に置換する
- <#TitleLine> を <#Calendar Month=December Year=1999 Size=Small> January 2000 <#Calendar Month=February Year=2000 Size=Small> に置換する
- <#MainBody> を <#Calendar Month=January Year=2000 Size=Large> に置換する

次のページプロデューサの OnHTMLTag イベントハンドラは最初のページプロデューサが作成したコンテンツを使い，<#Image Month=January Year=2000> タグを適切な HTML <IMG> タグに置換します。さらに別のページプロデューサが #Calendar タグを適切な HTML テーブルに置換します。

レスポンスでのデータベース情報の使い方

HTTP リクエストメッセージへのレスポンスには，データベースの情報を入れることができます。［Internet］パレットページのデータベース対応コンテンツプロデューサを使うと，HTML テーブルでデータベースのレコードを表す HTML を生成できます。

コンポーネントパレットの［InternetExpress］ページにある特殊コンポーネントを使う方法もあります。この特殊コンポーネントを使うと，多層データベースアプリケーションの一部としての Web サーバーを構築できます。詳細については，29-32 ページの「InternetExpress 使用による Web アプリケーションの構築」を参照してください。

Web モジュールへのセッションの追加

コンソール CGI アプリケーションと WinCGI アプリケーションはどちらも HTTP リクエストメッセージに応答して起動されます。これらの種類のアプリケーションでデータベースを操作する場合に

は，特に意識せずにデフォルトセッションを使ってデータベースの接続を管理できます。これは，リクエストメッセージごとに異なるアプリケーションインスタンスが生成されるからです。この場合，アプリケーションインスタンスごとに別々のデフォルトセッションが用意されます。

しかしながら，ISAPI アプリケーションまたは NSAPI アプリケーションを作成すると，各リクエストメッセージは 1 つのアプリケーションインスタンスの個別のスレッドとして処理されます。異なるスレッドからのデータベース接続が互いに妨害し合わないようにするには，スレッドごとに固有のセッションを割り当てなければなりません。

ISAPI アプリケーションまたは NSAPI アプリケーションの各リクエストメッセージは新しいスレッドを生成します。そのスレッドの Web モジュールは実行時に動的に生成されます。Web モジュールが入っているスレッドのデータベース接続を処理するには，Web モジュールに TSession オブジェクトを追加します。

Web モジュールの個々のインスタンスは，実行時にスレッドごとに生成されます。このため，これらのモジュールのそれぞれにセッションオブジェクトが割り当てられます。これらのセッションは，それぞれをユニークな名前を付けて識別しなければなりません。それにより，異なるスレッドからのデータベースへのアクセスによる影響を受けずにすみます。各モジュールのセッションオブジェクトが動的にそれぞれユニークな名前を生成するようにするには，セッションオブジェクトの AutoSessionName プロパティを true に設定します。各セッションオブジェクトは自身にユニークな名前を動的に生成し，モジュール内のすべてのデータセットの SessionName プロパティがそのユニークな名前を参照するようにします。これにより，リクエストスレッドごとのデータベースとの対話はほかのどのリクエストメッセージも妨害せずに処理できます。セッションについての詳細は，24-16 ページの「データベースセッションの管理」を参照してください。

HTML でデータベース情報を表す

［Internet］パレットページのデータベース対応のコンテンツプロデューサコンポーネントは，データセットのレコードに基づいた HTML コマンドを供給します。データベース対応のコンテンツプロデューサは，以下の 2 種類があります。

- データセットページプロデューサ。データセットの項目を HTML ドキュメントのテキストにフォーマットする
- テーブルプロデューサ。データセットのレコードを HTML テーブル形式にフォーマットする

データセットページプロデューサの使い方

データセットページプロデューサは，ページプロデューサコンポーネントと同様に動作します。HTML 透過タグを持つテンプレートを，最終的な HTML 表現に変換します。ただし，データセットページプロデューサは，データセット内の項目名と一致するタグ名を持つタグをその項目の現在の値に変換する特殊な機能を持っています。一般的なページプロデューサの使い方についての詳細は，33-14 ページの「ページプロデューサコンポーネントの使い方」を参照してください。

データセットページプロデューサを使用するには，TDataSetPageProducer コンポーネントを Web モジュールに追加し，HTML コンテンツに項目値を表示するデータセットを DataSet プロパティに設定します。データセットページプロデューサの出力を記述する HTML テンプレートを作成します。表示したい項目値すべてに，次の形式のタグを HTML テンプレートに含めます。

```
<#FieldName>
```

ここで，FieldName は値を表示したいデータセット内の項目名を指定します。

アプリケーションが Content，ContentFromString，または ContentFromStream メソッドを呼び出した場合，データセットページプロデューサは項目を表すタグを現在の項目値で置き換えます。

テーブルプロデューサの使い方

［Internet］パレットページには，データセットのレコードを表す HTML テーブルを作成する 2 つのコンポーネントがあります。

- データセットテーブルプロデューサ。データセットの項目を HTML ドキュメントのテキストにフォーマットする
- 問い合わせテーブルプロデューサ。リクエストメッセージによって供給されたパラメータを設定してから，問い合わせを実行し，その結果得られたデータセットを HTML テーブル形式にフォーマットする

どちらのテーブルプロデューサを使う場合でも，テーブルの色，境界，セパレータの太さなどのプロパティを設定することにより，生成される HTML テーブルの外観をカスタマイズできます。設計時にテーブルプロデューサのプロパティを設定するには，テーブルプロデューサコンポーネントをダブルクリックしてレスポンスエディタダイアログを表示します。

テーブル属性を指定する

データセットテーブルプロデューサは THTMLTableAttributes オブジェクトを使って，データセットのレコードを表示する HTML テーブルの表示形式を指定します。THTMLTableAttributes オブジェクトには，HTML ドキュメント内のテーブルの幅や間隔，背景色，境界線の太さ，セルのパディング，セルの間隔などを指定するプロパティが含まれています。これらのプロパティは，どれもテーブルプロデューサが作成する HTML <TABLE> タグのオプションとして設定されます。

これらのプロパティは，設計時にオブジェクトインスペクタを使って指定します。オブジェクトインスペクタでテーブルプロデューサオブジェクトを選択し，THTMLTableAttributes オブジェクトの表示プロパティにアクセスできるように TableAttributes プロパティの「+」記号をクリックして展開します。

これらのプロパティは実行時にプログラムで指定することもできます。

行属性を指定する

テーブル属性と同様に，データを表示するテーブルの行でセルの位置合わせや背景色を指定できます。RowAttributes プロパティは THTMLTableRowAttributes 型のオブジェクトです。

これらのプロパティは，設計時にオブジェクトインスペクタを使って RowAttributes プロパティを展開して指定します。これらのプロパティは実行時にプログラムで指定することもできます。

HTML テーブルでの行数も，MaxRows プロパティを設定して調整できます。

列を指定する

テーブルに入れるデータセットが設計時にわかっている場合は，列エディタを使って列の項目結合や表示属性をカスタマイズできます。まず，テーブルプロデューサコンポーネントを選択して右クリッ

クします。次に，コンテキストメニューで［レスポンスの設定］を選択します。これにより，テーブルの列の追加，削除，再配置ができます。個々の列の項目結合や表示プロパティは，列エディタで選択すれば，オブジェクトインスペクタで設定できます。

HTTP リクエストメッセージからデータセットの名前を取り出す場合，設計時に列エディタで項目を結合することはできません。ただし，実行時に THTMLTableColumn オブジェクトを作成し，データセットテーブルプロデューサの Columns プロパティにそれらのオブジェクトを追加すれば，実行時にプログラムで列を設定できます。Columns プロパティを設定しないと，データセットの項目に一致するデフォルトの列セットが作成され，特殊な表示特性は指定されません。

HTML ドキュメントにテーブルを埋め込む

データセットテーブルプロデューサの Header プロパティと Footer プロパティを使うと，データセットを表す HTML テーブルを大きなドキュメントに埋め込むことができます。テーブルより前にくるすべてのものを指定するには Header を使い，テーブルの後にくるすべてのものを指定するには Footer を使います。

ページプロデューサのような別のコンテンツプロデューサを使って Header プロパティと Footer プロパティの値を作成することもできます。

テーブルを大きなドキュメントに埋め込む場合，テーブルにキャプションを追加することもできます。テーブルにキャプションを追加するには，Caption プロパティと CaptionAlignment プロパティを使います。

データセットテーブルプロデューサを設定する

TDataSetTableProducer はデータセットの HTML テーブルを作成するテーブルプロデューサです。表示したいレコードが入っているデータセットを指定するには，TDataSetTableProducer の DataSet プロパティを設定します。従来のデータベースアプリケーションではほとんどのデータベース対応オブジェクトで DataSource プロパティを設定しますが，TDataSetTableProducer については DataSource プロパティは使用しません。これは，TDataSetTableProducer が自分のデータソースを内部で生成するからです。

Web アプリケーションが特定のデータセットのレコードを表示する場合は，DataSet の値は設計時に設定できます。データセットが HTTP リクエストメッセージの情報に基づいている場合は，実行時に DataSet プロパティを設定しなければなりません。

問い合わせテーブルプロデューサを設定する

問い合わせの結果を表示する HTML テーブルを作成できます。この場合，問い合わせのパラメータは HTTP リクエストメッセージから取り出します。これらのパラメータは TQueryTableProducer コンポーネントの Query プロパティとして TQuery オブジェクトを指定します。

リクエストメッセージが GET リクエストの場合，問い合わせのパラメータは HTTP リクエストメッセージのターゲットとして指定された URL の問い合わせフィールドから取り出します。リクエストメッセージが POST リクエストの場合，問い合わせのパラメータはリクエストメッセージのコンテンツから取り出します。

TQueryTableProducer の Content メソッドを呼び出すと，リクエストオブジェクトで見つけたパラメータを使って問い合わせが実行されます。次に，結果として生成されたデータセットのレコードを表示できるように HTML テーブルのフォーマットを作成します。

データセットテーブルプロデューサの場合と同様に，HTML テーブルの表示プロパティと列結合のカスタマイズや，大きな HTML ドキュメントへのテーブルの埋め込みができます。

第34章

WebSnap を使用しての Web サーバーアプリケーションの作成

WebSnap は，追加のコンポーネント，ウィザード，およびビューによって WebBroker を強化し，複雑なデータ駆動型 Web ページを提供する Web アプリケーションの構築を容易にします。WebSnap は，複数のモジュールおよびサーバー側スクリプティングをサポートすることにより，C++Builder 開発者と Web デザイナとのチームでの開発と保守が容易になります。

WebSnap では，Web サーバーの開発と保守に関して，チームの HTML デザイナがより効果的に貢献できるようになります。WebSnap 開発プロセスの最終製品には，スクリプティング可能な HTML ページテンプレート一式が含まれます。これらのページは，埋め込みスクリプトタグをサポートする Microsoft FrontPage などの HTML エディタや，単純なテキストエディタを使って変更できます。アプリケーションを配布した後でも，必要に応じてテンプレートを変更することができます。プロジェクトソースコードを変更する必要はなく，貴重な開発時間を節約することができます。また，WebSnap の複数モジュールサポートを利用すると，プロジェクトのコーディング段階で，アプリケーションを小さな部分に分割して進めることができます。そのため，C++Builder 開発者の作業の独立性を高めることができます。

ディスパッチャコンポーネントは，ページコンテンツのリクエスト，HTML フォームの送信，および動的イメージのリクエストを自動的に処理します。アダプタと呼ばれる WebSnap コンポーネントが，アプリケーションのビジネスルールに対するスクリプト可能なインターフェースを定義する手段を提供します。たとえば，TDataSetAdapter オブジェクトは，データセットコンポーネントをスクリプト可能にするために使用されます。WebSnap プロデューサコンポーネントを使用すると，複雑なデータ駆動型フォームおよびテーブルをすばやく構築したり，XSL を使用してページを生成できます。セッションコンポーネントを使用すると，エンドユーザーを追跡できます。ユーザーリストコンポーネントを使用すると，ユーザーの名前，パスワード，およびアクセス権限の機能を提供できます。

Web アプリケーションウィザードを使用すると，必要なコンポーネントを使用してカスタマイズされたアプリケーションをすばやく構築できます。Web ページモジュールウィザードを使用すると，アプリケーション内の新しいページを定義するモジュールを作成できます。または，Web データモジュールウィザードを使用すると，Web アプリケーション全体で共有されるコンポーネントのコンテナを作成できます。

ページモジュールビューは，アプリケーションを実行せずにサーバー側スクリプトの結果を表示します。生成された HTML を表示するには，［プレビュー］タブを使用して，埋め込まれたブラウザに表示するか，［HTML ソース］タブを使用して，テキスト形式で表示します。［HTML スクリプト］タブは，ページ用 HTML の生成に使用されるサーバー側スクリプティングを含むページを表示します。

この章の以降のセクションでは，WebSnap コンポーネントを使用して Web サーバーアプリケーションを作成する方法を説明します。

WebSnap コンポーネントの基本的なコンポーネント

WebSnap を使用して Web サーバーアプリケーションを作成する前に，WebSnap 開発で使用する基本的なコンポーネントについて理解しておく必要があります。このコンポーネントは以下の 3 つのカテゴリに分けられます。

- Web モジュール：アプリケーションを構成するコンポーネントとページを定義するコンポーネントが含まれる
- アダプタ：HTML ページと Web サーバーアプリケーション自体との間のインターフェースを提供する
- ページプロデューサ：エンドユーザーに提供される HTML ページを作成するルーチンが含まれる

以降のセクションでは，これらの各コンポーネントを詳しく説明します。

Web モジュール

Web モジュールは，WebSnap アプリケーションを構築する基本要素です。WebSnap サーバーアプリケーションには，少なくとも 1 つの Web モジュールが含まれていなければなりません。Web モジュールは必要に応じて追加することができます。Web モジュールには以下の 4 種類があります。

- Web アプリケーションページモジュール（TWebAppPageModule オブジェクト）
- Web アプリケーションデータモジュール（TWebAppDataModule オブジェクト）
- Web ページモジュール（TWebPageModule オブジェクト）
- Web データモジュール（TWebDataModule オブジェクト）

Web ページモジュールと Web アプリケーションページモジュールは，Web ページのコンテンツを提供します。Web データモジュールと Web アプリケーションデータモジュールは，アプリケーション間で共有されるコンポーネントのコンテナとして機能します。これは，通常のデータモジュールが通常の C++ Builder アプリケーションで受け持つ役割と同じ役割を WebSnap アプリケーションで受け持ちます。Web ページモジュールと Web データモジュールは，サーバーアプリケーションにいくつでも含めることができます。

作成するアプリケーションで Web モジュールがいくつ必要かが分からない場合があります。WebSnap アプリケーションでは，なんらかのタイプの Web アプリケーションモジュールが 1 つだけ必要です。それとは別に，Web ページモジュールと Web データモジュールは必要なだけいくつでも追加することができます。

Web ページモジュールについては，ページスタイルごとに 1 モジュールというのが大まかな目安です。既存のページのフォーマットを使用できるページを実装するのであれば，新しい Web ページモジュールは必要ないこともあります。既存のページモジュールに手を加えるだけで十分なこともあります。作成しようとするページが既存のモジュールと大きく違う場合は，新たなモジュールを作成することになるでしょう。たとえば，オンラインカタログ販売のサーバーを作成するとします。販売する製品について説明するページでは，同じレイアウトを使って同じ基本的な情報を提供するので，どのページも同じ Web ページモジュールを共有できます。しかし注文フォームでは，製品の説明ページとは形式と機能が異なるので，別の Web ページモジュールが必要になるでしょう。

Web データモジュールでは，ルールが異なります。他の多くの Web モジュールと共有できるコンポーネントは，Web データモジュールに配置して，共有アクセスを簡略化する必要があります。また，さまざまな Web アプリケーションから使用できるコンポーネントを，固有の Web データモジュールに配置したいことがあります。そうすれば，そのアイテムを容易に複数のアプリケーションで共有することができます。もちろん，このような状況がどれも当てはまらなければ，Web データモジュールをまったく使わないという選択もありえます。Web データモジュールを通常のデータモジュールと同じような方法で使ってみれば，その経験から以降の開発での指針が得られるでしょう。

Web アプリケーションモジュールの種類

Web アプリケーションモジュールは，Web アプリケーション内のビジネスルールおよび非ビジュアルコンポーネントを集中管理できます。Web アプリケーションモジュールには，以下の表に示す 2 種類があります。

表 34.1　Web アプリケーションモジュールの種類

Web アプリケーションモジュールの種類	説明
ページ	コンテンツページを作成する。ページモジュールには，ページコンテンツの生成を担当するページプロデューサが含まれる。ページプロデューサは，HTTP リクエストのパス情報がページ名に一致するときに，関連するページを表示する。パス情報が空白のときにこのページはデフォルトページとして機能することができる
データ	ほかのモジュールが共有するコンポーネント（たとえば，複数の Web ページモジュールが使用するデータベースコンポーネント）のコンテナとして使用される

Web アプリケーションモジュールは，アプリケーションの複数の機能（リクエストのディスパッチ，セッションの管理，ユーザーリストの保持など）をまとめて実行するアプリケーションコンポーネントのコンテナとして機能します。WebBroker アーキテクチャについてすでに理解していれば，Web アプリケーションモジュールを TWebApplication オブジェクトに似たものとして考えることができます。Web アプリケーションモジュールには，そのタイプに応じて通常の Web ページモジュールまたは Web データモジュールの機能を持たせることもできます。プロジェクトに入れられる Web アプリケーションモジュールは 1 つだけです。Web アプリケーションモジュールが 2 つ以上必要になることはなく，さらに機能を付加したい場合は通常の Web モジュールを追加することができます。

Web アプリケーションモジュールを使うと，サーバーアプリケーションの基本機能のほとんどを備えることができます。サーバーで何らかのホームページを保守する場合，Web アプリケーションモジュールを TWebAppDataModule ではなく TWebAppPageModule にすれば，そのページ用にさらに Web ページを作成する必要がなくなります。

Web ページモジュール

各 Web ページモジュールには，ページプロデューサが関連付けられています。リクエストが受信されると，ページディスパッチャがリクエストを分析し，適切なページモジュールを呼び出してリクエストを処理し，ページのコンテンツを返します。

Web データモジュールと同様に，Web ページモジュールはコンポーネントのコンテナです。ただし，Web ページモジュールは単なるコンテナではありません。Web ページモジュールは，Web ページの作成専用に使用されます。

Web ページモジュールにはすべて，プレビューと呼ばれるエディタビューがあり，作成中のページをプレビューすることができます。これによって，C++Builder が提供するビジュアルアプリケーション開発環境の利点をフルに活用することできます。

ページプロデューサコンポーネント

Web ページモジュールには，ページコンテンツの生成を担当するページプロデューサコンポーネントを識別するプロパティがあります（ページプロデューサの詳細については，34-6 ページの「ページプロデューサ」を参照してください）。Web ページモジュールを作成するときに，WebSnap ページモジュールウィザードは自動的にプロデューサを追加します。後でページプロデューサコンポーネントを変更するには，異なるプロデューサを WebSnap パレットからドロップします。ただし，ページモジュールにテンプレートファイルがある場合は，このファイルのコンテンツが，置換するプロデューサコンポーネントと互換性がある必要があります。

ページ名

Web ページモジュールには，HTTP リクエスト内またはアプリケーションのロジック内でページを参照するために使用できるページ名があります。Web ページモジュールのユニット内のファクトリは，Web ページモジュールのページ名を指定します。

プロデューサテンプレート

大部分のページプロデューサはテンプレートを使用します。HTML テンプレートには通常は，静的 HTML が透過タグやサーバー側スクリプティングと混在しています。ページプロデューサはコンテンツを作成するときに，透過タグを適切な値に置き換え，サーバー側スクリプトを実行して，クライアントブラウザが表示する HTML を作成します（XSLPageProducer は，これとは異なります。XSLPageProducer は，HTML ではなく XSL が含まれる XSL テンプレートを使用します。XSL テンプレートは，透過タグまたはサーバー側スクリプトをサポートしません）。

Web ページモジュールには，ユニットの一部として管理されるテンプレートファイルが関連付けられている場合があります。このテンプレートファイルはプロジェクトマネージャに表示され，基本ファイルの名前と位置がユニットのサービスファイルと同じになります。Web ページモジュールにテンプレートファイルが関連付けられていない場合，ページプロデューサコンポーネントのプロパティがテンプレートを指定します。

Web データモジュール

標準的なデータモジュールと同様に，Web データモジュールはコンポーネントパレットにあるコンポーネントのコンテナです。データモジュールは，コンポーネントを追加，削除，および選択するデザインサーフェスを提供します。Web データモジュールが標準的なデータモジュールと異なる点は，ユニットの構造，および Web データモジュールが実装するインターフェースです。

Web データモジュールは，アプリケーション全体で共有されるコンポーネントのコンテナとして使用します。たとえば，データモジュールにデータセットコンポーネントを入れて，以下の両方からこのデータセットにアクセスすることができます。

- グリッドを表示するページモジュール
- 入力フォームを表示するページモジュール

Web データモジュールを使うと，種類の異なる複数の Web サーバーアプリケーションから使用できるコンポーネントのセットを持つこともできます。

Web データモジュールユニットの構造

標準的なデータモジュールには，フォーム変数と呼ばれる変数があり，この変数を使用してデータモジュールオブジェクトにアクセスします。Web データモジュールでは，この変数のかわりに関数を使用します。関数は Web データモジュールのユニット内で定義され，Web データモジュールと同じ名前になります。関数の目的は，その関数で置き換えられた変数と同じです。WebSnap アプリケーションはマルチスレッドの場合があり，複数のリクエストを同時に処理する特定のモジュールのインスタンスが複数ある可能性があります。したがって，正しいインスタンスを返すためにこの関数が使用されます。

Web データモジュールユニットはファクトリも登録し，WebSnap アプリケーションがどのようにモジュールを管理するかを指定します。たとえば，モジュールをキャッシュしてほかのリクエストに再使用するかどうか，またはリクエストを処理した後にモジュールを破棄するかどうかをフラグで示します。

アダプタ

アダプタは，サーバーアプリケーションへのスクリプトインターフェースを定義します。アダプタを使用すると，ページへのスクリプト言語の挿入，およびスクリプトコードからアダプタへの呼び出しを行うことによる情報の取得ができます。たとえば，アダプタを使用すると，HTML ページに表示するデータフィールドを定義できます。スクリプトが追加された HTML ページには，HTML コンテンツ，およびそれらのデータフィールドの値を取得するスクリプト文が含まれることがあります。これは，WebBroker アプリケーションで使用される透過タグに似ています。アダプタは，コマンドを実行するアクションもサポートしています。たとえば，ハイパーリンクのクリックや HTML フォームの実行（送信）でアダプタのアクションを開始することができます。

アダプタは，HTML ページを動的に作成するタスクを簡単にします。アプリケーションにアダプタを使用すると，条件分岐ロジックとループをサポートするオブジェクト指向スクリプトを含めることができます。アダプタとサーバー側スクリプティングを使わなければ，C++ のイベントハンドラに HTML 生成ロジックをたくさん記述しなければなりません。アダプタを使うことで，開発時間が大幅に短縮されます。

スクリプティングについての詳細は，34-32 ページの「WebSnap でのサーバー側スクリプト」，および 34-35 ページの「リクエストとレスポンスのディスパッチ」を参照してください。

ページコンテンツの作成に使用できるアダプタコンポーネントは，フィールド，アクション，エラー，レコードの 4 種類です。

フィールド

フィールドとは，アプリケーションからのデータの取得，および Web ページへのコンテンツの表示のためにページプロデューサが使用するコンポーネントです。フィールドを使用すると，イメージを取得することもできます。この場合，Web ページに書き込まれるイメージのアドレスをフィールドが返します。ページがコンテンツを表示するときに，Web サーバーアプリケーションにリクエストが送信され，Web サーバーアプリケーションはアダプタディスパッチャを起動して，フィールドコンポーネントから実際のイメージを取得します。

アクション

アクションは，アダプタのかわりにコマンドを実行するコンポーネントです。ページプロデューサがページを生成するときに，スクリプト言語がアダプタアクションコンポーネントを呼び出して，アクションの名前を返し，コマンドの実行に必要なパラメータがあればこのパラメータも返します。たとえば，HTML フォーム上のボタンをクリックしてテーブルから 1 行を削除するとします。この場合，HTTP リクエスト内で，このボタンに関連付けられたアクション名，および行番号を示すパラメータが返されます。アダプタディスパッチャは，この名前のアクションコンポーネントを探し出し，この行番号をパラメータとしてこのアクションに渡します。

エラー

アダプタは，アクションの実行中に発生したエラーのリストを保持します。ページプロデューサは，このエラーリストにアクセスでき，アプリケーションがエンドユーザーに返す Web ページにこれを表示することができます。

レコード

TDataSetAdapter などのいくつかのアダプタコンポーネントは複数のレコードを表します。アダプタは，レコードの繰り返し処理を可能にするスクリプトインターフェースを提供します。一部のアダプタはページングをサポートし，現在のページのレコードだけを繰り返し処理します。

ページプロデューサ

ページプロデューサを使用すると，Web ページモジュールのかわりにコンテンツを生成することができます。ページプロデューサは以下の機能を提供します。

- HTML コンテンツを生成する
- HTMLFile プロパティを使用して外部ファイルを参照するか，HTMLDoc プロパティを使用して内部文字列を参照できる
- プロデューサを Web ページモジュールとともに使用する場合は，ユニットに関連付けたファイルをテンプレートにすることができる
- プロデューサは，透過タグまたはアクティブスクリプティングを使用してテンプレートに挿入できる HTML を動的に生成する。透過タグは，WebBroker アプリケーションと同じように使用できる。透過タグの使い方の詳細については，33-15 ページの「HTML 透過タグを変換する」を参照。アクティブスクリプティングをサポートするので，HTML ページの中に JScript または VBScript を埋め込むことができる

WebSnap でページプロデューサを使うための標準的な方法は以下のとおりです。Web ページモジュールを作成するときには，Web ページモジュールウィザードでページプロデューサの種類を選択しなければなりません。たくさんの選択肢がありますが，ほとんどの WebSnap 開発者は，アダプタページプロデューサ（TAdapterPageProducer）を使ってページのプロトタイプを作成します。アダプタページプロデューサを使うと，標準的なコンポーネントモデルに類似したプロセスによって Web ページのプロトタイプを作成できます。開発者は，ある種のフォーム（アダプタフォーム）をアダプタページプロデューサに追加します。必要に応じて，アダプタフォームにアダプタコンポーネント（たとえばアダプタグリッド）を追加できます。アダプタページプロデューサを使えば，C++Builder でユーザーインターフェースを作成する標準的な技法に近い方法で，Web ページを作成することができます。

状況によっては，アダプタページプロデューサから通常のページプロデューサへの切り替えた方が適切な場合があります。たとえば，アダプタページプロデューサの機能の 1 つに，ページテンプレート内のスクリプトを実行時に動的に生成することがあります。サーバーの最適化のためには，静的なスクリプトを使うことが考えられます。また，スクリプト作成の経験を積んだ開発者であれば，スクリプトに直接変更を加えたいと考えることもあります。このような場合には，動的なスクリプトと静的なスクリプトの競合を避けるために，通常のページプロデューサを使用しなければなりません。通常のページプロデューサを変更する方法については，34-24 ページの「HTML の高度な設計」を参照してください。

ページプロデューサを Web ディスパッチャアクションフィールドに関連付けることで，プロデューサを WebBroker アプリケーションで使用する場合と同じように使用することもできます。Web ページモジュールを使用する利点は，以下のとおりです。

- アプリケーションを実行せずにページのレイアウトをプレビューできる
- ページ名をモジュールに関連付けて，ページディスパッチャが自動的にページプロデューサを呼び出せるようにできる

WebSnap による Web サーバーアプリケーションの作成

WebSnap のソースコードを見れば，WebSnap は数百のオブジェクトで構成されていることが分かります。実際，WebSnap はオブジェクトと機能が豊富なので，アーキテクチャを完全に理解するには，ある程度の時間をかける必要があるでしょう。幸いなことに，実際には WebSnap の全システムを理解していなくても，サーバーアプリケーションの開発を始めることができます。

ここでは，Web サーバーアプリケーションを作成しながら，WebSnap のしくみを学んでいきます。

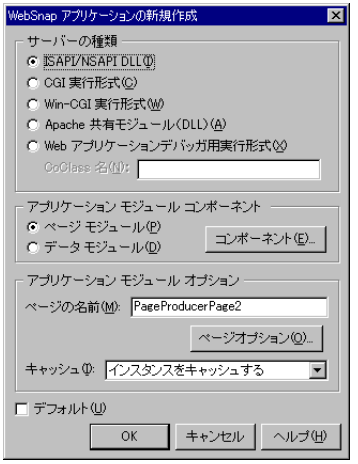
WebSnap アーキテクチャを使用して新しい Web サーバーアプリケーションを作成する手順は次のとおりです。

1. ［ファイル｜新規作成｜その他］を選択します。
2. ［新規作成］ダイアログボックスで，［WebSnap］タブを選択し，［WebSnap アプリケーション］を選択します。

   ダイアログボックスが表示されます（図 34.1 参照）。

3. 正しいサーバーの種類を指定します。
4. ［コンポーネント］ボタンを使用して，アプリケーションモジュールコンポーネントを指定します。
5. ［ページオプション］ボタンを使用して，アプリケーションモジュールオプションを選択します。

図 34.1 ［WebSnap アプリケーションの新規作成］ダイアログ



サーバーの種類の選択

アプリケーションで使う Web サーバーの種類に応じて，Web サーバーアプリケーションの種類を以下から 1 つ選択します。

表 34.2 Web サーバーアプリケーションの種類

サーバーの種類	説明
ISAPI と NSAPI	Web サーバーが要求するエクスポートされたメソッドを持つ DLL としてプロジェクトを設定する
Apache	Apache Web サーバーが要求するエクスポートされたメソッドを持つ DLL としてプロジェクトを設定する
CGI 実行形式	CGI（Common Gateway Interface）規格に準拠するコンソールアプリケーションとしてプロジェクトを設定する
WinCGI 実行形式	Windows CGI アプリケーション（Windows に採用された CGI 規格の派生形式）としてプロジェクトを設定する
Web アプリケーションデバッガ用実行形式ファイル	Web サーバーアプリケーションの開発およびテストのための環境を作成する。この種類のアプリケーションは配布を目的としていない

アプリケーションモジュールコンポーネントの指定

アプリケーションコンポーネントは，Web アプリケーションの機能を提供します。たとえば，アダプタディスパッチャコンポーネントを含めると，HTML フォームの送信，および動的に生成された

イメージの返信が自動的に処理されます。ページディスパッチャを含めると，HTTP リクエストのパス情報にページの名前が含まれている場合にページのコンテンツが自動的に表示されます。

［WebSnap アプリケーションの新規作成］ダイアログ（図 34.1 を参照）の［コンポーネント］ボタンを選択すると，別のダイアログが開いて，Web アプリケーションモジュールコンポーネントを選択することができます。この［Web アプリケーションコンポーネント］ダイアログを図 34.2 に示します。

図 34.2 ［Web アプリケーションコンポーネント］ダイアログ

使用できるコンポーネントとその説明を次の表に示します。

表 34.3 Web アプリケーションコンポーネント

コンポーネント型	説明
アプリケーション アダプタ	アプリケーションについての情報（タイトルなど）が含まれる。デフォルト型は TApplicationAdapter
エンドユーザー アダプタ	ユーザーについての情報（名前，アクセス権，ログインしているかどうかなど）が含まれる。デフォルト型は TEndUserAdapter。TEndUserSessionAdapter も選択できる
ページ ディスパッチャ	HTTP リクエストのパス情報を調べ，ページのコンテンツを返す適切なページモジュールを呼び出す。デフォルト型は TPageDispatcher
アダプタ ディスパッチャ	アダプタのアクションおよびフィールドコンポーネントを呼び出すことで，HTML フォームの送信，および動的イメージのリクエストを自動的に処理する。デフォルト型は TAdapterDispatcher
ディスパッチャ アクション	パス情報とメソッド型に基づいてリクエストを処理するアクションフィールドの集合を定義できる。アクションフィールドは，ユーザー定義のイベントを呼び出すか，ページプロデューサコンポーネントのコンテンツを要求する。デフォルト型は TWebDispatcher
ファイルサービス ロケータ	Web アプリケーションの実行中に，テンプレートファイル，およびスクリプトインクルードファイルのロードを制御する。デフォルト型は TLocateFileService
セッション サービス	短時間のあいだ必要なエンドユーザーについての情報を格納する。たとえば，セッションを使用すると，ログインしたユーザーの追跡，および非アクティブ状態が一定時間続いた後のユーザーの自動的ログアウトができる。デフォルト型は TSessionsService
ユーザーリスト サービス	許可されたユーザー，およびユーザーのパスワードとアクセス権を追跡する。デフォルト型は TWebUserList

上記の各コンポーネントに対してリストされたコンポーネント型は，C++Builder 出荷時のデフォルト型です。ユーザーが独自のコンポーネント型を作成したり，サードパーティのコンポーネント型を使用することもできます。

Web アプリケーションモジュールのオプションの選択

選択したアプリケーションモジュールの種類がページモジュールの場合，［WebSnap アプリケーションの新規作成］ダイアログボックスの［ページの名前］フィールドに名前を入力してページに名前を関連付けることができます。実行時に，このモジュールのインスタンスは，キャッシュ内に保持するか，リクエストの処理が済んだときにメモリから除去することができます。［キャッシュ］フィールドからいずれかのオプションを選択します。このほかのページモジュールオプションを選択するには，［ページオプション］ボタンを選択します。［アプリケーションモジュールページのオプション］ダイアログに以下のカテゴリが表示されます。

- ［Producer の選択］：ページのプロデューサの種類は，AdapterPageProducer，DataSetPageProducer，InetXPageProducer，PageProducer，または XSLPageProducer の 1 つに設定できる。選択したページプロデューサがスクリプトをサポートする場合は，［スクリプトエンジン］ドロップダウンリストを使用して，ページのスクリプトに使用する言語を選択する。

メモ AdapterPageProducer は JScript のみをサポートします。

- ［HTML］：選択したプロデューサが HTML テンプレートを使用する場合にこのグループが表示される
- ［XSL］：選択したプロデューサが XSL テンプレート（TXSLPageProducer など）を使用する場合にこのグループが表示される
- ［ファイルの新規作成］：テンプレートファイルを作成し，ユニットの一部として管理する場合は［ファイルの新規作成］をチェックする。管理されたテンプレートファイルはプロジェクトマネージャに表示され，ファイルの名前と位置がユニットのソースファイルと同じになる。プロデューサコンポーネントのプロパティ（通常は HTMLDoc または HTMLFile プロパティ）を使用する場合は［ファイルの新規作成］のチェックマークをはずす
- ［ひな形］：［ファイルの新規作成］をチェックした場合は，テンプレートファイルのデフォルトのコンテンツを［ひな形］ドロップダウンから選択する。標準的なテンプレートは，アプリケーションのタイトル，ページのタイトル，および公開されたページへのハイパーリンクを表示する。空白のテンプレートは空白のページを作成する
- ［ページ］：ページモジュールのページ名とタイトルを入力する。ページの名前は，HTTP リクエスト内またはアプリケーションのロジック内でページを参照するために使用され，タイトルは，ページがブラウザに表示されたときにエンドユーザーが目にする名前である
- ［公開する］：これをチェックすると，リクエストメッセージ内のパス情報とページ名が一致する HTTP リクエストにページが自動的に応答することができる
- ［ログインを必要とする］：これをチェックすると，ユーザーがページにアクセスする前にログインするように要求できる

ここまでは WebSnap サーバーアプリケーションの作成を開始する方法を見てきました。次の「WebSnap のチュートリアル」では，アプリケーションの開発を進めて完成に近づける方法を説明します。

WebSnap のチュートリアル

このチュートリアルでは，WebSnap アプリケーションを構築する手順を説明します。WebSnap HTML コンポーネントを使用して，テーブルの内容を編集するアプリケーションを構築する方法を，この WebSnap アプリケーションが示します。最初から最後まで，説明に従って学習を進めてください。途中で休憩したい場合は，［ファイル｜すべて保存］を使用すればいつでもプロジェクトを保存できます。

新しいアプリケーションの作成

このトピックでは，Country Tutorial という新しい WebSnap アプリケーションの作成手順を説明します。このアプリケーションは，さまざまな国に関する情報の一覧表を Web 上でユーザーに表示します。ユーザーは，国の追加と削除，既存の国についての情報の編集ができます。

ステップ 1：WebSnap アプリケーションウィザードを起動する

1. C++Builder を実行し，［ファイル｜新規作成｜その他］を選択します。
2. ［新規作成］ダイアログボックスで，［WebSnap］タブを選択し，［WebSnap アプリケーション］を選択します。
3. ［WebSnap アプリケーションの新規作成］ダイアログボックスで，次の操作を行います。
   - ［Web アプリケーションデバッガ用実行形式］を選択します。
   - ［CoClass 名］編集コントロールに CountryTutorial と入力します。
   - コンポーネントの種類として［ページモジュール］を選択します。
   - ［ページの名前］フィールドに Home と入力します。
4. ［OK］をクリックします。

ステップ 2：生成されたファイルとプロジェクトを保存する

Pascal ユニットのファイルとプロジェクトを保存する手順は次のとおりです。

1. ［ファイル｜すべて保存］を選択します。
2. 保存のダイアログで，Country Tutorial プロジェクトのファイルをすべて保存できる適切なディレクトリに変更します。
3. ［ファイル名］フィールドに HomeU.cpp と入力し，［保存］をクリックします。
4. ［ファイル名］フィールドに CountryU.cpp と入力し，［保存］をクリックします。
5. ［ファイル名］フィールドに CountryTutorial.bpr と入力し，［保存］をクリックします。

メモ アプリケーションが実行形式ファイルと同じディレクトリで HomeU.html ファイルを探すので，ユニットとプロジェクトは同じディレクトリに保存します。

ステップ 3：アプリケーションのタイトルを指定する

アプリケーションのタイトルとは，エンドユーザーに表示される名前です。アプリケーションのタイトルを指定する手順は次のとおりです。

1. ［表示｜プロジェクトマネージャ］を選択します。
2. プロジェクトマネージャで，CountryTutorial.exe を展開し，HomeU エントリをダブルクリックします。
3. オブジェクトインスペクタウィンドウの最上行で，プルダウンリストから［ApplicationAdapter］を選択します。
4. ［プロパティ］タブで，［ApplicationTitle］プロパティに Country Tutorial と入力します。
5. エディタウィンドウの［プレビュー］タブをクリックします。アプリケーションのタイトルが，ページ名である Home とともにページの上部に表示されます。

このページはごく基本的なものです。エディタの［HomeU.html］タブを使うか外部エディタを使えば，HomeU.html ファイルを編集して改良することもできます。ページテンプレートを編集する方法については，34-24 ページの「HTML の高度な設計」を参照してください。

CountryTable ページの作成

Web ページモジュールは，公開されるページを定義します。またデータコンポーネントのコンテナとしても機能します。Web ページをエンドユーザーに返す必要があるときには，Web ページモジュールは必要な情報をそのデータコンポーネントから取り出し，その情報を使ってページを作成します。

ここでは，WebSnap アプリケーションに新しいページモジュールを追加します。このページモジュールは，目に見えるもう 1 つのページを Country Tutorial アプリケーションに追加します。1 ページ目の Home は，アプリケーションを作成したときに定義されました。2 ページ目の CountryTable は，国に関する情報の一覧表を表示します。

ステップ 1：新しい Web ページモジュールを追加する

新しいページモジュールを追加する手順は次のとおりです。

1. ［ファイル｜新規作成｜その他］を選択します。
2. ［新規作成］ダイアログボックスで，［WebSnap］タブを選択し，［WebSnap ページモジュール］を選択し，［OK］をクリックします。
3. このダイアログボックスで，［Producer の選択｜種類］を［AdapterPageProducer］に設定します。
4. ［ページ｜名前］フィールドに CountryTable と入力します。この入力に合わせて［タイトル］も変更されます。
5. その他のフィールドはデフォルト値のままにします。

   ダイアログは図 34.3 のようになっているはずです。
6. ［OK］をクリックします。

図 34.3　CountryTable ページを作成する［WebSnap ページモジュールの新規作成］ダイアログボックス

これで CountryTable モジュールは，IDE に表示されています。モジュールを保存してから，CountryTable モジュールに新しいコンポーネントを追加します。

ステップ 2：新しい Web ページモジュールを保存する

ユニットをプロジェクトファイルのディレクトリに保存します。アプリケーションを実行すると，実行形式ファイルと同じディレクトリで CountryTableU.html ファイルが検索されます。

1. ［ファイル｜上書き保存］を選択します。
2. ［ファイル名］フィールドに CountryTableU.cpp と入力し，［保存］をクリックします。

CountryTable モジュールへのデータコンポーネントの追加

TTable と TDataSetAdapter はデータベース対応のコンポーネントであり，データへのアクセスを提供します。TTable は，HTML テーブルにデータを提供します。TDataSetAdapter を使用すると，サーバー側スクリプトが TTable コンポーネントにアクセスできます。ここでは，これらのデータベース対応コンポーネントをアプリケーションに追加します。

以下のステップ 1 とステップ 2 はデータベースプログラミングの経験があることを前提にしていますが，経験者でなくてもこのチュートリアルを完成できます。WebSnap は単に（アダプタコンポーネントを通して）データベースコンポーネントへのインターフェースとして機能するだけです。データベースプログラミングについて詳しく知りたい場合は，このマニュアルの第 II 部の各章を参照してください。

ステップ 1：データベース対応のコンポーネントを追加する

1. ［表示｜プロジェクトマネージャ］を選択します。

2. プロジェクトマネージャで，CountryTutorial.exe を展開し，CountryTableU エントリをダブルクリックします。
3. ［表示｜オブジェクトツリー］を選択します。オブジェクトツリーウィンドウがアクティブになります。
4. コンポーネントパレットの［BDE］タブを選択します。
5. Table コンポーネントを選択して，CountryTable Web ページモジュールに追加します。
6. Session コンポーネントを選択して，CountryTable Web ページモジュールに追加します。マルチスレッドアプリケーションでは BDE コンポーネント（TTable）を使用するので，Session コンポーネントが必要になります。
7. Web ページモジュールまたはオブジェクトツリーで，Session コンポーネントを選択します（デフォルトで Session1 という名前が付いています）。オブジェクトインスペクタに Session コンポーネントの値が表示されます。
8. オブジェクトインスペクタで，AutoSessionName プロパティを **true** に設定します。
9. Web ページモジュールまたはオブジェクトツリーで，Table コンポーネントを選択します。オブジェクトインスペクタに Table コンポーネントの値が表示されます。
10. オブジェクトインスペクタで，以下のプロパティを変更します。
    - DatabaseName プロパティを BCDEMOS に設定します。
    - Name プロパティに Country と入力します。
    - TableName プロパティを country.db に設定します。
    - Active プロパティを **true** に設定します。

ステップ 2：キー項目を指定する

キー項目は，テーブル内でレコードを識別するために使用されます。このことは，アプリケーションに編集ページを追加するときに重要になります。キー項目を指定する手順は次のとおりです。

1. オブジェクトツリーで，［Session］ノードと［BCDEMOS］ノードを展開し，country.db ノードを選択します。このノードは Country Table コンポーネントです。
2. country.db ノードを右クリックし，［項目の設定］を選択します。
3. ［CountryTable->Country］ウィンドウを右クリックし，［すべての項目の追加］を選択します。
4. 追加された項目のリストから Name 項目を選択します。
5. オブジェクトインスペクタで，ProviderFlags プロパティを展開します。
6. pfInKey プロパティを **true** に設定します。

ステップ 3：アダプタコンポーネントを追加する

ここまででデータベースコンポーネントの追加が終わりました。次は，TTable 内でサーバー側スクリプトを通してデータをエクスポーズするために，データセットアダプタ（TDataSetAdapter）コンポーネントを含めなければなりません。データセットアダプタを追加する手順は次のとおりです。

1. コンポーネントパレットの［WebSnap］タブから DataSetAdapter コンポーネントを選択します。このコンポーネントを CountryTable Web モジュールに追加します。

2. オブジェクトインスペクタで以下のプロパティを変更します。
   - DataSet プロパティを Country に設定します。
   - Name プロパティを Adapter に設定します。

   以上の作業が終わると，CountryTable Web ページモジュールは図 34.4 のようになっているでしょう。

図 34.4　CountryTable Web ページモジュール

モジュール内の要素はビジュアルではないので，モジュールのどこに表示されているかは問題ではありません。重要なのは，モジュールにこの図と同じコンポーネントが含まれていることです。

データを表示するグリッドの作成

ステップ 1：グリッドを追加する

Country テーブルデータベースからのデータを表示するグリッドを追加する手順は次のとおりです。

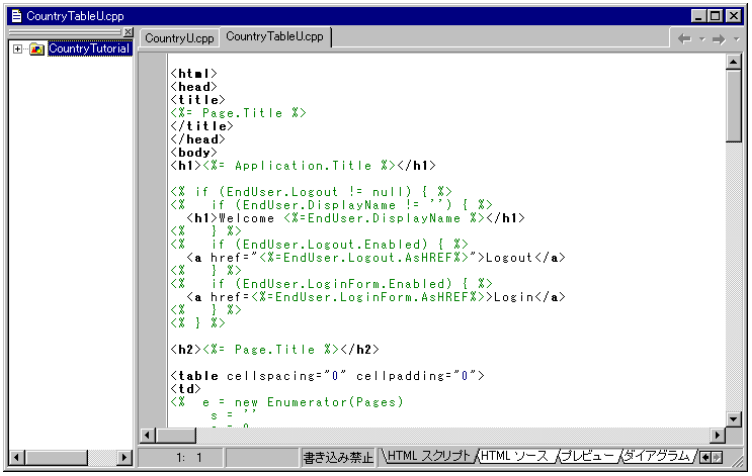
1. ［表示｜プロジェクトマネージャ］を選択します。
2. プロジェクトマネージャで，CountryTutorial.exe を展開し，CountryTableU エントリをダブルクリックします。
3. ［表示｜オブジェクトツリー］を選択し，オブジェクトツリー上でクリックします。
4. AdapterPageProducer コンポーネントを展開します。このコンポーネントは，HTML テーブルをすばやく作成するサーバー側スクリプトを生成します。
5. WebPageItems エントリを右クリックし，［コンポーネントの新規作成］を選択します。
6. ［Web コンポーネントの追加］ダイアログボックスで，AdapterForm を選択し，［OK］をクリックします。オブジェクトツリーに AdapterForm1 コンポーネントが表示されます。
7. AdapterForm1 を右クリックし，［コンポーネントの新規作成］を選択します。
8. ［Web コンポーネントの追加］ウィンドウで，AdapterGrid を選択し，［OK］をクリックします。オブジェクトツリーに AdapterGrid1 コンポーネントが表示されます。
9. オブジェクトインスペクタウィンドウで，Adapter プロパティを Adapter に設定します。

ページをプレビューするには，コードエディタの上部で［CountryTableU］タブを選択し，下部で［プレビュー］タブを選択します。［プレビュー］タブが表示されていない場合は，下部の右矢印を使用して，タブをスクロールします。プレビューは図 34.5 のようになっているはずです。

図 34.5　CountryTable の［プレビュー］タブ

［プレビュー］タブには，Web ブラウザに表示されるような，最終的な（静的な）HTML ページが表示されます。このページは，スクリプトを含む動的 HTML ページから生成されます。スクリプトコマンドをテキスト表示するには，エディタウィンドウの下部にある［HTML スクリプト］タブを選択します（図 34.6 参照）。

図 34.6　CountryTable の［HTML スクリプト］タブ



［HTML スクリプト］タブには，HTML とスクリプトの両方が表示されます。エディタ内の HTML とスクリプトは，フォントの色と属性で区別されます。デフォルトでは，HTML タグは黒の太字で表示され，HTML 属性名は黒，HTML 属性値は青で表示されます。スクリプトは，<% と %> で挟まれて，緑で表示されます。これらの項目のデフォルトのフォントの色と属性は，［エディタオプション］ダイアログの［色］タブで変更できます。このダイアログは，エディタで右クリックして［プロパティ］を選択すると表示できます。

HTML 関連のエディタタブは，他にあと 2 つあります。［HTML ソース］タブは，プレビューの HTML コンテンツをそのまま表示します。［HTML ソース］，［HTML スクリプト］，［プレビュー］は，すべて表示専用です。HTML 関連のもう 1 つのエディタタブの［CountryTable.html］は編集用に使用できます。

このページの外観を改良したいときは，［CountryTable.html］タブか外部エディタを使えば，いつでも HTML を追加することができます。ページテンプレートを編集する方法については，34-24 ページの「HTML の高度な設計」を参照してください。

ステップ 2：編集コマンドをグリッドに追加する

ユーザーがテーブルの行の削除，挿入，編集によって，テーブルの内容を更新する必要がある場合があります。ユーザーがこのような更新を行えるようにするには，コマンドコンポーネントを追加します。

コマンドコンポーネントを追加する手順は次のとおりです。

1. CountryTable のオブジェクトツリーで，AdapterPageProducer コンポーネントとそのブランチをすべて展開します。
2. AdapterGrid1 コンポーネントを右クリックし，［すべてのカラムを追加］を選択します。5 つのカラムがアダプタグループに追加されます。
3. AdapterGrid1 コンポーネントをもう一度右クリックし，［コンポーネントの新規作成］を選択します。
4. AdapterCommandColumn を選択し，［OK］をクリックします。AdapterCommandColumn1 エントリが AdapterGrid1 コンポーネントに追加されます。
5. AdapterCommandColumn1 を右クリックし，［コマンドの追加］を選択します。
6. DeleteRow，EditRow，および NewRow コマンドを複数選択し，［OK］をクリックします。
7. ページをプレビューするには，コードエディタの下部で［プレビュー］タブをクリックします。これで，テーブルの各行の端に 3 つの新しいボタン（［削除］，［編集］，［挿入］）が表示されます（図 34.7 を参照）。アプリケーションの実行中は，3 つのボタンのどれかを押すと，それぞれ関連付けられているアクションが実行されます。

先に進む前に，［すべて保存］ボタンをクリックしてアプリケーションを保存します。

図 34.7 編集コマンドの追加後の CountryTable のプレビュー

編集フォームの追加

こんどは，country テーブルの編集フォームを処理する Web ページモジュールを作成します。ユーザーが，編集フォームを通して CountryTutorial アプリケーションでデータを編集できるようにします。具体的には，ユーザーが［挿入］または［編集］ボタンを押したときに，編集フォームが表示されるようにします。ユーザーの編集フォームでの作業が終わったら，変更後の情報が自動的にテーブルに表示されるようにします。

ステップ 1：新しい Web ページモジュールを追加する

新しい Web ページモジュールを追加する手順は次のとおりです。

1. ［ファイル｜新規作成｜その他］を選択します。
2. ［新規作成］ダイアログボックスで，［WebSnap］タブを選択し，［WebSnap ページモジュール］を選択します。
3. このダイアログボックスで，［Producer の選択｜種類］を［AdapterPageProducer］に設定します。
4. ［ページ｜名前］フィールドを CountryForm に設定します。
5. ［公開する］ボックスのチェックを外して，このサイトの利用可能ページのリストにこのページが表示されないようにします。編集フォームは［編集］ボタンからアクセスされ，その内容は country テーブルのどの行が編集されるかによって変わります。
6. その他のフィールドと選択肢はデフォルト値に設定したままにします。［OK］をクリックします。

ステップ 2：新しいモジュールを保存する

このモジュールをプロジェクトファイルと同じディレクトリに保存します。アプリケーションを実行すると，実行形式ファイルと同じディレクトリで CountryFormU.html ファイルが検索されます。

1. ［ファイル｜上書き保存］を選択します。

2. ［ファイル名］フィールドに CountryFormU.cpp と入力し，［OK］をクリックします。

ステップ 3：新しいモジュールが CountryTableU にアクセスできるようにする

CountryTableU ユニットを include 指令に追加して，モジュールが Adapter コンポーネントにアクセスできるようにします。

1. ［ファイル｜ユニットヘッダーファイルの追加］を選択します。
2. リストから CountryTableU を選択し，［OK］をクリックします。
3. ［ファイル｜上書き保存］を選択します。

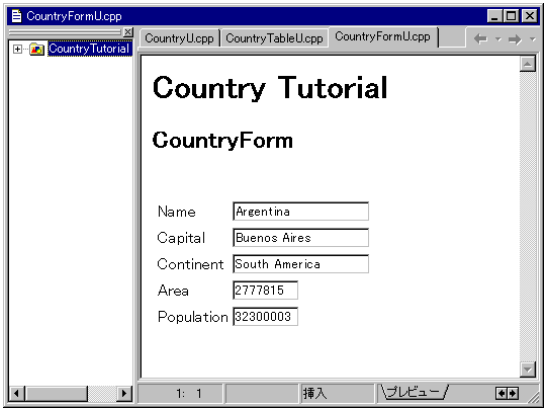
ステップ 4：入力フィールドを追加する

AdapterPageProducer コンポーネントにコンポーネントを追加して，HTML フォーム内にデータ入力フィールドを生成します。

入力フィールドを追加する手順は次のとおりです。

1. ［表示｜プロジェクトマネージャ］を選択します。
2. プロジェクトマネージャウィンドウで，CountryTutorial.exe を展開し，CountryFormU エントリをダブルクリックします。
3. オブジェクトツリーで，AdapterPageProducer コンポーネントを展開し，WebPageItems を右クリックし，［コンポーネントの新規作成］を選択します。
4. AdapterForm を選択し，［OK］をクリックします。オブジェクトツリーに AdapterForm1 エントリが表示されます。
5. AdapterForm1 を右クリックし，［コンポーネントの新規作成］を選択します。
6. AdapterFieldGroup を選択し，［OK］をクリックします。オブジェクトツリーに AdapterFieldGroup1 エントリが表示されます。
7. オブジェクトインスペクタウィンドウで，Adapter プロパティを CountryTable->Adapter に設定します。AdapterMode プロパティを Edit に設定します。
8. ページをプレビューするには，コードエディタの下部で［プレビュー］タブをクリックします。図 34.8 のようなプレビューになるはずです。

図 34.8　CountryForm のプレビュー

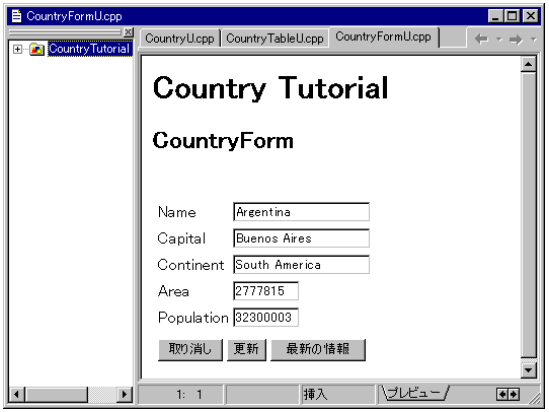


ステップ 5：ボタンを追加する

AdapterPageProducer コンポーネントにコンポーネントを追加して，HTML フォーム内に送信ボタンを生成します。コンポーネントを追加する手順は，次のとおりです。

1. オブジェクトツリーで，AdapterPageProducer コンポーネントとそのブランチをすべて展開します。
2. AdapterForm1 を右クリックし，［コンポーネントの新規作成］を選択します。
3. AdapterCommandGroup を選択し，［OK］をクリックします。オブジェクトツリーに AdapterCommandGroup1 エントリが表示されます。
4. オブジェクトインスペクタで，DisplayComponent プロパティを AdapterFieldGroup1 に設定します。
5. AdapterCommandGroup1 エントリを右クリックし，［コマンドの追加］を選択します。
6. Cancel，Apply，および RefreshRow コマンドをまとめて選択し，［OK］をクリックします。
7. ページをプレビューするには，コードエディタウィンドウの下部で［プレビュー］タブをクリックします。プレビューで country フォームが表示されない場合は，上部の［CountryU］タブをクリックしてから［CountryFormU］をクリックし，下部の［プレビュー］タブを再度クリックします。図 34.9 のようなプレビューになるはずです。

図 34.9　送信ボタンがある CountryForm



ステップ 6：フォームアクションをグリッドページにリンクする

ユーザーがボタンをクリックすると，目的のアクションを行うアダプタアクションが実行されます。アダプタアクションの実行後に表示されるページを指定する手順は次のとおりです。

1. オブジェクトツリーで，AdapterCommandGroup1 を展開して，CmdCancel，CmdApply，および CmdRefreshRow エントリを表示します。
2. CmdCancel を選択します。オブジェクトインスペクタウィンドウで，PageName プロパティに CountryTable と入力します。
3. CmdApply を選択します。オブジェクトインスペクタウィンドウで，PageName プロパティに CountryTable と入力します。

ステップ 7：グリッドアクションをフォームページにリンクする

グリッド内のボタンを押すとアダプタアクションが実行されます。アダプタアクションに応答して表示されるページを指定する手順は次のとおりです。

1. ［表示｜プロジェクトマネージャ］を選択します。
2. プロジェクトマネージャで，CountryTutorial.exe を展開し，CountryTableU エントリをダブルクリックします。
3. オブジェクトツリーで，AdapterPageProducer コンポーネントとそのブランチをすべて展開して，CmdNewRow，CmdEditRow，および CmdDeleteRow エントリを表示します。これらのエントリは，AdapterCommandColumn1 エントリの下に表示されます。
4. CmdNewRow を選択します。オブジェクトインスペクタで，PageName プロパティに CountryForm と入力します。
5. CmdEditRow を選択します。オブジェクトインスペクタで，PageName プロパティに CountryForm と入力します。

アプリケーションが動作すること，およびすべてのボタンがなんらかのアクションを実行することを確認するために，アプリケーションを実行します。アプリケーションの実行方法については，34-22 ページの「完成したアプリケーションの実行」を参照してください。アプリケーションを実行する

と，サーバーを実行していることになります。アプリケーションの動作をチェックするためには，Web ブラウザでそのページを表示する必要があります。これは，Web アプリケーションデバッガからアプリケーションを起動することによって行えます。

メモ 型が無効などのデータベースエラーは示されません。たとえば，Area フィールドに無効な値（'abc' など）を持つ国を新しく追加してみてください。

完成したアプリケーションの実行

完成したアプリケーションを実行する手順は次のとおりです。

1. ［実行｜実行］を選択します。フォームが表示されます。Web アプリケーションデバッガの実行可能な Web アプリケーションは COM サーバーであり，表示されたフォームは COM サーバーのコンソールウィンドウです。初めてプロジェクトを実行したときに，COM オブジェクトが登録され，Web アプリケーションデバッガが直接アクセスできるようになります。
2. ［ツール｜ Web アプリケーションデバッガ］を選択します。
3. デフォルトの URL リンクをクリックして ServerInfo ページを表示します。ServerInfo ページは，登録されたすべての Web アプリケーションデバッガ実行形式ファイルの名前を表示します。
4. このドロップダウンリストで［CountryTutorial］を選択し，［実行］ボタンをクリックします。

これでブラウザに Country Tutorial アプリケーションが表示されます。CountryTable リンクをクリックして CountryTable ページを表示します。

エラー報告の追加

この時点のアプリケーションは，ユーザーにエラーを表示しません。たとえば，country レコードの Area フィールドに文字を入力しても，エラーが発生したという通知は表示されません。ここでは，AdapterErrorList コンポーネントを追加して，country テーブルを編集するアダプタアクションの実行中に発生したエラーを表示します。

ステップ 1：エラーサポートをグリッドに追加する

エラーサポートをグリッドに追加する手順は次のとおりです。

1. CountryTable のオブジェクトツリーで，AdapterPageProducer コンポーネントとそのブランチをすべて展開して AdapterForm1 を表示します。
2. AdapterForm1 を右クリックし，［コンポーネントの新規作成］を選択します。
3. リストから AdapterErrorList を選択し，［OK］をクリックします。オブジェクトツリーに AdapterErrorList1 エントリが表示されます。
4. AdapterErrorList1 を AdapterGrid1 の上に移動します（ドラッグするか，オブジェクトツリーツールバーの上向き矢印を使用します）。
5. オブジェクトインスペクタで，Adapter プロパティを Adapter に設定します。

ステップ 2：エラーサポートをフォームに追加する

エラーサポートをフォームに追加する手順は次のとおりです。

1. CountryForm のオブジェクトツリーで，AdapterPageProducer コンポーネントとそのブランチをすべて展開して AdapterForm1 を表示します。
2. AdapterForm1 を右クリックし，［コンポーネントの新規作成］を選択します。
3. リストから AdapterErrorList を選択し，［OK］をクリックします。オブジェクトツリーに AdapterErrorList1 エントリが表示されます。
4. AdapterErrorList1 を AdapterFieldGroup1 の上に移動します（ドラッグするか，オブジェクトツリーツールバーの上向き矢印を使用します）。
5. オブジェクトインスペクタで，Adapter プロパティを CountryTable->Adapter に設定します。

ステップ 3：エラー報告メカニズムをテストする

ここでエラー報告メカニズムをテストするためには，C++Builder IDE を少し変更しておく必要があります。［ツール｜デバッガオプション］を選択します。［言語固有の例外］タブで，［Delphi 言語の例外で停止］チェックボックスのチェックをはずします。これによって，アプリケーションは例外が検出されたときに停止せずに動作を続けます。グリッドエラーをテストする手順は次のとおりです。

1. Web アプリケーションデバッガを使ってアプリケーションを実行し，CountryTable ページを表示します。その方法については，34-22 ページの「完成したアプリケーションの実行」を参照してください。
2. ブラウザウィンドウをもう 1 つ開き，CountryTable ページを表示します。
3. グリッドの 1 行目の［削除］ボタンをクリックします。
4. もう 1 つのブラウザを更新せずに，グリッドの 1 行目の［削除］ボタンをクリックします。

グリッドの上に，「Country の Row が見つかりません」というエラーメッセージが表示されます。

フォームエラーをテストする手順は次のとおりです。

1. Web アプリケーションデバッガを使ってアプリケーションを実行し，CountryTable ページを表示します。
2. ［編集］ボタンをクリックします。CountryForm ページが表示されます。
3. Area フィールドを 'abc' に変更し，［更新］ボタンをクリックします。

最初のフィールドの上に「項目 'Area' の値として不正です」というエラーメッセージが表示されます。

以上で，WebSnap チュートリアルが完成しました。操作を続ける前に，［Delphi 言語の例外で停止］チェックボックスのチェックを元に戻しておくとよいでしょう。また，完成したアプリケーションを後で参照できるようにするために，［ファイル｜すべて保存］を選択してアプリケーションを保存します。

HTML の高度な設計

アダプタとアダプタページプロデューサを使うことで，WebSnap は Web サーバーアプリケーションでのスクリプト付き HTML ページの作成を容易にしています。ニーズに見合う WebSnap ツールを使って，アプリケーションデータの Web フロントエンドを作成することができます。しかし，WebSnap の強力な特徴の 1 つは，Web 設計の専門技術を他のソースからアプリケーションに組み込める機能です。このセクションでは，Web サーバーの設計と保守のプロセスを拡張して，他のツールとプログラマ以外のチームメンバーを取り込むための指針について説明します。

WebSnap 開発の最終製品は，サーバーアプリケーションと，そのサーバーが生成するページのための HTML テンプレートです。テンプレートには，スクリプティングと HTML の両方が含まれます。これらは，最初に作成されたあとは，いつでも任意の HTML ツールを使って編集することができます（編集時に誤ってスクリプトを壊してしまわないためには，埋め込みのスクリプトタグをサポートする Microsoft FrontPage などのツールを使うのがよいでしょう）。IDE の外でテンプレートページを編集できる方法はたくさんあります。

たとえば，C++ Builder 開発者であれば，設計時に好みの外部エディタを使って HTML テンプレートを編集できます。これによって，外部 HTML エディタがサポートする（C++ Builder がサポートしていない）フォーマッティングやスクリプティングの高度な機能を利用することができます。IDE から外部 HTML エディタを使用できるようにする手順は次のとおりです。

1. メインメニューから［ツール｜環境オプション］を選択します。［環境オプション］ダイアログで［インターネット］タブをクリックします。
2. ［インターネットファイルタイプ］ボックスで HTML を選択し，［編集］ボタンをクリックして［ファイルタイプの編集］ダイアログボックスを表示します。
3. ［アクション］ボックスで，使用する HTML エディタに関連付けるアクションを選択します。たとえば，使用しているシステムのデフォルトの HTML エディタを選択するには，ドロップダウンリストから［Edit］を選択します。［OK］を 2 回クリックして，［ファイルタイプの編集］ダイアログボックスと［環境オプション］ダイアログボックスを閉じます。

HTML テンプレートを編集するには，そのテンプレートが含まれるユニットを開きます。編集ウィンドウで右クリックして，コンテキストメニューから［HTML エディタ］を選択します。別ウィンドウで HTML エディタが開き，編集するテンプレートが表示されます。この HTML エディタは IDE から独立して実行されるので，編集を終えたらテンプレートを保存してエディタを閉じてください。

製品が配布された後で，最終的な HTML ページの外観を変更したい場合があります。ソフトウェア開発チームは最終的なページレイアウトには責任がないかもしれません。その責任は，組織内のたとえば専任の Web ページデザイナに属するかもしれません。そのページデザイナは，C++ Builder での開発については経験がないかもしれませんが，その経験は必要ありません。ページデザイナは，製品の開発・保守サイクルのどの段階でも，ソースコードになんら変更を加えずに，ページテンプレートを編集することができます。したがって，WebSnap の HTML テンプレートを使用すると，サーバーの開発と保守の効率が向上します。

HTML ファイルでサーバー側スクリプトを操作する

ページテンプレート内の HTML は，開発サイクルのどの段階でも変更することができます。しかし，サーバー側スクリプティングはまた別の問題でもあります。テンプレート内のサーバー側スクリプトを C++ Builder の外で変更することはいつでも可能ですが，アダプタページプロデューサが生成したページについては，そのような変更はお勧めできません。アダプタページプロデューサは，実行時にページテンプレート内のサーバー側スクリプティングを変更できるという点で，通常のページプロデューサとは異なります。他のスクリプトが動的に追加された場合に，既存のスクリプトがどのように動作するかを予測するのは困難なことがあります。スクリプトを直接操作したい場合は，Web ページモジュールにアダプタページプロデューサではなくページプロデューサを含めるようにしてください。

アダプタページプロデューサを使用している Web ページモジュールは，通常のページプロデューサを使用するように変換することができ，その手順は次のとおりです。

1. 変換するモジュール（ここでは ModuleName とします）で，［HTML スクリプト］タブのすべての情報を［ModuleName.html］タブにコピーし，それまでの内容をそっくり置き換えます。
2. コンポーネントパレットの［Internet］タブにあるページプロデューサを Web ページモジュールにドロップします。
3. そのページプロデューサの ScriptEngine プロパティを，置き換えるアダプタページプロデューサの ScriptEngine プロパティと一致するように設定します。
4. Web ページモジュール内のページプロデューサを，アダプタページプロデューサから新しいページプロデューサに変更します。［プレビュー］タブをクリックして，ページのコンテンツが変わっていないことを確認します。
5. これで，アダプタページプロデューサは無視されるようになり，Web ページモジュールから削除してもかまいません。

ログインサポート

多くの Web サーバーアプリケーションはログインのサポートを必要とします。たとえば，サーバーアプリケーションが Web サイトの一部へのアクセス権を与える前に，ユーザーにログインを要求する場合があります。また，ユーザーに応じてページの外観を変える場合は，Web サーバーが適切なページを送信できるようにするためにログインが必要になります。さらに，サーバーのメモリとプロセッササイクルに物理的制限があるので，サーバーアプリケーションが一定時点のユーザー数を制限できるようにする必要がある場合があります。

WebSnap を使用すると，Web サーバーアプリケーションへのログインサポートの組み込みが，かなり単純で簡単になります。ここでは，開発プロセスの最初の段階で設計に組み込むか，既存のアプリケーションに後から付加することで，ログインサポートを追加する方法を説明します。

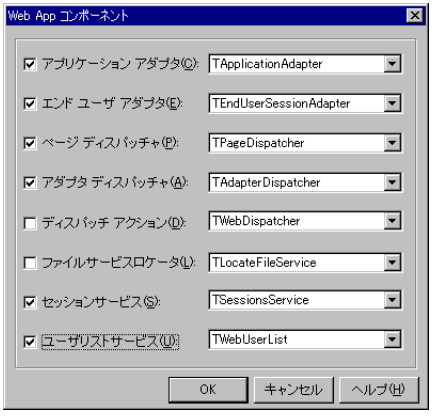
ログインサポートの追加

ログインサポートを実装するには，Web アプリケーションモジュールが以下のコンポーネントを持つ必要があります。

- ユーザーリストサービス（TWebUserList 型のオブジェクト）。これにはサーバーユーザーのユーザー名，パスワード，および許可が含まれる
- セッションサービス（TSessionsService）。サーバーに現在ログインしているユーザーについての情報を格納する
- エンドユーザーアダプタ（TEndUserSessionAdapter）。ログインに関連するアクションを処理する

Web サーバーアプリケーションを最初に作成するときに，［WebSnap アプリケーションの新規作成］ダイアログボックスを使用してこれらのコンポーネントを追加できます。このダイアログの［コンポーネント］ボタンをクリックして，［Web App コンポーネント］ダイアログボックスを表示します。［エンドユーザーアダプタ］，［セッションサービス］，および［ユーザーリストサービス］ボックスをチェックします。［エンドユーザーアダプタ］ボックスの横のドロップダウンメニューで TEndUserSessionAdapter を選択してエンドユーザーのアダプタタイプを選択します（デフォルトでは TEndUserAdapter が選択されますが，これは現在のユーザーを追跡できないのでログインサポートには不適切です）。以上の作業を終えると，ダイアログは次の図のようになります。［OK］を 2 回クリックして，ダイアログボックスを閉じます。これでログインサポートに必要なコンポーネントが Web アプリケーションモジュールに用意できました。

図 34.10［Web App コンポーネント］ダイアログでログインサポートのオプションを選択した状態



既存の Web アプリケーションモジュールにログインサポートを追加する場合は，コンポーネントパレットの［WebSnap］タブからこれらのコンポーネントをモジュールに直接ドロップします。Web アプリケーションモジュールは自動的に設定されます。

セッションサービスとエンドユーザーアダプタは設計段階では注意する必要がないかもしれませんが，Web ユーザーリストには注意する必要があるでしょう。デフォルトユーザーを追加し，WebUserList コンポーネントエディタによってユーザーの読み取り / 修正許可を設定できます。コンポーネントをダブルクリックしてエディタを表示すると，ユーザー名，パスワード，およびアクセス

権を設定できます。アクセス権の設定方法の詳細については，34-30 ページの「ユーザーアクセス権」を参照してください。

セッションサービスの使い方

セッションサービスは，TSessionsService 型のオブジェクトであり，Web サーバーアプリケーションにログインしているユーザーを追跡します。セッションサービスは，ユーザーごとに異なるセッションの割り当て，および名前と値のペア（ユーザー名など）とユーザーとの関連付けを行います。

セッションサービスに含まれる情報は，アプリケーションのメモリに格納されます。したがって，セッションサービスが機能するためには，Web サーバーアプリケーションがリクエスト間で実行し続ける必要があります。サーバーアプリケーションの種類によっては（CGI などは），リクエスト間で終了するものがあります。

メモ アプリケーションがログインをサポートするようにしたい場合は，リクエスト間で終了しないサーバーの種類を使用してください。Web アプリケーションデバッガ用実行形式ファイルを作成するプロジェクトの場合は，アプリケーションがページリクエストを受信する前にバックグラウンドで実行しているようにしなければなりません。そうしないと，ページリクエストを終えるたびにアプリケーションが終了し，ユーザーにログインページが渡されなくなります。

セッションサービスには 2 つの重要なプロパティがあり，これを使用すると，デフォルトのサーバー動作を変更できます。MaxSessions プロパティは，一定時点でシステムにログインできるユーザー数を指定します。MaxSessions のデフォルト値は -1 で，その場合はログイン可能なユーザー数はソフトウェアで制限されません。当然ながら，それでもサーバーのハードウェアでは新規ユーザーのためのメモリまたはプロセッササイクルが不足する可能性があり，システムの処理効率に悪影響を与えかねません。ユーザー数が多くなりすぎてサーバーが処理しきれなくなる心配がある場合は，必ず MaxSessions を適切な値に設定します。

DefaultTimeout プロパティは，デフォルトのタイムアウト時間を分単位で指定します。ユーザーの動作がない時間が DefaultTimeout に指定した時間（分単位）経過すると，そのセッションは自動的に終了されます。そのユーザーがログインしていた場合は，すべてのログイン情報が失われます。デフォルト値は 20 です。TimeoutMinutes プロパティを変更すれば，任意のセッションでこのデフォルト値をオーバーライドできます。

ログインページ

当然ながら，アプリケーションにはログインページも必要になります。ユーザーは，アクセスが制限されたページにアクセスしようとしたとき，またはアクセスする前に，認証のためにユーザー名とパスワードを入力します。ユーザーは，認証が完了したときに受信するページを指定することもできます。ユーザー名とパスワードが Web ユーザーリスト内のユーザーの 1 人と一致すると，そのユーザーは該当するアクセス権を取得し，ログインページで指定したページに転送されます。ユーザーが認証されない場合は，ログインページが再表示されるか（デフォルトのアクション），その他のなんらかのアクションが発生します。

幸いなことに，WebSnap では，Web ページモジュールとアダプタページプロデューサを使用して単純なログインページを簡単に作成することができます。ログインページを作成するには，まず新しい

Web ページモジュールを作成します。［ファイル｜新規作成｜その他］を選択して［新規作成］ダイアログボックスを表示し，［WebSnap］タブから［WebSnap ページモジュール］を選択します。ページプロデューサの種類として AdapterPageProducer を選択します。その他のオプションは好きなように指定します。ログインページに適した名前の 1 つとして Login が使用される傾向があります。

次に，ログインページのごく基本的なフィールドを追加する必要があります。つまり，ユーザー名のフィールド，パスワードのフィールド，ログイン後にユーザーが受信するページを選択する選択ボックス，およびログインページを送信してユーザーを認証する［ログイン］ボタンです。これらのフィールドを追加する手順は次のとおりです。

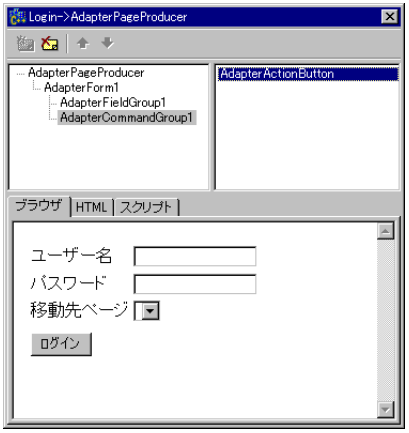
1. 作成したばかりの Web ページモジュールに LoginFormAdapter コンポーネントを追加します（このコンポーネントはコンポーネントパレットの［WebSnap］タブにあります）。
2. AdapterPageProducer コンポーネントをダブルクリックして，Web ページエディタウィンドウを表示します。
3. 左上のペインで AdapterPageProducer を右クリックし，［新規コンポーネント］を選択します。［Web コンポーネントの追加］ダイアログボックスで AdapterForm を選択し，［OK］をクリックします。
4. AdapterFieldGroup を AdapterForm に追加します（左上のペインで AdapterForm を右クリックし，［新規コンポーネント］を選択します。［Web コンポーネントの追加］ダイアログボックスで AdapterFieldGroup を選択し，［OK］をクリックします）。
5. 次に，オブジェクトインスペクタを表示し，AdapterFieldGroup の Adapter プロパティを自分の LoginFormAdapter に設定します。UserName，Password，および NextPage フィールドが，Web ページエディタの［ブラウザ］タブに自動的に表示されるはずです。

以上のように，WebSnap ではほとんどの作業を単純ないくつかのステップで行うことができます。このログインページには，認証のためにフォーム上の情報を送信する［ログイン］ボタンが欠けています。［ログイン］ボタンを追加する手順は次のとおりです。

1. AdapterCommandGroup を AdapterForm に追加します。
2. オブジェクトインスペクタで，DisplayComponent を AdapterFieldGroup1 に設定します。
3. AdapterActionButton を AdapterCommandGroup に追加します。
4. （Web ページエディタの右上のペインにリストされた）AdapterActionButton をクリックし，オブジェクトインスペクタを使用して ActionName プロパティを Login に変更します。ログインページのプレビューが Web ページエディタの［ブラウザ］タブに表示されます。

Web ページエディタは次の図のようになっているはずです。

図 34.11 Web ページエディタから見たログインページの例



AdapterFieldGroup の下にボタンが表示されない場合は，Web ページエディタ上の AdapterFieldGroup の下に AdapterCommandGroup がリストされていることを確認します。上に表示されている場合は，AdapterCommandGroup を選択し，Web ページエディタ上で下矢印をクリックします（一般に，Web ページの要素は，Web ページエディタでの表示順と同じ順序で縦に表示されます）。

このログインページを動作可能にするために必要なステップがあと 1 つあります。どのページがログインページかをエンドユーザーのセッションアダプタ内で指定する必要があります。それには，Web アプリケーションモジュール内で EndUserSessionAdapter コンポーネントを選択します。オブジェクトインスペクタで，LoginPage プロパティをログインページの名前に変更します。これでこのログインページが Web サーバーアプリケーションのすべてのページで使用可能になりました。

ログインを必要とするページの設定

実際に使用できるログインページが用意できたら，アクセスを制限する必要があるページでログインを要求するようにしなければなりません。ページでログインを要求するように設定するもっとも簡単な方法は，その要求をページに組み込んで設計することです。最初に Web ページモジュールを作成するときに，［WebSnap ページモジュールの新規作成］ダイアログボックスの［ページ］セクションの［ログインを必要とする］ボックスをチェックします。

ログインを要求しないページを作成した場合でも，後から変更できます。Web ページモジュールを作成した後でログインを要求するようにする手順は次のとおりです。

1. エディタ内で Web ページモジュールに関連付けられたソースコードファイルを開きます。
2. 静的 WebInit 関数の宣言までスクロールします。
3. パラメータリストの wpLoginRequired 部分のコメントを解除するために，これを囲む記号の /* と */ を削除します。これで WebInit 関数は次のようになるはずです。

```
static TWebPageInit WebInit(__classid(TAdapterPageProducerPage3), crOnDemand, caCache,
PageAccess << wpPublished << wpLoginRequired, ".html", "", "", "", "");
```

ページからログインの要求を削除するには，以上の手順の逆を行い，宣言の「wpLoginRequired」の部分を再度コメントにします。

メモ ページを公開するかどうかを設定するプロセスも同じです。「wpPublished」の部分を囲むコメント記号を必要に応じて追加または削除します。

ユーザーアクセス権

ユーザーアクセス権は，Web サーバーアプリケーションの重要な部分の 1 つです。サーバーが提供する情報を誰が表示および変更できるかを制限する必要があります。たとえば，オンライン小売販売を処理するサーバーアプリケーションを作成するとします。ユーザーがカタログの商品の表示はできても，価格の変更はできないようにする必要があります。これには，明らかにアクセス権が重要になります。

幸い，WebSnap ではページやサーバーコンテンツへのアクセスを制限する方法がいくつか用意されています。前のセクションでは，ログインを要求することでページのアクセスを制限する方法を説明しました。そのほかにも方法があります。次に例を示します。

- 適切な変更アクセス権を持つユーザーにはデータフィールドを編集ボックスに表示し，その他のユーザーはフィールドの内容を見ることはできても編集できないようにすることが可能です。
- 正しい表示アクセス権を持たないユーザーには特定のフィールドを非表示にすることができます。
- 権限のないユーザーが特定のページを受信できないようにすることができます。

このセクションではこれらの動作を実装する方法を説明します。

編集ボックスまたはテキストボックスとしてフィールドを動的に表示する

アダプタページプロデューサを使用する場合，ユーザーのアクセス権に応じてページ要素の外観を変更できます。たとえば，（Examples ディレクトリの WebSnap サブディレクトリ内にある）Biolife デモには，特定の種のすべての情報を表示するフォームページが含まれています。このフォームは，ユーザーがグリッド上の［詳細］ボタンをクリックしたときに表示されます。ユーザーが Will としてログインした場合は，データがプレーンテキストで表示されます。Will にはデータの変更が許可されていないので，このフォームでは変更のメカニズムが提供されません。ユーザー Ellen には変更許可が与えられているので，Ellen がこのフォームページを表示すると，フィールドの内容を変更できる一連の編集ボックスが表示されます。アクセス権をこのように使用すると，余分なページを作成する手間が省けます。

TAdapterDisplayField や TAdapterDisplayColumn などの一部のページ要素の外観は，ViewMode プロパティによって決まります。ViewMode が vmToggleOnAccess に設定された場合，変更アクセス権を持つユーザーにはこのページ要素が編集ボックスとして表示されます。変更アクセス権を持たないユーザーにはプレーンテキストが表示されます。ViewMode プロパティを vmToggleOnAccess に設定すると，ページ要素の外観と機能を動的に決定できます。

Web ユーザーリストは，TWebUserListItem オブジェクトのリストであり，システムにログインできるユーザーごとに 1 つのオブジェクトがあります。ユーザーに与えられた許可は，そのユーザーの Web ユーザーリストフィールドの AccessRights プロパティに格納されます。AccessRights はテキスト文字列なので，好きなように許可を指定できます。サーバーアプリケーションで使用したいあらゆる

種類のアクセス権の名前を作成します。1 人のユーザーが複数のアクセス権を持つようにしたい場合は，リスト内のアクセス権をスペース，セミコロン，またはカンマで区切ります。

フィールドのアクセス権は，ViewAccess プロパティと ModifyAccess プロパティによって制御されます。ViewAccess は，特定のフィールドを表示するために必要なアクセス権の名前を格納します。ModifyAccess は，フィールドデータを変更するために必要なアクセス権を指示します。これらのプロパティは，各フィールド内，およびそのフィールドを含むアダプタオブジェクト内の 2 か所に表示されます。

アクセス権のチェックは 2 ステップで行われます。ページ内の 1 つのフィールドの外観を決定するときに，アプリケーションはまずそのフィールド自身のアクセス権をチェックします。その値が空の文字列の場合は，そのフィールドを含むアダプタのアクセス権をチェックします。アダプタプロパティも空の場合，アプリケーションはデフォルトの動作を行います。変更アクセスの場合，デフォルトの動作は，Web ユーザーリストに含まれ，AccessRights プロパティが空でないあらゆるユーザーに変更を許可することです。表示アクセスの場合，表示アクセス権が指定されなければ自動的に許可が与えられます。

フィールドとその内容の非表示

適切な表示許可を与えられていないユーザーにはフィールドの内容を表示しないようにできます。まず，ユーザーが持っているべき許可と一致するようにフィールドの ViewAccess プロパティを設定します。次に，フィールドのページ要素の ViewAccess が vmToggleOnAccess に設定されていることを確認します。フィールドのキャプションは表示されますが，フィールドの値は表示されません。

当然ながら，多くの場合は，表示許可を与えられていないユーザーにはそのフィールドへの参照をすべて非表示にする必要があるでしょう。それには，フィールドのページ要素の HideOptions が hoHideOnNoDisplayAccess を含むように設定します。フィールドのキャプションも内容も表示されなくなります。

ページアクセスの防止

権限のないユーザーが特定のページにアクセスできないようにすることがあります。ページを表示する前にアクセス権をチェックするには，モジュールの WebInit 関数の宣言を変更します。この関数は，モジュールのソースコードに表示されます。

WebInit 関数は，最大 9 つの引数をとります。通常，WebSnap はそのうち 4 つをデフォルト値（空の文字列）のままにするので，この呼び出しは普通は次のようになります。

```
static TWebPageInit WebInit(__classid(TAdapterPageProducerPage3), crOnDemand, caCache,
PageAccess << wpPublished /* << wpLoginRequired */, ".html", "", "", "", "");
```

アクセス権を与える前に許可をチェックするには，必要な許可の文字列を 9 番目のパラメータで指定する必要があります。たとえば，必要な許可が Access であるとします。この場合，WebInit 関数を次のように変更できます。

```
static TWebPageInit WebInit(__classid(TAdapterPageProducerPage3), crOnDemand, caCache,
PageAccess << wpPublished /* << wpLoginRequired */, ".html", "", "", "", "Access");
```

Access 許可を持たないユーザーは，このページへのアクセスが拒否されるようになりました。

WebSnap でのサーバー側スクリプト

ページプロデューサテンプレートには，JScript または VBScript などのスクリプト言語を含めることができます。ページプロデューサは，プロデューサのコンテンツのリクエストに応答してスクリプトを実行します。このスクリプトは，（ブラウザが評価するクライアント側スクリプトとは対照的に）Web サーバーアプリケーションが評価するので，サーバー側スクリプトと呼ばれます。

ここでは，サーバー側スクリプトの概念の概要と，WebSnap アプリケーションでサーバー側スクリプトがどのように使われるかを示します。付録の「WebSnap サーバー側スクリプトリファレンス」では，各スクリプトオブジェクトとそのプロパティとメソッドについて詳細に説明しています。このリファレンスは，C++Builder のヘルプファイルにある VCL リファレンスのような，サーバー側スクリプティングについての API リファレンスであると考えることができます。この付録には，スクリプトをどのように使用すれば HTML ページを生成できるかを示す詳細なスクリプトの例もあります。

サーバー側スクリプティングは WebSnap の重要な部分ですが，WebSnap アプリケーションでのスクリプティングの使用は必須ではありません。スクリプティングは，HTML の生成のためにのみ使用されます。これによって，アプリケーションデータを HTML ページに挿入することが可能になります。実際，アダプタおよび他のスクリプト対応のオブジェクトでエクスポーズされるプロパティのほとんどが読み出し専用です。サーバー側スクリプトは，アプリケーションデータの変更には使用されません。その変更は，Pascal または C++ で記述されるコンポーネントとイベントハンドラで処理されます。

アプリケーションデータを HTML ページに挿入する方法は他にもあります。WebBroker の透過タグ，または他のタグベースの方法を使うことができます。たとえば，C++ Builder の Examples \ WebSnap フォルダ内のいくつかのプロジェクトでは，スクリプティングの代わりに XML と XSL を使用しています。しかし，スクリプティングを使わないとすれば，HTML 生成ロジックのほとんどを C++ で記述せざるをえなくなり，開発時間が長くなってしまいます。

WebSnap で使用されるスクリプティングは，オブジェクト指向であり，条件分岐ロジックとループをサポートし，ページ生成タスクを大幅に簡略化することができます。たとえば，一部のユーザーだけが編集でき，他のユーザーは編集できないデータフィールドをページに含めることができます。スクリプティングを使えば，承認ユーザーには編集ボックスを表示し，他のユーザーにはテキストのみを表示する条件分岐ロジックをテンプレートページに配置できます。タグベースの方法では，このような条件分岐は HTML 生成のソースコードにプログラミングしなければなりません。

アクティブスクリプティング

WebSnap は，アクティブスクリプティングを使ってサーバー側スクリプトを実装します。アクティブスクリプティングとは，Microsoft が作成したテクノロジーであり，これを使用すると，COM インターフェースによってアプリケーションオブジェクトとともにスクリプト言語を使用できます。Microsoft は，VBScript と JScript の 2 つのアクティブスクリプティング言語をリリースしています。その他の言語のサポートは，サードパーティによって提供されています。

スクリプトエンジン

ページプロデューサの ScriptEngine プロパティは，テンプレート内のスクリプトを評価するアクティブスクリプティングエンジンを識別します。これはデフォルトでは JScript をサポートするように設定されていますが，その他のスクリプティング言語（VBScript など）もサポートできます。

メモ WebSnap のアダプタは，JScript を作成するように設計されています。その他のスクリプティング言語の場合は，スクリプト生成ロジックを独自に提供する必要があります。

スクリプトブロック

スクリプトブロックは，HTML テンプレートに表示され，<% と %> によって区切られます。スクリプトエンジンは，スクリプトブロック内のテキストを評価します。その結果はページプロデューサのコンテンツの一部になります。ページプロデューサは，埋め込まれた透過タグを変換した後で，スクリプトブロックの外にテキストを書き込みます。スクリプトブロックは，テキストを囲むこともできるので，条件ロジックおよびループによってテキストの出力を指示できます。たとえば，次の JScript ブロックは，番号の付いた 5 行のリストを生成します。

```
<ul>
<% for (i=0;i<5;i++) { %>
     <li>Item <%=i %></li>
<% } %>
</ul>
```

（<%= 区切り記号は，Response.Write の短縮形です）。

スクリプトの作成

開発者は，WebSnap の機能を利用してスクリプトを自動的に生成できます。

ウィザードテンプレート

WebSnap ウィザードは，新しい WebSnap アプリケーションまたはページモジュールを作成するときに，ページモジュールテンプレートの最初のコンテンツの選択に使用されるテンプレートフィールドを提供します。たとえば，標準テンプレートは，アプリケーションのタイトル，ページ名，および公開されたページへのリンクを表示する JScript を生成します。

TAdapterPageProducer

TAdapterPageProducer は，HTML および JScript を生成してフォームおよびテーブルを構築します。生成された JScript は，アダプタオブジェクトを呼び出して，フィールド値，フィールドイメージパラメータ，およびアクションパラメータを取得します。

スクリプトの編集と表示

［HTML ソース］タブを使用して，実行されたスクリプトの結果の HTML を表示します。［プレビュー］タブを使用して，ブラウザに結果を表示します。Web ページモジュールが TAdapterPageProducer を使用する場合は，［HTML スクリプト］タブを使用できます。［HTML スク

リプト］タブは，TAdapterPageProducer オブジェクトが生成する HTML と JScript を表示します。このビューを見て，アダプタフィールドの表示およびアダプタアクションの実行をする HTML フォームを構築するスクリプトの作成方法を調べてください。

ページにスクリプトを取り込む

テンプレートには，ファイルまたは別のページからスクリプトを取り込むことができます。ファイルからスクリプトを取り込むには，次のコード文を使用します。

```
<!-- #include file="filename.html" -->
```

テンプレートに別のページからスクリプトを取り込んだ場合，そのスクリプトは取り込んだ側のページによって評価されます。次のコード文を使用すると，未評価の page1 のコンテンツが取り込まれます。

```
<!-- #include page="page1" -- >
```

スクリプトオブジェクト

スクリプトオブジェクトは，スクリプトコマンドが参照できるオブジェクトです。オブジェクトをスクリプトに使用できるようにするには，このオブジェクトへの IDispatch インターフェースをアクティブスクリプティングエンジンに登録します。スクリプトには以下のオブジェクトが使用できます。

表 34.4 スクリプトオブジェクト

スクリプトオブジェクト	説明
Application	Web アプリケーションモジュールのアプリケーションアダプタへのアクセスを提供する
EndUser	Web アプリケーションモジュールのエンドユーザーアダプタへのアクセスを提供する
Session	Web アプリケーションモジュールのセッションオブジェクトへのアクセスを提供する
Pages	アプリケーションページへのアクセスを提供する
Modules	アプリケーションモジュールへのアクセスを提供する
Page	現在のページへのアクセスを提供する
Producer	Web ページモジュールのページプロデューサへのアクセスを提供する
Response	WebResponse へのアクセスを提供する。このオブジェクトは，タグの置き換えをしない場合に使用する
Request	WebRequest へのアクセスを提供する
Adapter オブジェクト	現在のページ上のアダプタの全コンポーネントを限定なしで参照できる。ほかのモジュール内のアダプタは，Modules オブジェクトを使用して限定しなければならない

現在のページ上のスクリプトオブジェクトは，すべて同じアダプタを使用し，限定なしで参照できます。ほかのページ上のスクリプトオブジェクトは，ほかのページモジュールの一部であり，アダプタオブジェクトが異なります。ほかのページ上のスクリプトオブジェクトにアクセスするには，スクリプトオブジェクトの参照をアダプタオブジェクトの名前で始めます。次に例を示します。

```
<%= FirstName %>
```

この場合は，現在のページのアダプタの FirstName プロパティのコンテンツが表示されます。次のスクリプト行の場合は，別のページモジュール内の Adapter1 の FirstName プロパティが表示されます。

```
<%= Adapter1.FirstName %>
```

スクリプトオブジェクトについての詳細は，付録の「WebSnap サーバー側スクリプトリファレンス」を参照してください。

リクエストとレスポンスのディスパッチ

WebSnap を Web サーバーアプリケーション開発に使用する理由の 1 つは，WebSnap コンポーネントが HTML のリクエストとレスポンスを自動処理することです。ありふれたページ転送操作のイベントハンドラを作成することなく，その分の労力をビジネスロジックやサーバー設計に振り向けることができます。それでも，WebSnap アプリケーションがどのように HTML のリクエストとレスポンスを処理するかを理解しておくと役に立ちます。ここでは，その処理プロセスの概要を説明します。

リクエストを処理する前に，Web アプリケーションモジュールは（TWebContext 型の）Web コンテキストオブジェクトを初期化します。Web コンテキストオブジェクトは，グローバル関数 WebContext の呼び出しでアクセスされ，リクエストを処理するコンポーネントが使用する変数へのグローバルアクセスを提供します。たとえば，Web コンテキストには，HTTP リクエストメッセージを表す TWebRequest オブジェクトと，それに対して返されるレスポンスを表す TWebResponse オブジェクトが含まれます。

ディスパッチャコンポーネント

Web アプリケーションモジュール内のディスパッチャコンポーネントは，アプリケーションの処理の流れを制御します。ディスパッチャは，HTTP リクエストを調べることで，特定の種類の HTTP リクエストメッセージの処理方法を判別します。

アダプタディスパッチャコンポーネント（TAdapterDispatcher）は，アダプタアクションコンポーネントまたはアダプタイメージフィールドコンポーネントを識別するコンテンツフィールドまたは問い合わせフィールドを探します。アダプタディスパッチャは，コンポーネントを探し出すとそのコンポーネントに制御を渡します。

Web ディスパッチャコンポーネント（TWebDispatcher）は，HTTP リクエストメッセージの種類ごとに処理方法を規定しているアクションフィールド（TWebActionItem 型）の集合を保持しています。Web ディスパッチャは，リクエストに一致するアクションフィールドを探します。探し出せた場合は，そのアクションフィールドに制御を渡します。Web ディスパッチャは，リクエストを処理できる自動ディスパッチコンポーネントも探します。

ページディスパッチャコンポーネント（TPageDispatcher）は，TWebRequest オブジェクトの PathInfo プロパティを調べ，登録された Web ページモジュールの名前を探します。このディスパッチャは，Web ページモジュール名を探し出すと，そのモジュールに制御を渡します。

アダプタディスパッチャ操作

アダプタディスパッチャコンポーネント（TAdapterDispatcher）は，アダプタのアクションおよびフィールドコンポーネントを呼び出すことで，HTML フォームの送信，および動的イメージのリクエストを自動的に処理します。

アダプタコンポーネントを使用してのコンテンツの生成

WebSnap アプリケーションが自動的にアダプタアクションを実行し，アダプタフィールドから動的イメージを取得するには，HTML コンテンツを適切に作成しなければなりません。HTML コンテンツが適切に作成されないと，結果として生じる HTTP リクエストには，アダプタディスパッチャがアダプタのアクションおよびフィールドコンポーネントを呼び出すために必要とする情報が含まれません。

HTML ページの作成時のエラーを減らすために，アダプタコンポーネントは HTML 要素の名前と値を示します。アダプタコンポーネントは，アダプタフィールドの更新のために設計される HTML フォームに表示しなければならない非表示フィールドの名前と値を取得するメソッドを持っています。通常は，ページプロデューサがサーバー側スクリプティングを使用してアダプタコンポーネントから名前と値を取得し，この情報を使用して HTML を生成します。たとえば，次のスクリプトは，Adapter1 から Graphic という名前のフィールドを参照する <IMG> 要素を作成します。

```
<img src="<%=Adapter1.Graphic.Image.AsHREF%>" alt="<%=Adapter1.Graphic.DisplayText%>">
```

Web アプリケーションがスクリプトを評価するときに，HTML src 属性には，フィールドコンポーネントがイメージを取得するために必要なフィールドとパラメータを識別するために必要な情報が含まれます。この結果，次のような HTML になります。

```
<img src="?_lSpecies No=90090&__id=DM.Adapter1.Graphic" alt="(GRAPHIC)">
```

このイメージを取得する HTTP リクエストをブラウザが Web アプリケーションに送信するときに，アダプタディスパッチャは，モジュール DM において，Adapter1 の Graphic フィールドがパラメータ "Species No=90090" を使用して呼び出される必要があることを判別できます。アダプタディスパッチャは，適切な HTTP レスポンスを記述するために Graphic フィールドを呼び出します。

次のスクリプトは，Adapter1 の EditRow アクションを参照する <A> 要素を作成し，Details という名前のページへのハイパーリンクを作成します。

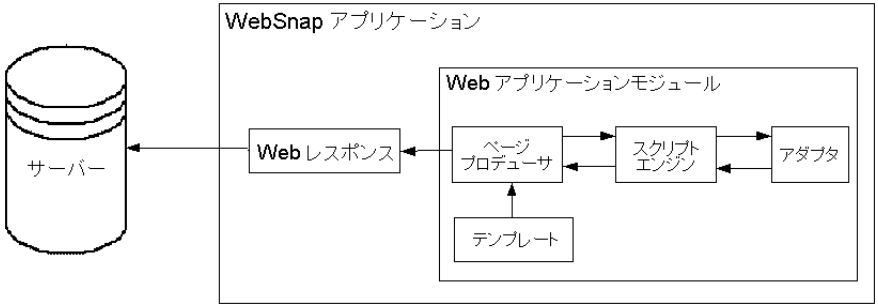
```
<a href="<%=Adapter1.EditRow.LinkToPage("Details", Page.Name).AsHREF%>">Edit...</a>
```

この結果，次のような HTML になります。

```
<a href="?&_lSpecies No=90310&__sp=Edit&__fp=Grid&__id=DM.Adapter1.EditRow">Edit...</a>
```

エンドユーザーがこのハイパーリンクをクリックし，ブラウザが HTTP リクエストを送信します。アダプタディスパッチャは，モジュール DM において，Adapter1 の EditRow アクションがパラメータ Species No=903010 を使用して呼び出される必要があることを判別できます。さらにアダプタディスパッチャは，アクションが正常に実行された場合は［Edit］ページ，アクションの実行が失敗した場合は［Grid］ページを表示します。次に，編集する行を探し出すために EditRow アクションを呼び出し，HTTP レスポンスを生成するために Edit という名前のページが呼び出されます。図 34.12 は，どのようにアダプタコンポーネントを使用してコンテンツを生成するかを示しています。

図 34.12 コンテンツの生成の流れ



アダプタリクエストの受信とレスポンスの生成

アダプタディスパッチャは，クライアントリクエストを受信すると，HTTP リクエストについての情報を保持するアダプタリクエストとアダプタレスポンスのオブジェクトを作成します。このアダプタリクエストとアダプタレスポンスのオブジェクトは，リクエストの処理中にアクセスできるように Web コンテキストに格納されます。

アダプタディスパッチャは，アクションとイメージの 2 種類のアダプタリクエストオブジェクトを作成します。アクションのリクエストオブジェクトは，アダプタアクションを実行するときに作成します。イメージのリクエストオブジェクトは，アダプタフィールドからイメージを取得するときに作成します。

アダプタレスポンスオブジェクトは，アダプタアクションまたはアダプタイメージのリクエストに対するレスポンスを示すためにアダプタコンポーネントによって使用されます。アダプタレスポンスオブジェクトには，アクションとイメージの 2 種類があります。

アクションリクエスト

アクションリクエストオブジェクトは，HTTP リクエストを分解して，アダプタアクションの実行に必要な情報にします。アダプタアクションの実行に必要な情報には，以下のリクエスト情報が含まれます。

表 34.5 アクションリクエスト内のリクエスト情報

リクエスト情報	説明
コンポーネント名	アダプタアクションコンポーネントを識別する
アダプタモード	モードを定義する。たとえば，TDataSetAdapter は，Edit，Insert，および Browse モードをサポートする。アダプタアクションは，モードに応じて異なる方法で実行できる。
成功ページ	アクションが正常に実行された後に表示されるページを識別する
失敗ページ	アクションの実行中にエラーが発生した場合に表示されるページを識別する
アクションリクエストパラメータ	アダプタアクションに必要なパラメータを識別する。たとえば，TDataSetAdapter の Apply アクションには，更新するレコードを識別するキー値が含まれる

表 34.5 アクションリクエスト内のリクエスト情報（つづき）

リクエスト情報	説明
アダプタフィールド値	HTML フォームが送信されたときに HTTP リクエストに渡されるアダプタフィールドの値を指定する。フィールドの値に含めることができるのは，エンドユーザーが入力した新しい値，アダプタフィールドの元の値，およびアップロードされたファイルである
レコードキー	各レコードを一意に識別するキーを指定する

アクションレスポンスの生成

アクションレスポンスオブジェクトは，アダプタアクションコンポーネントのかわりに HTTP レスポンスを生成します。アダプタアクションは，オブジェクト内でプロパティを設定することで，またはアクションレスポンスオブジェクト内でメソッドを呼び出すことで，レスポンスの種類を示します。このプロパティには以下のものがあります。

- RedirectOptions：HTML コンテンツを返さずに HTTP リダイレクトを実行するかどうかを示す
- ExecutionStatus：状態を成功に設定すると，デフォルトのアクションレスポンスが，アクションリクエストで識別された成功ページのコンテンツになる

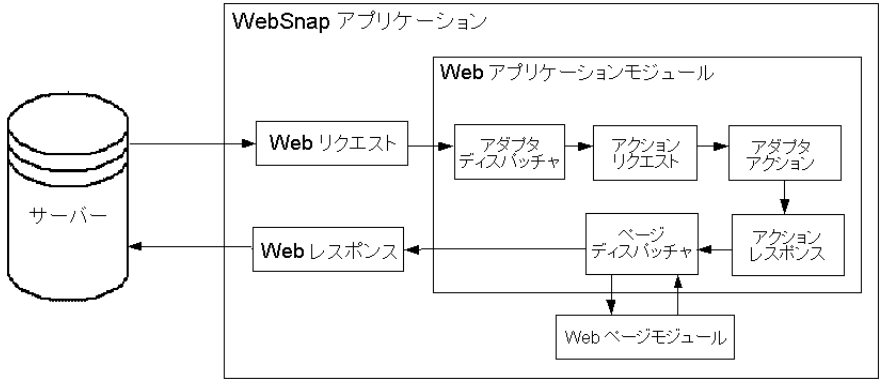
アクションレスポンスメソッドには以下のものがあります。

- RespondWithPage：特定の Web ページモジュールがレスポンスを生成する必要があるときに，アダプタアクションがこのメソッドを呼び出す
- RespondWithComponent：このコンポーネントを含む Web ページモジュールからのレスポンスが必要なときに，アダプタアクションがこのメソッドを呼び出す
- RespondWithURL：指定された URL へのリダイレクトがレスポンスであるときに，アダプタアクションがこのメソッドを呼び出す

ページで応答する場合，アクションレスポンスオブジェクトはページディスパッチャを使用してページコンテンツを生成しようと試みます。ページディスパッチャが見つからない場合は，Web ページモジュールが直接呼び出されます。

図 34.15 は，アクションリクエストおよびアクションレスポンスのオブジェクトがどのようにリクエストを処理するかを示しています。

図 34.13 アクションリクエストとアクションレスポンス



イメージリクエスト

イメージリクエストオブジェクトは，HTTP リクエストを分解し，アダプタイメージフィールドがイメージを生成するために必要な情報にします。イメージリクエストが表す情報の種類は以下のとおりです。

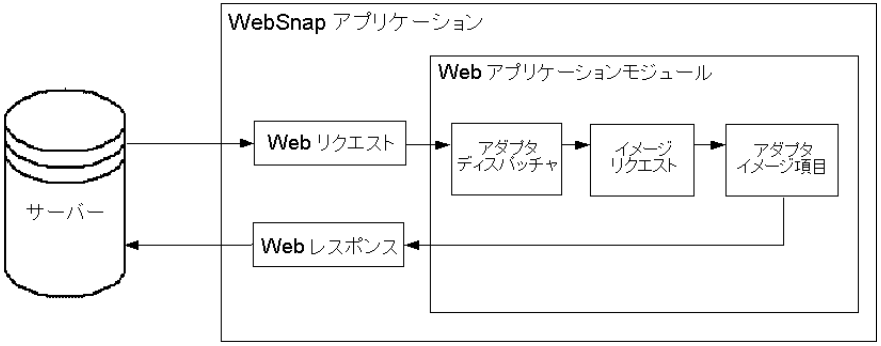
- コンポーネント名：アダプタフィールドコンポーネントを識別する
- イメージリクエストパラメータ：アダプタイメージに必要なパラメータを識別する。たとえば，TDataSetAdapterImageField オブジェクトは，イメージを含むレコードを識別するキー値を必要とする

イメージレスポンス

イメージレスポンスオブジェクトには，TWebResponse オブジェクトが含まれます。アダプタフィールドは，Web レスポンスオブジェクトにイメージを書き込むことで，アダプタリクエストに応答します。

図 34.14 は，アダプタイメージフィールドがどのようにリクエストに応答するかを示しています。

図 34.14 リクエストに対するイメージレスポンス



アクションフィールドのディスパッチ

リクエストに応答するときに，Web ディスパッチャ（TWebDispatcher）は，アクションフィールドのリスト内を検索して，以下のようなアクションフィールドを探します。

- ターゲット URL のリクエストメッセージの PathInfo 部分に一致するもの
- リクエストメッセージのメソッドとして指定されたサービスを提供できるもの

これを行うために，TWebRequest オブジェクトの PathInfo プロパティと MethodType プロパティが，アクションフィールドの同名のプロパティと比較されます。

ディスパッチャは，適切なアクションフィールドを見つけると，そのアクションフィールドを起動します。起動されたアクションフィールドは以下のいずれかを行います。

- レスポンスの内容を埋め，そのレスポンスを送信するか，リクエストの処理が完了したことを通知する
- レスポンスに追加を行い，他のアクションフィールドによって処理が完了できるようにする

- リクエストを他のアクションフィールドに任せる

ディスパッチャがアクションフィールドをすべてチェックした後もメッセージが正しく処理されていなければ，ディスパッチャは，アクションフィールドを使用しない自動ディスパッチコンポーネントが特別に登録されていないかチェックします。この自動ディスパッチコンポーネントは多層データベースアプリケーションに固有のコンポーネントです。リクエストメッセージの処理がなお完了していない場合，ディスパッチャはデフォルトのアクションフィールドを呼び出します。デフォルトアクションフィールドはターゲット URL にもリクエストのメソッドにも対応している必要はありません。

ページディスパッチャ操作

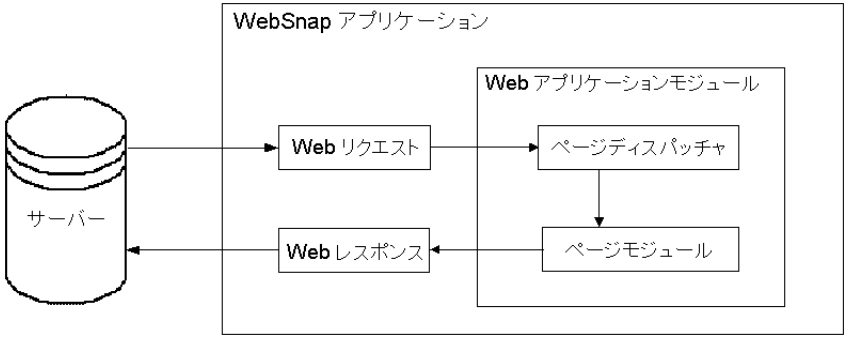
ページディスパッチャは，クライアントリクエストを受け取ると，ターゲット URL のリクエストメッセージの PathInfo 部分をチェックすることで，ページ名を判別します。PathInfo 部分が空白でない場合は，ページディスパッチャは PathInfo の最後の単語をページ名として使用します。PathInfo 部分が空白である場合は，ページディスパッチャはデフォルトのページ名を判別しようとします。

ページディスパッチャの DefaultPage プロパティにページ名が含まれている場合，ページディスパッチャはこの名前をデフォルトのページ名として使用します。DefaultPage プロパティが空白であり，Web アプリケーションモジュールがページモジュールである場合は，ページディスパッチャは Web アプリケーションモジュールの名前をデフォルトのページ名として使用します。

ページ名が空白でない場合は，一致する名前を持つ Web ページモジュールをページディスパッチャが検索します。ページディスパッチャは Web ページモジュールを探し出すと，そのモジュールを呼び出してレスポンスを生成します。ページ名が空白である場合，またはページディスパッチャが Web ページモジュールを探し出せない場合は，ページディスパッチャが例外を生成します。

図 34.15 は，ページディスパッチャがどのようにリクエストに応答するかを示しています。

図 34.15 ページのディスパッチ



第35章

XML ドキュメントの操作

XML（Extensible Markup Language）は，構造化データを記述するマークアップ言語です。HTML に似ていますが，表示特性ではなく情報の構造をタグで記述する点が異なります。XML ドキュメントは，単純なテキストベースの方法で情報を格納するので，情報の検索または編集が簡単にできます。XML ドキュメントは，多くの場合，Web アプリケーション，企業間通信などでの標準的で転送可能なデータ形式として使用されます。

XML ドキュメントは，データの本体を階層で表示します。XML ドキュメントのタグは，次のドキュメントに示すように，各データ要素の役割または意味を記述します。次のドキュメントは持ち株の集合を記述しています。

```
<?xml version="1.0" encoding=UTF-8 standalone="no" ?>
<!DOCTYPE StockHoldings SYSTEM "sth.dtd">
<StockHoldings>
   <Stock exchange="NASDAQ">
      <name>Borland</name>
      <price>15.375</price>
      <symbol>BORL</symbol>
      <shares>100</shares>
   </Stock>
   <Stock exchange="NYSE">
      <name>Pfizer</name>
      <price>42.75</price>
      <symbol>PFE</symbol>
      <shares type="preferred">25</shares>
   </Stock>
</StockHoldings>
```

この例は，XML ドキュメントによく使用されるいくつかの要素を示しています。1 行目は，XML 宣言と呼ばれる処理の指示です。XML 宣言は省略可能ですが，ドキュメントについての有益な情報を供給するのでこの宣言を含める必要があります。この例の XML 宣言が供給している情報は，このドキュメントが XML 仕様のバージョン 1.0 に準拠すること，UTF-8 の文字コード化を使用すること，およびドキュメントの種類の宣言（DTD）を外部ファイルに依存していることです。

<!DOCTYPE> タグで始まっている 2 行目は，ドキュメントの種類の宣言です。ドキュメントの種類の宣言（DTD）は，XML がドキュメントの構造をどのように定義するかを示します。DTD は，ドキュ

メントに含まれる要素（タグ）に構文規則を課します。この例の DTD は，別のファイル（sth.dtd）を参照します。この場合，XML ドキュメントそのものの中ではなく外部ファイルで構造が定義されています。XML ドキュメントの構造を記述するファイルの種類はこのほかにも，XDR（Reduced XML Data：縮小 XML データ），XSD（XML スキーマ）などがあります。

3 行目以降は，1 つのルートノード（<StockHoldings> タグ）を持つ階層に編成されています。この階層の各ノードには，子ノードのセットまたはテキスト値が含まれています。属性が含まれているタグもあります（<Stock> タグと <shares> タグ）。属性は Name=Value のペアであり，タグの解釈方法の詳細を示しています。

XML ドキュメント内のテキストを直接操作することも可能ですが，通常はアプリケーションが追加ツールを使用してデータを解析および編集します。W3C は，Document Object Model（DOM）と呼ばれる解析された XML ドキュメントを表す標準的なインターフェースのセットを定義しています。DOM インターフェースを実装して XML ドキュメントの解釈と編集が簡単にできるようにする XML パーサーは，数社のベンダーが提供しています。

C++Builder は，XML ドキュメントを操作するための追加ツールをいくつか用意しています。これらのツールは，別のベンダーが提供する DOM パーサーを使用し，XML ドキュメントの操作がさらに簡単にできるようにしています。この章では，これらのツールについて説明します。

メモ この章で説明するツールのほかにも，C++Builder には，C++Builder データベースアーキテクチャに統合されるデータパケットに XML ドキュメントを変換するためのツールとコンポーネントが用意されています。XML ドキュメントをデータベースアプリケーションに統合するツールについての詳細は，第 30 章「データベースアプリケーションでの XML の使用」を参照してください。

DOM の使い方

DOM（Document Object Model）は，解析された XML ドキュメントを表す標準的なインターフェースのセットです。デフォルトでは，Microsoft の DOM 実装のコピーが取得されます。さらに，その他のベンダーの追加の DOM 実装を XML フレームワークに統合できる登録メカニズムがあります。

XMLDOM ユニットには，W3C XML DOM レベル 2 仕様で定義されたすべての DOM インターフェースの宣言が含まれています。各 DOM ベンダーは，これらのインターフェースの実装を提供しています。

- Microsoft の実装を使用するには，ソースファイルに MSXMLDOM ユニットを含める。Microsoft の実装は COM ベースなので，msxml.dll ライブラリが登録されていない場合は，このライブラリ COM サーバーとして登録しなければならない。Regsvr32.exe を使用すると，この DLL を登録できる
- ほかの DOM 実装を使用するには，TDOMVendor クラスの下位クラスを定義するユニットを作成しなければならない。下位クラスでは，2 つのメソッドをオーバーライドしなければならない。つまり，ベンダーを識別する文字列を返す Description メソッドと，最上位のインターフェース（IDOMImplementation）を返す DOMImplementation メソッドをオーバーライドしなければならない。グローバル手続き RegisterDOMVendor を呼び出して，このベンダーを登録する必要がある。グローバル手続き UnRegisterDOMVendor を呼び出すと，このベンダーを登録解除できる。新しい DOMVendor の登録を終えると，このベンダーを登録解除するまで，ラップされた DOM 実装にアプリケーションがアクセスできるようになる

標準的な DOM インターフェースの拡張機能を提供するベンダーもあります。これらの拡張機能を使用できるようにするために，XMLDOM ユニットは IDOMNodeEx インターフェースも定義します。IDOMNodeEx は，これらの拡張機能のうちもっとも有用なものを含む標準的な IDOMNode の下位インターフェースです。

DOM インターフェースを直接使用して，XML ドキュメントを解析および編集することができます。単純に GetDOM 関数を呼び出して IDOMImplementation インターフェースを取得すれば，これを出発点として使用できます。

メモ DOM インターフェースについての詳細は，XMLDOM ユニットヘッダー内の宣言，DOM ベンダーが提供するマニュアル，または W3C の Web サイト（www.w3.org）で提供されている仕様を参照してください。

特殊な XML クラスを使用すると，DOM インターフェースを直接使用するより便利な場合があります。これらについて，以下で説明します。

XML コンポーネントの操作

VCL（または CLX）では，XML ドキュメントを操作するためのクラスとインターフェースをいくつか定義しています。これらは，XML ドキュメントのロード，編集，および保存のプロセスを単純にします。

TXMLDocument の使い方

XML ドキュメントの操作の出発点は，TXMLDocument コンポーネントです。TXMLDocument を使用して XML ドキュメントを直接操作する手順は次のとおりです。

1. フォームまたはデータモジュールに TXMLDocument コンポーネントを追加します。TXMLDocument は，コンポーネントパレットの［Internet］ページにあります。
2. DOMVendor プロパティを設定して，このコンポーネントが XML ドキュメントの解析と編集に使用する DOM 実装を指定します。オブジェクトインスペクタには，現在登録されているすべての DOM ベンダーがリストされます。DOM 実装についての詳細は，35-2 ページの「DOM の使い方」を参照してください。
3. 実装に応じて，ParseOptions プロパティを設定して，基底の DOM 実装が XML ドキュメントをどのように解析するかを設定できます。
4. 既存の XML ドキュメントを操作する場合は，以下の方法でこのドキュメントを指定します。
   - その XML ドキュメントがファイルに格納されている場合は，FileName プロパティをそのファイルの名前に設定します。
   - XML プロパティを使用することで，XML ドキュメントを文字列として指定できます。
5. Active プロパティを **true** に設定します。

アクティブな TXMLDocument オブジェクトが用意できたら，そのノードの階層を横断し，値の読み出しまたは設定ができます。この階層のルートノードは，DocumentElement プロパティとして使用できます。

XML ノードの操作

XML ドキュメントが DOM 実装によって解析されると，そのデータをノードの階層として使用できるようになります。各ノードは，ドキュメント内のタグ付き要素に対応します。たとえば，次の XML があるとします。

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE StockHoldings SYSTEM "sth.dtd">
<StockHoldings>
   <Stock exchange="NASDAQ">
      <name>Borland</name>
      <price>15.375</price>
      <symbol>BORL</symbol>
      <shares>100</shares>
   </Stock>
   <Stock exchange="NYSE">
      <name>Pfizer</name>
      <price>42.75</price>
      <symbol>PFE</symbol>
      <shares type="preferred">25</shares>
   </Stock>
</StockHoldings>
```

TXMLDocument が生成するノードの階層は次のようになります。階層のルートは StockHoldings ノードになります。StockHoldings には 2 つの子ノードがあり，これが 2 つの Stock タグに対応します。これらの 2 つの子ノードのそれぞれには，独自に 4 つの子ノード（name, price, symbol, および shares）があります。これらの 4 つの子ノードは，リーフノードとして機能します。これらに含まれるテキストは，各リーフノードの値として表示されます。

メモ このノード分割は，DOM 実装が XML ドキュメントのノードを生成する方法とは少し異なります。特に，DOM パーサーがすべてのタグ付き要素を内部ノードとして扱う点が異なります。（テキスト型のノードの）追加ノードが，name, price, symbol, および shares ノードの値に対して作成されます。そしてこれらのテキストノードは，name, price, symbol, および shares ノードの子として表示されます。

各ノードは，XML ドキュメントコンポーネントの DocumentElement プロパティの値であるルートノードを出発点として，IXMLNode インターフェースによってアクセスされます。

ノードの値の操作

IXMLNode インターフェースがある場合，IsTextElement プロパティをチェックすることで，このインターフェースが内部ノードまたはリーフノードのどちらを表すかをチェックできます。

- リーフノードを表す場合は，Text プロパティを使用して値の読み出しまたは設定ができます。
- 内部ノードを表す場合は，ChildNodes プロパティを使用して子ノードにアクセスできます。

したがって，たとえば上記の XML ドキュメントを使用すると，次のようにして Borland の株価を読み出すことができます。

```
_di_IXMLNode BorlandStock = XMLDocument1->DocumentElement->ChildNodes[0];
AnsiString Price = BorlandStock->ChildNodes->Nodes["price"]->Text;
```

ノードの属性の操作

ノードに属性が含まれる場合は，Attributes プロパティを使用してこの属性を操作できます。既存の属性名を指定すると，属性の値の読み出しまたは変更ができます。Attributes プロパティを設定するときに，次のように新しい属性名を指定すると，新しい属性を追加できます。

```
_di_IXMLNode BorlandStock = XMLDocument1->DocumentElement->ChildNodes[0];
BorlandStock->ChildNodes->Nodes["shares"]->Attributes["type"] = "common";
```

子ノードの追加と削除

AddChild メソッドを使用すると，子ノードを追加できます。AddChild は，XML ドキュメント内のタグ付き要素に対応する新しいノードを作成します。このようなノードは，要素ノードと呼ばれます。

新しい要素ノードを作成するには，新しいタグに表示される名前を指定し，必要であれば，新しいノードが表示される場所を指定します。たとえば，次のコードは，上記のドキュメントに新しい株式リストを追加します。

```
_di_IXMLNode NewStock = XMLDocument1->DocumentElement->AddChild("stock");
NewStock->Attributes["exchange"] = "NASDAQ";
_di_IXMLNode ValueNode = NewStock->AddChild("name");
ValueNode->Text = "Cisco Systems";
ValueNode = NewStock->AddChild("price");
ValueNode->Text = "62.375";
ValueNode = NewStock->AddChild("symbol");
ValueNode->Text = "CSCO";
ValueNode = NewStock->AddChild("shares");
ValueNode->Text = "25";
```

オーバーロードバージョンの AddChild を使用すると，タグ名が定義される名前空間 URI を指定できます。

ChildNodes プロパティのメソッドを使用すると，子ノードを削除できます。ChildNodes は IXMLNodeList インターフェースの 1 つであり，ノードの子を管理します。このプロパティの Delete メソッドを使用すると，位置または名前で識別される 1 つの子ノードを削除できます。たとえば，次のコードは，上記のドキュメントにリストされた最後の株式を削除します。

```
_di_IXMLNode StockList = XMLDocument1->DocumentElement;
StockList->ChildNodes->Delete(StockList->ChildNodes->Count - 1);
```

データバインドウィザードによる XML ドキュメントの抽象化

TXMLDocument コンポーネントや XML ドキュメント内のノードにアクセスする IXMLNode インターフェースを使って XML ドキュメントを操作することは可能です。あるいは，TXMLDocument を使わずに DOM インターフェースだけを処理することもできます。しかし，より単純で読みやすいコードを記述するには，XML データバインドウィザードを使用するのがよいでしょう。

データバインドウィザードは，XML スキーマまたはデータファイルを受け取り，その上にマップされるインターフェースのセットを生成します（インターフェースとは，純粋仮想メソッドのみが含まれるクラスです）。たとえば，次のような XML データがあるとします。

```
<customer id=1>
  <name>Mark</name>
  <phone>(831) 431-1000</phone>
</customer>
```

データバインドウィザードは，次のインターフェース（およびこれを実装するクラス）を生成します。

```
__interface INTERFACE_UUID("{F3729105-3DD0-1234-80e0-22A04FE7B451}") ICustomer :
    public IXMLNode
{
public:
  virtual int __fastcall Getid(void) = 0 ;
  virtual DOMString __fastcall Getname(void) = 0 ;
  virtual DOMString __fastcall Getphone(void) = 0 ;
  virtual void __fastcall Setid(int Value)= 0 ;
  virtual void __fastcall Setname(DOMString Value)= 0 ;
  virtual void __fastcall Setphone(DOMString Value)= 0 ;
  __property int id = {read=Getid, write=Setid};
  __property DOMString name = {read=Getname, write=Setname};
  __property DOMString phone = {read=Getphone, write=Setphone};
};
```

あらゆる子ノードは，ある条件を満たすプロパティにマップされます。その条件とは，プロパティの名前がその子ノードのタグ名に一致し，（子が内部ノードの場合は）プロパティの値がその子ノードのインターフェースであるか，（リーフノードの場合は）プロパティの値がその子ノードの値であることです。あらゆるノード属性もプロパティにマップされ，マップ先のプロパティは，名前が属性の名前と一致し，値が属性の値と一致するプロパティになります。

XML ドキュメント内のタグ付き要素のそれぞれに対してインターフェース（および下位の実装クラス）を作成するほかに，データバインドウィザードは，ルートノードへのインターフェースを取得するグローバル関数を作成します。たとえば，上記の XML の出典であるドキュメントのルートノードにタグ <Customers> がある場合，データバインドウィザードは次のグローバルルーチンを作成します。

```
extern PACKAGE _di_ICustomers __fastcall GetCustomers(TXMLDocument *XMLDoc);
extern PACKAGE _di_ICustomers __fastcall GetCustomers(_di_IXMLDocument XMLDoc);
extern PACKAGE _di_ICustomers __fastcall LoadCustomers(const WideString FileName);
extern PACKAGE _di_ICustomers __fastcall NewCustomers(void);
```

Get... 関数は，TXMLDocument インスタンスのインターフェースラッパー（または TXMLDocument インスタンスへのポインタ）を取得します。Load... 関数は，TXMLDocument インスタンスを動的に作成し，インターフェースポインタを返す前に，指定された XML ファイルを値としてロードします。New... 関数は，新しい（空の）TXMLDocument インスタンスを作成し，ルートノードへのインターフェースを返します。

生成されたインターフェースは XML ドキュメントの構造をより直接的に反映しているので，このインターフェースを使用するとコードを単純にできます。たとえば，このインターフェースを使用せずに記述したコードは次のようになります。

```
_di_IXMLNode CustIntf = XMLDocument1->DocumentElement;
CustName = CustIntf->ChildNodes->Nodes[0]->ChildNodes->Nodes["name"]->Value;
```

ところが，このインターフェースを使用すると，次のようなコードになります。

```
_di_ICustomers CustIntf = GetCustomers(XMLDocument1);
CustName = CustIntf->Nodes[0]->Name;
```

データバインドウィザードが生成したインターフェースはすべて，IXMLNode を継承しています。つまり，データバインドウィザードを使わない場合と同じように，子ノードの追加と削除ができます（35-5 ページの「子ノードの追加と削除」を参照してください）。さらに，（1 つのノードの子がすべて同じ型であるときに）子ノードが繰り返しの要素を表す場合，親ノードには，追加の繰り返しを追加するために Add と Insert の 2 つのメソッドが与えられます。これらのメソッドでは作成するノードの型を指定する必要がないので，AddChild を使用するより簡単です。

XML データバインドウィザードの使い方

データバインドウィザードを使用する手順は次のとおりです。

1. ［ファイル｜新規作成｜その他］を選択し，［新規作成］ページの下部から［XML データバインド］というラベルのアイコンを選択します。
2. XML データバインドウィザードが表示されます。
3. このウィザードの最初のページで，インターフェースを生成する XML ドキュメントまたは XML スキーマを指定します。サンプルの XML ドキュメント，ドキュメントタイプ定義（.dtd）ファイル，縮小 XML データ（.xdr）ファイル，または XML スキーマ（.xsd）ファイルを指定できます。
4. ［オプション］ボタンをクリックし，インターフェースと実装クラスを生成するときにウィザードが使用する命名方法，およびスキーマで定義された型からネイティブなデータ型へのデフォルトのマッピングを指定します。
5. このウィザードの 2 ページ目に進みます。このページでは，ドキュメントまたはスキーマ内のあらゆるノード型についての詳細な情報を指定できます。左側のツリービューには，ドキュメント内のすべてのノード型が表示されます。複合ノード（子を持つノード）の場合は，ツリービューを展開して子の要素を表示できます。このツリービューの中でノードを 1 つ選択すると，そのノードについての情報がダイアログの右側に表示され，そのノードの処理方法を指定できます。
   - ［元の名前］コントロールは，XML スキーマ内のノード型の名前を表示します。
   - ［元のデータ型］コントロールは，XML スキーマ内で指定されているノードの値の型を表示します。
   - ［説明］コントロールを使用すると，そのノードの用途または目的を記述するコメントをスキーマに追加できます。
   - 選択されたノードのコードをウィザードが生成する場合（すなわち，ノードが複合型であるためインターフェースと実装クラスをウィザードが生成する場合，または複合型の子要素の 1 つであるため複合型のインターフェース上のプロパティをウィザードが生成する場合），［バインドを生成］チェックボックスを使用して，ウィザードがそのノードのコードを生成する必要があるかどうかを指定できます。［バインドを生成］のチェックマークをはずすと，ウィザードは複合型のインターフェースまたは実装クラスを生成しないか，子の要素または属性の親インターフェース内でプロパティを作成しません。

- ［バインドオプション］セクションを使用すると，選択された要素に対してウィザードが生成するコードに影響を与えることができます。どのノードの場合も，［識別子の名前］（生成されたインターフェースまたはプロパティの名前）を指定できます。さらに，インターフェースの場合は，どれがドキュメントのルートノードを表すかを示さなければなりません。プロパティを表すノードの場合は，プロパティの型を指定でき，プロパティがインターフェースでない場合は，読み出し専用のプロパティかどうかを指定できます。

6. ウィザードが各ノードに対してどのコードを生成するかを指定したら，3 ページ目に進みます。このページでは，ウィザードがどのようにコードを生成するかについてのグローバルオプションを選択でき，生成されるコードをプレビューでき，将来の使用に備えて選択内容をどのように保存するかを指示できます。
   - ウィザードが生成するコードをプレビューするには，［バインドの一覧］リスト内でインターフェースを選択し，結果として生じるインターフェース定義を［生成コードのプレビュー］コントロール内で参照します。
   - ［データバインドの設定］を使用して，ウィザードがどのように選択内容を保存するかを指定します。ドキュメントに関連付けられるスキーマファイル（このダイアログの 1 ページ目で指定されたスキーマファイル）内に注釈として設定内容を格納するか，このウィザードだけが使用する独立したスキーマファイルの名前を指定することができます。
7. ［完了］をクリックすると，データバインドウィザードは，XML ドキュメント内のすべてのノード型に対してインターフェースと実装クラスを定義する新しいユニットを生成します。さらに，TXMLDocument オブジェクトを受け取り，データ階層のルートノードのインターフェースを返すグローバル関数を作成します。

XML データバインドウィザードが生成するコードの使い方

ウィザードがインターフェースと実装クラスのセットを生成すると，これらを使用して，このウィザードに提供したドキュメントまたはスキーマの構造に一致する XML ドキュメントを操作できます。組み込みの XML コンポーネントだけを使用する場合と同様に，出発点はコンポーネントパレットの［Internet］ページにある TXMLDocument コンポーネントです。

XML ドキュメントを操作する手順は次のとおりです。

1. XML ドキュメントのルートノードのインターフェースを取得します。その方法は以下の 3 つです。
   - フォームまたはデータモジュール内に TXMLDocument コンポーネントを配置します。FileName プロパティを設定して，TXMLDocument を XML ドキュメントにバインドします（または，実行時に XML プロパティを設定して，XML の文字列を使用することができます）。次に，コード内で，XML ドキュメントのルートノードのインターフェースを取得するためにウィザードが作成したグローバル関数を呼び出します。たとえば，XML ドキュメントのルート要素がタグ <StockList> であった場合，デフォルトでは，IStockListType インターフェースを返す関数 GetStockListType をウィザードが生成します。

```
XMLDocument1->FileName := "Stocks.xml";
_di_IStockListType StockList = GetStockListType(XMLDocument1);
```

- 生成された Load... 関数を呼び出して，TXMLDocument インスタンスの作成とバインド，およびそのインターフェースの取得をすべて 1 つのステップで行います。たとえば，上記と同じ XML ドキュメントを使用すると次のようになります。

```
_di_IStockListType StockList = LoadStockListType("Stocks.xml");
```

- アプリケーション内のすべてのデータを作成したい場合に，生成された New... 関数を呼び出して空のドキュメントの TXMLDocument インスタンスを作成します。

```
_di_IStockListType StockList = NewStockListType();
```

2. このインターフェースには，ドキュメントのルート要素のサブノードに対応するプロパティ，およびそのルート要素の属性に対応するプロパティがあります。これらを使用すると，XML ドキュメントの階層の横断，ドキュメント内のデータの変更などができます。
3. このウィザードが生成したインターフェースを使用して行った変更を保存するには，TXMLDocument コンポーネントの SaveToFile メソッドを呼び出すか，またはその XML プロパティを読み出します。

ヒント TXMLDocument オブジェクトの Options プロパティが doAutoSave を含むように設定する場合は，SaveToFile メソッドを明示的に呼び出す必要はありません。

第36章

Web サービスの使い方

Web サービスは，インターネットによって公開および起動ができる自己包含型のモジュラーアプリケーションです。Web サービスは，提供されるサービスを記述する，明確に定義されたインターフェースを提供します。クライアントブラウザ用の Web ページを生成する Web サーバーアプリケーションとは異なり，Web サービスは人間と直接対話するために設計されているのではありません。Web サービスは，クライアントアプリケーションによってプログラム的にアクセスされます。

Web サービスは，クライアントとサーバーとのゆるやかな結合を可能にするために設計されています。つまり，クライアントが特定のプラットフォームまたはプログラミング言語を使用することをサーバー実装が要求しません。言語に中立な方法でインターフェースを定義できるだけでなく，複数の通信メカニズムも可能にするように設計されています。

C++Builder による Web サービスのサポートは，SOAP（Simple Object Access Protocol）を使用して機能するように設計されています。SOAP は，非集中型分散環境において情報を交換するための標準的な軽量プロトコルです。SOAP は，XML を使用してリモート手続き呼び出しをエンコードし，通常は HTTP を通信プロトコルとして使用します。SOAP についての詳細は，以下の URL で SOAP の仕様を参照してください。

```
http://www.w3.org/TR/SOAP/
```

メモ Web サービスをサポートするコンポーネントは SOAP と HTTP を使用するように構築されていますが，フレームワークが十分な一般性を備えているので，その他のエンコードおよび通信プロトコルを使用するように拡張できます。

C++Builder では，SOAP ベースの Web サービスアプリケーション（サーバー）を構築できるだけでなく，SOAP エンコードまたは Document Literal スタイルを使用する Web サービスのクライアントのサポートを提供します。Document Literal スタイルは，.Net Web サービスで使用されます。

Web サービスをサポートするコンポーネントは，Windows と Linux の両方で使用できるので，クロスプラットフォームの分散アプリケーションの基礎として利用できます。特殊なクライアント実行時ソフトウェアのインストールは必要ありません（これに対し，CORBA を使用してアプリケーションを配布するときは必要になります）。このテクノロジーは HTTP メッセージに基づいているので，さまざまなマシンで広く使用できるという利点があります。Web サービスのサポートは，Web サーバーアプリケーションアーキテクチャ（WebBroker）の上に構築されています。

Web サービスアプリケーションは，使用可能なインターフェースについての情報，および WSDL（Web Service Definition Language）ドキュメントを使用して呼び出す方法についての情報を公開します。サーバー側では，該当する Web サービスを記述する WSDL ドキュメントをアプリケーションが公開できます。クライアント側では，公開された WSDL ドキュメントをウィザードまたはコマンドラインユーティリティがインポートして，必要なインターフェース定義と接続情報を提供することができます。実装したい Web サービスを記述する WSDL ドキュメントがすでにある場合は，その WSDL ドキュメントをインポートするときにサーバー側のコードも生成できます。

起動可能インターフェースについて

Web サービスをサポートするサーバーは，起動可能インターフェースを使用して構築されます。起動可能インターフェースとは，実行時型情報（RTTI）を含むようにコンパイルされる純粋仮想クラス（純粋仮想メソッドのみが含まれるクラス）です。サーバーでは，この RTTI は，クライアントから受信したメソッド呼び出しを正しくマーシャリングできるように解釈するときに使用されます。クライアントでは，この RTTI は，インターフェースのメソッドを呼び出すためのメソッドテーブルを動的に生成するために使用されます。

起動可能インターフェースを作成するには，**_declspec** キーワードを使用し，delphirtti 修飾子を使用してインターフェースクラスを宣言する必要があります。起動可能インターフェースの下位も起動可能です。ただし，起動可能でない別のインターフェースクラスを起動可能インターフェースが継承する場合，Web サービスは，起動可能インターフェースとその下位で定義されたメソッドしか使用することができません。起動可能でない上位を継承するメソッドは，型情報を使用してコンパイルされないので，Web サービスの一部として使用することができません。

メモ C++Builder におけるインターフェースクラスについての詳細は，13-2 ページの「継承とインターフェース」を参照してください。

sysmac.h ヘッダーファイルは，基本の起動可能インターフェースである IInvokable を定義し，Web サービスサーバーによってクライアントに公開されるインターフェースをこのインターフェースから派生させることができます。IInvokable は，基本のインターフェース（IInterface）とほぼ同じですが，異なる点は，このインターフェースとすべての下位が RTTI を含むように，**_declspec**(delphirtti) オプションを使用してコンパイルされる点です。

たとえば，次のコードは，数値のエンコードとデコードを行う 2 つのメソッドを含む起動可能インターフェースを定義しています。

```
__interface INTERFACE_UUID("{C527B88F-3F8E-1134-80e0-01A04F57B270}") IEncodeDecode :
    public IInvokable
{
public:
  virtual double __stdcall EncodeValue(int Value) = 0 ;
  virtual int __stdcall DecodeValue(double Value) = 0 ;
};
```

メモ この例では，宣言での **__interface** の使い方に注目してください。これは本当のキーワードではなく，慣例的にインターフェースに使用されるマクロです。これは **class** キーワードにマップします。INTERFACE_UUID マクロは，インターフェースに GUID（globally unique identifier）を割り当てます。このインターフェースクラスには純粋仮想メソッドしか含まれません。

Web サービスアプリケーションがこの起動可能インターフェースを使用できるようにするには，その前にインターフェースを起動レジストリに登録しなければなりません。サーバーでは，起動レジストリのエントリを使用すると，インボーカコンポーネント（THTTPSOAPCppInvoker）が，インターフェース呼び出しの実行に使用する実装クラスを識別することができます。クライアントアプリケーションでは，起動レジストリのエントリを使用すると，起動可能インターフェースを識別する情報をリモートインターフェースオブジェクト（THTTPRio）が検索でき，その呼び出し方法についての情報が提供されます。

通常は，Web サービスのクライアントまたはサーバーは，WSDL ドキュメントをインポートするか Web サービスウィザードを使用して，起動可能インターフェースを定義するコードを作成します。デフォルトでは，WSDL インポータまたは Web サービスウィザードがインターフェースを生成するときに，Web サービスと同名のヘッダーファイルに定義が追加されます。対応する .cpp ファイルには，インターフェースを登録するコードが含まれます（サーバーを作成する場合は実装クラスも含まれます）。上で説明したインターフェースの場合，この登録コードは次のようになります。

```
static void RegTypes()
{
  InvRegistry()->RegisterInterface(__delphirtti(IEncodeDecode), "", "");
}
#pragma startup RegTypes 32
```

Web サービスのインターフェースは，すべての可能な Web サービスの中のすべてのインターフェースからそのインターフェースを識別する名前空間を持たなければなりません。上の例では，インターフェースの名前空間が提供されていません。明示的に名前空間が提供されない場合は，起動レジストリによって自動的に生成されます。この名前空間は，アプリケーションをユニークに識別する文字列（AppNamespacePrefix 変数），インターフェース名，およびインターフェースを定義するユニットの名前から構築されます。自動生成された名前空間を使用したくない場合は，RegisterInterface 呼び出しの 2 番目のパラメータを使用して明示的に名前空間を指定できます。

メモ クライアントとサーバーの両方のアプリケーションで同じヘッダーファイルを使用して起動可能インターフェースを定義しないようにしてください。同じファイルを使用すると，名前空間に不一致が起きやすくなります。なぜなら，そのヘッダーが別のソースファイルに含まれる場合に，ユニットの名前がそのソースファイルに変更され，生成された名前空間の名前が変更されるからです。したがって，クライアントアプリケーションは，WSDL ドキュメントを使用して Web サービスをインポートする必要があります。

起動可能インターフェースでの非スカラー型の使用

Web サービスアーキテクチャには，以下のスカラー型のマーシャリングのサポートが自動的に含まれます。

- **bool**
- **char**
- **signed char**
- **unsigned char**
- **short**
- **unsigned short**

- **int**
- **unsigned int**
- **long**
- **unsigned long**
- **__int64**
- **unsigned __int64**
- **float**
- **double**
- **long double**
- AnsiString
- WideString
- Currency
- Variant

起動可能インターフェース上でこれらの型を使用する場合は，何か特殊なことをする必要はありません。ただし，その他の型を使用するプロパティまたはメソッドがインターフェースに含まれる場合は，アプリケーションがその型をリモート可能タイプレジストリに登録しなければなりません。リモート可能タイプレジストリについての詳細は，36-4 ページの「非スカラー型の登録」を参照してください。

列挙型，および **typedef** 文を使用して宣言された型の場合，リモート可能タイプレジストリが必要な型情報を型定義から取り出せるようにするために，余分な作業が少し必要になります。これらについては，36-6 ページの「typedef 宣言された型および列挙型の登録」で説明しています。

sysdyn.h で定義された動的配列型は自動的に登録されるので，この動的配列型用の特殊な登録コードをアプリケーションで追加する必要はありません。動的配列型の 1 つの TByteDynArray は特に注意を要します。なぜなら，ほかの動的配列型のように各配列要素を別個にマップせずに，「base64」ブロックのバイナリデータにマップするからです。

その他の型（静的配列，構造体，クラスなど）の場合は，その型をリモート可能クラスにマップしなければなりません。リモート可能クラスとは，実行時型情報（RTTI：Runtime Type Information）を含むクラスです。そしてインターフェースは，対応する静的配列，構造体，またはクラスではなくリモート可能クラスを使用しなければなりません。作成するリモート可能クラスは，リモート可能タイプレジストリに登録しなければなりません。

重要 すべての型は，インラインで宣言するのではなく，**typedef** 文で明示的に宣言する必要があります。なぜなら，そうすればネイティブな C++ 型名をリモート可能タイプレジストリが判別できるからです。

非スカラー型の登録

36-3 ページの「起動可能インターフェースでの非スカラー型の使用」にリストした組み込みのスカラー型以外の型を起動可能インターフェースが使用できるようにするには，まずアプリケーションがその型をリモート可能タイプレジストリに登録しなければなりません。リモート可能タイプレジストリにアクセスするには，ソースファイルに InvokeRegistry.hpp を含めなければなりません。このヘッダーは，リモート可能タイプレジストリの参照を返すグローバル関数 RemTypeRegistry() を宣言します。

メモ クライアントでは，WSDL ドキュメントをインポートしたときに，リモート可能タイプレジストリに型を登録するコードが自動的に生成されます。サーバーでは，リモート可能な型を使用するインターフェースを登録したときに，その型が自動的に登録されます。型を登録するコードを明示的に追加する必要があるのは，自動生成された値を使用せずに名前空間または型名を指定したい場合だけです。

リモート可能タイプレジストリには，型の登録に使用できるメソッドは，RegisterXSInfo と RegisterXSClass の 2 つがあります。RegisterXSInfo を使用すると，動的配列を登録できます。RegisterXSClass は，その他の型を表すために定義するリモート可能クラスを登録するメソッドです。

動的配列を使用する場合，起動レジストリは，コンパイラが生成した型情報から必要な情報を取得できます。したがって，インターフェースはたとえば次のような型を使用できます。

```
typedef DynamicArray<TXSDateTime> TDateTimeArray;
```

起動可能インターフェースを登録するときにこの型は自動的に登録されます。ただし，型を定義する名前空間，または型の名前を指定したい場合は，この動的配列を使用する起動可能インターフェースを登録する RegTypes 関数に，次の登録を追加しなければなりません。

```
void RegTypes()
{
  RemTypeRegistry()->RegisterXSInfo(__arraytypeinfo(TDateTimeArray),
       MyNameSpace, "DTarray", "DTarray");
  InvRegistry()->RegisterInterface(__delphirtti(ITimeServices));
}
```

メモ すべてが同じ基底の要素型にマップする複数の動的配列型は使用しないでください。なぜなら，相互に暗黙にキャストできる透過型としてコンパイラに処理されるので，実行時型情報が識別されないからです。

RegisterXSInfo の 1 番目のパラメータは，登録しようとしている型の型情報です。2 番目のパラメータは，型が定義される名前空間の名前空間 URI です。このパラメータを省略するか，空の文字列を指定すると，レジストリが名前空間を自動的に生成します。3 番目のパラメータは，ネイティブな C++ コードに表示される型の名前です。このパラメータには値を指定しなければなりません。最後のパラメータは，WSDL ドキュメントに表示される型の名前です。このパラメータを省略するか，空の文字列を指定すると，レジストリはネイティブな型名（3 番目のパラメータ）を使用します。

リモート可能クラスの登録方法は似ていますが，型情報のポインタではなくクラス参照を指定する点が異なります。たとえば，次のコードは，TXSRecord というリモート可能クラスを登録します。

```
void RegTypes()
{
  RemTypeRegistry()->RegisterXSclass(__classid(TXSRecord), MyNameSpace, "record", "",
      false);
  InvRegistry()->RegisterInterface(__delphirtti(ITimeServices));
}
```

1 番目のパラメータは，型を表すリモート可能クラスのクラス参照です。2 番目のパラメータは，新しいクラスの名前空間をユニークに識別する URI（Uniform Resource Identifier）です。空の文字列を指定すると，レジストリが自動的に URI を生成します。3 番目と 4 番目のパラメータは，クラスが表すデータ型のネイティブな名前と外部の名前を指定します。2 番目と 4 番目のパラメータを省略すると，タイプレジストリは 3 番目のパラメータを両方の値に使用します。両方のパラメータに空の文字列を指定すると，レジストリはクラス名を使用します。5 番目のパラメータは，クラスインスタンス

の値を文字列として転送できるかどうかを示します。（ここには表示されていない）6 番目のパラメータを追加して，同じオブジェクトインスタンスに対する複数の参照を SOAP パケット内でどのように表すかを制御することもできます。

typedef 宣言された型および列挙型の登録

起動可能インターフェースが列挙型を使用する場合，**__delphirtti** 関数が型情報を直接取り出すことはできません。これに対処するには，列挙型の型情報の取り出しに使用できる VCL 形式のホルダークラスを生成する必要があります。

```
class MyEnumType_TypeInfoHolder : public TObject {
  MyEnumType __instanceType;
public:
__published:
  __property MyEnumType __propType = {read=__instanceType };
};
```

この場合，クラス MyEnumType_TypeInfoHolder は，MyEnumType という列挙型から型情報を取り出すために定義されています。これは 1 つのパブリッシュプロパティ __propType を持ち，その型は，登録しようとしている列挙型です。ホルダークラスを一度定義すると，グローバル関数 GetClsMemberTypeInfo を呼び出すことで列挙型の型情報を取得できます。次のコードは，ホルダークラスを指定して列挙型を登録する方法を示しています。

```
void RegTypes()
{
  RemTypeRegistry()->RegisterXSInfo(
         GetClsMemberTypeInfo(__classid(MyEnumType_TypeInfoHolder), "__propType"),
         MyNameSpace, "MyEnumType");
}
```

RegisterClsMemberTypeInfo には 2 つのパラメータがあります。1 つはホルダークラスの型情報，もう 1 つは型情報を取り出そうとしている型のパブリッシュプロパティの名前です。（上の例のように）ホルダークラスにパブリッシュプロパティが 1 つしかない場合は，2 番目のパラメータを省略できます。

組み込みの C++ スカラー型に型をマップする **typedef** 文を使用して型を宣言する場合も，このようにしてでホルダークラスを使用しなければなりません（組み込みの C++ スカラー型は，36-3 ページの「起動可能インターフェースでの非スカラー型の使用」にリストされた型ですが，Object Pascal の型をエミュレートするクラスである型を除きます）。したがって，たとえば次の型があるとします。

```
typedef int CardNumber;
```

この場合は，次のようなホルダークラスを作成します。

```
class CardNumberType_TypeInfoHolder : public TObject {
  CardNumber __instanceType;
public:
__published:
  __property CardNumber __propType = {read=__instanceType };
};
```

そして次のようにして登録します。

```
RemTypeRegistry()->RegisterXSInfo(GetClsMemberTypeInfo(
         __classid(CardNumber_TypeInfoHolder), "__propType"),
         MyNameSpace, "CardNumber");
```

typedef 文を使用して宣言されるその他の型の場合は，動的配列クラスにマップする場合は RegisterXSInfo を，クラスにマップする場合は RegisterXSClass を呼び出します。動的配列およびクラスの登録方法についての詳細は，36-4 ページの「非スカラー型の登録」を参照してください。

メモ 複数の型を同じ基底の型にマップする **typedef** 文は使用しないようにしてください。これらの実行時型情報は明確ではありません。

リモート可能オブジェクトの使い方

TRemotable は，起動可能インターフェース上で複合データ型を表すクラスを定義するときの基本クラスとして使用します。たとえば，通常は構造体をパラメータとして渡す場合に，通常の方法をとらずに，構造体のあらゆるメンバーが新しいクラスのパブリッシュプロパティである TRemotable の下位を定義します。

TRemotable の下位のパブリッシュプロパティがその型の対応する SOAP エンコード内の要素ノードまたは属性のいずれとして表示されるかは，制御することができます。このプロパティを属性にするには，プロパティ定義上で **stored** 指令を使用して，AS_ATTRIBUTE を値として割り当てます。

```
__property bool MyAttribute =
                        {read=FMyAttribute, write=FMyAttribute, stored= AS_ATTRIBUTE;
```

メモ **stored** 指令を含めない場合，または **stored** 指令に（AS_ATTRIBUTE を返す関数にさえ）その他の値を割り当てる場合，このプロパティは属性ではなくノードとしてエンコードされます。

新しい TRemotable の下位の値が WSDL ドキュメント内のスカラー型を表す場合は，代わりの基本クラスとして TRemotableXS を使用する必要があります。TRemotableXS は TRemotable の下位クラスで，新しいクラスとその文字列表現の間で変換を行うメソッドを 2 つ導入します。これらのメソッドを実装するには，XSToNative および NativeToXS メソッドをオーバーライドします。

よく使われる特定の XML スカラー型用に，XSBuiltIns ユニットはすでにリモート可能クラスを定義および登録しています。次の表にこのリモート可能クラスを示します。

表 36.1　リモート可能クラス

XML 型	リモート可能クラス
dateTime timeInstant	TXSDateTime
date	TXSDate
time	TXSTime
duration timeDuration	TXSDuration
decimal	TXSDecimal
hexBinary	TXSHexBinary

リモート可能クラスを定義したら，36-4 ページの「非スカラー型の登録」で説明したように，このクラスはリモート可能タイプレジストリに登録されなければなりません。サーバーでは，このクラスを使用するインターフェースを登録したときにこの登録は自動的に行われます。クライアントでは，型を定義する WSDL ドキュメントをインポートしたときに，このクラスを登録するコードが自動的に生成されます。

ヒント TRemotable の下位を実装および登録するユニットは，サーバーアプリケーションのその他の部分（起動可能インターフェースを宣言および登録するユニットなど）から分けたほうがよいでしょう。そうすれば，複数のインターフェースに対してその型を使用できます。

TRemotable の下位を使用するときの問題の 1 つは，いつ作成および破棄するかということです。明らかに，サーバーアプリケーションはこれらのオブジェクトのローカルインスタンスを独自に作成しなければなりません。なぜなら，呼び出し側のインスタンスが別個のプロセス空間内にあるからです。これに対処するために，Web サービスアプリケーションは，受信したリクエストに対してデータコンテキストを作成します。このデータコンテキストは，サーバーがそのリクエストを処理する間は保持され，出力パラメータが戻りメッセージ内にマーシャリングされた後に解放されます。サーバーは，リモート可能オブジェクトのローカルインスタンスを作成すると，これをデータコンテキストに追加し，このインスタンスはデータコンテキストとともに解放されます。メソッド呼び出しの後もリモート可能オブジェクトのインスタンスが解放されないようにしたい場合もあります。たとえば，オブジェクトにステート情報が入っている場合は，あらゆるメッセージ呼び出しに対して 1 つのインスタンスを使用した方が効率的になることがあります。データコンテキストとともにリモート可能オブジェクトが解放されないようにするには，DataContext プロパティを変更します。

リモート可能オブジェクトの例

次の例は，起動可能インターフェースのパラメータ用に，既存のクラスを使用する代わりにリモート可能オブジェクトを作成する方法を示しています。この例では，既存のクラスは文字列リスト（TStringList）です。この例を小さくとどめるために，この文字列リストの Objects プロパティは複製していません。

この新しいクラスはスカラーではないので，TRemotableXS ではなく TRemotable を継承します。クライアントとサーバーの間で通信する文字列リストのあらゆるプロパティのパブリッシュプロパティが含まれています。これらの各リモート可能プロパティは，リモート可能型に対応します。さらに，新しいリモート可能クラスには，文字列リストへの変換および文字列リストからの変換を行うメソッドが含まれます。

```
class TRemotableStringList: public TRemotable
{
  private:
    bool FCaseSensitive;
    bool FSorted;
    Classes::TDuplicates FDuplicates;
    System::TStringDynArray FStrings;
  public:
    void __fastcall Assign(Classes::TStringList *SourceList);
    void __fastcall AssignTo(Classes::TStringList *DestList);
__published:
    __property bool CaseSensitive = {read=FCaseSensitive, write=FCaseSensitive};
    __property bool Sorted = {read=FSorted, write=FSorted};
    __property Classes::TDuplicates Duplicates = {read=FDuplicates, write=FDuplicates};
    __property System::TStringDynArray Strings = {read=FStrings, write=FStrings};
}
```

TRemotableStringList は，トランスポートクラスとしてのみ存在しています。したがって，（文字列リストの Sorted プロパティの値を転送するために）Sorted プロパティは持っていますが，格納している文字列をソートする必要はなく，文字列をソートすべきかどうかを記録する必要があるだけです。し

たがって，実装が非常に単純になります。文字列リストへの変換および文字列リストからの変換を行う Assign メソッドと AssignTo メソッドさえ実装すればよいからです。

```
void __fastcall TRemotableStringList::Assign(Classes::TStringList *SourceList)
{
  SetLength(Strings, SourceList->Count);
  for (int i = 0; i < SourceList->Count; i++)
    Strings[i] = SourceList->Strings[i];
  CaseSensitive = SourceList->CaseSensitive;
  Sorted = SourceList->Sorted;
  Duplicates = SourceList->Duplicates;
}
void __fastcall TRemotableStringList::AssignTo(Classes::TStringList *DestList)
{
  DestList->Clear();
  DestList->Capacity = Length(Strings);
  DestList->CaseSensitive = CaseSensitive;
  DestList->Sorted = Sorted;
  DestList->Duplicates = Duplicates;
  for (int i = 0; i < Length(Strings); i++)
    DestList->Add(Strings[i]);
}
```

新しいリモート可能クラスを登録してクラス名を指定できるようにすることもできます。このクラスを登録しない場合は，これを使用するインターフェースを登録したときに自動的に登録されます。同様に，このクラスを登録し，このクラスが使用する TDuplicates 型と TStringDynArray 型を登録しない場合は，これらの型は自動的に登録されます。次のコードは，TRemotableStringList クラスの登録方法を示しています。TStringDynArray は自動的に登録されます。なぜなら，sysdyn.h で宣言された組み込みの動的配列型の 1 つだからです。TDuplicates などの列挙型を明示的に登録する方法についての詳細は，36-6 ページの「typedef 宣言された型および列挙型の登録」を参照してください。

```
void RegTypes()
{
  RemTypeRegistry()->RegisterXSclass(__classid(TRemotableStringList), MyNameSpace,
                                     "stringList", "", false);
}
#pragma startup initServices 32
```

Web サービスをサポートするサーバーの記述

起動可能インターフェース，およびその純粋仮想メソッドを実装する下位クラスのほかにも，サーバーはディスパッチャとインボーカの 2 つのコンポーネントを必要とします。ディスパッチャ（THTTPSoapDispatcher）は，受信した SOAP メッセージを受け取り，インボーカに渡します。インボーカ（THTTPSoapCppInvoker）は，この SOAP メッセージを解釈し，このメッセージが呼び出す起動可能インターフェースを識別し，その呼び出しを実行し，レスポンスメッセージを組み立てます。

メモ THTTPSoapDispatcher と THTTPSoapCppInvoker は，SOAP リクエストを含む HTTP メッセージに応答するように設計されています。ただし，基底のアーキテクチャが十分な一般性を備えているので，異なるディスパッチャおよびインボーカコンポーネントと置換することによってほかのプロトコルをサポートできます。

起動可能インターフェースとその実装クラスを登録すると，ディスパッチャとインボーカが HTTP リクエストメッセージの SOAP Action ヘッダーの中でこのインターフェースを識別するメッセージを自動的に処理します。

Web サービスにはパブリッシャ（TWSDLHTMLPublish）も含まれます。パブリッシャは，アプリケーション内で Web サービスを呼び出す方法を記述した WSDL ドキュメントを作成することで，受信したクライアントリクエストに応答します。

Web サービスサーバーの構築

Web サービスを実装するサーバーアプリケーションを構築する手順は次のとおりです。

1. ［ファイル｜新規作成｜その他］を選択し，［Web Services］タブで［SOAP サーバーアプリケーション］アイコンをダブルクリックして SOAP サーバーアプリケーションウィザードを起動します。SOAP リクエストに応答するために必要なコンポーネントが含まれる 新しい Web サーバーアプリケーションが作成されます。SOAP アプリケーションウィザードおよびこのウィザードが生成するコードについての詳細は，36-11 ページの「SOAP アプリケーションウィザードの使い方」を参照してください。
2. SOAP サーバーアプリケーションウィザードを終了すると，Web サービスのインターフェースを定義するかどうかが確認されます。まったく最初から Web サービスを作成する場合は［はい］をクリックします。［Web サービスの新規作成］ウィザードが表示されます。このウィザードは，Web サービスの新しい起動可能インターフェースを宣言および登録するコードを追加します。生成されたコードを編集して Web サービスを定義および実装します。追加のインターフェースを追加したい場合（または後でインターフェースを定義したい場合）は，［ファイル｜新規作成｜その他］を選択し，［WebServices］タブで［SOAP Web サービスインターフェース］アイコンをダブルクリックします。［Web サービスの新規作成］ウィザードの使い方，およびこのウィザードが生成するコードの完成方法についての詳細は，36-12 ページの「新しい Web サービスの追加」を参照してください。
3. すでに WSDL ドキュメントで定義された Web サービスを実装する場合は，Web サービスインポートを使用して，アプリケーションに必要なインターフェース，実装クラス，および登録コードを生成します。Web サービスインポートが実装クラス用に生成するメソッドの本体を埋めるだけで完成できます。Web サービスインポートの使い方については，36-13 ページの「Web サービスインポートの使用」を参照してください。
4. SOAP リクエストを実行しようとしたときにアプリケーションが例外を生成する場合は，その例外が自動的に SOAP 障害パケット内にエンコードされ，メソッド呼び出しの結果のかわりに返されます。単なるエラーメッセージ以上の詳細な情報を伝達する必要がある場合は，エンコードされてクライアントに渡される独自の例外クラスを作成できます。これについては，36-14 ページの「Web サービス用のカスタム例外クラスの作成」で説明しています。
5. SOAP サーバーアプリケーションウィザードは，新しい Web サービスアプリケーションにパブリッシャコンポーネント（TWSDLHTMLPublish）を追加します。これによってこのアプリケーションは，Web サービスを記述する WSDL ドキュメントをクライアントに公開できます。WSDL パブリッシャの詳細については，36-15 ページの「Web サービスアプリケーション用の WSDL ドキュメントの生成」を参照してください。

SOAP アプリケーションウィザードの使い方

Web サービスアプリケーションは，特殊な形式の Web サーバーアプリケーションです。そのため，Web サービスのサポートが WebBroker アーキテクチャの上に構築されます。したがって，SOAP アプリケーションウィザードが生成するコードを理解するには，WebBroker アーキテクチャを理解することが役立ちます。Web サーバーアプリケーションについての一般的な情報，特に WebBroker についての情報は，第 32 章「インターネットサーバーアプリケーションの作成」および第 33 章「WebBroker の使い方」を参照してください。

SOAP アプリケーションウィザードを起動するには，［ファイル｜新規作成｜その他］を選択し，［WebServices］ページで，［SOAP サーバーアプリケーション］アイコンをダブルクリックします。Web サービスに使用する Web サーバーアプリケーションの種類を選択します。Web サーバーアプリケーションの種類についての詳細は，32-6 ページの「Web サーバーアプリケーションの種類」を参照してください。

ウィザードは，次の 3 つのコンポーネントが入った Web モジュールを含む新しい Web サーバーアプリケーションを生成します。

- インボーカコンポーネント（THTTPSOAPCppInvoker）：インボーカは，SOAP メッセージの間で，および Web サービスアプリケーション内で起動可能インターフェースを登録した場合はそのメソッドの間で変換を行います。
- ディスパッチャコンポーネント（THTTPSoapDispatcher）：ディスパッチャは，受信した SOAP メッセージに自動的に応答し，これをインボーカに転送します。その WebDispatch プロパティを使用すると，アプリケーションが応答する HTTP リクエストメッセージを識別できます。それには，PathInfo プロパティを設定して，アプリケーションに送られる URL のパス部分を示し，MethodType プロパティを設定してリクエストメッセージのメソッドヘッダーを示す必要があります。
- WSDL パブリッシャ（TWSDLHTMLPublish）：WSDL パブリッシャは，インターフェースとその呼び出し方法を記述した WSDL ドキュメントを公開します。この WSDL ドキュメントは，ここで作成する Web サービスアプリケーションの呼び出し方法をクライアントに示します。WSDL パブリッシャの使い方については，36-15 ページの「Web サービスアプリケーション用の WSDL ドキュメントの生成」を参照してください。

SOAP ディスパッチャと WSDL パブリッシャは自動ディスパッチコンポーネントです。つまり，自分自身を自動的に Web モジュールに登録することによって，WebDispatch プロパティで指定したパス情報を使用してアドレス指定された受信リクエストを転送します。この Web モジュールを右クリックすると，これらの自動ディスパッチコンポーネントのほかに，DefaultHandler という Web アクション項目が 1 つあることがわかります。

DefaultHandler はデフォルトのアクション項目です。つまり，Web モジュールが受信したリクエストのハンドラが見つからない場合（パス情報が一致しない場合），そのメッセージをデフォルトのアクション項目に転送します。DefaultHandler は，Web サービスを記述する Web ページを生成します。デフォルトアクションを変更するには，このアクション項目の OnAction イベントハンドラを編集します。

新しい Web サービスの追加

新しい Web サービスインターフェースをサーバーアプリケーションに追加するには，［ファイル｜新規作成｜その他］を選択し，［WebServices］タブで［SOAP サーバーインターフェース］アイコンをダブルクリックします。

［Web サービスの新規作成］ウィザードでは，クライアントにエクスポーズする起動可能インターフェースの名前を指定できます。また，このインターフェースとその下位の実装クラスを宣言および登録するコードが生成されます。デフォルトでは，サンプルのメソッドと追加の型定義を示すコメントも生成されるので，生成されたファイルの編集に着手しやすくなっています。

生成されたコードの編集

生成されたユニットのヘッダーファイルにインターフェースの定義が表示されます。このユニットの名前はウィザードで指定した名前になります。インターフェース宣言を変更するには，サンプルのメソッドを置き換えて，クライアントが使用できるようにするメソッドにします。

TInvokableClass と起動可能インターフェースを継承する実装クラスがウィザードによって生成されます。まったく最初から起動可能インターフェースを定義する場合は，実装クラスの宣言を編集して，生成された起動可能インターフェースに行った編集と一致させなければなりません。

起動可能インターフェースと実装クラスにメソッドを追加するときは，そのメソッドではリモート可能な型だけを使用しなければならないことに注意してください。リモート可能な型と起動可能インターフェースについては，36-3 ページの「起動可能インターフェースでの非スカラー型の使用」を参照してください。

異なる基本クラスの使用

Add New Web Service ウィザードは，TInvokableClass を継承する実装クラスを生成します。Web サービスを実装する新しいクラスを作成する方法としてはこれがもっとも簡単です。ただし，この生成されたクラスは，異なる基本クラスを持つ実装クラスで置き換えることができます（たとえば，既存のクラスを基本クラスとして使用できます）。生成された実装クラスを置き換えるときは，以下のようにいくつか考慮すべきポイントがあります。

- 新しい実装クラスは，起動可能インターフェースを直接継承しなければなりません。起動可能インターフェースとその実装クラスが登録される起動レジストリは，登録された各インターフェースをどのクラスが実装するかを追跡し，インボーカがインターフェースを呼び出す必要ができたときにインボーカコンポーネントが使用できるようにします。クラス宣言にインターフェースが直接含まれる場合，あるクラスがそのインターフェースを実装したことしか検出できません。基本クラスとともに継承された場合は，インターフェースのサポートは検出されません。
- 新しい実装クラスは，インターフェースの一部である IUnknown メソッドのサポートを含まなければなりません。このことは言うまでもないように思えますが，見落とされがちです。IUnknown についての詳細は，13-4 ページの「IUnknown をサポートするクラスの作成」を参照してください。
- 実装クラスを登録する生成されたコードを変更し，実装クラスのインスタンスを作成するファクトリメソッドを含めなければなりません。

この最後のポイントには少し説明が必要です。実装クラスが TInvokableClass を継承し，継承したコンストラクタを，1 つまたは複数のパラメータを含む新しいコンストラクタで置き換えない場合，起動レジストリはクラスのインスタンスが必要になったときに作成する方法を知っています。TInvokableClass を継承しない実装クラスを作成する場合，またはコンストラクタを変更する場合は，実装クラスのインスタンスを取得する方法を起動レジストリに指示しなければなりません。

実装クラスのインスタンスを取得する方法を起動レジストリに指示するには，ファクトリ手続きを使用して指定します。TInvokableClass を継承し，継承したコンストラクタを使用する実装クラスの場合でも，ファクトリ手続きを指定することができます。たとえば，起動可能インターフェースの呼び出しをアプリケーションが受信するたびに起動レジストリが新しいインスタンスを作成しなければならないのではなく，1 つのグローバルインスタンスの実装クラスを使用できます。

このファクトリ手続きは，TCreateInstanceProc 型でなければなりません。これは実装クラスのインスタンスを返します。この手続きが新しいインスタンスを作成する場合，実装オブジェクトは，インターフェースの参照カウントが 0 になったときにオブジェクト自身を解放する必要があります。なぜなら，起動レジストリはオブジェクトインスタンスを明示的に解放しないからです。次のコードは，もう 1 つの方法，すなわちファクトリ手続きが 1 つのグローバルインスタンスの実装クラスを返す方法を示しています。

```
void __fastcall CreateEncodeDecode(System::TObject* &obj)
{
  if (!FEncodeDecode)
  {
    FEncodeDecode = new TEncodeDecodeImpl();
    // インターフェースへの参照を保存し，グローバルインスタンスが自分自身を解放しないようにする
    TEncodeDecodeImpl->QueryInterface(FEncodeDecodeInterface);
  }
  obj = FEncodeDecode;
}
```

メモ 上の例では，FEncodeDecodeInterface は _di_IEncodeDecode 型の変数です。

ファクトリ手続きを実装クラスに登録するには，そのクラスを起動レジストリに登録する呼び出しの 2 番目のパラメータとして指定します。まず，実装クラスを登録するためにウィザードが生成した呼び出しを探し出します。これはクラスを定義するユニットの下部の RegTypes メソッド内に表示されます。これはたとえば次のようになります。

```
InvRegistry()->RegisterInvokableClass(__classid(TEncodeDecodeImpl));
```

ファクトリ手続きを指定する 2 番目のパラメータをこの呼び出しに追加します。

```
InvRegistry()->RegisterInvokableClass(__classid(TEncodeDecodeImpl), &CreateEncodeDecode);
```

Web サービスインポートの使用

Web サービスインポートを使用するには，［ファイル｜新規作成｜その他］を選択し，［WebServices］ページで，［WSDL インポート］というラベルのアイコンをダブルクリックします。表示されたダイアログで，WSDL ドキュメント（または XML ファイル）のファイル名を指定するか，そのドキュメントが公開された URL を指定します。

認証を必要とするサーバー上に WSDL ドキュメントが置かれている場合（または認証を要求するプロクシーサーバーを使用しなければ WSDL ドキュメントに到達できない場合）は，ウィザードが

WSDL ドキュメントを取得できるようにするには，その前にユーザー名とパスワードを指定しなければなりません。この情報を指定するには，［オプション］ボタンをクリックし，適切な接続情報を指定します。

［次へ］ボタンをクリックすると，Web サービスインポートは，Web サービスのフレームワークと互換性がある WSDL ドキュメント内のあらゆる定義に対して生成したコードを表示します。つまり，SOAP バインドを持つポートの種類だけを使用します。Web サービスインポートがコードを生成する方法を設定するには，［オプション］ボタンをクリックし，必要なオプションを選択します。

Web サービスインポートは，サーバーまたはクライアントのいずれかのアプリケーションを記述するときに使用できます。サーバーを記述する場合は，［オプション］ボタンをクリックし，表示されたダイアログで，サーバーコードの生成を指示するオプションにチェックマークを付けます。このオプションを選択すると，Web サービスインポートが起動可能インターフェースの実装クラスを生成するので，メソッドの本体を埋めるだけで済みます。

注意 WSDL ドキュメントをインポートして，すでに定義されている Web サービスを実装するサーバーを作成する場合でも，そのサービスに対する自分の WSDL ドキュメントを公開しなければなりません。インポートされた WSDL ドキュメントと生成された実装には少し違いがあります。たとえば，WSDL ドキュメントまたは XML スキーマファイルが使用する識別子が C++ のキーワードでもある場合は，生成されたコードがコンパイルできるように自動的に名前が調整されます。

［完了］をクリックすると，ドキュメント内で定義された操作の起動可能インターフェースを定義および登録し，ドキュメントが定義する型のリモート可能クラスを定義および登録する新しいユニットが作成されます。

もう 1 つの方法として，コマンドライン WSDL インポータを使用することもできます。サーバーの場合は，次のように **-S** オプションを付けてコマンドラインインポータを呼び出します。

```
WSDLIMP -S -C -V MyWSDLDoc.wsdl
```

クライアントアプリケーションの場合は，次のように -S オプションを付けずにコマンドラインインポータを呼び出します。

```
WSDLIMP -C -V MyWSDLDoc.wsdl
```

Web サービス用のカスタム例外クラスの作成

Web サービスアプリケーションが SOAP リクエストを実行しようとしたときに例外を生成する場合は，SOAP 障害パケット内でその例外についての情報が自動的にエンコードされ，メソッド呼び出しの結果のかわりに返されます。次に，クライアントアプリケーションが例外を生成します。

デフォルトでは，クライアントアプリケーションが ERemotableException 型の汎用の例外を生成し，SOAP 障害パケットからの情報が提供されます。ERemotableException の下位から派生させることで，アプリケーション固有の追加情報を転送できます。例外クラスに追加したパブリッシュプロパティの値は SOAP 障害パケットに含まれるので，クライアントは同等の例外を生成することができます。

ERemotableException の下位を使用するには，これをリモート可能タイプレジストリに登録しなければなりません。したがって，ERemotableException の下位を定義するユニットでは，InvokeRegistry.hpp をインクルードし，グローバル関数 RemTypeRegistry が返すオブジェクトの RegisterXSClass メソッドの呼び出しを追加しなければなりません。

クライアントも ERemotableException の下位を定義および登録する場合は，クライアントが SOAP 障害パケットを受信したときに，自動的に適切な例外クラスのインスタンスを生成し，すべてのプロパティを SOAP 障害パケット内の値に設定します。

Web サービスアプリケーション用の WSDL ドキュメントの生成

アプリケーションが使用可能にする Web サービスをクライアントアプリケーションがわかるようにするには，起動可能インターフェースを記述し，その呼び出し方法を示す WSDL ドキュメントを公開します。

メモ インポートされたサービスを実装した場合でも，または C++Builder を使用してクライアントを記述する場合でも，Web サービスアプリケーション用の WSDL ドキュメントを必ず公開しなければなりません。

Web サービスを記述する WSDL ドキュメントを公開するには，Web モジュールに TWSDLHTMLPublish コンポーネントを含めます（SOAP サーバーアプリケーションウィザードは，デフォルトでこのコンポーネントを追加します）。TWSDLHTMLPublish は自動ディスパッチコンポーネントなので，Web サービス用の WSDL ドキュメントのリストを要求するメッセージを受信すると自動的に応答します。WebDispatch プロパティを使用して，クライアントが WSDL ドキュメントのリストにアクセスするために使用する URL のパス情報を指定します。次に Web ブラウザが，サーバーアプリケーションの位置の後に WebDispatch プロパティ内のパスが続く URL を指定して，WSDL ドキュメントのリストを要求できます。この URL は，たとえば次のようになります。

```
http://www.myco.com/MyService.dll/WSDL
```

ヒント 物理的な WSDL ファイルを使用したい場合は，Web ブラウザに WSDL ドキュメントを表示し，これを保存すると WSDL ドキュメントファイルを生成できます。

WSDL ドキュメントの公開は，Web サービスを実装するものと同じアプリケーションから行う必要はありません。WSDL ドキュメントを単に公開するアプリケーションを作成するには，実装オブジェクトを実装および登録するコードを省略し，起動可能インターフェース，複合型を表すリモート可能クラス，およびリモート可能例外を定義および登録するコードだけを含めます。

デフォルトでは，WSDL ドキュメントを公開すると，その WSDL ドキュメントを公開したものと同じ URL（ただしパスは異なる）でサービスが利用できます。複数バージョンの Web サービスアプリケーションを配布する場合，または Web サービスを実装したものとは異なるアプリケーションから WSDL ドキュメントを公開する場合は，Web サービスを探し出せるように情報を更新して，WSDL ドキュメントを変更する必要があります。

URL を変更するには，WSDL アドミニストレータを使用します。まず最初に，WSDL アドミニストレータを使用可能にします。それには，TWSDLHTMLPublish コンポーネントの AdminEnabled プロパティを **true** に設定します。すると，ブラウザを使用して WSDL ドキュメントのリストを表示したときに，このドキュメントも管理するボタンが表示されます。WSDL アドミニストレータを使用して，Web サービスアプリケーションを配布した位置（URL）を指定します。

Web サービスのクライアントの記述

C++Builder は，SOAP ベースのバインドを使用する Web サービスを呼び出すためのクライアント側サポートを提供します。この Web サービスは，C++Builder で記述されたサーバーによって，または WSDL ドキュメント内で Web サービスを定義するその他のサーバーによって供給されます。

WSDL ドキュメントのインポート

Web サービスを使用できるようにするには，まず，Web サービスアプリケーションに含まれる起動可能インターフェースと型をアプリケーションで定義および登録しなければなりません。この定義を取得するには，このサービスを定義する WSDL ドキュメント（または XML ファイル）をインポートします。Web サービスインポートは，使用する必要があるインターフェースと型を定義および登録するユニットを作成します。Web サービスインポートの使い方については，36-13 ページの「Web サービスインポートの使用」を参照してください。

起動可能インターフェースの呼び出し

起動可能インターフェースを呼び出すには，起動可能インターフェース，および複合型を実装するリモート可能クラスを定義するヘッダーがクライアントアプリケーションに含まれなければなりません。

クライアントアプリケーションで起動可能インターフェースを宣言したら，その希望するインターフェースの THTTPRio のインスタンスを作成します。

```
X = new THTTPRio(NULL);
```

メモ 重要なことは，THTTPRio インスタンスを明示的に破棄しないということです。（上のコード行のように）Owner を使わずに作成された場合は，インターフェースが解放されるときに自動的に解放されます。Owner を使って作成された場合は，THTTPRio インスタンスの解放は Owner が行います。

次に，この THTTPRio オブジェクトがサーバーインターフェースを識別してサーバーを探し出すために必要とする情報を，このオブジェクトに提供します。この情報を提供する方法は以下の 2 つです。

- Web サービスの URL，または変更が必要な名前空間と SOAP Action ヘッダーが予想されない場合は，単にアクセスしたい Web サービスの URL を指定できます。THTTPRio は，この URL を使用してインターフェースの定義を参照し，起動レジストリ内の情報に基づいて名前空間とヘッダー情報を参照します。URL を指定するには，URL プロパティをサーバーの位置に設定します。

  ```
  X->URL = "http://www.myco.com/MyService.dll/SOAP/IServerInterface";
  ```

- 実行時に動的に WSDL ドキュメントから URL，名前空間，または SOAP Action ヘッダーを参照したい場合は，WSDLLocation，Service，および Port プロパティを使用して，WSDL ドキュメントから必要な情報を取り出すことができます。

  ```
  X.WSDLLocation = "Cryptography.wsdl";
  X.Service = "Cryptography";
  X.Port = "SoapEncodeDecode";
  ```

サーバーを探し出してインターフェースを識別する方法を指定したら，QueryInterface メソッドを使用して起動可能インターフェースのインターフェースポインタを取得できます。これを行うと，関連付けられたインターフェースの vtable がメモリ内に動的に作成され，インターフェース呼び出しができるようになります。

```
_di_IEncodeDecode InterfaceVariable;
X->QueryInterface(InterfaceVariable);
if (InterfaceVariable)
{
  Code = InterfaceVariable->EncodeValue(5);
}
```

QueryInterface の呼び出しは，起動可能インターフェースそのものではなく起動可能インターフェースの DelphiInterface ラッパーを引数としてとります。

THTTPRio は，起動レジストリに依存して起動可能インターフェースについての情報を取得します。クライアントアプリケーションが起動レジストリを持たない場合，または起動可能インターフェースが登録されない場合，THTTPRio はメモリ内の vtable を構築できません。

注意 THTTPRio から取得したインターフェースをグローバル変数に割り当てる場合，アプリケーションをシャットダウンする前にその割り当てを NULL に変更しなければなりません。たとえば，上のコード例の InterfaceVariable がスタック変数ではなくグローバル変数である場合，THTTPRio オブジェクトが解放される前にインターフェースを解放しなければなりません。通常，このコードはフォームまたはデータモジュールの OnDestroy イベントハンドラに置かれます。

```
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
  InterfaceVariable = NULL;
}
```

グローバルインターフェース変数を NULL に割り当て変更しなければならない理由は，THTTPRio がメモリ内で動的に vtable を構築するからです。その vtable は，インターフェースが解放されるときにも存在していなければなりません。このインターフェースをフォームまたはデータモジュールとともに解放しない場合，シャットダウン時にグローバル変数が解放されるときに解放されます。グローバル変数のメモリは，THTTPRio オブジェクトを含むフォームまたはデータモジュールの後で解放される可能性があり，その場合，インターフェースが解放されたときに vtable は使用できなくなります。

第37章

ソケットの操作

この章では，ソケットコンポーネントについて説明します。ソケットコンポーネントを使うと，TCP/IP および関連するプロトコルを使ってほかのシステムと通信するアプリケーションを作成できます。ソケットを使うと，使用しているネットワークソフトウェアの詳細を意識せずにほかのマシンと接続して読み書きができます。ソケットは TCP/IP プロトコルに基づいた接続能力を提供しますが，User Datagram Protocol（UDP），Xerox Network System（XNS），DEC の DECnet プロトコル，Novell の IPX/SPX ファミリーなどの関連プロトコルとともに動作できる一般性も備えています。

ソケットを使用すると，ほかのシステムとの間で読み書きを行うネットワークサーバーやクライアントアプリケーションを記述できます。サーバーやクライアントアプリケーションは，ハイパーテキスト転送プロトコル（HTTP）やファイル転送プロトコル（FTP）などの単一のサービス専用にするのが普通です。サーバーソケットは，こういったサービスのいずれかを提供するアプリケーションが，そのサービスを利用するクライアントアプリケーションとリンクできるようにします。クライアントソケットは，いずれかのサービスを利用するアプリケーションが，そのサービスを提供しているサーバーアプリケーションとリンクできるようにします。

サービスの実装

ソケットは，ネットワークサーバーまたはクライアントアプリケーションを記述するために必要な要素の一部を提供します。HTTP や FTP など，多くのサービスについてはサードパーティのサーバーを簡単に利用できます。一部のサービスはオペレーティングシステムにバンドルされていて，開発者が記述する必要はありません。ただし，サービスの実装形式を細かく制御したい場合や，アプリケーションとネットワーク通信をより緊密に統合したい場合，または必要な特定のサービスを提供するサーバーがない場合は，独自のサーバーまたはクライアントアプリケーションを作成することになります。たとえば，分散データセットを使って作業する場合，ほかのシステム上のデータベースと通信するための層を記述します。

サービスプロトコルについて

ネットワークサーバーまたはクライアントを記述する前に，アプリケーション自身が提供または利用するサービスを理解しておかなければなりません。サービスの多くで標準プロトコルが使われますが，作成するネットワークアプリケーションではこれらのプロトコルをサポートしなければなりません。HTTP や FTP，さらには finger や time まで含め，こういった標準的なサービスのネットワークアプリケーションを記述する場合，ほかのシステムとの通信に使われる各プロトコルをまず理解する必要があります。ネットワークアプリケーションの記述を始める前に，提供または利用しようとしている特定のサービスについてのマニュアルを参照してください。

ほかのシステムと通信するアプリケーションに新しいサービスを提供する場合，そのサービスのサーバーとクライアント用の通信プロトコルを設計することから始めます。どのようなメッセージを送信するのか，それらのメッセージをどのように調整するのか，情報はどのようにコード化するのか，といった点を考慮します。

アプリケーションと通信する

ネットワークソフトウェアと，サービスを利用するアプリケーションとの間の層をネットワークサーバーまたはクライアントアプリケーションが提供することがよくあります。たとえば，HTTP サーバーは，コンテンツを提供したり HTTP リクエストメッセージに応答する Web サーバーアプリケーションと，インターネットとの間に位置します。

ソケットは，ネットワークサーバーまたはクライアントアプリケーションとネットワークソフトウェアの間のインターフェースを提供します。開発者は，作成するアプリケーションとそのアプリケーションを使うクライアントとの間のインターフェースを用意しなければなりません。サードパーティ製の標準的なサーバーの API（Apache など）をコピーしたり，独自の API を設計して公開したりできます。

サービスとポート

ほとんどの標準的なサービスは，特定のポート番号に関連付けられることになっています。ポート番号についての詳細は後述します。当面は，ポート番号はサービスのコード番号であると考えてください。

クロスプラットフォームアプリケーションで使用する標準的なサービスを実装する場合，Linux ソケットオブジェクトにはサービスのポート番号を調べられるメソッドがあります。新しいサービスを提供する場合は，関連付けたポート番号を /etc/services ファイルで指定できます。services ファイルについての詳細は，使用している Linux のドキュメントを参照してください。

ソケット接続の種類

ソケット接続は，クライアント接続，リスニング（監視）接続，サーバー接続という基本的な 3 つの種類に分けられます。

- クライアント接続

- リスニング接続
- サーバー接続

いったんクライアントソケットへの接続が完了すると，サーバー接続はクライアント接続と区別できなくなります。両方のエンドポイントが同じ機能を持ち，同じ種類のイベントを受信します。リスニング接続だけは根本的に異なり，単一のエンドポイントしか持ちません。

クライアント接続

クライアント接続は，ローカルシステム上のクライアントソケットをリモートシステム上のサーバーソケットに接続します。クライアント接続はクライアントソケットが開始します。まず，クライアントソケットには接続先のサーバーソケットを記述しておかなければなりません。次に，クライアントソケットはそのサーバーソケットを探し，サーバーが見つかると接続を要求します。サーバーソケットは接続をすぐには完了しない場合があります。サーバーソケットはクライアントからのリクエストをキューで順番に管理し，処理可能な時間が空いたら接続作業を行い，完了します。サーバーソケットがクライアント接続を受け入れると，接続先のサーバーソケットに関する情報がサーバーソケットからクライアントソケットに送信され，クライアントソケットが接続状態になります。

リスニング接続

サーバーソケットはクライアントを探しません。かわりに，クライアントのリクエストを待つ受動的な「半接続」を形成します。サーバーソケットはリスニング接続のためのキューを用意し，クライアントからの接続を待ちます。接続リクエストを受信するとそのリクエストをキューに記録します。サーバーソケットがクライアントからの接続リクエストを受け入れると，新しいソケットを作成してリクエスト元クライアントに接続します。これにより，速やかにほかのクライアントのリクエストを受け入れられるようにリスニング接続を開いておくことができます。

サーバー接続

リスニングソケットによってクライアントのリクエストを受け入れると，サーバーソケットはサーバー接続を作成します。サーバーソケットは接続を確立させるために，サーバーソケットの情報をクライアントに送信します。クライアントソケットがサーバーの情報を受信して接続状態を確立させると，相互接続が確立します。

ソケットの記述

ソケットは，ネットワークアプリケーションがネットワーク上でほかのシステムと通信するための手段を提供します。各ソケットをネットワーク接続のエンドポイントとみなすことができます。このため，ソケットは以下の情報を持っています。

- アプリケーションを実行するシステム
- アプリケーションが認識するインターフェース
- アプリケーションが接続のために使用するポート

ソケット接続を完全に記述するには，接続の両端にある各ソケットのアドレスを指定しなければなりません。各ソケットのエンドポイントのアドレスを記述するには，IP アドレスまたはホスト名，およびポート番号の両方を指定します。

ソケット接続を作成する前に，ソケット接続のエンドポイントを形成するソケットを記述しておかなければなりません。情報の一部は，アプリケーションを実行するシステムから取得できます。たとえば，クライアントソケットのローカル IP アドレスを記述する必要はありません。この情報はオペレーティングシステムから取得できます。

提供しなければならない情報は，操作するソケットの種類に応じて異なります。クライアントソケットは接続したいサーバーを記述しなければなりません。また，リスニングサーバーソケットは，提供するサービスを表すポートを記述しなければなりません。

ホストの記述

ホストは，ソケットを持つアプリケーションを実行しているシステムを表します。IP アドレスを指定することでソケットのホストを記述できます。IP アドレスは，インターネットのドット付き標準表記法で表した 4 つの数値（バイト）からなる次のような文字列です。

```
123.197.1.2
```

1 台のシステムが複数の IP アドレスを使用する場合もあります。

IP アドレスは覚えにくく，書き間違いやすいものです。かわりに使えるのがホスト名です。ホスト名は IP アドレスのエリアスで，URL によく見られます。ドメイン名とサービスを含む次のような文字列です。

```
http://www.ASite.com
```

ほとんどのイントラネットには，そのネット上の各システムの IP アドレスを表すホスト名が用意されています。（存在する）IP アドレスに関連付けられたホスト名は，コマンドプロンプトから次のコマンドを実行すると調べることができます。

```
nslookup IPADDRESS
```

ここで *IPADDRESS* は，調べようとしている IP アドレスです。ホスト名を持たないローカル IP アドレスにホスト名を関連付けたい場合は，ネットワーク管理者に問い合わせてください。コンピュータが localhost という名前および IP 番号 127.0.0.1 を使用して自分自身を参照することは一般的です。

サーバーソケットはホスト名を指定する必要はありません。ローカル IP アドレスはシステムから読み出せます。ローカルシステムが複数の IP アドレスをサポートする場合，サーバーソケットは，すべての IP アドレスに対するクライアントのリクエストを同時に監視します。サーバーソケットが接続リクエストを受け入れると，クライアントソケットがリモート IP アドレスを提供します。

クライアントソケットは，リモートホストのホスト名か IP アドレスのどちらかを提供してリモートホストを指定しなければなりません。

ホスト名と IP アドレスのどちらかを選択する

ほとんどのアプリケーションでは，ホスト名を使ってシステムを指定します。ホスト名の方が覚えやすく，誤記を見つけやすくなります。さらに，サーバーによっては，特定のホスト名に関連付けられたシステム（IP アドレス）が変更されることがあります。ホスト名を使うと，クライアントソケッ

トは，ホスト名が表す抽象的なサイト名が新しい IP アドレスに移行した場合でもそのサイトを探し出せます。

ホスト名がわからない場合，クライアントソケットは IP アドレスを使ってサーバーシステムを指定しなければなりません。IP アドレスを指定してサーバーシステムを指定した方が速やかにサーバーシステムを探し出せます。ホスト名を指定した場合，ソケットは，そのホスト名に関連付けられた IP アドレスを検索しないとサーバーシステムを探し出せません。

ポートの使い方

ソケット接続の相手側のシステムを見つけるのに十分な情報が IP アドレスで提供されますが，そのシステムのポート番号も必要です。ポート番号がないと，システムは一度に 1 つの接続しか形成できません。ポート番号はユニークな識別子であり，接続ごとに別々のポート番号を与えることによって，1 つのシステムが同時に複数の接続に対応できるようになります。

前の説明で，ポート番号はネットワークアプリケーションが実装するサービスの数字コードであるとしました。実際は，クライアントソケットがリスニングサーバー接続を見つけ出せるように，リスニングサーバー接続が自らを固定のポート番号で利用可能にするための取り決めにすぎません。サーバーソケットは，リスニング接続を形成して提供しているサービスに関連付けられたポート番号で接続リクエストを待ちます。サーバーソケットがクライアントソケットとの接続を受け入れると，異なる任意のポート番号を使う別のソケット接続がそのサーバーソケットによって作成されます。これにより，サービスに関連付けられたポート番号ではリスニング接続が継続して接続リクエストを監視することができます。

クライアントソケットは任意のローカルなポート番号を使います。ほかのソケットがクライアントソケットを見つけ出す必要はないからです。クライアントソケットは，サーバーアプリケーションを見つけられるように，接続先のサーバーソケットのポート番号を指定します。目的のサービス名を指定することで間接的にポート番号を指定する場合がよくあります。

ソケットコンポーネントの使い方

［Internet］パレットページには 3 つのソケットコンポーネントがあり，これらのコンポーネントを使うと，ネットワークアプリケーションがほかのマシンと接続でき，その接続を通じて情報を読み書きできます。この 3 つのソケットコンポーネントは，以下のとおりです。

- TcpServer
- TcpClient
- UdpSocket

これらのソケットコンポーネントにはそれぞれソケットオブジェクトが関連付けられています。ソケットオブジェクトは実際のソケット接続のエンドポイントを表します。ソケットコンポーネントは，ソケットオブジェクトを使ってソケットサーバー呼び出しをカプセル化します。そのため，接続の確立やソケットメッセージの管理についての詳細をアプリケーションで考慮する必要はありません。

開発者のためにソケットコンポーネントが作成する接続の詳細をカスタマイズしたい場合は，ソケットオブジェクトのプロパティ，イベント，およびメソッドを使います。

接続についての情報を取得する

いったんクライアントまたはサーバーのソケットとの接続が完了したら，ソケットコンポーネントに関連付けられたクライアントまたはサーバーのソケットオブジェクトを使って，接続に関する情報を取得できます。LocalHost プロパティと LocalPort プロパティを使って，ローカルのクライアントまたはサーバーのソケットが使ったアドレスとポート番号を確認するか，RemoteHost プロパティと RemotePort プロパティを使って，リモートのクライアントまたはサーバーのソケットが使ったアドレスとポート番号を確認できます。GetSocketAddr メソッドを使用すると，ホスト名とポート番号に基づいた有効なソケットアドレスを作成できます。LookupPort メソッドを使用すると，ポート番号を参照できます。LookupProtocol メソッドを使用すると，プロトコル番号を参照できます。LookupHostName メソッドを使用すると，ホストマシンの IP アドレスに基づいたホスト名を参照できます。

ソケットを出入りするネットワークトラフィックを表示するには，BytesSent プロパティおよび BytesReceived プロパティを使用します。

クライアントソケットの使い方

アプリケーションを TCP/IP または UDP クライアントに変えるには，フォームまたはデータモジュールに，TcpClient コンポーネントまたは UdpSocket コンポーネントを追加します。クライアントソケットでは，接続したいサーバーソケットとそのサーバーに提供させたいサービスを指定できます。必要な接続を記述しておくと，クライアントソケットコンポーネントを使ってサーバーとの接続を確立できます。

各クライアントソケットコンポーネントは，接続のクライアントエンドポイントを表す単一のクライアントソケットオブジェクトを使います。

目的のサーバーを指定する

クライアントソケットコンポーネントには，接続したいサーバーシステムとポートを指定するためのいくつかのプロパティがあります。RemoteHost プロパティは，リモートホストサーバーのホスト名または IP アドレスによってそのサーバーを指定します。

サーバーシステムに加えて，クライアントソケットが接続するサーバーシステム上のポートを指定しなければなりません。RemotePort プロパティを使うと，サーバーのポート番号を直接指定するか，ターゲットサービスによって間接的にポート番号を指定することができます。

接続する

クライアントソケットコンポーネントのプロパティを設定して，接続したいサーバーを記述してあれば，Open メソッドを呼び出すことで実行時に接続できます。アプリケーションの起動時に自動的に接続したい場合は，オブジェクトインスペクタを使って設計時に Active プロパティを **true** に設定します。

接続についての情報を取得する

いったんサーバーソケットとの接続が完了したら，クライアントソケットコンポーネントに関連付けられたクライアントソケットオブジェクトを使って，接続に関する情報を取得できます。LocalHost プロパティと LocalPort プロパティを使って，クライアントソケットとサーバーソケットが接続のエンドポイントを形成するのに使ったアドレスとポート番号を確認できます。Handle プロパティを使うと，ソケット呼び出しを行うときに使うソケット接続のハンドルを取得できます。

接続を閉じる

ソケット接続を通じたサーバーアプリケーションとの通信を終了したら，Close メソッドを呼び出して接続を閉じることができます。接続がサーバー側から閉じられる場合もあります。その場合，OnDisconnect イベントで通知されます。

サーバーソケットの使い方

アプリケーションを IP サーバーに変えるには，フォームまたはデータモジュールに，サーバーソケットコンポーネント（TcpServer または UdpSocket）を追加します。サーバーソケットでは，提供するサービスを指定したり，クライアントのリクエストを監視するのに使うポートを指定できます。クライアントの接続リクエストを待ち，受け入れるのにサーバーソケットコンポーネントを使えます。

各サーバーソケットコンポーネントは，リスニング接続のサーバーエンドポイントを表す単一のサーバーソケットオブジェクトを使います。サーバーが受け入れるクライアントソケットとのアクティブな接続ごとのサーバーエンドポイントを表すサーバークライアントソケットオブジェクトも使います。

ポートを指定する

サーバーソケットがクライアントのリクエストを監視するには，サーバーが接続リクエストを監視するポートを指定しておかなければなりません。このポートは LocalPort プロパティで指定できます。特定のポート番号に関連付けられることになっている標準的なサービスをサーバーアプリケーションで提供する場合は，LocalPort プロパティを使ってサービス名を指定することもできます。ポート番号を指定するときには入力ミスが起こりがちなので，ポート番号の代わりにサービス名を使うことをお勧めします。

クライアントのリクエストを監視する

サーバーソケットコンポーネントのポート番号またはサービス名を設定したら，Open メソッドを呼び出すことで実行時にリスニング接続を形成できます。アプリケーションの起動時に自動的にリスニング接続を形成したい場合は，設計時にオブジェクトインスペクタを使って Active プロパティを **true** に設定します。

クライアントへの接続

リスニング接続が開始されているサーバーソケットコンポーネントは，クライアントの接続リクエストを受信すると，そのリクエストを自動的に受け入れます。これが行われるたびに，OnAccept イベントで通知されます。

サーバー接続を閉じる

リスニング接続をシャットダウンするときには，Close メソッドを呼び出すか，Active プロパティを **false** に設定します。Close メソッドを呼び出すと，クライアントアプリケーションへの，開いているすべての接続が閉じ，受け入れられなかった保留中の接続がすべて解除されます。その後，サーバーソケットコンポーネントが新しい接続を受け入れないようにリスニング接続が閉じます。

TCP クライアントがサーバーソケットとの個々の接続を閉じると，OnDisconnect イベントで通知されます。

ソケットイベントに対するレスポンス

ソケットを使用するアプリケーションを作成する場合，ソケットの書き込みと読み出しはプログラムのどこにでも記述できます。ソケットを開いた後は，プログラムで SendBuf，SendStream，または Sendln メソッドを使ってソケットに書き込むことができます。これと似た名前の ReceiveBuf および Receiveln メソッドを使うと，ソケットから読み出すことができます。ソケットに対して何らかの書き込みまたは読み出しが行われるたびに，OnSend イベントおよび OnReceive イベントが起動されます。この 2 つのイベントはフィルタリングに使用できます。読み出しまたは書き込みを行うと，必ず読み出しイベントまたは書き込みイベントが起動されます。

クライアントソケットとサーバーソケットは両方とも，接続からエラーメッセージを受信するとエラーイベントを生成します。

ソケットコンポーネントも，接続を開いてから完了するまでの間に 2 つのイベントを受信します。ソケットを開く方法にアプリケーション側で関与する必要がある場合は，SendBuf メソッドと ReceiveBuf メソッドを使って，これらのクライアントイベントまたはサーバーイベントに応答しなければなりません。

エラーイベント

クライアントソケットとサーバーソケットは，接続からエラーメッセージを受信すると OnError イベントを生成します。これらのエラーメッセージに応答するには OnError イベントハンドラを記述します。イベントハンドラは，以下に関する情報を送ります。

- どのソケットオブジェクトがエラー通知を受信したか
- エラーが発生したときに，ソケットが何をしようとしていたか
- エラーメッセージによって提供されたエラーコード

エラーに対してはイベントハンドラで応答でき，ソケットが例外を生成しないようにエラーコードを 0 に変更できます。

クライアントイベント

クライアントソケットが接続を開くと，次のイベントが発生します。

- ソケットがセットアップされ，イベント通知のために初期化されます。

- サーバーとサーバーソケットが作成されると，OnCreateHandle イベントが発生します。この時点で，Handle プロパティを通じて利用できるソケットオブジェクトは，接続の相手側を形成するサーバーソケットまたはクライアントソケットについての情報を提供できます。これが接続に使われた実際のポートを取得できる最初の機会です。このポートは，接続を受け入れたリスニングソケットのポートとは異なる場合があります。
- サーバーが接続リクエストを受け入れ，クライアントソケットが接続リクエストを完了します。
- 接続が確立すると，OnConnect 通知イベントが発生します。

サーバーイベント

サーバーソケットコンポーネントは，リスニング接続とクライアントアプリケーションへの接続の 2 種類の接続を作成します。両方の状態に関連するイベントが，サーバーソケットに送られます。

リスニング接続でのイベント

リスニング接続が形成される直前に，OnListening イベントが発生します。Handle プロパティを使うと，ソケットを開く前にそのソケットの値を変更できるので，接続リクエストを監視することができます。たとえば，サーバーとの接続リクエストを出すクライアントの IP アドレスに制限を加えたい場合には，OnListening イベントハンドラの中で制限します。

クライアント接続でのイベント

サーバーソケットがクライアントの接続リクエストを受け入れると，次のイベントが発生します。

- OnAccept イベントが発生し，新しい TTcpClient オブジェクトをイベントハンドラに渡します。ここで初めて，TTcpClient のプロパティを使ってクライアントへの接続のサーバーエンドポイントに関する情報を取得できます。
- BlockMode が bmThreadBlocking の場合は，OnGetThread イベントが発生します。TServerSocketThread の下位オブジェクトをカスタマイズして提供したい場合，OnGetThread イベントハンドラの中でこの下位オブジェクトを作成でき，その下位オブジェクトが TServerSocketThread のかわりに使われます。スレッドの初期化を実行したい場合，またはスレッドが接続を通じて読み書きを開始する前にソケットの API 呼び出しを行いたい場合，これらの処理にも OnGetThread イベントハンドラを使う必要があります。
- クライアントが接続を完了し，OnAccept イベントが発生します。非ブロッキングサーバーを含めて，ソケット接続を通じた読み書きは，この時点で開始するとよいでしょう。

ソケット接続を通じての読み書き

ほかのマシンとのソケット接続を形成するのは，それらの接続を通じて情報の読み書きができるようにするためです。何の情報をいつ読み書きするかは，ソケット接続に関連付けられたサービスによって異なります。

ソケットを通じての読み書きは非同期に行われるため，ネットワークアプリケーション内のほかのコードの実行は妨げられません。これを非ブロッキング接続といいます。ブロッキング接続を形成す

ることもできます。ブロッキング接続では，アプリケーションは，読み書きが完了するまで待ってからコードの次の行を実行します。

非ブロッキング接続

非ブロッキング接続は非同期の読み書きを行います。そのため，アプリケーションの実行はブロックされません。クライアントソケットまたはサーバーソケットの非ブロッキング接続を作成するには，BlockMode プロパティを bmNonBlocking に設定します。

非ブロッキング接続の場合，相手側のソケットがデータを読み書きすると，自分のソケットに読み書きのイベントが届きます。

読み書きのイベント

非ブロッキング接続で使われるソケットは，接続相手と読み書きする必要が発生した場合に，読み書きイベントを生成します。この通知に対して，OnReceive または OnSend の各イベントハンドラで応答します。

読み書きイベントハンドラのパラメータとして，ソケット接続に関連付けられたソケットオブジェクトが渡されます。このソケットオブジェクトは，接続相手とデータを読み書きするための多くのメソッドを提供しています。

ソケット接続先からデータを読み出すには，ReceiveBuf メソッドまたは Receiveln メソッドを使います。ソケット接続先に書き込むには，SendBuf，SendStream，または Sendln メソッドを使います。

ブロッキング接続

ブロッキング接続の場合には，ソケットが接続を通じて読み書きを開始しなければなりません。ソケット接続からの通知を受動的に待つことはできません。いつ読み書きするかを接続の自分側で管理しなければならない場合，ブロッキング接続のためのソケットを使います。

クライアントソケットまたはサーバーソケットの場合，ブロッキング接続を作成するには，BlockMode プロパティを bmBlocking に設定します。クライアントアプリケーションがほかに何をするかに応じて，読み書き用に新しい実行スレッドを作成するとよいでしょう。こうするとアプリケーションは，接続先との読み書きが完了するのを待っている間に，ほかのスレッド上でコードの実行を続けられます。

サーバーソケットの場合，ブロッキング接続を作成するには，BlockMode プロパティを bmBlocking または bmThreadBlocking に設定します。ブロッキング接続では，接続を通じて情報が読み書きされるのをソケットが待っている間はほかのすべてのコードの実行が停止されるので，BlockMode が bmThreadBlocking の場合，サーバーソケットコンポーネントは必ずそれぞれのクライアント接続用に新しい実行スレッドを生成します。BlockMode が bmBlocking の場合，プログラムの実行は新しい接続が確立されるまでブロックされます。

第 IV 部

COM ベースアプリケーションの開発

第 IV 部「COM ベースアプリケーションの開発」の各章では，COM ベースアプリケーションの開発に必要な概念とテクニックについて説明します。このようなアプリケーションには，オートメーションコントローラ，オートメーションサーバー，ActiveX コントロール，および COM+ アプリケーションがあります。

メモ COM クライアントのサポートは C++Builder のすべての版で利用できますが，サーバーの作成には Professional 版または Enterprise 版が必要です。

第38章

COM テクノロジーの概要

C++Builder は，Microsoft のコンポーネントオブジェクトモデル（COM: Component Object Model）に基づいたアプリケーションの実装を容易にするためのウィザードとクラスを提供しています。これらのウィザードを使えば，アプリケーション内で使用する COM ベースのクラスおよびコンポーネントを作成したり，COM オブジェクトを実装した完全に機能する COM クライアントまたはサーバー，オートメーションサーバー（Active Server オブジェクトなど），ActiveX コントロール，または ActiveForm を作成することができます。

メモ COM コンポーネント（コンポーネントパレットの［ActiveX］ページ，［COM+］ページ，［Servers］ページなどにあるコンポーネント）は，CLX アプリケーションには使用できません。COM 技術は Windows で使用するためのものであり，クロスプラットフォーム用ではありません。

COM は，Windows プラットフォーム上で動作するソフトウェアコンポーネントとアプリケーションの間の対話を可能にするための，言語に依存しないソフトウェアコンポーネントモデルです。COM の最も重要な機能は，明確に定義されたインターフェースを通じて，コンポーネント間，アプリケーション間，およびクライアントとサーバー間の通信を可能にすることです。インターフェースは，クライアントが COM コンポーネントに対して実行時にどの機能をサポートするかを問い合わせるための手段を提供します。コンポーネントに機能を追加するには，追加機能用のインターフェースを追加するだけで済みます。

アプリケーションは，同じコンピュータ上にある COM コンポーネントのインターフェースにアクセスするか，分散 COM（DCOM）と呼ばれるメカニズムを使ってネットワーク上の別のコンピュータ上にある COM コンポーネントのインターフェースにアクセスすることができます。クライアント，サーバー，およびインターフェースについての詳細は，第 38 章-2 の「COM アプリケーションの各部分」を参照してください。

この章では，オートメーションと ActiveX コントロールの土台となっている技術の基本概念について説明します。この後の各章では，C++Builder でオートメーションオブジェクトと ActiveX コントロールを作成する方法を詳しく説明します。

仕様としての COM と実装としての COM

COM は，仕様であると同時に実装されたものでもあります。COM の仕様は，オブジェクトの作成とオブジェクト相互の通信がどのように行われるかを定義しています。この仕様に従って，COM オブジェクトをさまざまな言語で書いて，さまざまなプロセス空間およびプラットフォームで実行することができます。書かれた仕様に従ってさえいれば，オブジェクトは互いに通信することができます。したがって，古いコードを 1 つのコンポーネントとして，オブジェクト指向言語で書かれた新しいコンポーネントと統合することができます。

COM の実装は Win32 サブシステムに組み込まれています。この Win32 サブシステムは，COM 仕様をサポートするための多数のコアサービスを提供します。COM ライブラリには，COM オブジェクトの基本機能を定義する標準インターフェースセットや，COM オブジェクトの作成と管理を目的としたいくつかの API 関数が入っています。

アプリケーションの中で C++Builder のウィザードと VCL オブジェクトを使うということは，COM 仕様の C++Builder の実装を使うということです。さらに，直接実装されていない機能（ActiveX ドキュメントなど）に代わる COM サービスのためのラッパーも提供されています。

メモ C++Builder は，Microsoft Active Template Library（ATL）を使用し，クラスとマクロで変更を加えた，COM 仕様に準拠するオブジェクトを実装します。

COM 拡張機能

COM は，進化するにつれて基本的な COM サービスをはるかに超えて拡張されました。COM は，オートメーション，ActiveX コントロール，ActiveX ドキュメント，Active ディレクトリといったほかの技術の土台として機能します。COM 拡張機能についての詳細は，38-10 ページの「COM 拡張機能」を参照してください。

さらに，大規模な分散環境において作業する場合に，トランザクション COM オブジェクトを作成することができます。Windows 2000 以前は，トランザクションオブジェクトはアーキテクチャ的には COM の一部ではなく，MTS（Microsoft Transaction Server）環境の中で動作していました。Windows 2000 の登場により，このサポートが COM+ に統合されました。トランザクションオブジェクトについての詳細は，第 44 章「MTS オブジェクトまたは COM+ オブジェクトの作成」に記載されています。

C++Builder は，上記の技術を C++Builder 環境に統合するアプリケーションを簡単に作成するためのウィザードを提供しています。詳細については，38-19 ページの「ウィザードを使って COM オブジェクトを実装する」を参照してください。

COM アプリケーションの各部分

COM アプリケーションを実装するには，以下のものが必要です。

COM インターフェース オブジェクトが自身のサービスをクライアントに対して外部に公開するための手段。COM オブジェクトは，関連するメソッドとプロパティのセット 1 つにつきインターフェースを 1 つ提供する。COM プロパティは VCL オブジェクト上のプロパティと同一ではない。COM プロパティは常にアクセスメソッド **read** および **write** を使用し，**__property** キーワードを使用して宣言されるのではない

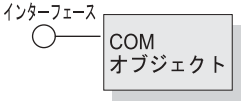
COM サーバー COM オブジェクトのコードが入っているモジュール（EXE，DLL，OCX のいずれか）。オブジェクトの実装はサーバー上にある。1 つの COM オブジェクトが 1 つまたは複数のインターフェースを実装する

COM クライアント 要求されたサービスをサーバーから取得するためにインターフェースを呼び出すコード。クライアントは，自分がインターフェースを介してサーバーから何を取得したいかを知っているが，サーバーがサービスを提供する仕組みの中身は知らない。C++Builder では，COM サーバー（Word 文書や PowerPoint スライドなど）をコンポーネントパレット上のコンポーネントとしてインストールできるので，クライアント作成のプロセスが簡単になっている。これにより，サーバーと接続し，オブジェクトインスペクタを使ってサーバーのイベントをフックすることができる

COM インターフェース

COM クライアントは，COM インターフェースを介してオブジェクトと通信します。インターフェースは，サービスの提供者（サーバーオブジェクト）とそのクライアントとの間の通信を提供する，論理的または意味的に関連したルーチンのグループです。COM インターフェースを図示する標準的な方法を図 38.1 に示します。

図 38.1　COM インターフェース



たとえば，各 COM オブジェクトは，基本的なインターフェース IUnknown を実装しなければなりません。IUnknown 内の QueryInterface というルーチンによって，クライアントは，サーバーが実装するその他のインターフェースを要求できます。

オブジェクトは複数のインターフェースを持つことができ，各インターフェースがそれぞれ 1 つの機能を実装します。インターフェースは，オブジェクトがサービスをどこでどのようにして提供するかという実装の詳細を知らせずに，どのようなサービスを提供するかをクライアントに知らせる手段を提供します。

COM インターフェースの主な特長は以下のとおりです。

- いったん公開されたインターフェースは不変です。インターフェースは，特定の機能のセットを提供することを保証します。追加の機能はインターフェースの追加によって提供されます。
- 規則により，COM インターフェース識別子は大文字の I（アイ）で始まり，その後にインターフェースを定義するシンボル名が続きます（例：IMalloc，IPersist など）。
- インターフェースは，**グローバルユニーク識別子**（GUID: Globally Unique Identifier）と呼ばれるユニークな識別子を持つことが保証されています。GUID はランダムに生成される 128 ビットの数値です。インターフェースの GUID は**インターフェース識別子**（IID: Interface Identifier）と呼ばれます。IID によって，同一製品の異なるバージョン間や異なる製品間での名前の衝突が避けられます。

- インターフェースは言語に依存しません。COM インターフェースの実装には，ポインタの構造をサポートしていて，ポインタを通じて明示的または暗黙に関数を呼び出すことのできる言語であれば，どのような言語でも使えます。
- インターフェース自体はオブジェクトではなく，オブジェクトにアクセスする手段を提供するものです。したがって，クライアントはデータに直接アクセスするわけではなく，インターフェースポインタを介してデータにアクセスします。Windows 2000 にはインターセプタという間接層が追加され，この層を通して即時アクティブ化，オブジェクトプーリングといった COM+ 機能を提供しています。
- インターフェースは，常に基本インターフェース IUnknown から継承されます。
- スレッド間，プロセス間，およびネットワーク化されたマシンの間でインターフェースメソッド呼び出しを行えるようにするために，プロクシーを通じてインターフェースが COM によってリダイレクトされるようにすることができます（クライアントまたはサーバーオブジェクトがこのリダイレクトに気付くことはありません）。詳細については，第 38 章-6 の「インプロセス，アウトオブプロセス，およびリモートサーバー」を参照してください。

基本 COM インターフェース：IUnknown

COM オブジェクトはすべて，IUnknown（基本インターフェース型 IInterface の **typedef**）という基本インターフェースをサポートしなければなりません。IUnknown には以下のルーチンが含まれています。

QueryInterface	そのオブジェクトがサポートしているほかのインターフェースにポインタを提供します。
AddRef と Release	オブジェクトが自分の存続期間を追跡し，そのサービスをクライアントが必要としなくなったときに自分を削除できるようにする，単純な参照カウントのためのメソッドです。

クライアントは，IUnknown メソッド QueryInterface を通じて，ほかのインターフェースへのポインタを取得します。QueryInterface は，サーバーオブジェクト内のすべてのインターフェースを知っているので，要求されたインターフェースへのポインタをクライアントに与えることができます。クライアントは，インターフェースへのポインタを受け取ると，そのインターフェースのどのメソッドでも呼び出せることが保証されます。

オブジェクトは，参照カウントのためのメソッドである IUnknown メソッドの AddRef および Release を通じて，自分の存続期間を追跡します。オブジェクトの参照カウントがゼロ以外である間，そのオブジェクトはメモリ内に留まります。参照カウントがゼロになったら，インターフェースの実装は，その基礎となるオブジェクトを破棄しても問題はありません。

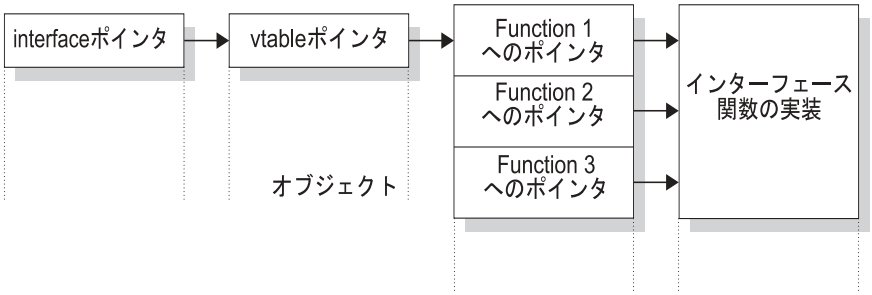
COM インターフェースポインタ

インターフェースポインタは，オブジェクトのインスタンスを指し示し，それによってさらにインターフェース内の各メソッドの実装を指し示すポインタです。実装へのアクセスには，これらのメソッドへのポインタの配列が使われます。この配列は**仮想テーブル**（vtable）と呼ばれます。仮想テーブルは，C++ で仮想関数をサポートするために使用されているメカニズムと類似しています。

この類似点があるので，コンパイラは，C++ クラス上のメソッドへの呼び出しの解決と同じように，インターフェース上のメソッドへの呼び出しを解決できます。

仮想テーブルは，1 つのオブジェクトクラスのすべてのインスタンスの間で共有されるので，オブジェクトコードは各オブジェクトインスタンスについて，プライベートなデータを入れるための第 2 の構造体を割り当てます。したがって，クライアントのインターフェースポインタは，次の図に示すように，仮想テーブルへの*ポインタ*へのポインタです。

図 38.2　インターフェース仮想テーブル



Windows 2000 以降では，オブジェクトが COM+ で動作しているとき，インターフェースポインタと vtable ポインタの間に追加の間接層（インターセプタ）が提供されます。クライアントに示すインターフェースポインタはインターセプタを指し，インターセプタは vtable を指します。これにより，COM+ は即時アクティブ化などのサービスを提供し，クライアントに認識されずにサーバーを動的に起動 / 停止することができます。これを実現するため，COM+ は，インターセプタが通常の vtable ポインタであるかのように動作させます。

COM サーバー

COM サーバーは，クライアントのアプリケーションまたはライブラリにサービスを提供するアプリケーションまたはライブラリです。COM サーバーは 1 つまたは複数の COM オブジェクトで構成され，COM オブジェクトは一群のプロパティ（データメンバーまたは内容）とメソッド（またはメンバー関数）で構成されます。

クライアントは，COM オブジェクトが自分のサービスをどのようにして実行するかを知りません。オブジェクトの実装はカプセル化されたままです。前述のように，オブジェクトは自分のインターフェースを通じて自分のサービスを提供できるようにします。

クライアントは，COM オブジェクトがどこに存在するかを知る必要もありません。COM は，オブジェクトの場所に関係なく，透過的なアクセスを提供します。

クライアントは，COM オブジェクトからのサービスを要求するときに，クラス識別子（CLSID）を COM に渡します。CLSID は単に COM オブジェクトを識別する GUID です。CLSID はシステムレジストリに登録されており，COM はこの CLSID を使用して，適切なサーバーの実装を探し出します。適切なサーバーが探し出されると，COM はそのコードをメモリに読み込んで，サーバーがそのクライアントのためのオブジェクトインスタンスをインスタンス化するようにさせます。このプロセスは，要求時にオブジェクトのインスタンスを作成する（インターフェースに基づく）クラスファクトリと呼ばれる特殊なオブジェクトを介して，間接的に処理されます。

COM サーバーは，最低でも以下のことを実行する必要があります。

- サーバーモジュールをクラス識別子（CLSID）と関連付けるシステムレジストリにエントリを登録する
- 特定の CLSID の別のオブジェクトを作成するクラスファクトリオブジェクトを実装する
- クラスファクトリを COM に公開する
- クライアントにサービスを提供していないサーバーをメモリから削除するためのアンロード機構を提供する

メモ C++Builder のウィザードは，38-19 ページの「ウィザードを使って COM オブジェクトを実装する」で説明されているように，COM オブジェクトとサーバーの作成を自動化します。

CoClass とクラスファクトリ

COM オブジェクトは，1 つまたは複数の COM インターフェースを実装するクラス **CoClass** のインスタンスです。COM オブジェクトは，自分のインターフェースによって定義されたサービスを提供します。

CoClass は，クラスファクトリと呼ばれる特殊な種類のオブジェクトによってインスタンス化されます。クライアントがオブジェクトのサービスを要求するたびに，クラスファクトリはその特定のクライアントのためのオブジェクトインスタンスを作成します。通常は，別のクライアントが同じオブジェクトのサービスを要求した場合に，クラスファクトリはその 2 番目のクライアントにサービスを提供する別のオブジェクトインスタンスを作成します（クライアントは，それをサポートするために自分自身を登録する実行中の COM オブジェクトにバインドすることもできます）。

別のモジュールから外部的にインスタンス化するために，CoClass にはクラスファクトリとクラス識別子（CLSID: class identifier）がなければなりません。CoClass のこれらのユニークな識別子を使うと，新しいインターフェースがそのクラスに実装されるたびに更新できます。新しいインターフェースは，（DLL を使うときによく問題が起きるように）古いバージョンに影響を与えないでメソッドを変更または追加できます。

C++Builder のウィザードは，クラス識別子の割り当て，およびクラスファクトリの実装とインスタンス化を行います。

インプロセス，アウトオブプロセス，およびリモートサーバー

COM の場合，クライアントはオブジェクトが存在する場所を知る必要がなく，オブジェクトのインターフェースを呼び出すだけです。呼び出しのために必要な手順は COM が実行します。これらの手順は，オブジェクトがクライアントと同じプロセスの中にあるか，クライアントマシン上の別のプロセスの中にあるか，それともネットワーク上の別のマシン上にあるかによって異なります。サーバーには以下の種類があります。

インプロセスサーバー クライアントと同じプロセス空間で実行中のライブラリ（DLL）。たとえば，Internet Explorer または Netscape で表示されている Web ページに埋め込まれた ActiveX コントロールがこれに当たる。つまり，ActiveX コントロールはクライアントマシンにダウンロードされ，Web ブラウザと同じプロセスの中で呼び出される

クライアントは，COM インターフェースへの直接呼び出しを使ってインプロセスサーバーと通信する

アウトオブプロセスサーバー（ローカルサーバー） クライアントと同じマシンの別のプロセス空間で動作している別のアプリケーション（EXE）。たとえば，Word 文書の中に埋め込まれた Excel スプレッドシートがこれに当たる。Word と Excel は同じマシン上で動作している 2 つの別個のアプリケーション

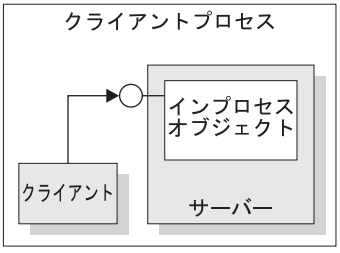
ローカルサーバーは，COM を使ってクライアントと通信する

リモートサーバー クライアントとは別のマシンで動作している DLL または別のアプリケーション。たとえば，ネットワーク内の別のマシン上のアプリケーションサーバーに接続している C++Builder データベースアプリケーションがこれに当たる

リモートサーバーは分散 COM（DCOM）を使ってインターフェースにアクセスし，アプリケーションサーバーと通信する

インプロセスサーバーの場合は，図 38.3 に示したように，オブジェクトインターフェースへのポインタがクライアントと同じプロセス空間にあるので，COM はオブジェクトの実装に対して直接呼び出しを実行します。

図 38.3 インプロセスサーバー

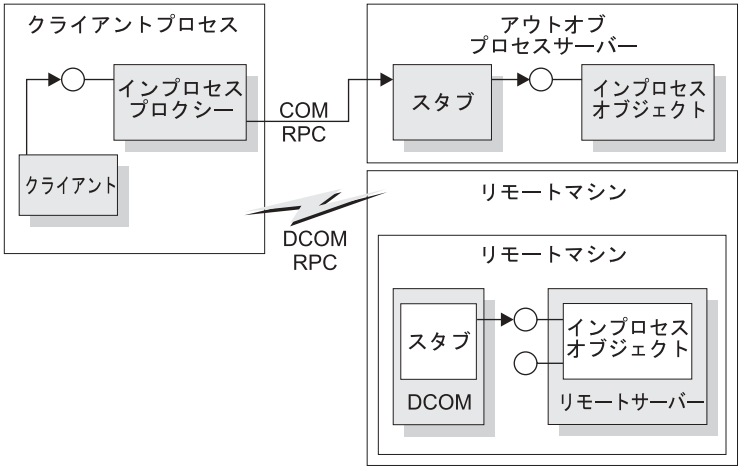


メモ これは COM+ では必ずしも当てはまりません。異なるコンテキストにおけるオブジェクトへの呼び出しをクライアントが行うと，COM+ はこの呼び出しをインターセプトすることによって，サーバーがインプロセスサーバーであっても，アウトオブプロセスサーバー（後述）への呼び出しのように動作するようにします。COM+ の使い方についての詳細は，第 44 章「MTS オブジェクトまたは COM+ オブジェクトの作成」を参照してください。

プロセスが別のプロセスか別のマシンにある場合，COM は図 38.4 に示すように，プロクシーを使ってリモート手続きを呼び出します。**プロクシー**は，クライアントと同じプロセスの中にあるので，クライアントの観点からはすべてのインターフェース呼び出しが同じように見えます。プロクシーは，クライアントの呼び出しをインターセプトして，実際のオブジェクトが動作している場所へ転送します。クライアントが，別のプロセス空間内または別のマシン上にあるオブジェクトに，

それが自分のプロセス内に存在するかのようにアクセスすることを可能にする機構を**マーシャリング**と呼びます。

図 38.4 アウトオブプロセスサーバーとリモートサーバー



アウトオブプロセスサーバーとリモートサーバーの違いは，使用されるプロセス間通信の種類にあります。プロクシーは COM を使ってアウトオブプロセスサーバーと通信し，分散 COM（DCOM）を使ってリモートマシンと通信します。DCOM は，別のマシン上で動作しているリモートオブジェクトに，ローカルオブジェクト要求を透過的に転送します。

メモ リモート手続き呼び出しについては，DCOM はオープングループの分散コンピューティング環境（DCE）が提供している RPC プロトコルを使います。分散セキュリティについては，DCOM は NTLM（NT Lan Manager）セキュリティプロトコルを使います。ディレクトリサービスについては，ドメインネームシステム（DNS）を使います。

マーシャリング機構

マーシャリングは，クライアントが別のプロセスまたは別のマシンにあるリモートオブジェクトに対してインターフェース関数呼び出しを実行できるようにする機構です。マーシャリングは，以下のように機能します。

- サーバーのプロセスからインターフェースポインタを取り，クライアントプロセスのコードがプロクシーポインタを利用できるようにします。
- インターフェース呼び出しの引数をクライアントから渡されたとおりに転送し，その引数をリモートオブジェクトのプロセス空間に入れます。

どのようなインターフェース呼び出しでも，クライアントは引数をスタックにプッシュし，インターフェースポインタを介して関数呼び出しを実行します。オブジェクトへの呼び出しがインプロセスでない場合，その呼び出しはプロクシーに渡されます。プロクシーは引数をマーシャリングパケットにパックし，その構造体をリモートオブジェクトへ送信します。オブジェクトのスタブはパケットをアンパックし，引数をスタックにプッシュして，オブジェクトの実装を呼び出します。基本的に，オブジェクトはクライアントの呼び出しを自分自身のアドレス空間内に再作成します。

発生するマーシャリングの種類は，COM オブジェクトが何を実装するインターフェースによって決まります。オブジェクトは，IDispatch インターフェースが提供する標準マーシャリング機構を使うことができます。これは，システム標準リモート手続き呼び出し（RPC）を介した通信を可能にする，一般的なマーシャリング機構です。IDispatch インターフェースについての詳細は，41-12 ページの「オートメーションインターフェース」を参照してください。オブジェクトが IDispatch を実装しない場合でも，オブジェクトが自分自身をオートメーション互換型に制限し，登録されたタイプライブラリを持つ場合は，COM は自動的にマーシャリングのサポートを提供します。

アプリケーションが自分自身をオートメーション互換型に制限せず，タイプライブラリも登録していない場合は，独自のマーシャリングを提供しなければなりません。マーシャリングを提供するには，IMarshal インターフェースを実装するか，別途生成したプロクシー / スタブ DLL を使います。C++Builder は，プロクシー / スタブ DLL の自動生成をサポートしていません。

集合体

サーバーオブジェクトが別の COM オブジェクトを利用して，その関数のいくつかを実行することがあります。たとえば，ある在庫管理オブジェクトが，1 つの独立したインボイス作成オブジェクトを利用して顧客インボイスを処理しているとします。しかし，この在庫管理オブジェクトがインボイスインターフェースをクライアントに提供する必要がある場合は，問題が生じます。在庫インターフェースを持つクライアントは QueryInterface を呼び出してインボイスインターフェースを取得できますが，インボイスオブジェクトは作成されたときに在庫管理オブジェクトのことを知らないので，QueryInterface の呼び出しに応答して在庫インターフェースを返すことができません。インボイスインターフェースを持つクライアントは，在庫インターフェースに戻ることができません。

この問題を回避するために，一部の COM オブジェクトは**集合体**をサポートしています。在庫管理オブジェクトは，インボイスオブジェクトのインスタンスを作成するときに，自分自身の IUnknown インターフェースのコピーを渡します。すると，インボイスオブジェクトはこの IUnknown インターフェースを使用して，サポートしないインターフェース（たとえば在庫インターフェース）を要求する QueryInterface 呼び出しを処理することができます。このような現象が起きる場合に，2 つのオブジェクトをまとめて 1 つの集合体と呼びます。インボイスオブジェクトは集合体の内部オブジェクトと呼ばれ，在庫オブジェクトは外部オブジェクトと呼ばれます。

メモ 集合体の外部オブジェクトとして機能するには，COM オブジェクトは Windows API CoCreateInstance または CoCreateInstanceEx を使用して内部オブジェクトを作成し，内部オブジェクトが QueryInterface 呼び出しに使用できるパラメータとして自分の IUnknown ポインタを渡さなければなりません。TComInterface オブジェクトをインスタンス化し，その CreateInstance メソッドをこの目的に使うこともできます。

集合体の内部オブジェクトとして機能するオブジェクトは，IUnknown インターフェースの 2 つの実装を提供します。1 つは，外部オブジェクトの制御 IUnknown への QueryInterface 呼び出しを処理できない場合に呼び出しを延期するもの，もう 1 つは，処理できない QueryInterface 呼び出しを受け取ったときにエラーを返すものです。C++Builder のウィザードは，内部オブジェクトとして機能するためにこのサポートを含むオブジェクトを自動的に作成します。

COM クライアント

クライアントは，COM オブジェクトが何を提供できるかを判断するために，そのオブジェクトのインターフェースに常に問い合わせることができます。すべての COM オブジェクトでは，既知のインターフェースをクライアントが要求できます。さらに，サーバーが IDispatch インターフェースをサポートする場合に，クライアントはそのインターフェースがサポートするメソッドについての情報をサーバーに問い合わせることができます。サーバーオブジェクトは，そのオブジェクトを使用するクライアントについて，何も予想しません。同様に，クライアントは，オブジェクトがどのようにして（またはどこで）サービスを提供するかを知る必要がなく，サーバーオブジェクトがインターフェースを通じて公表するサービスを提供することをあてにするだけで済みます。

COM クライアントには，コントローラとコンテナの 2 種類があります。コントローラはサーバーを起動し，そのインターフェースを通じてサーバーと対話します。コントローラは COM オブジェクトにサービスを要求するか，COM オブジェクトを別個のプロセスとして駆動します。コンテナは，そのユーザーインターフェースに表示されるビジュアルコントロールまたはオブジェクトのホストになります。コンテナは，定義済みのインターフェースを使用して，表示の問題をサーバーオブジェクトとネゴシエートします。DCOM を介してコンテナ関係を持つことはできません。たとえば，コンテナのユーザーインターフェースに表示するビジュアルコントロールは，ローカルに配置しなければなりません。これは，コントロールが自分自身をペイントすると想定されるからです。コントロールが自分自身をペイントするには，ローカルの GDI リソースにアクセスできるようにする必要があります。

C++Builder は，COM クライアントの開発を容易にするために，コンポーネントラッパーにタイプライブラリまたは ActiveX コントロールをインポートし，サーバーオブジェクトをほかの VCL コンポーネントと同じように見せることができるようにしています。このプロセスについての詳細は，第 40 章「COM クライアントの作成」を参照してください。

COM 拡張機能

COM は，もともとは基本通信機能を提供し，拡張を通じてその機能を拡大できるようにするために設計されました。COM 自体は，特定の目的に合わせて特殊化したインターフェースのセットを定義することによって，基本機能を拡張してきました。

COM 拡張機能が現在提供しているサービスには，以下のようなものがあります。この後の各セクションでは，これらのサービスについて詳しく説明します。

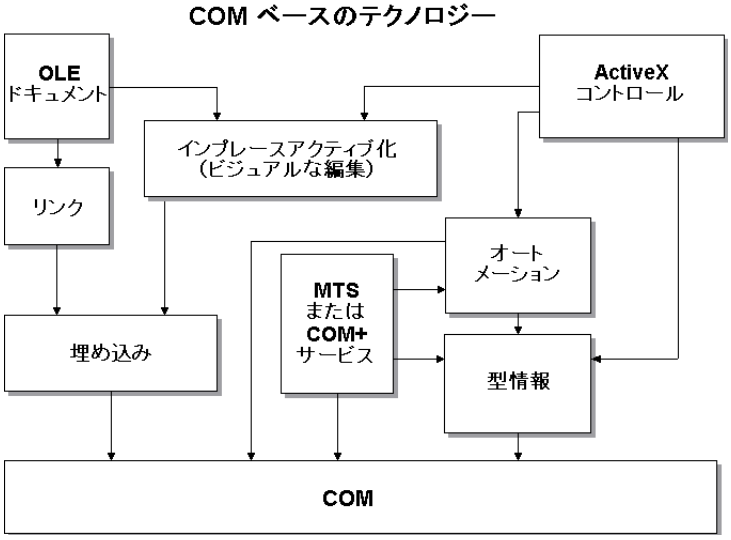
オートメーションサーバー　オートメーションとは，あるアプリケーションが別のアプリケーション内のオブジェクトをプログラム的に制御する機能のことをいう。オートメーションサーバーは，実行時にほかの実行可能ファイルによって制御できるオブジェクト

ActiveX コントロール　ActiveX コントロールは，特殊なインプロセスサーバーであり，一般にクライアントアプリケーションに埋め込まれるよう意図されている。ActiveX コントロールは，設計時と実行時の動作およびイベントを提供する

ASP（Active Server Page）	ASP は，HTML ページを生成するスクリプトである。スクリプト言語には，オートメーションオブジェクトの作成と実行のための構文が含まれる。つまり，ASP はオートメーションコントローラとして機能する
ActiveX ドキュメント	リンクと埋め込み，ドラッグアンドドロップ，ビジュアル編集，およびインプレースアクティベーションをサポートするオブジェクト。Word 文書と Excel スプレッドシートは，ActiveX ドキュメントの例
トランザクションオブジェクト	多数のクライアントに応答するための追加サポートを含むオブジェクト。これには，即時アクティブ化，トランザクション，リソースプーリング，セキュリティサービスなどの機能が含まれる。これらの機能は，元来は MTS によって処理されていたが，COM+ の登場によって COM に組み込まれた
COM+ イベントオブジェクトおよびイベントサブスクリプションオブジェクト	疎結合の COM+ イベントモデルをサポートするオブジェクト。ActiveX コントロールが使用する密結合モデルとは異なり，COM+ イベントモデルでは，イベントサブスクライバから独立してイベントパブリッシャを開発できる
タイプライブラリ	静的データ構造の集まり。多くの場合，オブジェクトとそのインターフェースに関する詳しい型情報を提供するリソースとして保存される。オートメーションサーバー，ActiveX コントロール，およびトランザクションオブジェクトのクライアントは，型情報が使用可能であるものと想定する

次の図に，COM 拡張機能の関係，およびこれらの拡張機能が COM をベースにどのように構築されているかを示します。

図 38.5　COM ベースのテクノロジー



COM オブジェクトは，ビジュアルの場合も非ビジュアルの場合もあります。一部の COM オブジェクトは，クライアントと同じプロセス空間で実行しなければなりません。それ以外の COM オブジェクトは，そのオブジェクトがマーシャリングサポートを提供してさえいれば，別のプロセスまたはリモートマシンで実行することができます。表 38.1 は，作成可能な COM オブジェクトの型，ビジュア

ルかどうかの区別，実行できるプロセス空間，マーシャリングの提供方法，タイプライブラリの必要性を示します。

表 38.1　COM オブジェクトの要件

オブジェクト型	ビジュアルオブジェクト	プロセス空間	通信プロセス	タイプライブラリ
ActiveX ドキュメント	通常はビジュアル	インプロセスまたはアウトオブプロセス	OLE Verb	なし
オートメーションサーバー	場合による	インプロセス，アウトオブプロセス，またはリモート	IDispatch インターフェースを使って自動的にマーシャリングされる（アウトオブプロセスサーバーおよびリモートサーバーの場合）	自動マーシャリングのために必要
ActiveX コントロール	通常はビジュアル	インプロセス	IDispatch インターフェースを使って自動的にマーシャリングされる	必要
MTS または COM+	場合による	MTS はインプロセス，COM+ はどれでも可能	タイプライブラリを介して自動的にマーシャリングされる	必要
インプロセスのカスタムインターフェースオブジェクト	オプション	インプロセス	インプロセスサーバーの場合はマーシャリングは不要	望ましい
その他のカスタムインターフェースオブジェクト	オプション	インプロセス，アウトオブプロセス，またはリモート	タイプライブラリを介して自動的にマーシャリングされるか，カスタムインターフェースを使って手動でマーシャリングされる	望ましい

オートメーションサーバー

オートメーションとは，たとえば一度に複数のアプリケーションを操作できるマクロのように，アプリケーションが別のアプリケーションの中のオブジェクトをプログラム的に制御する能力です。操作対象のサーバーオブジェクトをオートメーションオブジェクトと呼び，オートメーションオブジェクトのクライアントをオートメーションコントローラと呼びます。

オートメーションは，インプロセス，ローカル，リモートのいずれのサーバーでも使えます。

オートメーションは，2 つの重要な点を特徴としています。

- オートメーションオブジェクトは一連のプロパティとコマンドを定義し，型記述によってその機能を記述する。この条件を満足するには，オブジェクトのインターフェース，インターフェースメソッド，メソッドの引数の情報を提供する手段を用意しなければならない。通常はこの情報はタイプライブラリで入手できる。オートメーションサーバーは，IDispatch インターフェース（後述）を介して問い合わせがあったときに型情報を動的に生成することもできる
- オートメーションオブジェクトは，ほかのアプリケーションが自分のメソッドにアクセスして使えるようにする。このために IDispatch インターフェースを実装する。このインターフェースを経由すれば，オブジェクトはそのすべてのメソッドとプロパティをエクスポーズできる。型情報によってオブジェクトのメソッドがわかったら，このインターフェースの一次メソッドを使ってそのメソッドを呼び出せる

オートメーション IDispatch インターフェースはマーシャリングのプロセスを自動化するので，開発者はよくオートメーションを使って，任意のプロセス空間で動作する非ビジュアル OLE オブジェクトを作成および使用します。ただし，オートメーションを使うと，利用できる型が制限されます。

タイプライブラリで一般に有効な型の一覧と，特にオートメーションインターフェースについては，39-10 ページの「有効な型」を参照してください。

オートメーションサーバーの作成についての詳細は，第 41 章「単純な COM サーバーの作成」を参照してください。

ASP（Active Server Page）

ASP（Active Server Page）テクノロジーは，Web サーバーを介してクライアントが起動できる単純なスクリプト，すなわち ASP の作成を可能にします。クライアント上で実行される ActiveX コントロールとは異なり，ASP はサーバー上で実行され，結果の HTML ページをクライアントに返します。

ASP は，Jscript または VB script で作成されます。このスクリプトは，サーバーが Web ページをロードするたびに実行されます。すると，そのスクリプトは，埋め込まれたオートメーションサーバー（または Enterprise Java Bean）を起動できます。たとえば，ビットマップを作成したりデータベースに接続するようなオートメーションサーバーを作成し，このサーバーを使って，クライアントが Web ページをロードするたびに更新されるデータにアクセスするようにできます。

ASP は，Web ページにサービスを提供することに関しては Microsoft Internet Information Server（IIS）環境に依存しています。

C++Builder ウィザードは，Active Server オブジェクトの作成を可能にします。このオブジェクトは，特に ASP で使用するために設計されたオートメーションオブジェクトです。これらの型のオブジェクトの作成方法と使い方についての詳細は，第 42 章「ASP（Active Server Page）の作成」を参照してください。

ActiveX コントロール

ActiveX を使うと，COM コンポーネント，特にコントロールを，よりコンパクトで効率的にすることができます。これは，クライアントがダウンロードして使用するイントラネットアプリケーション向けのコントロールに，特に重要です。

ActiveX コントロールは，インプロセスサーバーとしてのみ実行するビジュアルコントロールであり，ActiveX コントロールコンテナアプリケーションにプラグインできます。ActiveX コントロールそのものは完全なアプリケーションではありませんが，各種のアプリケーションで再使用可能な作成済み OLE コントロールとみなせます。ActiveX コントロールは，表示されるユーザーインターフェースを持ち，定義済みのインターフェースに依存して，入出力と表示の問題をホストコンテナとネゴシエートします。

ActiveX コントロールはオートメーションを使ってそのプロパティ，メソッド，イベントをエクスポーズします。ActiveX コントロールの機能には，イベントを送出し，データソースにバインドし，ライセンス付与をサポートする機能などがあります。

ActiveX コントロールの使い方の 1 つは，Web ページ内の対話型オブジェクトとして Web サイトに使用することです。このように，ActiveX は World Wide Web 用の対話型コンテンツをターゲットとする標準であり，Web ブラウザを経由して HTML 以外のドキュメントを表示するための ActiveX ドキュメントも使われています。ActiveX 技術についての詳細は，Microsoft ActiveX Web サイトを参照してください。

C++Builder のウィザードを使うと，ActiveX コントロールを簡単に作成できます。これらの型のオブジェクトの作成方法と使い方についての詳細は，第 43 章「ActiveX コントロールの作成」を参照してください。

ActiveX ドキュメント

ActiveX ドキュメント（以前は OLE ドキュメントと呼ばれていました）は，リンクと埋め込み，ドラッグアンドドロップ，およびビジュアル編集をサポートする COM サービスのセットです。ActiveX ドキュメントは，形式が異なるデータまたはオブジェクト（サウンドクリップ，スプレッドシート，テキスト，ビットマップなど）をシームレスに統合できます。

ActiveX コントロールと異なり，ActiveX ドキュメントはインプロセスサーバーに限定されずクロスプロセスアプリケーションでも使えます。

また，オートメーションオブジェクトがビジュアルな場合はほとんどありませんが，ActiveX ドキュメントオブジェクトはほかのアプリケーションでビジュアルに操作できます。したがって，ActiveX ドキュメントオブジェクトは 2 種類の型のデータと関連付けられます。ディスプレイまたは出力デバイスでオブジェクトをビジュアルに表示するために使われるプレゼンテーションデータと，オブジェクトの編集に使われるネイティブデータです。

ActiveX ドキュメントオブジェクトは，ドキュメントコンテナの場合もドキュメントサーバーの場合もあります。C++Builder は ActiveX ドキュメントを作成するための自動ウィザードを提供していませんが，VCL クラス TOleContainer を使えば，既存の ActiveX ドキュメントのリンクと埋め込みをサポートできます。

TOleContainer を ActiveX ドキュメントコンテナの土台として使うこともできます。ActiveX ドキュメントサーバー用のオブジェクトを作成するには，COM オブジェクトウィザードを使用し，オブジェクトがサポートする必要のあるサービスに応じて適切なインターフェースを実装します。ActiveX ドキュメントサーバーの作成と使い方についての詳細は，Microsoft ActiveX Web サイトを参照してください。

メモ ActiveX ドキュメント仕様には，クロスプロセスアプリケーションにおけるマーシャリングのサポートが組み込まれていますが，ActiveX ドキュメントはマシン上のシステムに固有の型（ウィンドウハンドル，メニューハンドルなど）を使うので，リモートサーバー上では実行されません。

トランザクションオブジェクト

C++Builder では，（Windows 2000 より前のバージョンの Windows の場合は）MTS（Microsoft Transaction Server），または（Windows 2000 以降の場合は）COM+ が提供するトランザクションサービス，セキュリティ，およびリソース管理を利用するオブジェクトのことを，「トランザクションオ

ブジェクト」と呼びます。このオブジェクトは大規模な分散環境において機能するように設計されています。

トランザクションサービスは，動作が常に完了またはロールバックされるようにする堅固さを提供します（サーバーが動作を部分的に完了することはありません）。セキュリティサービスを使うと，さまざまなレベルのサポートをさまざまなクラスのクライアントにエクスポーズすることができます。リソース管理を使うと，リソースプーリングによって，または使用時のみオブジェクトをアクティブにすることによって，より多くのクライアントをオブジェクトが処理できます。システムがこれらのサービスを提供できるようにするには，オブジェクトが IObjectControl インターフェースを実装しなければなりません。トランザクションオブジェクトはこれらのサービスにアクセスするために，MTS または COM+ によって作成される IObjectContext というインターフェースを使用します。

MTS では，サーバーオブジェクトをライブラリ（DLL）に組み込み，このライブラリを MTS 実行時環境にインストールしなければなりません。つまり，サーバーオブジェクトとは，MTS 実行時プロセス空間において実行されるインプロセスサーバーです。COM+ では，すべての COM 呼び出しがインターセプタ経由で送られるので，この制約が当てはまりません。クライアントから見れば，MTS と COM+ の違いはありません。

MTS または COM+ サーバーは，同じプロセス空間で実行されるトランザクションオブジェクトをグループ化します。このグループは，MTS では MTS パッケージと呼ばれ，COM+ では COM+ アプリケーションと呼ばれます。1 つのマシンが複数の MTS パッケージ（または COM+ アプリケーション）を実行し，それぞれを別個のプロセス空間において実行することができます。

クライアントには，トランザクションオブジェクトはその他の COM サーバーオブジェクトと同じように見えます。クライアントは，トランザクションそのものを起動すること以外は，トランザクション，セキュリティ，または即時アクティブ化について知っている必要はありません。

MTS および COM+ はいずれも，トランザクションオブジェクトを管理するための独立したツールを提供します。このツールを使うと，オブジェクトをパッケージまたは COM+ アプリケーションに設定し，1 台のコンピュータにインストールされたパッケージまたは COM+ アプリケーションを表示し，含まれているオブジェクトの属性を表示または変更し，トランザクションを監視および管理し，クライアントがオブジェクトを使用できるようにするなどのことができます。MTS では，このツールは MTS エクスプローラです。COM+ ではコンポーネントサービスです。

COM+ イベントおよびイベントサブスクライバオブジェクト

COM+ イベントシステムは，イベントを生成するアプリケーション（すなわちパブリッシャ）を，イベントに応答するアプリケーション（すなわちサブスクライバ）から切り離す中間層のソフトウェアを導入します。パブリッシャとサブスクライバは，相互に密接に結合しているのではなく，独立して開発，配布，および起動することができます。

COM+ イベントモデルでは，まず C++Builder の COM+ イベントオブジェクトウィザードを使用してイベントインターフェースが作成されます。イベントインターフェースは実装を持たず，パブリッシャが生成してサブスクライバが応答するイベントメソッドを定義するだけです。そして COM+ イベントオブジェクトが COM+ アプリケーションの COM+ カタログ内にインストールされます。これを行うには，C++Builder IDE を使用するか，TComAdminCatalog オブジェクトを使用してプログラムで行うか，システム管理者がコンポーネントサービスを使って行います。

イベントサブスクライバは，イベントインターフェースの実装を提供します。イベントサブスクライバコンポーネントを作成するには，C++Builder の COM+ Event Subscription ウィザードを使用します。このウィザードを使用すると，実装したいイベントオブジェクトを選択でき，C++Builder はインターフェースの各メソッドについてスタブを作成します。イベントオブジェクトがまだ COM+ カタログにインストールされていない場合は，タイプライブラリを選択することもできます。

最後に，実装が済んだサブスクライバコンポーネントも COM+ カタログもインストールしなければなりません。この場合も，C++Builder IDE を使用するか，TComAdminCatalog オブジェクトを使用するか，コンポーネントサービス管理ツールを使用します。

パブリッシャがイベントを生成する必要がある場合は，単に（サブスクライバコンポーネントではなく）イベントオブジェクトのインスタンスを作成し，イベントインターフェース上で適切なメソッドを呼び出します。そして COM+ が間に入り，そのイベントオブジェクトにサブスクライブしたすべてのアプリケーションを一度に 1 つずつ同時に呼び出して通知します。したがって，パブリッシャは，イベントにサブスクライブするアプリケーションのことを知る必要がありません。サブスクライバは，イベントインターフェースの実装のことさえ知っていればすみ，サブスクライブしたいパブリッシャを選択する以外は何もする必要がありません。その他のことは COM+ が処理します。

COM+ イベントシステムについての詳細は，44-20 ページの「COM+ でのイベントの生成」を参照してください。

タイプライブラリ

タイプライブラリは，オブジェクトのインターフェースから調べる場合より多くのオブジェクトの型情報を取得するための手段を提供します。タイプライブラリに入っている型情報は，オブジェクトとそのインターフェースについて必要な情報を提供します。たとえば，どのオブジェクトにどのようなインターフェースが存在するか（CLSID を指定された場合），各インターフェースにどのようなメンバー関数が存在するか，それらの関数がどのような引数を必要とするか，などの情報が提供されます。

型情報は，オブジェクトの実行中インスタンスに問い合わせるか，タイプライブラリをロードして読み出せば入手できます。どのようなメンバー関数が必要で，それらのメンバー関数に何を渡すべきかがわかっていれば，この型情報を使って，希望のオブジェクトを使うためのクライアントを実装することができます。

オートメーションサーバー，ActiveX コントロール，およびトランザクションオブジェクトのクライアントは，型情報が使用可能であるものと予想します。C++Builder のウィザードはすべて，自動的にタイプライブラリを生成します。この型情報は，第 39 章「タイプライブラリの操作」で説明されているように，タイプライブラリエディタを使って表示および編集することができます。

このセクションでは，タイプライブラリに何が入っているか，どのように作成していつ使うか，どのようにアクセスするかを説明します。異なる言語間でインターフェースを共有する場合のために，このセクションの最後にタイプライブラリツールの使い方についてのヒントを示します。

タイプライブラリの内容

タイプライブラリには，どのインターフェースがどの COM オブジェクトにあるか，それにインターフェースメソッドへの引数の型と数などを示す型情報が入っています。これらの記述には CoClass のユニークな識別子（CLSID）やインターフェースのユニークな識別子（IID）が含まれているので，

CoClass やインターフェースに正しくアクセスできます。また，これらの記述にはオートメーションインターフェースのメソッドやプロパティのディスパッチ識別子（dispID）も含まれています。

タイプライブラリには，ほかにも以下のような情報が入っていることがあります。

- カスタムインターフェースに関連付けられたカスタム型情報
- オートメーションサーバーか ActiveX サーバーがエクスポートするがインターフェースメソッドではないルーチン
- 列挙，レコード（構造体），共用体，エリアス，およびモジュールデータ型に関する情報
- ほかのタイプライブラリからの型記述への参照

タイプライブラリを作成する

従来の開発ツールを使ってタイプライブラリを作成するには，インターフェース定義言語（IDL: Interface Definition Language）またはオブジェクト記述言語（ODL: Object Description Language）でスクリプトを作成し，コンパイルして実行します。ただし，C++Builder では，［新規作成］ダイアログボックスの［ActiveX］または［多層サポート］ページのウィザードのいずれかを使って，（ActiveX コントロール，オートメーションオブジェクト，リモートデータモジュールなどの）COM オブジェクトを作成すると，自動的にタイプライブラリが生成されます。メインメニューからもタイプライブラリを作成できます。［ファイル｜新規作成｜その他］を選択し，［ActiveX］タブで［タイプライブラリ］を選択します。

作成したタイプライブラリは，C++Builder のタイプライブラリエディタを使って表示することができます。タイプライブラリはタイプライブラリエディタを使って簡単に編集することができ，編集後タイプライブラリを保存すると，対応する .tlb ファイル（バイナリタイプライブラリファイル）が C++Builder により自動的に更新されます。ウィザードを使って作成されたインターフェースと CoClass に変更を加えると，タイプライブラリエディタは実装ファイルも更新します。タイプライブラリエディタを使ってインターフェースや CoClass を作成する方法については，第 39 章「タイプライブラリの操作」を参照してください。

いつタイプライブラリを使うか

外部ユーザーにエクスポーズする一連のオブジェクトごとにタイプライブラリを作成することが重要です。たとえば以下のようなことです。

- ActiveX コントロールにはタイプライブラリが必要である。この場合，ActiveX コントロールが入っている DLL にリソースとしてタイプライブラリが入っていなければならない
- 仮想テーブルのカスタムインターフェースのバインディングをサポートする，エクスポーズされたオブジェクトは，コンパイル時に仮想テーブル参照がバインドされるので，タイプライブラリで記述しなければならない。クライアントは，インターフェースについての情報をタイプライブラリからインポートし，その情報を使用してコンパイルする。仮想テーブルとコンパイル時のバインディングについての詳細は，41-12 ページの「オートメーションインターフェース」を参照
- オートメーションサーバーを実装するアプリケーションの場合，クライアントがアーリーバインディングできるようにタイプライブラリを用意しなければならない
- IProvideClassInfo インターフェースをサポートするクラスからインスタンス化されたオブジェクトには，タイプライブラリが必要である

- OLE ドラッグアンドドロップで使うオブジェクトを特定する場合，タイプライブラリは必須ではないが，あると役立つ

タイプライブラリにアクセスする

バイナリタイプライブラリは通常はリソースファイル（.res）の一部か，ファイル名拡張子が .tlb のスタンドアロンファイルの一部です。タイプライブラリは，リソースファイルに含まれている場合はサーバー（.dll，.ocx，または .exe）の中にバインドできます。

タイプライブラリをいったん作成すると，オブジェクトブラウザやコンパイラなどのツールは特殊な型インターフェースを経由してタイプライブラリにアクセスできます。

インターフェース	内容
ITypeLib	型の記述のライブラリにアクセスするためのメソッドを提供する
ITypeLib2	ドキュメント文字列，カスタムデータ，およびタイプライブラリについての解析結果のサポートを含むように，ITypeLib を拡大する
ITypeInfo	タイプライブラリ内の各オブジェクトの記述を提供する。たとえば，ブラウザはこのインターフェースを使って，オブジェクトに関する情報をタイプライブラリから抽出する
ITypeInfo2	追加のタイプライブラリ情報（カスタムデータ要素にアクセスするためのメソッドなど）にアクセスするように，ITypeInfo を拡大する
ITypeComp	コンパイラがインターフェースをバインドするときに必要とする情報にすばやくアクセスする手段を提供する

C++Builder では，［プロジェクト｜タイプライブラリの取り込み］を選択して，タイプライブラリをほかのアプリケーションからインポートして使うことができます。

タイプライブラリを使う利点

タイプライブラリはアプリケーションに必須ではありませんが，タイプライブラリを使うと以下の利点があります。

- オートメーションに対してアーリーバインディングを使うことができ（バリアントを通じて呼び出す場合とは対照的），仮想テーブルもデュアルインターフェースもサポートしないコントローラでもコンパイル時に dispID を符号化できるので実行時の処理効率が上がる
- ブラウザがライブラリをスキャンできるので，開発者のオブジェクトの特徴をクライアントが調べることができる
- RegisterTypeLib 関数を使って，エクスポーズしたオブジェクトを登録データベースに登録できる
- UnRegisterTypeLib 関数を使って，アプリケーションのタイプライブラリをシステムレジストリから完全に解除できる
- オートメーションはタイプライブラリの情報を使って，ほかのプロセスでオブジェクトに渡されるパラメータをパッケージするので，ローカルサーバーがアクセスしやすくなる

タイプライブラリツールを使う

タイプライブラリを操作するツールには，以下のものがあります。

- TLIBIMP（タイプライブラリインポート）ツールは既存のタイプライブラリを取り出して C++Builder インターフェースファイル（_TLB.cpp および _TLB.h ファイル）を作成する。このツールはタイプライブラリエディタに組み込まれている。TLIBIMP は，タイプライブラリエディタ内にはない追加的オプションを提供する
- TRegSvr はサーバーとタイプライブラリを登録および登録解除するためのツールで，C++Builder に添付されている。TRegSvr のソースは，例として Examples ディレクトリに収められている
- Microsoft IDL コンパイラ（MIDL）は IDL スクリプトをコンパイルしてタイプライブラリを作成する。MIDL にはヘッダーファイルを生成するオプションスイッチが装備されている。MS Win32 SDK を参照
- RegSvr32.exe はサーバーとタイプライブラリを登録および登録解除するための標準 Windows ユーティリティである
- OLEView はタイプライブラリブラウザツールであり，Microsoft の Web サイトにある

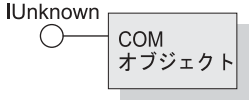
ウィザードを使って COM オブジェクトを実装する

C++Builder では，必要となる細かい操作の多くを処理するウィザードが用意されているので，COM サーバーアプリケーションを簡単に作成することができます。C++Builder では，以下のものを作成するために，それぞれ別個のウィザードが用意されています。

- 単純な COM オブジェクト
- オートメーションオブジェクト
- Active Server オブジェクト（ASP に埋め込む）
- ActiveX コントロール
- ActiveX フォーム
- トランザクションオブジェクト
- COM+ イベントオブジェクト
- COM+ サブスクリプションオブジェクト
- プロパティページ
- タイプライブラリ
- ActiveX ライブラリ

これらのウィザードは，それぞれの型の COM オブジェクトの作成にかかわる作業の多くを処理します。また，それぞれの型のオブジェクトに必要な COM インターフェースを提供します。単純な COM オブジェクトの場合，ウィザードは図 38.6 に示したように，オブジェクトへのインターフェースポインタを提供する必須 COM インターフェースである IUnknown を実装します。

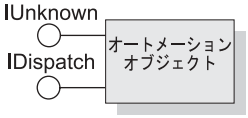
図 38.6　単純な COM オブジェクトのインターフェース



IDispatch の子孫をサポートするオブジェクトの作成を指定する場合，COM オブジェクトウィザードは，IDispatch の実装も提供します。

オートメーションオブジェクトおよび Active Server オブジェクトの場合，ウィザードは図 38.7 に示したように，IUnknown のほかに，自動マーシャリングを提供する IDispatch を実装します。

図 38.7　オートメーションオブジェクトのインターフェース



ActiveX コントロールオブジェクトおよび ActiveX フォームの場合，ウィザードは図 38.8 に示したように，IUnknown から IDispatch，IOleObject，IOleControl などに至るまで，必要な ActiveX コントロールインターフェースをすべて実装します。

図 38.8　ActiveX オブジェクトのインターフェース

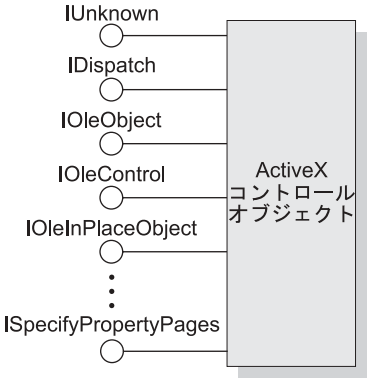


表 38.2 に示したように，さまざまなウィザードが以下のインターフェースを実装します。

表 38.2　COM，オートメーション，ActiveX オブジェクトを実装する C++Builder ウィザード

ウィザード	実装するインターフェース	ウィザードが実行すること
COM サーバー	IUnknown（IDispatch から継承されたデフォルトインターフェースを選択する場合は IDispatch も実装する）	サーバーの登録，クラスの登録，サーバーのロードとアンロード，オブジェクトのインスタンス化を行うルーチンをエクスポートする サーバー上に実装するオブジェクト用クラスファクトリを作成および管理する 選択されたスレッドモデルを指定するオブジェクトのレジストリエントリを提供する 要求があれば，イベント生成のサーバー側のサポートを提供する 選択されたインターフェースを実装するメソッドの宣言によって，中身を埋めるだけで完成できるスケルトン実装を提供する タイプライブラリを提供する . タイプライブラリに登録された任意のインターフェースを選択し，実装できる。その場合，タイプライブラリを使用しなければならない
オートメーションサーバー	IUnknown，IDispatch	COM サーバーウィザードの作業（上記）に加えて，次のことを実行 指定されたインターフェースを実装する（デュアルまたはディスパッチ）
Active Server オブジェクト	IUnknown，IDispatch，（IASPObject）	オートメーションオブジェクトウィザードの作業（上記）を実行し，Web ブラウザへのロードが可能な .ASP ページをオプションで生成する。必要であればオブジェクトのプロパティやメソッドを変更できるように，タイプライブラリエディタをアクティブにしておく ASP 組み込みオブジェクトをプロパティとして公開し，ASP アプリケーションおよびこれを起動した HTTP メッセージについての情報を簡単に取得できるようにする
ActiveX コントロール	IUnknown，IDispatch，IPersistStreamInit，IOleInPlaceActiveObject，IPersistStorage，IViewObject，IOleObject，IViewObject2，IOleControl，IPerPropertyBrowsing，IOleInPlaceObject，ISpecifyPropertyPages	オートメーションサーバーウィザードの作業（上記）に加えて，以下のことを実行 ActiveX コントロールの土台となり，すべての ActiveX インターフェースを実装する VCL コントロールに対応する実装 CoClass を生成する ユーザーが実装クラスを変更できるように，ソースコードエディタをアクティブにする
ActiveForm	ActiveX コントロールのインターフェースと同じ	ActiveX コントロールウィザードの作業に加えて，次のことを実行 ActiveX コントロールウィザードにおいて既存の VCL クラスに代わる TActiveForm の子孫を作成する。この新しいクラスを使うと，Windows のアプリケーションでフォームを設計する場合と同じように ActiveForm を設計できる

表 38.2 COM，オートメーション，ActiveX オブジェクトを実装する C++Builder ウィザード（つづき）

ウィザード	実装するインターフェース	ウィザードが実行すること
トランザクションオブジェクト	IUnknown，IDispatch，IObjectControl	MTS または COM+ オブジェクト定義を含む現在のプロジェクトに新規のユニットを追加する。C++Builder がオブジェクトを適切にインストールできるように独自の GUID をタイプライブラリに挿入し，オブジェクトがクライアントにエクスポーズするインターフェースを定義できるようにタイプライブラリエディタをアクティブにしておく。オブジェクトは，構築した後に単独でインストールしなければならない
プロパティページ	IUnknown，IPropertyPage	フォームデザイナで設計できる新規のプロパティページを作成する
COM+ イベントオブジェクト	なし（デフォルト）	タイプライブラリエディタを使って定義できる COM+ イベントオブジェクトを作成する。イベントオブジェクトは実装を持たない（実装はイベントサブスクライバオブジェクトによって提供される）ので，COM+ イベントオブジェクトウィザードはほかのオブジェクトウィザードとは異なり，実装ユニットを作成しない
COM+ サブスクリプションオブジェクト	なし（デフォルト）	選択されたイベントインターフェースを実装する COM+ サブスクライバオブジェクトを作成する
タイプライブラリ	なし（デフォルト）	新規のタイプライブラリを作成してアクティブなプロジェクトと関連付ける
ActiveX ライブラリ	なし（デフォルト）	新規の ActiveX または COM サーバー DLL を作成し，必要なエクスポート関数をエクスポーズする

COM オブジェクトを追加したり，既存の実装を実装し直すこともできます。新しいオブジェクトを追加するには，もう一度ウィザードを使用する方法がもっとも簡単です。なぜなら，ウィザードはタイプライブラリと実装クラスを関連付けるので，タイプライブラリエディタで行った変更が実装オブジェクトに自動的に適用されるからです。

ウィザードによって生成されるコード

C++Builder のウィザードは，COM サポートの土台として Microsoft Active Template Library（ATL）を使用するコードを生成します。ATL は，COM アプリケーション開発の実装の詳細の多くを処理するテンプレートクラスのフレームワークです。ATL はテンプレートを土台とするので，DLL にはリンクしません。そのかわり，オブジェクトコードにコンパイルされる ATL ヘッダーファイルがプロジェクトに含まれます。C++Builder のウィザードは，（Project1_ATL.cpp や Project1_ATL.h のように）_ATL サフィックスが付いたユニット内に，このヘッダーファイルの include 文を生成します。

メモ C++Builder の Include \ ATL ディレクトリ内の ATL ヘッダーファイルは，Microsoft が供給する ATL ヘッダーファイルとは少し異なります。この違いは，C++Builder のコンパイラがヘッダーをコンパイルできるようにするために必要となります。このヘッダーを別のバージョンの ATL と置き換えることはできません。なぜなら，正しくコンパイルされないからです。

ATL ファイルの include 文（および ATL クラスが VCL クラスとともに使用できるようにする追加ファイル）だけでなく，生成された _ATL ユニットヘッダーには，_Module と呼ばれるグローバル変数の宣言も含まれます。_Module は ATL クラス CComModule のインスタンスの 1 つであり，スレッ

ドと登録の問題を処理する方法において，DLL と EXE の違いからアプリケーションのその他の部分を隔離します。プロジェクトファイル（Project1.cpp）において，_Module は TComModule のインスタンスに割り当てられます。TComModule は，CComModule の子孫であり，C++Builder スタイルの COM 登録をサポートします。通常はこのオブジェクトを直接使用する必要はありません。

ウィザードはプロジェクトファイルにオブジェクトマップも追加します。これは，たとえば次のような ATL マクロのセットです。

```
BEGIN_OBJECT_MAP(ObjectMap)
  OBJECT_ENTRY(CLSID_MyObj, TMyObjImpl)
END_OBJECT_MAP()
```

BEGIN_OBJECT_MAP 行と END_OBJECT_MAP 行の間の各エントリは，クラス ID とその ATL 実装クラスとの関連付けを定義します。_Module オブジェクトはこのマップを使用してコンポーネントを登録します。オブジェクトマップは，ATL オブジェクトのクリエータクラスによっても使用されます。ウィザードを使用せずにアプリケーションに COM オブジェクトを追加した場合は，このオブジェクトが正しく登録および作成できるようにオブジェクトマップを更新しなければなりません。それには，OBJECT_ENTRY マクロを使用する行を追加し，クラス ID と実装クラス名をパラメータとして割り当てます。

作成しようとしている特定型の COM オブジェクトの実装ユニットが，ウィザードによって生成されます。この実装ユニットには，COM オブジェクトを実装するクラスの宣言が含まれます。このクラスは，ATL クラス CComObjectRootEx，ATL クラス CComCoClass，および作成しようとしているオブジェクトの型に依存するその他のクラスの（直接または間接的な）子孫です。

CComCoClass は，クラスを作成するためのクラスファクトリサポートを提供します。プロジェクトファイルに追加されたオブジェクトマップを使用します。

CComObjectRootEx は，IUnknown の基底のサポートを提供するテンプレートクラスです。これはインターフェースマップを使用して QueryInterface メソッドを実装します（インターフェースマップは実装ユニットヘッダー内にあります）。このインターフェースマップは次のようになります。

```
BEGIN_COM_MAP(TMyObjImpl)
  COM_INTERFACE_ENTRY(IMyObj)
END_COM_MAP()
```

インターフェースマップ内の各エントリは，QueryInterface メソッドがエクスポーズするインターフェースです。実装クラスにインターフェースを追加する場合は，（COM_INTERFACE_ENTRY マクロを使用して）インターフェースマップに追加し，その実装クラスの上位として追加しなければなりません。

CComObjectRootEx は，参照カウントの基底のサポートを供給します。ただし，AddRef および Release メソッドを宣言しません。これらのメソッドは，インターフェースマップの最後で END_COM_MAP() マクロによって実装クラスに追加されます。

メモ ATL についての詳細は，Microsoft の資料を参照してください。ただし，C++Builder の COM サポートは，登録，ActiveX コントロール，またはプロパティページサポートに ATL を使用しません（ActiveX コントロールは VCL オブジェクトを土台とします）。

ウィザードはタイプライブラリおよび関連付けられたユニットも生成し，その名前は Project1_TLB という形式になります。この Project1_TLB ユニットには，タイプライブラリ内で定義された型定義

とインターフェースを使用するためにアプリケーションが必要とする定義が含まれます。このファイルの内容についての詳細は，40-5 ページの「タイプライブラリ情報のインポート時に生成されるコード」を参照してください。

ウィザードが生成したインターフェースは，タイプライブラリエディタを使用して変更できます。変更を加えると，その内容を反映するために実装クラスが自動的に更新されます。生成されたメソッドの中身を埋めるだけで，実装が完成します。

第39章

タイプライブラリの操作

この章では，C++Builder のタイプライブラリエディタを使ってタイプライブラリを作成および編集する方法を説明します。タイプライブラリは，COM オブジェクトのデータ型，インターフェース，メンバー関数，オブジェクトクラスについての情報が入ったファイルです。タイプライブラリは，サーバー上で使うことができるオブジェクトとインターフェースの型を識別する手段を提供します。タイプライブラリを使うべきときと理由についての詳細は，38-16 ページの「タイプライブラリ」を参照してください。

タイプライブラリには，以下の一部または全部を入れることができます。

- エリアス，列挙型，構造体，および共用体などのカスタムデータ型に関する情報
- インターフェース，ディスパッチインターフェース，または CoClass などの，1 つまたは複数の COM 要素の記述。通常はこれらの記述のそれぞれを型情報と呼ぶ
- 外部モジュールで定義されている定数とメソッドの記述
- ほかのタイプライブラリから型記述への参照

COM アプリケーションまたは ActiveX ライブラリにタイプライブラリを組み込むと，そのアプリケーション内のオブジェクトについての情報を，COM のタイプライブラリツールおよびインターフェース経由でほかのアプリケーションやプログラミングツールから使えるようになります。

従来型の開発ツールでは，インターフェース定義言語（IDL: Interface Definition Language）でスクリプトを作成し，それをコンパイラで実行することによって，タイプライブラリを作成していました。これに対し，タイプライブラリエディタを使うと，作業の一部が自動的に処理されるので，タイプライブラリを簡単に作成および変更することができます。

C++Builder のウィザードを使って（ActiveX コントロール，オートメーションオブジェクト，リモートデータモジュールなどの）なんらかの型の COM サーバーを作成すると，ウィザードがタイプライブラリを自動的に生成します。生成されたオブジェクトのカスタマイズ作業のほとんどはタイプライブラリから開始します。なぜなら，クライアントにエクスポーズするプロパティとメソッドをここで定義するからです。ウィザードが生成した CoClass のインターフェースを変更するには，タイプライブラリエディタを使用します。タイプライブラリエディタはオブジェクトの実装ユニットを自動的に更新するので，生成されたメソッドの中身を埋めるだけで済みます。

タイプライブラリエディタ

タイプライブラリエディタでは，開発者が COM オブジェクトの型情報を調べたり作成することができます。タイプライブラリエディタを使用すると，COM オブジェクトの開発作業が大幅に簡単になります。たとえば，インターフェース，CoClass，および型の定義作業の集中化，新しいインターフェースの GUID の取得，インターフェースと CoClass との関連付け，実装ユニットの更新などができます。

タイプライブラリエディタは，タイプライブラリの内容を表す 2 種類のファイルを出力します。

表 39.1　タイプライブラリエディタのファイル

ファイル	内容
.TLB ファイル	バイナリタイプライブラリファイル。タイプライブラリはリソースとして自動的にアプリケーション内にコンパイルされるので，デフォルトではこのファイルを使用する必要はありません。しかし，このファイルを使用すると，タイプライブラリを別のプロジェクト内に明示的にコンパイルしたり，.exe または .ocx と切り離してタイプライブラリを配布することができます。詳細については，39-12 ページの「既存のタイプライブラリを開く」および 39-18 ページの「タイプライブラリの配布」を参照してください。
_TLB ユニット	このユニット（.cpp ファイルと .h ファイル）は，アプリケーションが使用できるようにタイプライブラリの内容を反映します。このユニットには，タイプライブラリ内で定義された要素を使用するためにアプリケーションが必要とするすべての宣言が含まれます。コードエディタでこのファイルを開くことはできますが，編集はしないでください。このファイルはタイプライブラリエディタによって保守されるので，変更を加えてもタイプライブラリエディタによって上書きされます。このファイルの内容についての詳細は，40-5 ページの「タイプライブラリ情報のインポート時に生成されるコード」を参照してください。

タイプライブラリエディタの要素

タイプライブラリエディタの主な要素を以下に示します。

表 39.2　タイプライブラリエディタのパーツ

パーツ	内容
ツールバー	新しい型，CoClass，インターフェース，インターフェースメンバーをタイプライブラリに追加するボタンがある。実装ユニットを更新するボタン，タイプライブラリを登録するボタン，および情報とともに IDL ファイルをタイプライブラリに保存するボタンもある
オブジェクトリストペイン	タイプライブラリの既存の要素を表示する。オブジェクトリストペインの項目を選択すると，そのオブジェクトにとって有効なページが表示される
ステータスバー	無効な型をタイプライブラリに追加しようとすると，構文エラーを表示する
ページ	選択されているオブジェクトに関する情報を表示する。どのページが表示されるかは選択されたオブジェクト型によって決まる

図 39.1 に，これらの要素を示します。この図のタイプライブラリエディタには，cyc という名前の COM オブジェクトの型情報が表示されています。

図 39.1　タイプライブラリエディタ

ツールバー

タイプライブラリエディタのツールバーはタイプライブラリエディタの一番上にあります。ツールバーのボタンを選択することにより，新しいオブジェクトをタイプライブラリに追加できます。

最初のグループのボタンを使用すると，タイプライブラリに要素を追加できます。ツールバーのボタンを選択すると，その要素のアイコンがオブジェクトリストペインに表示されます。右側のペインでその属性をカスタマイズできます。右側に表示される情報ページは，選択したアイコンの種類によって異なります。

タイプライブラリに追加できる要素は以下のとおりです。

アイコン	説明
	インターフェース
	ディスパッチインターフェース
	CoClass
	列挙型（Enum）
	エリアス
	レコード
	共用体
	モジュール

以上の要素の 1 つをオブジェクトリストペイン内で選択すると，2 番目のボタングループに，その要素に有効なメンバーが表示されます。たとえば，インターフェースを選択すると，インターフェース

の定義にはメソッドとプロパティを追加できるので，2 番目のグループでメソッドとプロパティのアイコンが使用可能になります。列挙型を選択すると，2 番目のボタングループは，列挙型情報で唯一の有効なメンバーである Const を表示するように変わります。

オブジェクトリストペイン内の要素に追加できるメンバーは以下のとおりです。

アイコン	説明
	インターフェース，ディスパッチインターフェース，またはモジュール内のエントリポイントのメソッド
	インターフェースまたはディスパッチインターフェース上のプロパティ
	書き込み専用プロパティ（プロパティボタン上のドロップダウンリストから選択可能）
	読み書き可能プロパティ（プロパティボタン上のドロップダウンリストから選択可能）
	読み出し専用プロパティ（プロパティボタン上のドロップダウンリストから選択可能）
	レコードまたは共用体内のフィールド
	列挙型またはモジュール内の定数

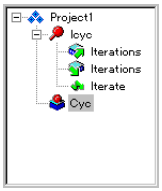
39-17 ページの「タイプライブラリ情報の保存と登録」で説明しているように，3 番目のボタングループでは，タイプライブラリを更新するか，登録するか，エクスポートする（IDL ファイルとして保存する）ことができます。

オブジェクトリストペイン

オブジェクトリストペインには，現在のタイプライブラリのすべての要素がツリービューで表示されます。ツリーのルートはタイプライブラリ自身を表し，次のアイコンで表示されます。

タイプライブラリノードの下位は，そのタイプライブラリ内の要素です。

図 39.2　オブジェクトリストペイン



これらの要素のいずれか（タイプライブラリ自身を含む）を選択すると，その要素に関連する情報だけを反映するように右側の型情報ページが変わります。このページを使用すると，選択した要素の定義とプロパティを編集できます。

オブジェクトリストペイン内の要素を操作するには，右クリックしてオブジェクトリストペインのコンテキストメニューを表示します。このメニューのコマンドを使用すると，Windows のクリップボードを使用して既存の要素を移動またはコピーしたり，新しい要素を追加したり，タイプライブラリエディタの外観をカスタマイズできます。

ステータスバー

タイプライブラリを編集または保存すると，構文，変換のエラーや警告のリストがステータスバーペインに表示されます。

たとえば，タイプライブラリエディタでサポートしていない型を指定した場合，構文エラーが表示されます。タイプライブラリエディタがサポートしている型の全リストについては，39-10 ページの「有効な型」を参照してください。

型情報のページ

オブジェクトリストペインで要素を選択すると，選択した要素について有効な型情報のページが，タイプライブラリエディタに表示されます。表示されるページは，次の表に示すように，オブジェクトリストペインで選択される要素によって異なります。

表 39.3　タイプライブラリページ

型情報要素	型情報のページ	ページの内容
タイプライブラリ	属性	タイプライブラリの名前，バージョン，GUID，およびそのタイプライブラリをヘルプにリンクする情報
	使用	このタイプライブラリが依存する定義が入っているほかのタイプライブラリのリスト
	フラグ	ほかのアプリケーションがそのタイプライブラリをどのように使用できるかを決定するフラグ
	テキスト	タイプライブラリ自身を定義するすべての定義および宣言（後述）
インターフェース	属性	インターフェースの名前，バージョン，GUID，そのインターフェースの上位のインターフェースの名前，およびそのインターフェースをヘルプにリンクする情報
	フラグ	インターフェースが，隠れている，デュアル，オートメーション互換，拡張可能であるかどうかを示すフラグ
	テキスト	インターフェースの定義と宣言（後述）
ディスパッチインターフェース	属性	インターフェースの名前，バージョン，GUID，およびこれをヘルプにリンクする情報
	フラグ	ディスパッチインターフェースが，隠れている，デュアル，拡張可能であるかどうかを示すフラグ
	テキスト	ディスパッチインターフェースの定義と宣言（後述）
CoClass	属性	CoClass の名前，バージョン，GUID，およびこれをヘルプにリンクする情報
	実装するインターフェース	CoClass が実装するインターフェース，およびその属性のリスト
	COM+	トランザクションオブジェクトの属性，すなわち，トランザクションモデル，同期化サポート，即時アクティブ化，オブジェクトのプールなど。COM+ イベントオブジェクトの属性も含む
	フラグ	CoClass のさまざまな属性を示すフラグ。たとえば，クライアントがどのようにインスタンスを作成および使用できるか，ブラウザ内でユーザーに見えるようにするか，ActiveX コントロールであるかどうか，集約できる（集合体の一部として機能できる）かどうかを示す
	テキスト	CoClass の定義と宣言（後述）
列挙型	属性	列挙型の名前，バージョン，GUID，およびこれをヘルプにリンクする情報

表 39.3 タイプライブラリページ（つづき）

型情報要素	型情報のページ	ページの内容
	テキスト	列挙型の定義と宣言（後述）
エリアス	属性	エリアスの名前，バージョン，GUID，エリアスが表す型，およびこれをヘルプにリンクする情報
	テキスト	エリアスの定義と宣言（後述）
レコード	属性	レコードの名前，バージョン，GUID，およびこれをヘルプにリンクする情報
	テキスト	レコードの定義と宣言（後述）
共用体	属性	共用体の名前，バージョン，GUID，およびこれをヘルプにリンクする情報
	テキスト	共用体の定義と宣言（後述）
モジュール	属性	モジュールの名前，バージョン，GUID，関連付けられた DLL，およびこれをヘルプにリンクする情報
	テキスト	モジュールの定義と宣言（後述）
メソッド	属性	名前，ディスパッチ ID，または DLL エントリポイント，およびこれをヘルプにリンクする情報
	パラメータ	メソッドの戻り型，すべてのパラメータおよびその型と修飾子のリスト
	フラグ	クライアントがどのようにメソッドを表示および使用できるか，これがインターフェースのデフォルトメソッドであるかどうか，および置き換え可能であるかどうかを示すフラグ
	テキスト	メソッドの定義と宣言（後述）
プロパティ	属性	プロパティアクセスメソッド（取得と設定）の名前，ディスパッチ ID，型，およびこれをヘルプにリンクする情報
	パラメータ	プロパティアクセスメソッドの戻り型，すべてのパラメータおよびその型と修飾子のリスト
	フラグ	クライアントがどのようにプロパティを表示および使用できるか，これがインターフェースのデフォルトであるかどうか，およびこのプロパティが置き換え可能，バインド可能であるかどうかなどを示すフラグ
	テキスト	プロパティアクセスメソッドの定義と宣言（後述）
定数	属性	（モジュール定数の）名前，値，型，およびこれをヘルプにリンクする情報
	フラグ	クライアントがどのように定数を表示および使用できるか，これがデフォルト値を表すかどうか，この定数がバインド可能などであるかどうかなどを示すフラグ
	テキスト	定数の定義と宣言（後述）
フィールド	属性	名前，型，およびこれをヘルプにリンクする情報
	フラグ	クライアントがどのようにフィールドを表示および使用できるか，これがデフォルト値を表すかどうか，このフィールドがバインド可能などであるかどうかなどを示すフラグ
	テキスト	フィールドの定義と宣言（後述）

メモ 型情報のページで設定できるさまざまなオプションについての詳細は，タイプライブラリエディタのオンラインヘルプを参照してください。

型情報の各ページを使用すると，そのページに表示される値を調べたり編集することができます。ほとんどのページでは，情報をいくつかのコントロールに編成しているので，対応する宣言の構文を知

らなくても，値を入力したりリストから選択することができます。したがって，限定されたセットから値を指定すれば，スペルミスなどの小さな誤りを防止できます。ただし，宣言を直接入力した方が速いと感じる人もいるでしょう。直接入力するには，［テキスト］ページを使用します。

すべてのタイプライブラリの要素には，その要素の構文を表示する［テキスト］ページがあります。この構文は，Microsoft インターフェース記述言語（IDL: Interface Definition Language）の IDL サブセットで表示されます。要素のほかのどのページに行った変更も，［テキスト］ページに反映されます。［テキスト］ページにコードを直接追加すると，その変更は，タイプライブラリエディタのほかのページにも反映されます。

サポートしていない識別子が追加されると，タイプライブラリエディタは構文エラーを生成します。現在のタイプライブラリエディタは，タイプライブラリサポートに関連する識別子だけをサポートしており，C++ コードの生成やマーシャリングのサポートを行うために Microsoft IDL コンパイラが使用する RPC サポートまたは構文に関連する識別子はサポートしていません。

タイプライブラリの要素

タイプライブラリのインターフェースは，一見するととてつもなく複雑なように見えます。なぜなら，多数の要素についての情報が表示され，各要素が独自の特性を持っているからです。しかし，このような特性の多くはすべての要素に共通しています。たとえば，（タイプライブラリ自身を含む）あらゆる要素に以下のものがあります。

- ［名前］は，要素の説明，およびコード内でのその要素の参照に使用される
- ［GUID］（globally unique identifier）は，ユニークな 128 ビットの値であり，COM が要素を識別するため使用する。これはタイプライブラリ自身に対して，および CoClass とインターフェースに対して，常に提供する必要がある。それ以外に対してはオプションである
- ［バージョン］番号は，複数のバージョンの要素を区別する。これは常にオプションだが，バージョン番号を持たない CoClass とインターフェースを使用できないツールがあるので，これらに対してはバージョン番号を提供する必要がある
- その要素をヘルプトピックにリンクする情報。［ヘルプ文字列］，［ヘルプコンテキスト］，［ヘルプ文字列コンテキスト］などの値である。［ヘルプコンテキスト］は，独立したヘルプファイルをタイプライブラリが持つ従来型の Windows ヘルプシステムに使用される。［ヘルプ文字列コンテキスト］は，別個の DLL によってヘルプを供給するときに使用される。［ヘルプコンテキスト］または［ヘルプ文字列コンテキスト］は，タイプライブラリの［属性］ページで指定されたヘルプファイルまたは DLL を参照する。これは常にオプションである。

インターフェース

インターフェースは，仮想関数テーブル（vtable）を通じてアクセスしなければならないオブジェクトのメソッドおよびプロパティ（「get」と「set」関数で記述される）を表します。インターフェースに dual というフラグが付いている場合は IDispatch から継承され，アーリーバインディングの vtable アクセスと OLE オートメーションによる実行時バインディングの両方をオブジェクトが提供できます。デフォルトでは，追加したすべてのインターフェースに dual のフラグが付きます。

インターフェースには，メンバーを割り当てることができます。すなわち，メソッドとプロパティを割り当てられます。これらはオブジェクトリストペイン内ではインターフェースノードの子として表示されます。インターフェースのプロパティは，プロパティのデータの読み出しと書き込みに使用される「get」および「set」メソッドとして表されます。プロパティは，ツリービューの中でそれぞれの目的を表す特別なアイコンを使って表示されます。

メモ プロパティが Write By Reference として指定されている場合は，そのプロパティが値ではなくポインタとして渡されることを意味します。Visual Basic など一部のアプリケーションは，処理効率を最適化するために，もし存在していれば Write By Reference を使用します。プロパティを値ではなく参照によってのみ渡すには，プロパティ型 By Reference Only を使用します。プロパティを値で渡すことも参照によって渡すこともある場合は，［読み込み｜書き込み｜参照による書き込み］を選択します。このメニューを呼び出すには，ツールバーで，プロパティアイコンの隣の矢印を選択します。

ツールバーのボタンまたはオブジェクトリストペインのコンテキストメニューを使用してプロパティまたはメソッドを追加したら，プロパティまたはメソッドを選択して型情報のページを使用して，構文と属性を記述します。

［属性］ページを使用すると，（IDispatch を使用して呼び出せるように）プロパティまたはメソッドに名前とディスパッチ ID を指定できます。プロパティの場合は，型も指定します。関数宣言は［パラメータ］ページを使用して作成します。このページでは，パラメータの追加，除去，並べ替え，パラメータの型と修飾子の設定，および関数の戻り型の指定が可能です。

インターフェースに割り当てたプロパティとメソッドは，インターフェースに関連付けられた CoClass に暗黙に割り当てられます。このため，タイプライブラリエディタを使用してプロパティとメソッドを CoClass に直接追加することはできません。

ディスパッチインターフェース

オブジェクトのプロパティおよびメソッドを記述する場合，ディスパッチインターフェースよりもインターフェースのほうが一般的に使用されます。インターフェースが仮想テーブルを介して静的バインディングができるのに対し，ディスパッチインターフェースは，動的バインディングでのみアクセスできます。

ディスパッチインターフェースにメソッドとプロパティを追加する方法は，インターフェースに追加する方法と同じです。ただし，ディスパッチインターフェースのプロパティを作成するときは，関数の種類やパラメータの型を指定できません。

CoClass

CoClass は，1 つまたは複数のインターフェースを実装するユニークな COM オブジェクトを記述します。CoClass を定義するときには，実装されたインターフェースのどれがオブジェクトのデフォルトかを指定し，さらにオプションとして，どのディスパッチインターフェースをイベントのデフォルトソースとするかを指定しなければなりません。タイプライブラリエディタでは CoClass にプロパティまたはメソッドを追加しません。プロパティとメソッドはインターフェースによってクライアントにエクスポーズされ，このインターフェースは［実装するインターフェース］ページを使用して CoClass に関連付けられます。

型定義

列挙型，エリアス，レコード，および共用体はすべて型を宣言し，この型はタイプライブラリ内のほかの場所で使用できるようになります。

列挙型は定数のリストで構成され，各定数は数値でなければなりません。数値入力は，通常，10 進または 16 進形式の整数です。基準値のデフォルトは 0 です。列挙型に定数を追加するには，オブジェクトリストペイン内で列挙型を選択し，ツールバーの［Const］ボタンを選択するか，オブジェクトリストペインのコンテキストメニューから［新規作成｜ Const］コマンドを選択します。

メモ 列挙型には，その意味を明確にするために，できるだけヘルプ文字列を提供するようにしてください。次のコードに，マウスボタンの列挙型のサンプルエントリを示します。このエントリには列挙型要素ごとのヘルプ文字列も入っています。

```
typedef enum TxMouseButton
{
   [helpstring("mbLeft")]
   mbLeft = 0,
   [helpstring("mbRight")]
   mbRight = 1,
   [helpstring("mbMiddle")]
   mbMiddle = 2
} TxMouseButton;
```

エリアスによって，型のエリアス（typedef）が作成されます。エリアスを使って，レコードや共用体など，ほかの型情報で使用する型を定義できます。［属性］ページの［型］属性を設定して，基底の型定義とエリアスを関連付けます。

レコードは，C 言語の構造体です。レコードは構造体メンバーまたはフィールドのリストで構成されます。共用体は，C 言語の共用体を定義します。レコードと同様に，共用体は構造体メンバーまたはフィールドのリストで構成されます。ただし，レコードのメンバーと異なる点は，共用体の各メンバーが同じ物理アドレスを占めるので，1 つの論理値しか保存できないことです。

レコードまたは共用体にフィールドを追加するには，オブジェクトリストペインでレコードまたは共用体を選択し，ツールバーの［Field］ボタンを選択するか，右クリックしてオブジェクトリストペインのコンテキストメニューから［新規作成｜ Field］を選択します。各フィールドは名前と型を持ち，名前と型を割り当てるには，フィールドを選択し，［属性］ページを使って値を指定します。レコードと共用体は，オプションタグを使って C の構造体として定義できます。

メンバーは任意の組み込み型を指定するか，レコードを定義する前に，エリアスを使って型を指定することもできます。

メモ C++Builder は，switch_type，first_is，last_is などの，構造体と共用体のマーシャル関連のキーワードをサポートしていません。

モジュール

モジュールによって，関数のグループ，通常は DLL の一連のエントリポイントが定義されます。以下の操作によってモジュールを定義します。

- モジュールが表す DLL を［属性］ページで指定する

- ツールバー，またはオブジェクトリストペインのコンテキストメニューを使用して，メソッドと定数を追加する。各メソッドまたは各定数に対して属性を定義しなければならない。それには，メソッドまたは定数をオブジェクトリストペイン内で選択し，［属性］ページで値を設定する

モジュールのメソッドに対しては，［属性］ページを使用して名前と DLL エントリポイントを指定しなければなりません。パラメータページを使用して関数のパラメータと戻り型を宣言します。

モジュールの定数に対しては，［属性］ページを使用して，名前，型，および値を指定します。

メモ タイプライブラリエディタでは，モジュールに関連する宣言または実装は生成されません。指定された DLL が別個のプロジェクトとして作成されなければなりません。

タイプライブラリエディタの使い方

タイプライブラリエディタを使うと，新しいタイプライブラリを作成したり，既存のタイプライブラリを編集することができます。通常，アプリケーション開発者はウィザードを使用して，タイプライブラリ内でエクスポーズされるオブジェクトを作成し，C++Builder が自動的にタイプライブラリを生成できるようにします。すると，自動生成されたタイプライブラリがタイプライブラリエディタ内で開かれるので，インターフェースの定義（または変更），型定義の追加などができます。

ただし，オブジェクトの定義にウィザードを使用しなくても，タイプライブラリエディタを使用すれば新しいタイプライブラリを定義できます。この場合，ウィザードによってタイプライブラリに関連付けられなかった CoClass のコードをタイプライブラリエディタが生成しないので，開発者が実装クラスを作成しなければなりません。

後述のように，タイプライブラリエディタはタイプライブラリの中の有効な型のサブセットをサポートします。

この節の最後では，以下の方法について説明します。

- 新規タイプライブラリを作成する
- 既存のタイプライブラリを開く
- タイプライブラリにインターフェースを追加する
- インターフェースを変更する
- タイプライブラリにプロパティとメソッドを追加する
- タイプライブラリに CoClass を追加する
- CoClass にインターフェースを追加する
- タイプライブラリに列挙型を追加する
- タイプライブラリにエリアスを追加する
- タイプライブラリにレコードまたは共用体を追加する
- タイプライブラリにモジュールを追加する
- タイプライブラリ情報を保存および登録する

有効な型

タイプライブラリエディタは，タイプライブラリで以下の IDL 型をサポートします。オートメーション互換の列は，Automation フラグまたは Dispinterface フラグにチェックマークが付いているインターフェースがその型を使えるかどうかを表します。これらは，COM がタイプライブラリを介して自動的にマーシャルできる型です。

IDL の型	バリアントの型	オートメーション互換	内容
short	VT_I2	あり	符号付き 2 バイト整数
long	VT_I4	あり	符号付き 4 バイト整数
single	VT_R4	あり	4 バイト実数
double	VT_R8	あり	8 バイト実数
CURRENCY	VT_CY	あり	通貨
DATE	VT_DATE	あり	日付
BSTR	VT_BSTR	あり	BSTR 文字列
IDispatch	VT_DISPATCH	あり	IDispatch インターフェースへのポインタ
SCODE	VT_ERROR	あり	OLE エラーコード
VARIANT_BOOL	VT_BOOL	あり	True = -1，False = 0
VARIANT	VT_VARIANT	あり	OLE バリアントへのポインタ
IUnknown	VT_UNKNOWN	あり	IUnknown インターフェースへのポインタ
DECIMAL	VT_DECIMAL	あり	16 バイト固定小数点
byte	VT_I1	なし *	符号付き 1 バイト整数
unsigned char	VT_UI1	あり	符号なし 1 バイト整数
unsigned short	VT_UI2	なし *	符号なし 2 バイト整数
unsigned long	VT_UI4	なし *	符号なし 4 バイト整数
__int64	VT_I8	なし	符号付き 8 バイト実数
uint64	VT_UI8	なし	符号なし 8 バイト実数
int	VT_INT	なし *	システム依存の整数（Win32 = Integer）
unsigned int	VT_UINT	なし *	システム依存の符号なし整数
void	VT_VOID	あり	C スタイルの VOID
HRESULT	VT_HRESULT	なし	32 ビットエラーコード
SAFEARRAY	VT_SAFEARRAY	あり	OLE Safe Array
LPSTR	VT_LPSTR	なし	ヌルで終わる文字列
LPWSTR	VT_LPWSTR	なし	ヌルで終わるワイド文字列

* 一部のアプリケーションではオートメーション互換

メモ unsigned char 型（VT_UI1）はオートメーション互換ですが，多くのオートメーションサーバーはこの値を正しく処理しないので，Variant または OleVariant では使えません。

メモ CORBA 開発に有効な型については，31-5 ページの「オブジェクトインターフェースを定義する」を参照してください。

これらの IDL 型のほかに，ライブラリに定義されているかまたは参照されるライブラリに定義されているインターフェースと型を，タイプライブラリ定義で使えます。

タイプライブラリエディタは，型情報を，生成されたタイプライブラリ（.TLB）ファイルにバイナリ形式で保存します。

SafeArray

COM では，SafeArray として知られる特別なデータ型を介して配列が渡される必要があります。SafeArray を作成および破棄するには，それを実行する特別な COM 関数を呼び出します。このとき，SafeArray 内のすべての要素は有効でオートメーション互換の型でなければなりません。

タイプライブラリエディタでは，SafeArray が要素の型を指定しなければなりません。たとえば，テキストページにおける次の行は，要素の型が long でパラメータが SafeArray のメソッドを宣言しています。

```
HRESULT _stdcall HighlightLines(SAFEARRAY(long) Lines);
```

メモ タイプライブラリエディタで SafeArray 型を宣言するときに要素の型を指定しなければなりませんが，生成された _TLB ユニット内の宣言は要素の型を示しません。

新規タイプライブラリの作成

特定の COM オブジェクトから独立したタイプライブラリの作成が必要になる場合もあります。たとえば，ほかのいくつかのタイプライブラリで使用する型定義が入るタイプライブラリを定義したい場合などです。その場合は，基本的定義のタイプライブラリを作成し，これをその他のタイプライブラリの［使用］ページに追加することができます。

まだ実装していないオブジェクトのタイプライブラリも作成できます。いったんタイプライブラリにインターフェース定義を入れると，COM オブジェクトウィザードを使用して CoClass と実装を生成することができます。

新規タイプライブラリを作成する手順は次のとおりです。

1. ［ファイル｜新規作成｜その他］を選択して［新規作成］ダイアログボックスを開きます。
2. ［ActiveX］ページを選択します。
3. ［タイプライブラリ］アイコンを選択します。
4. ［OK］を押します。

   タイプライブラリエディタが開きます。
5. タイプライブラリ名を入力します。作成したタイプライブラリに要素を追加します。

既存のタイプライブラリを開く

ウィザードを使って ActiveX コントロール，オートメーションオブジェクト，ActiveForm，ASP オブジェクト，COM オブジェクト，トランザクションオブジェクト，リモートデータモジュール，またはトランザクションデータモジュールを作成すると，実装ユニットとともにタイプライブラリが自動的に作成されます。さらに，システム上で使用可能なその他の製品（サーバー）に関連付けられたタイプライブラリがある場合があります。

現在プロジェクトの一部になっていないタイプライブラリを開く手順は次のとおりです。

1. IDE のメインメニューから［ファイル｜開く］を選択します。
2. ［開く］ダイアログボックスで［ファイルの種類］をタイプライブラリに設定します。
3. 目的のタイプライブラリファイルを選択し，［開く］を選択します。

現在のプロジェクトに関連付けられたタイプライブラリを開くには，次の手順を実行します。

1. ［表示｜タイプライブラリ］を選択します。

ここで，インターフェース，CoClass，および列挙型，プロパティ，メソッドなどのタイプライブラリのほかの要素を追加できます。

参考 クライアントアプリケーションを作成する場合は，タイプライブラリを開く必要はありません。必要なのは，タイプライブラリエディタがタイプライブラリから作成する Project_TLB ユニットだけであり，タイプライブラリ自体は必要ありません。このファイルはクライアントプロジェクトに直接追加できます。または，タイプライブラリがシステム上に登録されている場合は，［タイプライブラリの取り込み］ダイアログ（［プロジェクト｜タイプライブラリの取り込み］）を使用できます。

タイプライブラリにインターフェースを追加する

インターフェースを追加する手順は次のとおりです。

1. ツールバーの［インターフェース］アイコンを選択します。

   インターフェースがオブジェクトリストペインに追加され，名前の追加を求められます。

2. インターフェース名を入力します。

新規インターフェースには，必要に応じて変更できるデフォルト属性が入ります。

インターフェースの目的に合ったプロパティ（取得 / 設定関数で表される）とメソッドを追加できます。

タイプライブラリを使ってインターフェースを変更する

作成されたインターフェースまたはディスパッチインターフェースを変更するには，以下のようにいくつかの方法があります。

- 変更したい情報が入った型情報のページを使って，インターフェースの属性を変更できる。オブジェクトリストペインでインターフェースを選択し，適切な型情報ページのコントロールを使用する。たとえば，［属性］ページを使って親インターフェースを変更したり，［フラグ］ページを使ってデュアルインターフェースかどうかを変更する場合である
- オブジェクトリストペインでインターフェースを選択し，［テキスト］ページで宣言を編集することによって，インターフェース宣言を直接編集できる
- インターフェースにプロパティとメソッドを追加できる（次の節を参照）
- 型情報を変更することによって，インターフェース内にすでにあるプロパティとメソッドを変更できる
- オブジェクトリストペインで CoClass を選択し，［実装するインターフェース］ページを右クリックし，［インターフェースの追加］を選択することによって，CoClass と関連付けできる

ウィザードが生成した CoClass にインターフェースを関連付ける場合は，ツールバーの［更新］ボタンを選択すれば，変更を実装ファイルに適用するようにタイプライブラリエディタに指示できます。

インターフェースまたはディスパッチインターフェースにプロパティとメソッドを追加する

インターフェースまたはディスパッチインターフェースにプロパティまたはメソッドを追加する手順は次のとおりです。

1. インターフェースを選択し，ツールバーから［プロパティ］アイコンまたは［メソッド］アイコンを選択します。プロパティを追加する場合は，［プロパティ］アイコンを直接選択して（取得 / 設定メソッドの両方を持つ）読み出し / 書き込みプロパティを作成するか，下矢印を選択してプロパティ型のメニューを表示します。

   プロパティアクセスメソッドメンバーまたはメソッドメンバーがオブジェクトリストペインに追加され，名前の追加を求められます。

2. メンバー名を入力します。

新規メンバーの［属性］，［パラメータ］，［フラグ］ページにはデフォルト設定が入り，そのメンバーに合わせて設定を変更できます。たとえば，［属性］ページでプロパティに型を指定できます。メソッドを追加する場合は，［パラメータ］ページでパラメータを指定できます。

もう 1 つの方法としては，IDL 構文を使って，直接［テキスト］ページに入力することによって，プロパティとメソッドを追加することができます。たとえば，インターフェースの［テキスト］ページには次のプロパティ宣言を入力できます。

```
[
  uuid(5FD36EEF-70E5-11D1-AA62-00C04FB16F42),
  version(1.0),
  dual,
  oleautomation
]
interface Interface1: IDispatch
{ // 中カッコで囲まれたものをすべて追加
[propget, id(0x00000002)]
  HRESULT _stdcall AutoSelect([out, retval] long Value );
  [propget, id(0x00000003)]
  HRESULT _stdcall AutoSize([out, retval] VARIANT_BOOL Value );
  [propput, id(0x00000003)]
  HRESULT _stdcall AutoSize([in] VARIANT_BOOL Value );
};
```

インターフェースの［テキスト］ページを使ってインターフェースに追加されたメンバーは，オブジェクトリストペイン内に別個の項目として表示され，それぞれに独自の［属性］，［フラグ］，［パラメータ］ページがあります。新規のプロパティまたはメソッドをそれぞれ変更するには，オブジェクトリストペインで選択してからこれらのページを使用するか，［テキスト］ページで直接編集します。

ウィザードが生成した CoClass にインターフェースを関連付ける場合は，ツールバーの［更新］ボタンを選択すれば，変更を実装ファイルに適用するようにタイプライブラリエディタに指示できます。タイプライブラリエディタは新規のメソッドを実装クラスに追加して，新規のメンバーを反映します。この新規のメソッドを実装ユニットのソースコード内で探し出し，中身を埋めると実装が完成します。

タイプライブラリに CoClass を追加する

もっとも簡単に CoClass をプロジェクトに追加するには，IDE のメインメニューから［ファイル｜新規作成｜その他］を選択し，［新規作成］ダイアログの［ActiveX］または［多層サポート］ページ上の適切なウィザードを使用します。この方法の長所は，ウィザードが CoClass とそのインターフェースをタイプライブラリに追加する点だけでなく，ウィザードが実装ユニットを追加し，この新規の実装ユニットを含むようにプロジェクトファイルを更新し，プロジェクトファイル内のオブジェクトマップに新規の CoClass を追加する点です。

ただし，ウィザードを使用しなくても，CoClass を作成できます。それには，ツールバーの［CoClass］アイコンを選択し，その属性を指定します。（［属性］ページを使うと）この新規の CoClass に名前を付けることができます。また，［フラグ］ページを使うと，CoClass がアプリケーションオブジェクトであるかどうか，ActiveX コントロールであるかどうかなどの情報を指定できます。

メモ タイプライブラリへの CoClass の追加にウィザードではなくツールバーを使う場合は，自分で CoClass の実装を生成し，その CoClass のインターフェースの要素を 1 つ変更するたびに手動で実装を更新しなければなりません。手動で CoClass の実装を追加する場合は，プロジェクトファイル内のオブジェクトマップに新規の CoClass を追加する必要があります。

CoClass にメンバーを直接追加することはできません。そのかわりに，CoClass にインターフェースを追加するときに暗黙にメンバーが追加されます。

CoClass にインターフェースを追加する

CoClass は，クライアントに提供するインターフェースによって定義されます。CoClass の実装クラスにはプロパティとメソッドをいくつでも追加できますが，CoClass に関連付けられたインターフェースがエクスポーズするプロパティとメソッドしか，クライアントは見ることができません。

CoClass にインターフェースを関連付けるには，そのクラスの［実装するインターフェース］ページを右クリックし，［インターフェースの追加］を選択して，選択できるインターフェースのリストを表示します。このリストには，現在のタイプライブラリおよび現在のタイプライブラリが参照するタイプライブラリに定義されているインターフェースが表示されます。クラスの実装先インターフェースを選択します。そのインターフェースは，GUID およびその他の属性とともにそのページに追加されます。

CoClass がウィザードによって生成された場合，この方法で追加したインターフェースの（プロパティアクセスメソッドなどの）メソッドのスケルトンメソッドを含むように，実装クラスがタイプライブラリエディタによって自動的に更新されます。

タイプライブラリに列挙型を追加する

タイプライブラリに列挙型を追加する手順は次のとおりです。

1. ツールバーの［Enum］アイコンをクリックします。

   列挙型がオブジェクトリストペインに追加され，名前の追加を求められます。

2. 列挙型名を入力します。

   新規列挙型は空で，［属性］ページには変更できるデフォルト属性が表示されます。

［Const］ボタンをクリックして，列挙型に値を追加します。その後それぞれの値を選択し，［属性］ページを使って名前を付けます（場合によっては値も指定します）。

列挙型を追加すると，追加先のタイプライブラリ，または［使用］ページでこれを参照するほかのタイプライブラリが，この新規の型を使用できるようになります。たとえば，プロパティまたはパラメータの型として列挙型を使用できます。

タイプライブラリにエリアスを追加する

タイプライブラリにエリアスを追加する手順は次のとおりです。

1. ツールバーの［Alias］アイコンをクリックします。

   エリアス型がオブジェクトリストペインに追加され，名前の追加を求められます。

2. エリアス名を入力します。

   デフォルトでは，新規のエリアスは long 型を表します。自分が希望する型に変更するには，［属性］ページを使用します。

エリアスを追加すると，追加先のタイプライブラリ，または［使用］ページでこれを参照するほかのタイプライブラリが，この新規の型を使用できるようになります。たとえば，プロパティまたはパラメータの型としてエリアスを使用できます。

タイプライブラリにレコードまたは共用体を追加する

タイプライブラリにレコードまたは共用体を追加する手順は次のとおりです。

1. ツールバーの［Record］アイコンまたは［Union］アイコンをクリックします。

   選択した型の要素がオブジェクトリストペインに追加され，名前の追加を求められます。

2. レコード名または共用体名を入力します。

   この時点で，新規のレコードまたは共用体にはフィールドがありません。

3. オブジェクトリストペインでレコードまたは共用体を選択して，ツールバーの［Field］アイコンをクリックします。［属性］ページを使用してこのフィールドの名前と型を指定します。

4. 必要なフィールドの数だけ手順 3 を繰り返します。

レコードまたは共用体を定義すると，追加先のタイプライブラリ，または［使用］ページでこれを参照するほかのタイプライブラリが，この新規の型を使用できるようになります。たとえば，プロパティまたはパラメータの型としてレコードまたは共用体を使用できます。

タイプライブラリにモジュールを追加する

タイプライブラリにモジュールを追加する手順は次のとおりです。

1. ツールバーの［Module］アイコンをクリックします。

   選択したモジュールがオブジェクトリストペインに追加され，名前の追加を求められます。

2. モジュール名を入力します。

3. ［属性］ページで，そのモジュールがエントリポイントを表す DLL の名前を指定します。

4. 手順 3 で指定した DLL からメソッドを追加します。それには，ツールバーの［Method］アイコンをクリックし，［属性］ページを使用してそのメソッドを記述します。
5. そのモジュールが定義する必要がある定数を追加するために，ツールバーの［Const］アイコンをクリックします。定数ごとに，名前，型，および値を指定します。

タイプライブラリ情報の保存と登録

タイプライブラリを変更したら，タイプライブラリ情報を保存，登録します。

タイプライブラリを保存すると，以下のものが自動的に更新されます。

- バイナリタイプライブラリファイル（.tlb 拡張子）
- その内容を表す Project_TLB ユニット
- ウィザードが生成した CoClass の実装コード

メモ タイプライブラリは個別のバイナリ（.TLB）ファイルとして格納されますが，サーバー（.EXE, DLL，または .OCX）にもリンクされます。

タイプライブラリエディタでは，タイプライブラリ情報の格納のオプションを利用できます。どのオプションを選択するかは，以下のようにタイプライブラリ実装の段階に応じて異なります。

- ［保存］は，.TLB と Project_TLB ユニットの両方をディスクに保存する
- ［更新］は，タイプライブラリユニットをメモリ上でのみ更新する
- ［登録］は，タイプライブラリのエントリをシステムの Windows レジストリに追加する。これは，.TLB が関連付けられたサーバーが自身を登録するときに自動的に実行される
- ［書き出し］は，型とインターフェースの定義が IDL 構文で収められた .IDL ファイルを保存する

上記の方法はすべて，構文チェックを実行します。タイプライブラリを更新，登録，または保存すると，C++Builder は，ウィザードを使って作成された CoClass の実装ユニットを自動的に更新します。

タイプライブラリの保存

タイプライブラリを保存すると，以下のことが行われます。

- 構文と妥当性の検査が実行される
- 情報が .TLB ファイルに保存される
- 情報が Project_TLB ユニットに保存される
- ウィザードによって生成された CoClass にタイプライブラリが関連付けられている場合は，実装を更新するように IDE のモジュールマネージャに通知される

タイプライブラリを保存するには，C++Builder のメインメニューから［ファイル｜上書き保存］を選択します。

タイプライブラリの更新

タイプライブラリを更新すると，以下のことが実行されます。

- 構文検査が実行される
- C++Builder のタイプライブラリユニットがメモリ上にのみ再生成される。ファイルはディスクには保存されない

- ウィザードによって生成された CoClass にタイプライブラリが関連付けられている場合は，実装を更新するように IDE のモジュールマネージャに通知される

タイプライブラリを更新するには，タイプライブラリエディタツールバーの［更新］アイコンを選択します。

メモ タイプライブラリの項目の名前を変更した場合，実装を更新すると，重複するエントリが作成されることがあります。この場合，コードを正しいエントリに移動して，重複するエントリを削除しなければなりません。同様に，タイプライブラリの項目を削除した場合，実装を更新すると，削除したエントリが CoClass から除去されずに残ります。なぜなら，クライアントから見えなくなるようにしただけだとみなされるからです。不要になった項目は，実装ユニットから手動で削除しなければなりません。

タイプライブラリの登録

通常は，タイプライブラリを明示的に登録する必要はありません。なぜなら，COM サーバーアプリケーションを登録するときに自動的に登録されるからです（41-17 ページの「COM オブジェクトを登録する」を参照）。ただし，タイプライブラリウィザードを使ってタイプライブラリを作成すると，サーバーオブジェクトに関連付けられません。この場合は，ツールバーを使って直接タイプライブラリを登録できます。

タイプライブラリを登録すると以下のことが行われます。

- 構文検査が実行される
- そのタイプライブラリに関するエントリを Windows レジストリに追加する

タイプライブラリを登録するには，タイプライブラリエディタツールバーの［登録］アイコンを選択します。

IDL ファイルのエクスポート

タイプライブラリをエクスポートすると，以下のことが実行されます。

- 構文検査が実行される
- 型情報宣言を含む IDL ファイルが作成される。このファイルは型情報を Microsoft IDL で記述する

タイプライブラリをエクスポートするには，タイプライブラリエディタのツールバーから［書き出し］アイコンを選択します。

タイプライブラリの配布

ActiveX またはオートメーションサーバープロジェクトの一部としてタイプライブラリを作成すると，デフォルトで，タイプライブラリがリソースとして .DLL，.OCX，または .EXE ファイル内に自動的にリンクされます。

ただし，C++Builder がタイプライブラリを保持するため，必要に応じて，タイプライブラリを個別の .TLB としてアプリケーションとともに配布できます。

従来，オートメーションアプリケーションのタイプライブラリは，.TLB 拡張子の付いた個別のファイルとして格納されていました。現在では，一般的なオートメーションアプリケーションは，タイプライブラリを直接 .OCX ファイルまたは .EXE ファイルにバインドします。オペレーティングシステムは，タイプライブラリを，実行形式（.DLL，.OCX，または .EXE）ファイル内の最初のリソースとして取得します。

主なプロジェクト以外のタイプライブラリをアプリケーション開発者が使えるようにする場合，タイプライブラリを以下のいずれかの形式にすることができます。

- リソース。このリソースには，TYPELIB 型と整数の ID が必要である。リソースコンパイラを使ってタイプライブラリを構築した場合は，このリソースをリソース（.RC）ファイルの中で次のように宣言しなければならない。

  ```
  1 typelib mylib1.tlb
  2 typelib mylib2.tlb
  ```

  1 つの ActiveX ライブラリには複数のタイプライブラリリソースを入れることができます。アプリケーション開発者は，リソースコンパイラを使用して .TLB ファイルを ActiveX ライブラリに追加することができます。

- スタンドアロンのバイナリファイル。タイプライブラリエディタが出力する .TLB ファイルはバイナリファイルになる

第40章

COM クライアントの作成

COM クライアントとは，ほかのアプリケーションまたはライブラリが実装する COM オブジェクトを使用するアプリケーションです。もっとも一般的なものは，オートメーションサーバーを制御するアプリケーション（オートメーションコントローラ），および ActiveX コントロールのホストであるアプリケーション（ActiveX コンテナ）です。

一見したところ，これら 2 種類の COM クライアントは大きく異なるように見えます。典型的なオートメーションコントローラは，外部サーバー EXE を起動し，そのサーバーにタスクを実行させるコマンドを発行します。このオートメーションサーバーは，通常は非ビジュアルのアウトオブプロセスサーバーです。一方，典型的な ActiveX クライアントは，ビジュアルコントロールのホストであり，コンポーネントパレットのコントロールと同じようにこのコントロールを使用します。ActiveX サーバーは常にインプロセスサーバーです。

このように大きな違いはありますが，これら 2 種類の COM クライアントを作成する作業はきわめて似ています。なぜなら，クライアントアプリケーションがサーバーオブジェクトに対するインターフェースを取得し，そのプロパティとメソッドを使用するからです。C++Builder は，クライアント上のコンポーネントでサーバー CoClass をラップできるようにして，この作業を著しく簡単にします。このコンポーネントはコンポーネントパレットにインストールすることさえできます。このようなコンポーネントラッパーのサンプルは，コンポーネントパレットの 2 ページにわたって表示されています。ActiveX ラッパーのサンプルは［ActiveX］ページに表示され，オートメーションオブジェクトのサンプルは［Servers］ページに表示されています。

COM クライアントを作成するときは，サーバーがクライアントにエクスポーズするインターフェースを理解しなければなりません。コンポーネントパレットのコンポーネントをアプリケーションに使用するときに，そのプロパティとメソッドを理解しなければならないのと同じです。このインターフェース（またはインターフェースのセット）はサーバーアプリケーションによって決定され，通常はタイプライブラリ内でパブリッシュに設定されます。特定のサーバーアプリケーションのパブリッシュに設定されたインターフェースについての詳細は，サーバーアプリケーション付属のマニュアルを参照してください。

コンポーネントラッパーでサーバーオブジェクトをラップしてコンポーネントパレットにインストールするように選択しない場合でも，アプリケーションがインターフェース定義を使用できるようにしなければなりません。そのために，サーバーの型情報をインポートすることができます。

メモ COM API を使って型情報を直接問い合わせることもできますが，C++Builder ではこれに対する特殊なサポートは用意されていません。

オブジェクトのリンクと埋め込み（OLE）などの一部の古い COM 技術は，タイプライブラリ内に型情報を提供しません。そのかわりに，定義済みインターフェースの標準セットに依存しています。これらについては，40-17 ページの「タイプライブラリを持たないサーバーのクライアントの作成」を参照してください。

タイプライブラリの情報のインポート

COM サーバーについての情報をクライアントアプリケーションが使用できるようにするには，サーバーのタイプライブラリに格納されるサーバーについての情報をインポートしなければなりません。すると，その結果として生成されるクラスをアプリケーションが使用して，サーバーオブジェクトを制御できるようになります。

タイプライブラリの情報をインポートする方法は次のとおりです。

- ［タイプライブラリの取り込み］ダイアログを使用すると，サーバー型，オブジェクト，およびインターフェースについての使用可能なすべての情報をインポートできる。これはもっとも一般的な方法である。なぜなら，どのタイプライブラリからも情報をインポートできる上に，タイプライブラリ内で Hidden，Restricted，または PreDeclID のフラグが付いていないすべての作成可能な CoClass 用のコンポーネントラッパーをオプションで生成できるからである
- ActiveX コントロールのタイプライブラリからインポートする場合は，［ActiveX コントロールの取り込み］ダイアログを使用できる。この方法でもインポートされる型情報は同じだが，ActiveX コントロールを表す CoClass 用のコンポーネントラッパーしか作成されない
- コマンドラインユーティリティ tlibimp.exe を使用する。このユーティリティは，IDE 内では使用できない環境設定オプションを提供する
- ウィザードで作成したタイプライブラリを，メニュー項目の［タイプライブラリの取り込み］と同じメカニズムで自動的に取り込む

どの方法でタイプライブラリ情報をインポートしても，その結果表示されるダイアログで TypeLibName_TLB という名前のユニットが作成され，TypeLibName にはタイプライブラリの名前が入ります。このファイルには，タイプライブラリ内で定義されているクラス，型，およびインターフェースの宣言が入ります。このファイルをプロジェクトに含めることによって，アプリケーションがこれらの定義を使用できるようになるので，オブジェクトの作成やオブジェクトのインターフェースの呼び出しが可能になります。IDE 側で，このファイルを再作成することがあります。このため，このファイルに手作業で変更を加えるのは避けるようにしてください。

このダイアログによって TypeLibName_TLB ユニットに型定義が追加されるだけでなく，タイプライブラリで定義されている CoClass 用の VCL クラスラッパーも作成され，TypeLibName_OCX という名前の別のユニットに入れられます。［タイプライブラリの取り込み］ダイアログを使用する場合，

これらのラッパーはオプションです。［ActiveX コントロールの取り込み］ダイアログを使用する場合，コントロールを表すすべての CoClass に対して常にラッパーが生成されます。

メモ コンポーネントラッパーを生成する場合は，［ActiveX コントロールの取り込み］ダイアログによって TypeLibName_OCX ユニットが生成されますが，プロジェクトには追加されません（TypeLibName_TLB ユニットだけがプロジェクトに追加されます）。TypeLibName_OCX ユニットをプロジェクトに明示的に追加するには，［プロジェクト｜プロジェクトに追加］を選択します。

生成されたクラスラッパーは，アプリケーションに対して CoClass を表し，そのインターフェースのプロパティとメソッドをエクスポーズします。CoClass がイベント生成のインターフェース（IConnectionPointContainer および IConnectionPoint）をサポートする場合，VCL クラスラッパーはイベントシンクを作成するので，イベントに対するイベントハンドラの割り当てが，ほかのコンポーネントに対する割り当てと同じように簡単にできます。生成された VCL クラスをコンポーネントパレットにインストールするように指示した場合は，オブジェクトインスペクタを使用してプロパティ値とイベントハンドラを割り当てることができます。

メモ ［タイプライブラリの取り込み］ダイアログでは，COM+ イベントオブジェクト用のクラスラッパーは作成されません。COM+ イベントオブジェクトが生成するイベントに応答するクライアントを作成するには，プログラム的にイベントシンクを作成しなければなりません。このプロセスについては，40-16 ページの「COM+ イベントの処理」を参照してください。

タイプライブラリをインポートするときに生成されるコードについての詳細は，40-5 ページの「タイプライブラリ情報のインポート時に生成されるコード」を参照してください。

［タイプライブラリの取り込み］ダイアログの使い方

タイプライブラリをインポートする手順は次のとおりです。

1. ［プロジェクト｜タイプライブラリの取り込み］を選択します。
2. リストからタイプライブラリを選択します。

   システムに登録されているすべてのライブラリがダイアログにリストされます。タイプライブラリがリストにない場合は，［追加］ボタンを選択して，タイプライブラリ（TLB）ファイルを検索，選択し，［OK］を選択します。これによってタイプライブラリが登録され，使用可能になります。手順 2 を繰り返します。選択するタイプライブラリは，スタンドアロンのタイプライブラリファイル（.tlb，.olb），またはタイプライブラリを提供するサーバー（.dll，.ocx，.exe）のどちらでもかまいません。
3. タイプライブラリ内の CoClass をラップする VCL コンポーネントを生成したい場合は，［コンポーネントラッパーの作成］にチェックマークを付けます。このコンポーネントを生成しなくても，TypeLibName_TLB ユニット内の定義を使用すれば CoClass を使用できます。ただし，サーバーオブジェクトを作成する呼び出しを独自に作成し，必要であれば，イベントシンクをセットアップしなければなりません。

   ［タイプライブラリの取り込み］ダイアログがインポートする CoClass は，CanCreate フラグが付いていて，Hidden，Restricted，または PreDeclID フラグが付いていないものだけです。コマンドラインユーティリティの tlibimp.exe を使うと，これらのフラグをオーバーライドできます。

4. 生成されたコンポーネントラッパーをコンポーネントパレットにインストールしたくない場合は，［ユニットの作成］を選択します。これによって TypeLibName_TLB ユニットが生成されます。手順 3 で［コンポーネントラッパーの作成］にチェックマークを付けた場合は，TypeLibName_OCX ユニットが生成されます。これで［タイプライブラリの取り込み］ダイアログが終了します。
5. 生成されたコンポーネントラッパーをコンポーネントパレットにインストールしたい場合は，このコンポーネントを配置する［パレットページ名］を選択してから［インストール］を選択します。［ユニットの作成］ボタンを選択した場合と同様に，TypeLibName_TLB ユニットと TypeLibName_OCX ユニットが生成されます。そして［インストール］ダイアログが表示されるので，そのコンポーネントを配置するパッケージ（既存または新規のパッケージ）を指定できます。このボタンは，そのタイプライブラリについてコンポーネントを作成できないときは淡色表示されます。

［タイプライブラリの取り込み］ダイアログを終了すると，［ユニットディレクトリ名］コントロールで指定されたディレクトリに新規の TypeLibName_TLB ユニットと TypeLibName_OCX ユニットが生成されます。TypeLibName_TLB ユニットには，タイプライブラリ内で定義された要素の宣言が入ります。［コンポーネントラッパーの作成］にチェックマークを付けた場合は，生成されたコンポーネントラッパーが TypeLibName_OCX ユニットに入ります。

さらに，生成されたコンポーネントラッパーをインストールした場合は，タイプライブラリに記述されていたサーバーオブジェクトはコンポーネントパレットに配置されます。オブジェクトインスペクタを使って，プロパティを設定したり，サーバーのイベントハンドラを書くことができます。フォームまたはデータモジュールにこのコンポーネントを追加した場合は，設計時に右クリックすると，（プロパティページをサポートしていれば）プロパティページを表示できます。

メモ コンポーネントパレットの［Servers］ページには，この方法でインポートされたオートメーションサーバーの例が多数入っています。

［ActiveX コントロールの取り込み］ダイアログの使い方

ActiveX コントロールをインポートする手順は次のとおりです。

1. ［コンポーネント｜ ActiveX コントロールの取り込み］を選択します。
2. リストからタイプライブラリを選択します。

   ActiveX コントロールを定義し，登録されているすべてのライブラリがダイアログにリストされます（これは［タイプライブラリの取り込み］ダイアログにリストされたライブラリのサブセットです）。タイプライブラリがリストにない場合は，［追加］ボタンを選択して，タイプライブラリ（TLB）ファイルを検索，選択し，［OK］を選択します。これによってタイプライブラリが登録され，使用可能になります。手順 2 を繰り返します。選択するタイプライブラリは，スタンドアロンのタイプライブラリファイル（.tlb，.olb），または ActiveX サーバー（.dll，.ocx）のどちらでもかまいません。
3. ActiveX コントロールをコンポーネントパレットにインストールしたくない場合は，［ユニットの作成］を選択します。これによって TypeLibName_TLB ユニットと TypeLibName_OCX ユニットが生成されます。これで［ActiveX コントロールの取り込み］ダイアログが終了します。

4. ActiveX コントロールをコンポーネントパレットにインストールしたい場合は，このコンポーネントを配置する［パレットページ名］を選択してから［インストール］を選択します。［ユニットの作成］ボタンを選択した場合と同様に，TypeLibName_TLB ユニットと TypeLibName_OCX ユニットが生成されます。そして［インストール］ダイアログが表示されるので，そのコンポーネントを配置するパッケージ（既存または新規のパッケージ）を指定できます。

［ActiveX コントロールの取り込み］ダイアログを終了すると，［ユニットディレクトリ名］コントロールで指定されたディレクトリに新規の TypeLibName_TLB ユニットと TypeLibName_OCX ユニットが生成されます。TypeLibName_TLB ユニットには，タイプライブラリ内で定義された要素の宣言が入ります。TypeLibName_OCX ユニットには，ActiveX コントロール用に生成されたコンポーネントラッパーが入ります。

メモ ［タイプライブラリの取り込み］ダイアログではオプションですが，［ActiveX コントロールの取り込み］ダイアログでは常にコンポーネントラッパーが生成されます。なぜなら，ActiveX コントロールはビジュアルコントロールなので，VCL フォームに収まることができるようにするには，コンポーネントラッパーの追加サポートが必要だからです。

生成されたコンポーネントラッパーをインストールした場合は，ActiveX コントロールはコンポーネントパレットに配置されます。オブジェクトインスペクタを使って，プロパティを設定したり，このコントロールのイベントハンドラを書くことができます。フォームまたはデータモジュールにこのコントロールを追加した場合は，設計時に右クリックすると，（プロパティページをサポートしていれば）プロパティページを表示できます。

メモ コンポーネントパレットの［ActiveX］ページには，この方法でインポートされた ActiveX コントロールの例が多数入っています。

タイプライブラリ情報のインポート時に生成されるコード

タイプライブラリをインポートすると，生成された TypeLibName_TLB ユニットを見ることができます。そのユニットのソースファイルは，タイプライブラリおよびそのインターフェースと CoClass の GUID にシンボル名を与える定数を定義します。これらの定数の名前は，以下のように生成されます。

- タイプライブラリの GUID は LIBID_TypeLibName の形式になる。TypeLibName はそのタイプライブラリの名前
- インターフェースの GUID は IID_InterfaceName の形式になる。InterfaceName はそのインターフェースの名前
- ディスパッチインターフェースの GUID は DIID_InterfaceName の形式になる。InterfaceName はそのディスパッチインターフェースの名前
- CoClass の GUID は CLSID_ClassName の形式になる。ClassName はその CoClass の名前

ソースファイルの中を右クリックし，［ソース / ヘッダーファイルを開く］を選択すると，以下の定義が表示されます。

- タイプライブラリ内の CoClass の宣言。これは各 CoClass をデフォルトインターフェースにマップする。さらに，TComInterface テンプレートを使用して，各 CoClass のラッパーが宣言される。このラッパーの名前は，CoClassNamePtr の形式になる

- タイプライブラリ内のインターフェースとディスパッチインターフェースの宣言
- インターフェースおよびディスパッチインターフェースのクラスラッパーの宣言。インターフェースの場合，クラスラッパーは TComInterface テンプレートを使用し，名前は TComInterfaceName の形式になる。ディスパッチインターフェースの場合，クラスラッパーは TAutoDriver テンプレートを使用し，名前は InterfaceNameDisp の形式になる
- デフォルトインターフェースが仮想テーブルバインディングをサポートする各 CoClass のクリエータクラスの宣言。このクリエータクラスには，Create と CreateRemote の 2 つの静的メソッドがあり，これらを使用すると，ローカル（Create）またはリモート（CreateRemote）で CoClass をインスタンス化できる。これらのメソッドは，CoClass のデフォルトインターフェースのクラスラッパーを返す（デフォルトインターフェースは TComInterface テンプレートを使用して TypeLibName_TLB.h で定義される）
- あらゆるイベントインターフェースのプロクシークラスラッパー。このクラスラッパーは名前が TEvents_CoClassName の形式になる。CoClassName はイベントを生成する CoClass の名前。このプロクシークラスは，すべてのクライアントイベントシンクのリストを保持し，サーバーオブジェクトがイベントを送出するときにすべての適切なイベントシンク上で適切なメソッドを呼び出す

これらの宣言は，CoClass のインスタンスの作成とそのインターフェースのアクセスに必要なものを提供します。開発者が行う必要があることは，CoClass にバインドしてインターフェースを呼び出したいユニット内に，生成された TypeLibName_TLB.h ファイルをインクルードすることだけです。

警告 生成されたラッパークラスは，静的オブジェクト（TInitOleT）を使用して COM の初期化を処理します。複数のスレッドからオブジェクトを作成する場合は，これによって問題が生じる可能性があります。なぜなら，サーバーに接続する最初のスレッド上で COM が初期化されるだけだからです。COM サーバーに接続するその他すべてのスレッド上で，TInitOleT または TInitOle の別個のインスタンスを明示的に作成しなければなりません。

メモ タイプライブラリエディタか，コマンドラインユーティリティの TLIBIMP を使用したときも，TypeLibName_TLB ユニットが生成されます。

ActiveX コントロールを使用したい場合は，前述した宣言のほかにも，生成された VCL ラッパーが必要です。VCL ラッパーは，そのコントロールのウィンドウ管理に関する処理を行います。［タイプライブラリの取り込み］ダイアログでその他の CoClass の VCL ラッパーを生成した場合もあります。この VCL ラッパーは，サーバーオブジェクトの作成タスクとそのメソッドの呼び出しタスクを簡単にします。クライアントアプリケーションがイベントに応答するようにしたい場合は，これを使用することを特にお勧めします。

生成された VCL ラッパーの宣言は，TypeLibName_OCX ユニットに表示されます。ActiveX コントロールのコンポーネントラッパーは，TOleControl の下位オブジェクトです。オートメーションオブジェクトのコンポーネントラッパーは，TOleServer の下位オブジェクトです。生成されたコンポーネントラッパーは，CoClass のインターフェースがエクスポーズするプロパティ，イベント，およびメソッドを追加します。このコンポーネントは，ほかの VCL コンポーネントと同じように使用できます。

警告 生成された TypeLibName_TLB ユニットまたは TypeLibName_OCX ユニットを編集してはいけません。これらは，タイプライブラリが更新されるたびに再生成されるので，変更しても上書きされます。

メモ 生成されたコードについての最新情報は，自動生成される TypeLibName_TLB ユニットおよび utilcls.h 内のコメントを参照してください。

インポートされたオブジェクトの制御

タイプライブラリ情報のインポートを終えると，インポートされたオブジェクトのプログラミングを開始できるようになります。どのような方法で進めるかは，ある部分はオブジェクトに応じて決まり，ある部分はコンポーネントラッパーの作成を選択したかどうかに応じて決まります。

コンポーネントラッパーの使い方

サーバーオブジェクトのコンポーネントラッパーを生成した場合，COM クライアントアプリケーションの作成は，VCL コンポーネントが入ったほかのアプリケーションの作成と大差ありません。サーバーオブジェクトのプロパティ，メソッド，およびイベントは，すでに VCL コンポーネントにカプセル化されています。イベントハンドラを割り当て，プロパティ値を設定し，メソッドを呼び出すだけで済みます。

サーバーオブジェクトのプロパティ，メソッド，およびイベントの使い方については，サーバーのマニュアルを参照してください。コンポーネントラッパーは，可能であればデュアルインターフェースを自動的に提供します。C++Builder は，タイプライブラリ内の情報から仮想テーブルのレイアウトを判断します。

さらに，新規のコンポーネントは，基本クラスから特定の重要なプロパティとメソッドを継承します。

ActiveX ラッパー

ActiveX コントロールのホストには常にコンポーネントラッパーを使用してください。なぜなら，コンポーネントラッパーは ActiveX コントロールのウィンドウを VCL フレームワークに組み込むからです。

ActiveX コントロールが TOleControl から継承するプロパティとメソッドを使うと，基底のインターフェースにアクセスしたり，そのコントロールについての情報を取得することができます。ただし，大部分のアプリケーションでは，これらを使用する必要はありません。そのかわりに，ほかの VCL コントロールを使用する場合と同じように，インポートされたコントロールを使用します。

通常，ActiveX コントロールは，プロパティを設定できるプロパティページを提供します。プロパティページは，フォームデザイナでコンポーネントをダブルクリックしたときに，一部のコンポーネントで表示されるコンポーネントエディタに似ています。ActiveX コントロールのプロパティページを表示するには，右クリックして［プロパティ］を選択します。

大部分のインポートされた ActiveX コントロールの使い方は，サーバーアプリケーションによって決まります。ただし，ActiveX コントロールがデータベース項目のデータを表す場合は，標準セットの通知を使用します。このような ActiveX コントロールのホスト方法については，40-9 ページの「データベース対応の ActiveX コントロールの使い方」を参照してください。

オートメーションオブジェクトラッパー

オートメーションオブジェクトのラッパーを使うと，サーバーオブジェクトに対する接続をどのように形成するかを，以下のように制御できます。

- ConnectKind プロパティは，サーバーがローカルまたはリモートのいずれであるか，およびすでに動作中のサーバーに接続するか新規のインスタンスを起動するかを示す。リモートサーバーに接続する場合は，RemoteMachineName プロパティを使用してマシン名を指定しなければならない
- いったん ConnectKind を指定すると，以下の 3 つの方法でコンポーネントをサーバーに接続できる
  - コンポーネントの Connect メソッドを呼び出してサーバーに明示的に接続できる
  - AutoConnect プロパティを **true** に設定して，アプリケーションの起動時に自動的にコンポーネントを接続できる
  - サーバーに明示的に接続する必要はない。コンポーネントを使ってサーバーのプロパティまたはメソッドの 1 つを使用すれば，自動的に接続が形成される

以下のように，メソッドの呼び出しまたはプロパティのアクセスの方法は，その他のコンポーネントを使用する場合と同じです。

```
TServerComponent1->DoSomething();
```

オブジェクトインスペクタを使用してイベントハンドラを作成できるので，簡単にイベントを処理できます。ただし，コンポーネント上のイベントハンドラのパラメータが，タイプライブラリ内のイベントに対して定義されたパラメータとは少し違うことがあります。インターフェースポインタであるパラメータは，通常は TComInterface テンプレートを使ってラップされ，生インターフェースポインタとして表示されるのではありません。その結果できるインターフェースラッパーの名前は，InterfaceNamePtr の形式になります。

例として，ExcelApplication イベントのイベントハンドラ OnNewWorkBook のコードを次に示します。このイベントハンドラには，ほかの CoClass（ExcelWorkbook）のインターフェースを提供するパラメータがあります。ただし，このインターフェースは ExcelWorkbook インターフェースポインタとして渡されるのではなく，ExcelWorkbookPtr オブジェクトとして渡されます。

```
void _fastcall TForm1::XLappNewWorkbook(TObject *Sender, ExcelWorkbookPtr Wb)
{
  ExcelWorkbook1->ConnectTo(Wb);
}
```

この例では，イベントハンドラが ExcelWorkbook コンポーネント（ExcelWorkbook1）にワークブックを割り当てます。これは ConnectTo メソッドを使用してコンポーネントラッパーを既存のインターフェースに接続する方法を示します。ConnectTo メソッドは，コンポーネントラッパーに対して生成されたコードに追加されます。

アプリケーションオブジェクトを持つサーバーは，そのオブジェクト上で Quit メソッドをエクスポーズして，クライアントが接続を終了できるようにします。Quit は，通常，アプリケーションを終了するために［ファイル｜終了］を使用するのと同じ機能を提供します。Quit メソッドを呼び出すコードは，コンポーネントの Disconnect メソッド内で生成されます。パラメータを何も指定せずに Quit メソッドを呼び出すことが可能ならば，コンポーネントラッパーには AutoQuit プロパティもあります。AutoQuit は，コンポーネントが解放されたときに，コントローラが Quit を呼び出せるようにします。それ以外のときに切断したい場合，または Quit メソッドにパラメータが必要である場合

は，このメソッドを明示的に呼び出さなければなりません。Quit は生成されたコンポーネント上にパブリックなメソッドとして表示されます。

データベース対応の ActiveX コントロールの使い方

C++Builder アプリケーションでデータベース対応の ActiveX コントロールを使用する場合は，そのコントロールが表すデータのデータベースに関連付けなければなりません。それには，データベース対応の VCL コントロールにデータソースが必要であるのと同様に，データソースコンポーネントが必要です。

データベース対応の ActiveX コントロールをフォームデザイナに配置した後，その DataSource プロパティを，希望するデータセットを表すデータソースに割り当てます。データソースを指定し終えると，データバインディングエディタを使用してコントロールのデータベース対応プロパティをデータセットの項目にリンクできるようになります。

データバインディングエディタを表示するには，まず，データベース対応の ActiveX コントロールを右クリックして，オプションを表示します。基本オプションのほかに，追加の［データのバインド］項目も表示されます。この項目を選択するとデータバインディングエディタが表示され，データセット内の項目名，および ActiveX コントロールのバインド可能プロパティがリストされます。

項目をプロパティにバインドする手順は次のとおりです。

1. ActiveX データバインディングエディタダイアログで，項目およびプロパティ名を選択します。

   ［項目名］にはデータベースの項目がリストされ，［プロパティ］にはデータベース項目にバインドできる ActiveX コントロールのプロパティがリストされます。プロパティの dispID は Value(12) のようにカッコで囲まれています。

2. ［バインド］，［OK］を選択します。

メモ ダイアログに何もプロパティが表示されない場合，ActiveX コントロールにはデータベース対応プロパティがありません。ActiveX コントロールのプロパティ用に単純なデータバインディングを有効にするには，43-11 ページの「タイプライブラリを使って単純なデータバインディングを有効にする」で説明しているタイプライブラリを使います。

次の例では，C++Builder コンテナにあるデータベース対応 ActiveX コントロールを使う手順を説明します。この例では，Microsoft Office 97 に付属している Microsoft Calendar Control を使います。

1. C++Builder のメインメニューから，［コンポーネント｜ ActiveX コントロールの取り込み］を選択します。
2. Microsoft Calendar Control 8.0 など，データベース対応 ActiveX コントロールを選択し，そのクラス名を TCalendarAxControl に変更して［インストール］を選択します。
3. インストールダイアログで［OK］を選択して，コントロールをデフォルトのユーザーパッケージに追加します。これでコントロールをパレットで使用できるようになります。
4. ［すべて閉じる］を選択してから，［ファイル｜新規作成｜アプリケーション］を選択し，新しいアプリケーションを開きます。

5. ［ActiveX］タブから，上でパレットに追加した TCalendarAxControl オブジェクトをフォームにドロップします。
6. ［Data Access］タブから DataSource オブジェクトを，［BDE］タブから Table オブジェクトをフォームにドロップします。
7. DataSource オブジェクトを選択して，その DataSet プロパティを Table1 に設定します。
8. Table オブジェクトを選択して，以下の手順を実行します。
   - DatabaseName プロパティを BCDEMOS に設定します。
   - TableName プロパティを EMPLOYEE.DB に設定します。
   - Active プロパティを **true** に設定します。
9. TCalendarAXControl オブジェクトを選択して，その DataSource プロパティを DataSource1 に設定します。
10. TCalendarAXControl オブジェクトを右クリックし，［データのバインド］を選択して ActiveX コントロールデータバインディングエディタを呼び出します。

    ［項目名］には，アクティブデータベースのすべての項目がリストされます。［プロパティ名］にはデータベース項目に結合できる ActiveX コントロールのプロパティがリストされます。プロパティの dispID はカッコで囲まれています。
11. HireDate 項目と Value プロパティ名を選択し，［バインド］，［OK］を選択します。

    これで項目名とプロパティが結合されました。
12. Data Controls タブから DBGrid オブジェクトをフォームにドロップし，その DataSource プロパティを DataSource1 に設定します。
13. Data Controls タブから DBNavigator オブジェクトをフォームにドロップし，その DataSource プロパティを DataSource1 に設定します。
14. アプリケーションを実行します。
15. 次のようにアプリケーションをテストします。

    DBGrid オブジェクトに表示される HireDate 項目を使い，Navigator オブジェクトを使ってデータベース内を移動します。データベース内を移動すると ActiveX コントロールの日付が変わることを確認してください。

例題：Microsoft Word による文書の印刷

Office 97 の Microsoft Word 8 を使って文書を印刷するオートメーションコントローラを作成する手順は，次のとおりです。

操作を開始するにあたって，フォーム，ボタン，およびオープンダイアログボックス（TOpenDialog）で構成される新規プロジェクトを作成します。これらのコントロールがオートメーションコントローラを構成します。

ステップ 1：C++Builder の準備をする

C++Builder では，コンポーネントパレット上に，Word，Excel，PowerPoint など多くの一般的なサーバーが用意されています。サーバーのインポートのしかたを説明するために，Word を例としてとりあげます。これはすでにコンポーネントパレットに存在するので，この最初のステップでは，Word をパレットにインストールする方法を試せるように，Word を含むパッケージをいったん削除します。ステップ 4 で，コンポーネントパレットを通常の状態に戻す方法を説明します。

Word をコンポーネントパレットから削除する手順は，次のとおりです。

1. ［コンポーネント｜パッケージのインストール］を選択します。
2. ［Borland C++Builder COM Server Components Sample Package］をクリックし，［削除］を選択します。

   C++Builder によって提供されていたサーバーが，コンポーネントパレットの［Servers］ページからなくなります（それ以外のサーバーがインポートされていなければ，［Servers］ページもなくなります）。

ステップ 2：Word タイプライブラリを取り込む

Word タイプライブラリを取り込む手順は次のとおりです。

1. ［プロジェクト｜タイプライブラリの取り込み］を選択します。
2. ［タイプライブラリの取り込み］ダイアログでの手順は次のとおりです。
   1. ［Microsoft Office 8.0 Object Library］を選択します。

      Word (Version 8) がリストにない場合は，［追加］ボタンを選択し，Program Files \ Microsoft Office \ Office に移動して Word タイプライブラリファイル MSWord8.olb を選択し，［開く］を選択します。
   2. ［パレットページ名］では［Servers］を選択します。
   3. ［インストール］を選択します。

      ［インストール］ダイアログが表示されます。［新規パッケージに追加］タブを選択して WordExample と入力し，このタイプライブラリを含める新しいパッケージを作成します。
3. ［Servers］パレットページを開き，WordApplication を選択してそれをフォーム上に配置します。
4. ステップ 3 で後述するように，ボタンオブジェクトのイベントハンドラを書きます。

ステップ 3：仮想テーブルインターフェースオブジェクトまたはディスパッチインターフェースオブジェクトを使って Microsoft Word を制御する

仮想テーブルオブジェクトまたはディスパッチオブジェクトを使って，Microsoft Word を制御できます。

仮想テーブルインターフェースオブジェクトの使用

WordApplication オブジェクトのインスタンスをフォームにドロップすると，仮想テーブルインターフェースオブジェクトを使ってこのコントロールに簡単にアクセスできます。作成したクラスのメソッドを呼び出すだけで済みます。Word の場合は，TWordApplication クラスです。

1. このボタンを選択し，OnClick イベントハンドラをダブルクリックし，次のイベント処理コードを入力します。

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  if (OpenDialog1->Execute())
  {
    TVariant FileName = OpenDialog1->FileName.c_str();
    WordApplication1->Documents->Open(&FileName);
    WordApplication1->ActiveDocument->PrintOut();
  }
}
```

2. このプログラムをビルドして実行します。このボタンをクリックすると，Word が印刷するファイルの指定を求めます。

ディスパッチインターフェースオブジェクトの使用

レイトバインディングのためにディスパッチインターフェースを使うこともできます。ディスパッチインターフェースオブジェクトを使うには，次のように，_ApplicationDisp ディスパッチラッパークラスを使ってアプリケーションオブジェクトを作成および初期化します。ディスパッチインターフェースメソッドは，ソースでは仮想テーブルインターフェースを返すものとして記述されていますが，実際には戻り値をディスパッチインターフェースにキャストしなければなりません。

1. このボタンを選択し，OnClick イベントハンドラをダブルクリックし，次のイベント処理コードを入力します。

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
if (OpenDialog1->Execute())
  {
    TVariant FileName = OpenDialog1->FileName.c_str();

    _ApplicationDisp MyWord;
    MyWord.Bind(CLSID_WordApplication);
    DocumentsDisp MyDocs = MyWord->Documents;
    MyDocs->Open(&FileName);

    _DocumentDisp MyActiveDoc = MyWord->ActiveDocument;
    MyActiveDoc->PrintOut();
    MyWord->Quit();
}
```

2. このプログラムをビルドして実行します。このボタンをクリックすると，Word が印刷するファイルの指定を求めます。

ステップ 4：例題をクリーンアップする

この例題を終了したら，C++Builder を元の状態に戻しておきます。

1. ［Servers］ページのオブジェクトを削除します。
   - ［コンポーネント｜パッケージのインストール］を選択します。
   - リストの WordExample パッケージを選択し，［削除］を選びます。
   - ［確認］メッセージボックスで［はい］を選択します。

- ［OK］をクリックして［プロジェクトオプション］ダイアログを終了します。

2. ［Borland C++Builder COM Server Components Sample Package］パッケージを元に戻します。
   - ［コンポーネント｜パッケージのインストール］を選択します。
   - ［追加］ボタンをクリックします。
   - 表示されたダイアログで，Office 97 のコンポーネントの場合は bcb97axserver60.bpl を，Office 2000 のコンポーネントの場合は bcb2kaxserver60.bpl を選択します。
   - ［OK］をクリックして［プロジェクトオプション］ダイアログを終了します。

タイプライブラリ定義に基づいたクライアントコードの作成

ActiveX コントロールのホストにはコンポーネントラッパーを使用しなければなりませんが，TypeLibName_TLB ユニットにあるタイプライブラリから定義だけを使用してオートメーションコントローラを作成することができます。このプロセスは，コンポーネントに作業させる場合に比べて多少複雑です。特に，イベントに応答する必要がある場合は複雑になります。

サーバーへの接続

コントローラアプリケーションからオートメーションサーバーを駆動できるようにするには，まずサポートするインターフェースへの参照を取得しなければなりません。一般には，サーバーに接続するときはサーバーのメインインターフェースを介します。たとえば，Microsoft Word には WordApplication コンポーネントを介して接続します。

メインインターフェースがデュアルインターフェースである場合は，TypeLibName_TLB.h ファイル内のクリエータオブジェクトを使用できます。クリエータクラスは，CoClass と同じ名前を持ち，プレフィクスの「Co」が追加されます。Create メソッドを呼び出すと同じマシン上のサーバーに接続でき，CreateRemote メソッドを使用するとリモートマシン上のサーバーに接続できます。Create と CreateRemote は静的メソッドなので，このメソッドの呼び出しにクリエータクラスのインスタンスは必要ありません。

```
pInterface = CoServerClassName.Create();
pInterface = CoServerClassName.CreateRemote("Machine1");
```

Create と CreateRemote は，CoClass のデフォルトインターフェースのクラスラッパーを返します（デフォルトインターフェースは TComInterface テンプレートを使用して TypeLibName_TLB.h で定義されます）。

デフォルトインターフェースがディスパッチインターフェースである場合は，CoClass に対して生成される Creator クラスはありません。そのかわり，デフォルトインターフェースに対して自動生成されるラッパークラスのインスタンスを作成しなければなりません。このクラスは TAutoDriver テンプレートを使用して定義され，名前が InterfaceNameDisp の形式になります。次に，Bind メソッドを呼び出し，CoClass の GUID を渡します（この GUID の定数は _TLB ユニットの一番上で定義されています）。

デュアルインターフェースによるオートメーションサーバーの制御

自動生成されたクリエータクラスを使用してサーバーに接続したら，「->」演算子を使用してインターフェースラッパーオブジェクトのメソッドを呼び出します。たとえば，次のようにします。

```
TComApplication AppPtr = CoWordApplication_.Create();
AppPtr->DoSomething;
```

インターフェースラッパーとクリエータクラスは，タイプライブラリのインポート時に自動生成される TypeLibName_TLB ユニット内に定義されます。このインターフェースのラッパークラスを使うことの利点は，破棄されるときに基底のインターフェースを自動的に解放する点です。

デュアルインターフェースについての詳細は，41-13 ページの「デュアルインターフェース」を参照してください。

ディスパッチインターフェースによるオートメーションサーバーの制御

一般には，前述のようにデュアルインターフェースを使ってオートメーションサーバーを制御します。ただし，使用可能なデュアルインターフェースがないために，ディスパッチインターフェースを使ってオートメーションサーバーを制御する必要があることもあります。

ディスパッチインターフェースのメソッドを呼び出す手順は次のとおりです。

1. ディスパッチインターフェースのラッパークラスの Bind メソッドを使用してサーバーに接続します。サーバーへの接続方法についての詳細は，40-13 ページの「サーバーへの接続」を参照してください。
2. ディスパッチインターフェースのラッパーオブジェクトのメソッドを呼び出すことにより，オートメーションサーバーを制御します。

ラッパークラスは，ディスパッチインターフェースのプロパティとメソッドを，自分のプロパティとメソッドとして公開します。さらに，TAutoDriver を継承しているので，IDispatch メカニズムを使用すると，サーバーオブジェクトのプロパティとメソッドを呼び出すことができます。プロパティまたはメソッドにアクセスできるようにするには，まずディスパッチ ID を取得しなければなりません。それには，GetIDsOfNames メソッドを使用して，名前，およびその名前のディスパッチ ID を受け取る DISPID 変数への参照を渡します。ディスパッチ ID を取得すると，これを使用して OlePropertyGet，OlePropertyPut，または OleFunction を呼び出して，サーバーオブジェクトのプロパティとメソッドにアクセスすることができます。

ディスパッチインターフェースのもう 1 つの使い方は，Variant にディスパッチインターフェースを割り当てることです。この方法は，インターフェースを値に持つプロパティまたはパラメータの VCL オブジェクトによって使われることがあります。Variant 型には，OlePropertyGet，OlePropertySet，OleFunction，および OleProcedure メソッドを通じてのディスパッチインターフェース呼び出しのサポートが組み込まれています。これらのメソッドは，ディスパッチインターフェースのラッパークラス上のメソッドに相当します。Variant を使用する方法は，ディスパッチインターフェースのラッパークラスを使用する方法よりやや遅いことがあります。なぜなら，Variant のメソッドは，呼び出されるたびにディスパッチ ID を動的に参照するからです。ただし，タイプライブラリをインポートしなくても使えるという利点があります。

警告 ディスパッチインターフェースのラッパーではなく Variant にインターフェースを割り当てる場合は，注意しなければなりません。ラッパーは AddRef および Release の呼び出しを自動的に処理しますが，

Variant は処理しません。したがって，たとえばインターフェースを値に持つ Variant をインターフェースラッパーに割り当てる場合，このインターフェースラッパーは AddRef を呼び出しませんが，デストラクタは Release を呼び出します。このため，VCL オブジェクトのプロパティとしてすでに登場している場合以外は，Variant を使用しない方がよいでしょう。

ディスパッチインターフェースについての詳細は，41-12 ページの「オートメーションインターフェース」を参照してください。

オートメーションコントローラ内のイベント処理

開発者が自分で取り込んだタイプライブラリを持つオブジェクトのコンポーネントラッパーを生成する場合，生成したコンポーネントに追加されたイベントを使うだけで，簡単にイベントに応答できます。ただし，コンポーネントラッパーを使用しない場合（またはサーバーが COM+ イベントを使用する場合）は，開発者がイベントシンクコードを作成しなければなりません。

オートメーションイベントのプログラム的処理

イベントを処理できるようにするには，まずイベントシンクを定義しなければなりません。イベントシンクとは，サーバーのタイプライブラリで定義されるイベントディスパッチインターフェースを実装するクラスです。

イベントシンクは，TEventDispatcher の子孫です。TEventDispatcher はテンプレート化されたクラスであり，イベントシンクのクラス，およびイベントシンクが処理するイベントインターフェースの GUID の 2 つのパラメータを必要とします。

```
class MyEventSinkClass: TEventDispatcher<MyEventSinkClass, DIID_TheServerEvents>
{
...// ここで DIID_TheServerEvents のメソッドを宣言する
}
```

イベントシンククラスのインスタンスが用意できたら，ConnectEvents メソッドを呼び出して，このイベントシンクの存在をサーバーに知らせます。このメソッドは，サーバーの IConnectionPointContainer および IConnectionPoint インターフェースを使用して，オブジェクトをイベントシンクとして登録します。これでこのオブジェクトは，イベントが起きたときにサーバーから呼び出しを受け取るようになります。

```
pInterface = CoServerClassName.CreateRemote("Machine1");
MyEventSinkClass ES;
ES.ConnectEvents(pInterface);
```

イベントシンクを解放する前に，接続を終了しなければなりません。それには，イベントシンクの DisconnectEvents メソッドを呼び出します。

```
ES.DisconnectEvents(pInterface);
```

メモ イベントシンクを解放する前に，サーバーがこのイベントシンクとの接続を解放したことを確認しなければなりません。DisconnectEvents が起動する切断通知にサーバーがどのように応答するかが開発者にはわからないので，この呼び出しの直後にイベントシンクを解放する場合は，サーバー側の処理と競合する可能性があります。TEventDispatcher は，このような事態を防止するために，サーバーがイベントシンクのインターフェースを解放するまでデクリメントされない独自の参照カウントを保持しています。

COM+ イベントの処理

COM+ では，サーバーがイベントを生成するために，特殊なインターフェースのセット（IConnectionPointContainer および IConnectionPoint）ではなく，特殊なヘルパーオブジェクトを使用します。したがって，TEventDispatcher を継承するイベントシンクは使用できません。TEventDispatcher は，COM+ イベントオブジェクトではなくこれらのインターフェースを使用するように設計されています。

クライアントアプリケーションは，イベントシンクを定義せずに，サブスクライバオブジェクトを定義します。サブスクライバオブジェクトは，イベントシンクと同様に，イベントインターフェースの実装を提供します。イベントシンクとの違いは，サーバーの接続ポイントに接続するのではなく，特定のイベントオブジェクトにサブスクライブすることです。

サブスクライバオブジェクトを定義するには，COM オブジェクトウィザードを使用して，このイベントオブジェクトのインターフェースを実装対象として選択します。実装ユニットが生成され，このユニットのスケルトンメソッドの中身を埋めるとイベントハンドラを作成できます。COM オブジェクトウィザードを使って既存のインターフェースを実装する方法についての詳細は，41-2 ページの「COM オブジェクトウィザードの使い方」を参照してください。

メモ 実装可能なインターフェースのリストにイベントオブジェクトのインターフェースが表示されない場合は，ウィザードを使用してレジストリに追加する必要がある場合があります。

サブスクライバオブジェクトを作成したら，イベントオブジェクトのインターフェースに，またはそのインターフェース上の個別のメソッド（イベント）にサブスクライブしなければなりません。サブスクリプションは以下の 3 種類から選択できます。

- **一時的サブスクリプション：** 従来のイベントシンクと同様に，一時的サブスクリプションはオブジェクトインスタンスの存続期間と連動する。サブスクライバオブジェクトが解放されると，サブスクリプションが終了し，COM+ がイベントを転送しなくなる
- **持続的サブスクリプション**：持続的サブスクリプションは，特定のオブジェクトインスタンスではなくオブジェクトクラスに連動する。イベントが発生すると，COM がサブスクライバオブジェクトのインスタンスを探し出すか起動し，イベントハンドラを呼び出す。インプロセスオブジェクト（DLL）は，このサブスクリプションを使用する
- **ユーザーごとのサブスクリプション**：このサブスクリプションは，一時的サブスクリプションのセキュリティを向上したものである。サブスクライバオブジェクト，およびイベントを送出するサーバーオブジェクトの両方が，同じマシン上の同じユーザーアカウントで実行していなければならない

メモ COM+ イベントにサブスクライブするオブジェクトは，COM+ アプリケーションにインストールしなければなりません。

タイプライブラリを持たないサーバーのクライアントの作成

オブジェクトのリンクと埋め込み（OLE）などの一部の古い COM 技術は，タイプライブラリ内に型情報を提供しません。そのかわりに，定義済みインターフェースの標準セットに依存しています。このようなオブジェクトのホストになるクライアントを作成するには，TOleContainer コンポーネントを使用します。このコンポーネントは，コンポーネントパレットの［System］ページにあります。

TOleContainer は，Ole2 オブジェクトのホストサイトとして機能します。IOleClientSite インターフェースを実装し，オプションで IOleDocumentSite を実装します。通信の処理には OLE Verb が使用されます。

TOleContainer を使用する手順は次のとおりです。

1. TOleContainer コンポーネントをフォーム上に配置します。
2. ActiveX ドキュメントのホストにしたい場合は，AllowActiveDoc プロパティを **true** に設定します。
3. AllowInPlace プロパティを設定して，ホスト下に置かれるオブジェクトを TOleContainer または独立したウィンドウのどちらに表示するかを指定します。
4. オブジェクトの起動，停止，移動，またはサイズ変更が行われたときに応答するイベントハンドラを作成します。
5. 設計時に TOleContainer オブジェクトをバインドするには，右クリックして［オブジェクトの挿入］を選択します。［オブジェクトの挿入］ダイアログで，ホストになるサーバーオブジェクトを選択します。
6. 実行時に TOleContainer オブジェクトをバインドする場合は，いくつかのメソッドを使用でき，サーバーオブジェクトをどのように識別したいかに応じて選択します。CreateObject はプログラム ID をとり，CreateObjectFromFile はオブジェクトの保存先ファイル名をとり，CreateObjectFromInfo はオブジェクトの作成方法の情報が入った構造体をとり，CreateLinkToFile はオブジェクトの保存先ファイル名をとり，このファイルに埋め込むのではなくリンクします。
7. オブジェクトのバインドを終えると，OleObjectInterface プロパティを使用してインターフェースにアクセスできるようになります。ただし，Ole2 オブジェクトとの通信は OLE Verb に基づいていたので，サーバーへのコマンド送信には DoVerb メソッドを使用したいと考える場合が多くなります。
8. サーバーオブジェクトを解放したい場合は，DestroyObject メソッドを呼び出します。

第41章

単純な COM サーバーの作成

C++Builder は，さまざまな COM オブジェクトの作成に役立つウィザードを提供しています。もっとも単純な COM オブジェクトは，クライアントが呼び出すことができるデフォルトインターフェースを通じてプロパティとメソッド（および場合によってはイベント）をエクスポーズするサーバーです。

メモ COM サーバーとオートメーションは，CLX アプリケーションでは使用できません。COM 技術は Windows で使用するためのものであり，クロスプラットフォーム用ではありません。

単純な COM オブジェクトの作成プロセスを特に容易にするウィザードは以下の 2 つです。

- COM オブジェクトウィザードは，デフォルトインターフェースを IUnknown から継承するか，システムにすでに登録されているインターフェースを実装する，軽量の COM オブジェクトを構築する。作成できる COM オブジェクトの種類の柔軟性がもっとも高いウィザード
- オートメーションオブジェクトウィザードは，デフォルトインターフェースを IDispatch から継承する単純なオートメーションオブジェクトを作成する。IDispatch は，標準マーシャリング機構，およびインターフェース呼び出しのレイトバインディングのサポートを導入する

メモ COM は，特定の状況に対処するための標準的なインターフェースおよび機構を多数定義しています。C++Builder のウィザードは，もっとも一般的なタスクを自動化します。ただし，カスタムマーシャリングのように C++Builder のウィザードがサポートしないタスクもあります。カスタムマーシャリング，および C++Builder が明示的にサポートしないその他の技術については，Microsoft Developer's Network（MSDN）のマニュアルを参照してください。Microsoft の Web サイトでも，COM サポートの最新情報が提供されています。

COM オブジェクトの作成の概要

オートメーションオブジェクトウィザードを使用して新規のオートメーションサーバーを作成する場合でも，COM オブジェクトウィザードを使用してそれ以外の種類の COM オブジェクトを作成する場合でも，そのプロセスは同じです。その手順は次のとおりです。

1. COM オブジェクトを設計します。
2. COM オブジェクトウィザードまたはオートメーションオブジェクトウィザードを使って，サーバーオブジェクトを作成します。
3. ［プロジェクトオプション］ダイアログの［ATL］ページでオプションを指定して，そのオブジェクトが入るアプリケーションを COM がどのように呼び出すか，および必要なデバッグサポートの種類を指示します。
4. オブジェクトがクライアントにエクスポーズするインターフェースを定義します。
5. COM オブジェクトを登録します。
6. アプリケーションをテストおよびデバッグします。

COM オブジェクトを設計する

COM オブジェクトを設計するときには，どのような COM インターフェースを実装したいかを決定する必要があります。すでに定義済みのインターフェースを実装する COM オブジェクトを作成するか，オブジェクトが実装するインターフェースを新規に定義することができます。さらに，オブジェクトが複数のインターフェースをサポートすることもできます。サポートが必要になる可能性がある標準 COM インターフェースについては，MSDN のマニュアルを参照してください。

- 既存のインターフェースを実装する COM オブジェクトを作成するには，COM オブジェクトウィザードを使用する
- 新規に定義するインターフェースを実装する COM オブジェクトを作成するには，COM オブジェクトウィザードまたはオートメーションオブジェクトウィザードを使用する。COM オブジェクトウィザードは，*IUnknown* を継承する新規のデフォルトインターフェースを生成でき，オートメーションオブジェクトウィザードは，IDispatch を継承するデフォルトインターフェースをオブジェクトに与える。どちらのウィザードを使用する場合でも，後でいつでもタイプライブラリエディタを使用して，ウィザードが生成するデフォルトインターフェースの親インターフェースを変更できる

サポートするインターフェースを決定するだけでなく，COM オブジェクトをインプロセスサーバーにするか，アウトオブプロセスサーバーにするか，それともリモートサーバーにするかを決定しなければなりません。インプロセスサーバーの場合，およびタイプライブラリを使用するアウトオブプロセスサーバーとリモートサーバーの場合は，COM がデータをマーシャリングします。それ以外の場合は，どのようにしてデータをアウトオブプロセスサーバーにマーシャリングするかを決める必要があります。サーバーの種類についての詳細は，第 38 章-6 の「インプロセス，アウトオブプロセス，およびリモートサーバー」を参照してください。

COM オブジェクトウィザードの使い方

COM オブジェクトウィザードは以下の処理を行います。

- 新規のユニットを作成する

- ATL クラスの CComObjectRootEx および CComCoClass を継承する新規のクラスを定義する。基本クラスについての詳細は，38-22 ページの「ウィザードによって生成されるコード」を参照
- タイプライブラリをプロジェクトに追加し，オブジェクトとそのインターフェースをタイプライブラリに追加する

COM オブジェクトを作成する前に，実装したい機能を含んでいるアプリケーションのプロジェクトを作成するか開きます。ニーズに応じて，アプリケーションと ActiveX ライブラリのどちらかをプロジェクトにすることができます。

COM オブジェクトウィザードを起動する手順は次のとおりです。

1. ［ファイル｜新規作成｜その他］を選択して［新規作成］ダイアログボックスを開きます。
2. ［ActiveX］タブを選択します。
3. ［COM オブジェクト］アイコンをダブルクリックします。

ウィザードでは，以下のものを指定しなければなりません。

- **CoClass 名**：クライアントに表示されるオブジェクト名です。オブジェクトを実装するために作成されるクラスは，この名前の先頭に「T」を付けた名前になります。既存のインターフェースの実装を選択しない場合，この名前の先頭に「I」を付けた名前のデフォルトインターフェースが CoClass に与えられます。
- **実装するインターフェース**：デフォルトでは，IUnknown を継承するデフォルトインターフェースがオブジェクトに与えられます。ウィザードを終了した後は，タイプライブラリエディタを使用してこのインターフェースにプロパティとメソッドを追加できます。ただし，オブジェクトに対して定義済みインターフェースの実装を選択することもできます。COM オブジェクトウィザードの［一覧］ボタンを選択するとインターフェース選択ウィザードが表示され，システムに登録されたタイプライブラリで定義されたデュアルインターフェースまたはカスタムインターフェースを選択できます。選択したインターフェースは，新規の CoClass のデフォルトインターフェースになります。生成された実装クラスにはこのインターフェース上の全メソッドが追加されるので，実装ユニット内のメソッドの中身を埋めるだけで済みます。既存のインターフェースを選択する場合は，このインターフェースはプロジェクトのタイプライブラリに追加されません。つまり，オブジェクトを配布する際に，このインターフェースを定義するタイプライブラリも配布しなければなりません。
- **スレッドモデル：**通常は，オブジェクトに対する複数のクライアント要求は，異なる実行スレッドに入ります。COM がオブジェクトを呼び出すときに，これらのスレッドをどのようにシリアル化するかを指定できます。スレッドモデルを選択すると，オブジェクトの登録方法が決まります。開発者は，自分が選択したスレッドモデルに対するスレッドサポートを提供する責任があります。選択可能なスレッドモデルについては，41-5 ページの「スレッドモデルを選択する」を参照してください。アプリケーションにスレッドサポートを提供する方法については，第 11 章「マルチスレッドアプリケーションの作成」を参照してください。
- **イベントサポートのためのコードを生成：**クライアントが応答できるイベントをオブジェクトに生成させるかどうかを指定しなければなりません。ウィザードは，イベント生成に必要なインターフェース，およびクライアントイベントハンドラ呼び出しのディスパッチのサポートを提供できます。イベントがどのように機能するか，およびイベントを実装するときに何をする必要が

あるかについては，41-11 ページの「クライアントにイベントをエクスポーズする」を参照してください。

- **OLE オートメーションの印をつける：**オートメーション互換型しか使用しない場合は，インプロセスサーバーを生成しないときにマーシャリングを COM に処理させることができます。タイプライブラリ内でオブジェクトのインターフェースに OleAutomation としてマークを付けると，プロクシーとスタブを，およびプロセスの境界を越えてパラメータを渡すハンドルを，COM がセットアップできます。このプロセスについての詳細は，38-8 ページの「マーシャリング機構」を参照してください。新規のインターフェースを生成する場合は，そのインターフェースがオートメーション互換であるかどうかしか指定できません。既存のインターフェースを選択する場合，その属性はタイプライブラリですでに指定されています。オブジェクトのインターフェースが OleAutomation としてマークが付いていない場合は，インプロセスサーバーを作成するか，自分のマーシャリングコードを作成しなければなりません。
- **上位インターフェースを実装：**継承されたインターフェースにウィザードがスタブルーチンを提供するようにしたい場合に選択するオプションです。継承されたインターフェースのうち，ウィザードが実装しないものは，IUnknown，IDispatch，および IAppServer の 3 つです。IUnknown と IDispatch が実装されない理由は，ATL がこれら 2 つのインターフェースの実装を独自に提供するからです。IAppServer が実装されない理由は，クライアントデータセットとデータセットプロバイダを使用するときに自動的に実装されるからです。

COM オブジェクトについてのコメントをオプションで追加できます。このコメントは，オブジェクトのタイプライブラリに表示されます。

オートメーションオブジェクトウィザードの使い方

オートメーションオブジェクトウィザードは以下の処理を行います。

- 新規のユニットを作成する
- ATL クラスの CComObjectRootEx および CComCoClass を継承する新規のクラスを定義する。基本クラスについての詳細は，38-22 ページの「ウィザードによって生成されるコード」を参照
- タイプライブラリをプロジェクトに追加し，オブジェクトとそのインターフェースをタイプライブラリに追加する

オートメーションオブジェクトを作成するにあたって，提供したい機能を含むアプリケーションのプロジェクトを作成または開きます。ニーズに応じて，アプリケーションと ActiveX ライブラリのどちらかをプロジェクトにすることができます。

オートメーションウィザードを表示する手順は次のとおりです。

1. ［ファイル｜新規作成｜その他］を選択します。
2. ［ActiveX］タブを選択します。
3. ［オートメーションオブジェクト］アイコンをダブルクリックします。

ウィザードダイアログで以下のものを指定します。

- **CoClass 名**：クライアントに表示されるオブジェクト名です。オブジェクトのデフォルトインターフェースは，この CoClass 名の先頭に「I」を付けた名前で作成され，オブジェクトを実装するために作成されるクラスは，この CoClass 名の先頭に「T」を付けた名前になります。
- **スレッドモデル**：通常は，オブジェクトに対する複数のクライアント要求は，異なる実行スレッドに入ります。COM がオブジェクトを呼び出すときに，これらのスレッドをどのようにシリアル化するかを指定できます。スレッドモデルを選択すると，オブジェクトの登録方法が決まります。開発者は，自分が選択したスレッドモデルに対するスレッドサポートを提供する責任があります。選択可能なスレッドモデルについては，41-5 ページの「スレッドモデルを選択する」を参照してください。アプリケーションにスレッドサポートを提供する方法については，第 11 章「マルチスレッドアプリケーションの作成」を参照してください。
- **イベントサポートのためのコードを生成**：クライアントが応答できるイベントをオブジェクトに生成させるかどうかを指定しなければなりません。ウィザードは，イベント生成に必要なインターフェース，およびクライアントイベントハンドラ呼び出しのディスパッチのサポートを提供できます。イベントがどのように機能するか，およびイベントを実装するときに何をする必要があるかについては，41-11 ページの「クライアントにイベントをエクスポーズする」を参照してください。

COM オブジェクトについてのコメントをオプションで追加できます。このコメントは，オブジェクトのタイプライブラリに表示されます。

オートメーションオブジェクトは，仮想テーブルによるアーリー（コンパイル時）バインディングと IDispatch インターフェースによるレイト（実行時）バインディングを両方ともサポートする**デュアルインターフェース**を実装します。詳細については，41-13 ページの「デュアルインターフェース」を参照してください。

スレッドモデルを選択する

ウィザードを使ってオブジェクトを作成するときには，オブジェクトがサポートするスレッドモデルを選択します。COM オブジェクトにスレッドのサポートを付け加えると，複数のクライアントが同時にアプリケーションにアクセスできるので，処理効率を上げることができます。

表 41.1 に，指定できるスレッドモデルを示します。

表 41.1　COM オブジェクトのスレッドモデル

スレッドモデル	内容	実装の長所と短所
シングル	サーバーがスレッドのサポートを提供しない。アプリケーションが一度に 1 つの要求を受け取るように，クライアントからの要求を COM がシリアル化する	クライアントは一度に 1 つずつ処理されるので，スレッドのサポートが不要 処理効率上の利点はない
アパートメント（またはシングルスレッドアパートメント）	オブジェクトを呼び出せるクライアントスレッドが一度に 1 つだけであることを COM が保証する。そのオブジェクトが作成されたスレッドをすべてのクライアント呼び出しが使用する	オブジェクトは自分のインスタンスデータには安全にアクセスできるが，グローバルデータは，クリティカルセクションやほかの形のシリアル化を使って保護しなければならない スレッドのローカル変数は複数の呼び出しにわたって信頼できる 処理効率上の利点がある
フリー（マルチスレッドアパートメントとも呼ばれる）	オブジェクトはいつでも任意の数のスレッドの呼び出しを受け取ることができる	オブジェクトはクリティカルセクションやほかの形のシリアル化を使って，すべてのインスタンスデータとグローバルデータを保護しなければならない スレッドのローカル変数は，複数の呼び出しにわたると信頼できない
フリー / アパートメント両用	フリースレッドモデルとほぼ同じだが，（コールバックなどの）外向き呼び出しが同じスレッドで実行することが保証される点が異なる	処理効率と柔軟性は最高 外向き呼び出しに供給されるパラメータのスレッドサポートをアプリケーションが提供する必要がない
ニュートラル	複数のクライアントが異なるスレッド上で同時にオブジェクトを呼び出すことができるが，2 つの呼び出しが衝突しないことを COM が保証する	グローバルデータ，および複数のメソッドがアクセスするインスタンスデータが関与するスレッド衝突を，開発者が防止しなければならない。 ユーザーインターフェース（ビジュアルコントロール）を持つオブジェクトには使用してはならないモデル このモデルは COM+ のみで使用できる。COM ではアパートメントモデルにマップされる

メモ　ローカル変数（コールバック内の変数を除く）は，スレッドモデルに関係なく常に安全です。これは，ローカル変数がスタックに保存され，各スレッドが自分自身のスタックを持っているためです。フリースレッドを使う場合のコールバックでは，ローカル変数が安全でない可能性があります。

ウィザードで選択したスレッドモデルは，システムレジストリ内でオブジェクトがどのように登録されるかを決定します。開発者は，選択したスレッドモデルに忠実に，オブジェクトを実装しなければなりません。スレッドセーフコードの作成方法についての概要は，第 11 章「マルチスレッドアプリケーションの作成」を参照してください。

メモ　指定されたスレッドモデルに従って COM がオブジェクトを呼び出すようにするには，そのスレッドモデルをサポートするように COM を初期化しなければなりません。COM の初期化方法を指定するには，［プロジェクトオプション］ダイアログの［ATL］ページを使用します。

インプロセスサーバーの場合，ウィザードでスレッドモデルを設定すると，CLSID レジストリエントリ内でスレッドモデルキーが設定されます。

アウトオブプロセスサーバーは EXE として登録され，必要とされるもっとも高いスレッドモデル用に COM を初期化しなければなりません。たとえば，EXE にフリースレッドオブジェクトが含まれている場合は，フリースレッド用に初期化されます。つまり，EXE に含まれているどのフリースレッドまたはアパートメントスレッドオブジェクトに対しても，期待通りのサポートを提供できます。COM の初期化方法を指定するには，［プロジェクトオプション］ダイアログの［ATL］ページを使用します。

フリースレッドモデルをサポートするオブジェクトを作成する

複数のスレッドからオブジェクトにアクセスする必要がある場合は，アパートメントスレッドではなくフリー（または両用）スレッドモデルを使ってください。典型的な例として，リモートマシン上のオブジェクトに接続するクライアントアプリケーションがあります。リモートクライアントがそのオブジェクトのメソッドを呼び出すと，サーバーはサーバーマシン上のスレッドプールの中の 1 つのスレッドで，その呼び出しを受信します。この受信側スレッドは実際のオブジェクトをローカルに呼び出します。このオブジェクトはフリースレッドモデルをサポートしているので，スレッドはオブジェクトを直接呼び出すことができます。

オブジェクトがフリースレッドではなくアパートメントスレッドモデルをサポートしている場合は，そのオブジェクトが作成されたスレッドに呼び出しを転送しなければなりません。さらに，結果をクライアントに返す前に，まず受信側スレッドに転送する必要があります。この方法は余計なマーシャリングを必要とします。

フリースレッドをサポートするためには，それぞれのメソッドについて，どのようにしてインスタンスデータにアクセスできるかを考慮する必要があります。インスタンスデータに対して書き込みを行うメソッドの場合は，クリティカルセクションやほかの形のシリアル化を使ってインスタンスデータを保護しなければなりません。重要な呼び出しをシリアル化することによるオーバーヘッドは，COM のマーシャリングコードを実行することによるオーバーヘッドより小さいことがほとんどです。

インスタンスデータが読み出し専用の場合には，シリアル化は不要です。

フリースレッドのインプロセスサーバーは，フリースレッドのマーシャラとの集約において外部オブジェクトとして機能することで，処理効率を向上できます。フリースレッドのマーシャラは，フリースレッドでないホスト（クライアント）によってフリースレッドの DLL が呼び出されたときに，COM の標準スレッド処理のショートカットを提供します。

フリースレッドのマーシャラと集約するには，以下のことをしなければなりません。

- CoCreateFreeThreadedMarshaler を呼び出して，結果としてできるフリースレッドのマーシャラが使用する，オブジェクトの IUnknown インターフェースを渡す

  ```
  CoCreateFreeThreadedMarshaler(static_cast<IUnknown *>(this), &FMarshaler);
  ```

  この行は，フリースレッドのマーシャラのインターフェースをクラスメンバー FMarshaler に割り当てる
- タイプライブラリエディタを使って，CoClass が実装するインターフェースのセットに IMarshal インターフェースを追加する

- オブジェクトの QueryInterface メソッドにおいて，（前述の FMarshaler として格納された）フリースレッドのマーシャラに IDD_IMarshal の呼び出しを委任する

警告 フリースレッドのマーシャラでは，COM マーシャリングによって効率が向上するという通則が当てはまりません。使用の際は注意する必要があります。特に，インプロセスサーバーにおいてフリースレッドのオブジェクトとしか集約しないようにする必要があり，（別のスレッドではなく）これを使用するオブジェクトによってのみインスタンス化するようにする必要があります。

アパートメントスレッドモデルをサポートするオブジェクトを作成する

アパートメントスレッド（シングルスレッド）モデルを実装するためには，以下のルールに従わなければなりません。

- 作成されるアプリケーションの中の最初のスレッドが COM のメインスレッドである。通常これは，WinMain が呼び出されたスレッドである。このスレッドは，COM を初期化解除する最後のスレッドでもなければならない
- アパートメントスレッドモデルの中の各スレッドは，メッセージループを持っていなければならず，頻繁にメッセージキューをチェックしなければならない
- スレッドが COM インターフェースへのポインタを取得したとき，そのポインタはそのスレッドでのみ使用できる

シングルスレッドアパートメントモデルは，スレッドのサポートを提供しないことと，フリースレッドモデルの完全なマルチスレッドサポートとの中間に位置します。アパートメントモデルを使っているサーバーでは，そのすべてのグローバルデータ（オブジェクトカウントなど）に対するアクセスがシリアル化されています。これは，複数の異なるオブジェクトが異なるスレッドからグローバルデータにアクセスしようと試みる可能性があるからです。ただし，メソッドは常に同じスレッドから呼び出されるので，オブジェクトのインスタンスデータは安全です。

通常，Web ブラウザ内で使われるコントロールはアパートメントスレッドモデルを使います。ブラウザアプリケーションが自分のスレッドを常にアパートメントとして初期化するからです。

ニュートラルスレッドモデルをサポートするオブジェクトを作成する

COM+ では，フリースレッドとアパートメントスレッドの中間のスレッドモデルであるニュートラルモデルも使用できます。フリースレッドモデルと同様に，ニュートラルモデルでは複数のスレッドが同時にオブジェクトにアクセスできます。オブジェクトが作成されたスレッドに転送するための余計なマーシャリングがありません。ただし，衝突する呼び出しをオブジェクトが受け取らないことが保証されます。

ニュートラルスレッドモデルを使用するオブジェクトの作成ルールは，アパートメントスレッドオブジェクトの作成ルールとほぼ同じですが，オブジェクトのインターフェース内の別のメソッドがインスタンスデータをアクセスできる場合は，このデータをスレッドの衝突から保護する必要がある点が異なります。1 つのインターフェースメソッドしかアクセスできないインスタンスデータは，自動的にスレッドセーフになります。

ATL オプションを指定する

ウィザードを使用して COM オブジェクトを作成すると，［プロジェクトオプション］ダイアログに［ATL］ページが追加されます。このページを使用すると，COM の初期化またはアプリケーションの登録を行う ATL 呼び出し用に生成されるパラメータを制御するアプリケーションレベルのフラグ，および ATL を使用するコード内でデバッグトレースが使用できるかどうかを制御するフラグを設定できます。

［ATL］ページでは以下のオプションを設定します。

- **インスタンスの生成：**アプリケーションがインプロセスサーバーでない場合，［インスタンスの生成］は，オブジェクトクライアントのインスタンスを 1 つのプロセス空間内で複数作成できるかどうかを決定します。［シングルインスタンス］を指定した場合，いったんクライアントがオブジェクトをインスタンス化すると，COM がそのアプリケーションを見えなくしてしまうので，ほかのクライアントは各自でアプリケーションのインスタンスを起動しなければなりません。［マルチインスタンス］を指定した場合，複数のクライアントがオブジェクトのインスタンスを各自で作成でき，すべてのインスタンスが同じプロセス空間内で実行されます。
- **OLE 初期化時のフラグ：**このフラグは，アプリケーションによって COM がどのように初期化されるかを決定します。アパートメントスレッドを指定すると，各オブジェクトは常に自分が作成されたスレッドで呼び出され，マルチスレッドを指定すると，オブジェクトを複数のスレッドで呼び出すことができます。これらの COINIT フラグは，アプリケーション内のスレッドモデルオブジェクトが何を使用できるかに影響を与えます。COINIT フラグをアパートメントスレッドに設定すると，アプリケーションの中でシングルスレッド（アパートメントスレッド）オブジェクトしか使用できません。その他のスレッドモデルで登録されているオブジェクトは，デフォルトでアパートメントスレッドになります。
- **デバッグ：**IDE でアプリケーションを実行するときに ATL 呼び出しをイベントログに記録するように指定するフラグを設定することもできます。IUnknown メソッド（QueryInterface および参照カウント呼び出し）の呼び出しをトレースしたり，すべての ATL 呼び出しをトレースすることができます。

COM オブジェクトのインターフェースを定義する

ウィザードを使用して COM オブジェクトを作成すると，タイプライブラリが自動生成されます。タイプライブラリは，そのオブジェクトが何をできるかをホストアプリケーションが調べられる手段を提供します。タイプライブラリエディタを使用すると，オブジェクトのインターフェースを定義することもできます。タイプライブラリエディタで定義するインターフェースは，オブジェクトがクライアントにエクスポーズするプロパティ，メソッド，およびイベントを定義します。

メモ COM オブジェクトウィザードで既存のインターフェースを選択した場合は，プロパティとメソッドを追加する必要はありません。既存のインターフェースの定義は，そのインターフェースが定義されたタイプライブラリからインポートされます。そのかわりに，インポートされたインターフェースのメソッドを実装ユニットで探し出し，中身を埋めるだけで済みます。

オブジェクトのインターフェースにプロパティを追加する

タイプライブラリエディタを使ってオブジェクトのインターフェースにプロパティを追加すると，そのプロパティの値を読みだすメソッドと，プロパティの値を設定するメソッドのどちらかまたは両方が，自動的に追加されます。タイプライブラリエディタは，これらのメソッドを実装クラスに追加し，中身を埋めるだけで完成できる空のメソッド実装を実装ユニット内で作成します。

オブジェクトのインターフェースにプロパティを追加する手順は次のとおりです。

1. タイプライブラリエディタ内で，オブジェクトのデフォルトインターフェースを選択します。

   デフォルトのインターフェースは，オブジェクトの名前の前に「I」を付けたものです。デフォルトを調べるには，タイプライブラリエディタで［CoClass］を選択して［実装するインターフェース］タブを選択し，実装されるインターフェースの一覧から「Default」と表示されているインターフェースを確認します。

2. 読み出しプロパティまたは書き込みプロパティを提供するには，ツールバーの［プロパティ］ボタンをクリックします。または，［プロパティ］ボタンの横にある矢印をクリックし，提供するプロパティの種類をクリックします。

3. ［属性］タブで，プロパティの名前と型を指定します。

4. ツールバーの［ソースコードの更新］ボタンを選択します。

   プロパティアクセスメソッドの定義とスケルトン実装が，オブジェクトの実装ユニットに挿入されます。

5. 実装ユニット内で，プロパティのアクセスメソッドを探し出します。このメソッドは，get_PropertyName および set_PropertyName の形式の名前を持ち，例外を捕らえて HRESULT を返すコードだけが入っています。オブジェクトのプロパティ値を取得または設定するコードを（**try** 文と **catch** 文の間に）追加します。このコードは，アプリケーション内の既存の関数の呼び出し，オブジェクト定義に追加するデータメンバーへのアクセス，またはその他の方法でのプロパティの実装のような単純なものになります。

オブジェクトのインターフェースにメソッドを追加する

タイプライブラリエディタを使ってオブジェクトのインターフェースにメソッドを追加すると，タイプライブラリエディタは，このメソッドを実装クラスに追加し，中身を埋めるだけで完成できる空の実装を実装ユニット内で作成します。

オブジェクトのインターフェースを介してメソッドをエクスポーズする手順は次のとおりです。

1. タイプライブラリエディタ内で，オブジェクトのデフォルトインターフェースを選択します。

   デフォルトのインターフェースは，オブジェクトの名前の前に「I」を付けたものです。デフォルトを調べるには，タイプライブラリエディタで［CoClass］を選択して［実装するインターフェー

ス］タブを選択し，実装されるインターフェースの一覧から「Default」と表示されているインターフェースを確認します。

2. ［メソッドの新規作成］ボタンをクリックします。
3. ［属性］タブで，メソッドの名前を指定します。
4. ［パラメータ］タブで，メソッドの戻り型を指定し，適切なパラメータを追加します。
5. ツールバーの［ソースコードの更新］ボタンを選択します。

   メソッドの定義とスケルトン実装が，オブジェクトの実装ユニットに挿入されます。
6. 実装ユニットで，新規に挿入されたメソッド実装を探し出します。このメソッドは完全に空です。このメソッドが表すタスクを実行するようにメソッドの中身を埋めます。

クライアントにイベントをエクスポーズする

COM オブジェクトが生成できるイベントは，従来型のイベントと COM+ イベントの 2 種類です。

- COM+ イベントの場合は，イベントオブジェクトウィザードを使って別個のイベントオブジェクトを作成し，サーバーオブジェクトからそのイベントオブジェクトを呼び出すコードを追加する必要がある。COM+ イベントの生成方法についての詳細は，44-20 ページの「COM+ でのイベントの生成」を参照
- イベントオブジェクトウィザードを使うと，従来型のイベントの生成作業の大部分を処理できる。このプロセスについては後述する

オブジェクトにイベントを生成させる手順は次のとおりです。

1. ウィザードで，［イベントサポートのためのコードを生成］チェックボックスにチェックマークを付けます。

   イベントインターフェースおよびデフォルトインターフェースを含むオブジェクトが作成されます。このイベントインターフェースの名前は ICoClassnameEvents の形式になります。これは外向き（ソース）インターフェースです。つまり，オブジェクトが実装するインターフェースではなく，クライアントが実装しなければならず，オブジェクトが呼び出すインターフェースです（これを確認するには，CoClass を選択し，［実装するインターフェース］ページを表示し，イベントインターフェースの Source 列に **true** と表示されているのを確認します）。イベントインターフェースの GUID が接続ポイントマップに追加され，このマップがオブジェクトの宣言内のインターフェースマップの下に表示されます。接続ポイントマップについての詳細は，ATL のドキュメントを参照してください。

   イベントインターフェースのほかにも，追加の基本クラスがオブジェクトに追加されます。追加の基本クラスとは，IConnectionPointContainer インターフェースの実装（IConnectionPointContainerImpl），全クライアントへのイベント送出を管理するテンプレート化さ

れた基本クラス（TEvents_CoClassName）などです。この後者のクラスのテンプレートは，_TLB ユニットヘッダーにあります。

2. タイプライブラリエディタ内で，オブジェクトの外向きイベントインターフェースを選択します（これは ICoClassNameEvents 形式の名前を持つインターフェースです）。
3. タイプライブラリツールバーの［メソッドの新規作成］ボタンをクリックします。イベントインターフェースに追加した各メソッドは，クライアントが実装しなければならないイベントハンドラを表します。
4. ［属性］タブで，イベントハンドラの名前（たとえば MyEvent）を指定します。
5. ツールバーの［ソースコードの更新］ボタンを選択します。

   これでオブジェクト実装には，クライアントイベントシンクを受け入れ，イベントが発生したときに呼び出すインターフェースのリストを保守するために必要なものがすべて揃いました。これらのインターフェースを呼び出すには，（TEvents_CoClassName が実装する）イベントを送出するメソッドを呼び出して，クライアント上でイベントを生成します。各イベントハンドラに対して，このメソッドは Fire_EventHandlerName 形式の名前を持ちます。

6. イベントを送出する際にイベントの発生がクライアントに通知されるようにする必要がある場合は，そのたびに，すべてのイベントシンクにそのイベントをディスパッチするメソッドを呼び出します。

   ```
   if (EventOccurs) Fire_MyEvent; // イベントを送出するために作成したメソッドを呼び出す
   ```

オートメーションオブジェクト内のイベントの管理

サーバーは，従来型の COM イベントをサポートするために，クライアントによって実装される外向きインターフェースを定義しなければなりません。この外向きインターフェースには，サーバーイベントに応答するためにクライアントが実装しなければならないすべてのイベントハンドラが含まれます。

クライアントは外向きイベントインターフェースを実装したときに，サーバーの IConnectionPointContainer インターフェースに問い合わせることで，イベントの通知を受けたいということを登録します。IConnectionPointContainer インターフェースは，サーバーの IConnectionPoint インターフェースを返し，クライアントはこのインターフェースを使用して，（シンクと呼ばれる）イベントハンドラの実装へのポインタをサーバーに渡します。

サーバーは，前述のように，すべてのクライアントシンクのリストを保守し，イベントが発生したときにはそれらのシンク上でメソッドを呼び出します。

オートメーションインターフェース

オートメーションオブジェクトウィザードは，デフォルトでデュアルインターフェースを実装します。したがって，オートメーションオブジェクトは以下のバインディングを両方ともサポートします。

- 実行時のレイトバインディング（IDispatch インターフェースを介した）。これはディスパッチインターフェース，つまり **dispinterface** として実装される

- コンパイル時のアーリーバインディング。これはオブジェクトの仮想関数テーブル（VTable）内のメンバー関数の 1 つを直接呼び出すことによって実装される。これは**カスタムインターフェース**と呼ばれる

メモ COM オブジェクトウィザードによって生成され，IDispatch を継承しないインターフェースは，仮想テーブル呼び出しのみをサポートします。

デュアルインターフェース

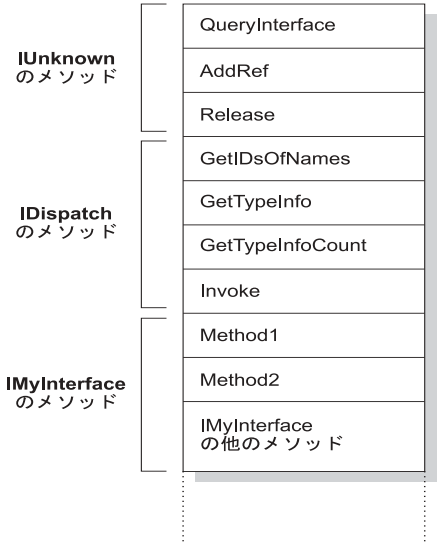
デュアルインターフェースはカスタムインターフェースであると同時にディスパッチインターフェースでもあります。これは IDispatch から派生した COM VTable インターフェースとして実装されます。実行時にしかオブジェクトにアクセスできないコントローラのために，ディスパッチインターフェースが用意されています。コンパイル時バインディングを利用できるオブジェクトには，より効率的な仮想テーブルインターフェースが使われます。

デュアルインターフェースは以下に示したように，仮想テーブルインターフェースとディスパッチインターフェース両方の利点を備えています。

- 型情報を取得できないオートメーションコントローラに対しては，ディスパッチインターフェースがオブジェクトへの実行時アクセスを提供する
- インプロセスサーバーの場合は，仮想テーブルインターフェースを介した高速アクセスという利点がある
- アウトオブプロセスサーバーの場合は，仮想テーブルインターフェースについてもディスパッチインターフェースについても COM がデータをマーシャリングする。COM はタイプライブラリの情報に基づいてインターフェースをマーシャリングできるユニバーサルなプロクシー / スタブ実装を提供する。マーシャリングについての詳細は，第 41 章-15 の「データのマーシャリング」を参照

次の図は，IMyInterface というデュアルインターフェースをサポートするオブジェクトの中の IMyInterface インターフェースを示しています。デュアルインターフェースの仮想テーブルの最初の 3 つのエントリは IUnknown インターフェースを参照し，次の 4 つのエントリは IDispatch インターフェースを参照します。残りのエントリはカスタムインターフェースのメンバーに直接アクセスするための COM エントリです。

図 41.1 デュアルインターフェースの仮想テーブル



ディスパッチインターフェース

オートメーションコントローラは，COM IDispatch インターフェースを使って COM サーバーオブジェクトにアクセスするクライアントです。コントローラはまずオブジェクトを作成してから，そのオブジェクトの IUnknown インターフェースに対して IDispatch インターフェースへのポインタを問い合わせる必要があります。IDispatch は，インターフェースメンバーを示すユニークな識別番号であるディスパッチ識別子（dispID）によって，メソッドとプロパティを内部的に追跡します。コントローラは IDispatch を介してディスパッチインターフェース用にオブジェクトの型情報を取り出してから，インターフェースメンバー名を特定の dispID にマップします。これらの dispID は実行時に入手できるので，コントローラは IDispatch メソッドの GetIDsOfNames を呼び出して dispID を取得します。

dispID を取得したコントローラは，適切なコード（プロパティまたはメソッド）を実行するために IDispatch メソッド Invoke を呼び出し，そのプロパティまたはメソッドに関するパラメータを Invoke パラメータの 1 つにパッケージ化します。Invoke にはコンパイル時の固定した名前が含まれているので，インターフェースメソッドを呼び出すときに引数をいくつでも受け付けることができます。

オートメーションオブジェクトの Invoke の実装はパラメータのパッケージを解除し，プロパティまたはメソッドを呼び出さなければなりません。エラーが発生した場合の処理に備える必要もあります。プロパティまたはメソッドからリターンしたら，オブジェクトはその戻り値をコントローラに返します。

コントローラがコンパイル時ではなく実行時にプロパティまたはメソッドにバインドするので，これをレイトバインディングと呼んでいます。

メモ C++Builder は，デュアルインターフェースでないディスパッチインターフェースを持つ CoClass を作成できません。なぜなら，C++Builder の COM サポートは ATL に依存しており，ATL はデュアル以外のディスパッチインターフェースをサポートしないからです。

メモ タイプライブラリをインポートするとき，C++Builder は，コード生成時に dispID を問い合わせます。このため，生成されたラッパークラスは GetIDsOfNames を呼び出さずに Invoke を呼び出すことができます。その結果，コントローラの実行時パフォーマンスが大幅に向上します。

カスタムインターフェース

カスタムインターフェースは，クライアントが仮想テーブルでの順序と引数の型に基づいてインターフェースメソッドを呼び出せるようにする，ユーザー定義インターフェースです。仮想テーブルはオブジェクトのメンバーであるすべてのプロパティとメソッドのアドレスを一覧にしたもので，オブジェクトがサポートするインターフェースのメンバー関数も含まれています。オブジェクトが IDispatch をサポートしない場合，そのオブジェクトのカスタムインターフェースのメンバーのエントリが IUnknown のメンバーのすぐ後に続きます。

オブジェクトにタイプライブラリがある場合は，VTable レイアウトを通じてカスタムインターフェースにアクセスできます。これは，タイプライブラリエディタを使って取得できます。オブジェクトにタイプライブラリがあって，IDispatch もサポートしている場合は，クライアントも IDispatch インターフェースの dispID を取得して VTable オフセットに直接バインドすることができます。C++Builder のタイプライブラリインポータ（TLIBIMP）はインポート時に dispID を取得するので，ディスパッチインターフェースラッパーを使用するクライアントは，GetIDsOfNames への呼び出しを回避できます。この情報は，すでに _TLB ユニット内に存在します。ただし，この場合でもクライアントは Invoke を呼び出す必要はあります。

データのマーシャリング

アウトオブプロセスサーバーとリモートサーバーについては，COM が現在のプロセスの外にあるデータをどのようにマーシャリングするかを考慮する必要があります。マーシャリングは以下の方法で提供できます。

- IDispatch インターフェースを使うことによって自動的に行われる
- サーバーにタイプライブラリを作成し，インターフェースに OLE Automation フラグを付けることによって自動的に行われる。COM はタイプライブラリ内のすべての**オートメーション互換型**をマーシャリングする方法を知っていて，プロクシーとスタブを自動的にセットアップする。自動マーシャリングを有効にするために，型に関するいくつかの制約が適用される
- IMarshal インターフェースのすべてのメソッドを実装することによって手動で行う。これは**カスタムマーシャリング**と呼ばれる

メモ 最初の方法（IDispatch を使用する方法）は，オートメーションサーバー上でしか使用できません。ウィザードで作成され，かつタイプライブラリを使用するオブジェクトであれば，すべて自動的に 2 番目の方法を使用できます。

オートメーション互換型

デュアルインターフェース，ディスパッチインターフェース，および OLE Automation のマークを付けたインターフェースの中で宣言されるメソッドの関数の結果とパラメータの型は，**オートメーション互換**型でなければなりません。以下の型は OLE オートメーション互換です。

- short，int，single，double，WideString などの定義済みの有効な型。完全なリストについては，39-10 ページの「有効な型」を参照
- タイプライブラリの中で定義された列挙型。OLE オートメーション互換の列挙型は 32 ビット値として保存され，パラメータとして渡す目的で Integer 型の値として取り扱われる
- タイプライブラリの中で OLE オートメーションが安全であると定義されているインターフェース型。つまり，IDispatch から派生して OLE オートメーション互換型だけを含んでいるもの
- タイプライブラリの中で定義された dispinterface 型
- タイプライブラリの中で定義されたカスタムのレコード型
- IFont，IStrings，および IPicture。以下のマッピングを行うためにヘルパーオブジェクトをインスタンス化しなければならない
  - IFont を TFont に
  - IStrings を TStrings に
  - IPicture を TPicture に

  ActiveX コントロールウィザードと Active フォームウィザードは，必要なときにこれらのヘルパーオブジェクトを自動的に作成します。これらのヘルパーオブジェクトを使うには，それぞれグローバルルーチン GetOleFont，GetOleStrings，GetOlePicture を呼び出します。

自動マーシャリングのための型の制約

インターフェースが自動マーシャリング（オートメーションマーシャリング，タイプライブラリマーシャリングとも呼ばれる）をサポートするためには，以下の制約が適用されます。タイプライブラリエディタを使ってオブジェクトを編集すると，エディタがこれらの制約を強制します。

- 型はプラットフォーム間通信で互換性がなければならない。たとえば，データ構造（別のプロパティオブジェクトを実装する場合以外），符号なしの引数，AnsiString などは使えない
- 文字列データ型は，BSTR として転送しなければならない。PChar と AnsiString は，安全にマーシャリングすることはできない
- デュアルインターフェースのすべてのメンバーは関数の戻り値として HRESULT を渡さなければならない
- ほかの値を返す必要があるデュアルインターフェースのメンバーは，それらのパラメータを **var** または **out** と指定して，関数値を返す出力パラメータであることを示さなければならない

メモ オートメーションの型の制約を回避するには，IDispatch インターフェースとカスタムインターフェースを別々に実装するという方法があります。それによって，すべての引数型を使えるようになります。この場合，COM クライアントはカスタムインターフェースを使うことができ，オートメー

ションコントローラは引き続きカスタムインターフェースにアクセスできます。ただし，マーシャリングコードを手動で実装しなければなりません。

カスタムマーシャリング

一般に，アウトオブプロセスサーバーとリモートサーバーでは自動マーシャリングを使います。COM が自動的に作業を実行してくれるので，簡単だからです。ただし，カスタムマーシャリングの方がマーシャリングの処理効率を上げられると判断した場合は，カスタムマーシャリングを提供することもできます。独自のカスタムマーシャリングを実装する場合は，IMarshal インターフェースをサポートしなければなりません。この方法についての詳細は，Microsoft の資料を参照してください。

COM オブジェクトを登録する

サーバーオブジェクトは，インプロセスサーバーまたはアウトオブプロセスサーバーとして登録できます。サーバータイプについての詳細は，38-6 ページの「インプロセス，アウトオブプロセス，およびリモートサーバー」を参照してください。

メモ COM オブジェクトをシステムから削除するときは，その前に登録を解除しなければなりません。

インプロセスサーバーの登録

インプロセスサーバー（DLL または OCX）を登録する手順は次のとおりです。

- ［実行｜ ActiveX サーバーの登録］を選択します。

インプロセスサーバーを登録解除する手順は次のとおりです。

- ［登録｜ ActiveX サーバーの削除］を選択します。

アウトオブプロセスサーバーの登録

アウトオブプロセスサーバーを登録する手順は次のとおりです。

- **/regserver** コマンドラインオプションを付けてサーバーを実行します。

  コマンドラインオプションは，［実行｜実行時引数］ダイアログボックスで設定できます。

  サーバーを実行することによって，登録することもできます。

アウトオブプロセスサーバーを登録解除する手順は次のとおりです。

- **/unregserver** コマンドラインオプションを付けてサーバーを実行します。

かわりに，コマンドラインから **tregsvr** コマンドを使うか，オペレーティングシステムから regsvr32.exe を実行するという方法もあります。

メモ COM+ での使用を目的とする COM サーバーの場合は，登録するのではなく，COM+ アプリケーションにインストールする必要があります（COM+ アプリケーションにオブジェクトをインストー

ルすると，登録が自動的に処理されます)。COM+ アプリケーションにオブジェクトをインストールする方法については，44-27 ページの「トランザクションオブジェクトのインストール」を参照してください。

アプリケーションのテストとデバッグ

COM サーバーアプリケーションをテストおよびデバッグする手順は次のとおりです。

1. 必要であれば，［プロジェクト｜オプション］ダイアログボックスの［コンパイラ］タブを使って，デバッグ情報をオンにします。さらに，［ツール｜デバッガオプション］ダイアログの中の［統合開発環境を使う］をオンにします。
2. インプロセスサーバーの場合は，［実行｜実行時引数］を選択して，［ホストアプリケーション］ボックス内にオートメーションコントローラの名前を入力し，［OK］を選択します。
3. ［実行｜実行］を選択します。
4. オートメーションサーバー内にブレークポイントを設定します。
5. オートメーションコントローラを使って，オートメーションサーバーと対話します。

ブレークポイントに達すると，オートメーションサーバーは一時停止します。

さらに，アプリケーションがインターフェースに対して行う呼び出しをトレースすると役立つことがあります。COM 呼び出しの流れを調べると，アプリケーションが期待どおりの動作をしているかどうかを確認できます。COM インターフェースが呼び出されるたびにイベントログにメッセージを追加するように C++Builder に指示するには，［プロジェクトオプション］ダイアログの［ATL］ページを使用してデバッグオプションを指定します。

メモ オートメーションコントローラも作成する場合は，プロセス間 COM サポートを有効にしてインプロセスサーバーの内部をデバッグすることもできます。プロセス間サポートを有効にするには，［ツール｜デバッガオプション］ダイアログの［一般］ページを使用します。

第42章

ASP（Active Server Page）の作成

自分の Web ページに IIS（Microsoft Internet Information Server）環境を使用すると，ASP（Active Server Page）を使って動的な Web ベースのクライアント / サーバーアプリケーションを作成できます。ASP を使うと，サーバーが Web ページをロードするたびに呼び出されるスクリプトを作成できます。このスクリプトは，オートメーションオブジェクトを呼び出して，生成される HTML ページに含まれる情報を取得することができます。たとえば，ビットマップを作成したりデータベースに接続したりする C++Builder オートメーションサーバーを書いて，このコントロールを使って，サーバーが Web ページをロードするたびに更新されるデータにアクセスするようにできます。

クライアント側では，ASP は，標準 HTML ドキュメントのように動作し，ユーザーはどのプラットフォーム上でどの Web ブラウザを使っても表示できます。

ASP アプリケーションは，C++Builder の WebBroker 技術を使って作成したアプリケーションに似ています。WebBroker 技術についての詳細は，第 32 章「インターネットサーバーアプリケーションの作成」を参照してください。ただし，ASP は，ビジネスルールまたは複雑なアプリケーションロジックの実装から UI 設計を分離する方法が異なります。

- UI 設計が，ASP によって管理される。これは本質的には HTML ドキュメントだが，ビジネスルールまたはアプリケーションロジックを反映するコンテンツを供給するために Active Server オブジェクトを呼び出す埋め込みスクリプトを含むことができる
- アプリケーションロジックは，ASP に単純なメソッドをエクスポーズする Active Server オブジェクトによってカプセル化され，必要なコンテンツを供給する

メモ ASP はアプリケーションロジックから UI 設計を分離する点が優れていますが，処理効率のスケールが限られています。非常に多数のクライアントに応答する Web サイトの場合は，ASP ではなく Web ブローカ技術を使用する方法をお勧めします。

ASP 内のスクリプト，およびアクティブサーバーページに埋め込むオートメーションオブジェクトは，ASP 組み込みオブジェクト（現在のアプリケーション，ブラウザからの HTTP メッセージなどの情報を提供する組み込みオブジェクト）を利用できます。

この章では，Active Server オブジェクトウィザードを使って Active Server オブジェクトを作成する方法について説明します。そうすれば，この特殊なオートメーションコントロールは，ASP によって呼び出され，コンテンツを供給することができます。

Active Server オブジェクトを作成する手順は次のとおりです。

- アプリケーションの Active Server オブジェクトを作成する
- Active Server オブジェクトのインターフェースを定義する
- Active Server オブジェクトを登録する
- アプリケーションをテストおよびデバッグする

Active Server オブジェクトの作成

Active Server オブジェクトは，ASP アプリケーション全体，およびブラウザと通信するために使用する HTTP メッセージについての情報にアクセスできるオートメーションオブジェクトです。このオブジェクトは TASPObject または TMTSASPObject（および ATL 基本クラスの CComObjectRootEx と CComCoClass）を継承し，オートメーションプロトコルをサポートし，ほかのアプリケーション（または ASP 内のスクリプト）で使用できるように自分自身をエクスポーズします。Active Server オブジェクトは，Active Server オブジェクトウィザードを使って作成します。

Active Server オブジェクトのプロジェクトは，必要に応じて，実行可能ファイル（exe）またはライブラリ（dll）のどちらかにすることができます。ただし，アウトオブプロセスサーバーの使用については，短所を理解しておく必要があります。この短所については，42-7 ページの「インプロセスまたはアウトオブプロセスサーバー用の ASP の作成」を参照してください。

Active Server オブジェクトウィザードを表示する手順は次のとおりです。

1. ［ファイル｜新規作成｜その他］を選択します。
2. ［ActiveX］タブを選択します。
3. ［Active Server オブジェクト］アイコンをダブルクリックします。

このウィザードで，新規の Active Server オブジェクトに名前を付け，サポートしたいスレッドモデルを指定します。このモデルに忠実に（たとえばスレッドの衝突を防止するように）実装を作成しなければなりません。選択できるスレッドモデルは，ほかの COM オブジェクトで選択できるものと同じです。詳細については，41-5 ページの「スレッドモデルを選択する」を参照してください。

Active Server オブジェクトのユニークな点は，ASP アプリケーションについての情報，および ASP とクライアント Web ブラウザとの間で渡される HTTP メッセージについての情報にアクセスできることです。この情報は，ASP 組み込みオブジェクトを使ってアクセスされます。このウィザードでは，Active Server の種類を設定して，オブジェクトがこれらにどのようにアクセスするかを指定できます。

- IIS 3 または IIS 4 を使用している場合は，［ページレベルイベントメソッド］を使用します。このモデルでは，オブジェクトが *OnStartPage* メソッドおよび *OnEndPage* メソッドを実装し，ASP がロードおよびアンロードするときにこれらのメソッドが呼び出されます。オブジェクトがロードされると，自動的に IScriptingContext インターフェースを取得し，このインターフェースを使用

して ASP 組み込みオブジェクトにアクセスします。そして，これらのインターフェースは，基本クラス（TASPObject）から継承されたプロパティとして公開されます。

- IIS5 以降を使用している場合は，［Object Context］を使用します。このモデルでは，オブジェクトが IObjectContext インターフェースをフェッチし，これを使用して ASP 組み込みオブジェクトにアクセスします。この方法でも，これらのインターフェースは，継承された基本クラス（TMTSASPObject）内のプロパティとして公開されます。この後者の方法の方が優れている点の 1 つは，IObjectContext を介して使用できるその他のサービスのすべてにオブジェクトがアクセスできることです。IObjectContext インターフェースにアクセスするには，Active Server オブジェクトの ObjectContext プロパティを使用します。IObjectContext を介して使用できるサービスについての詳細は，第 44 章「MTS オブジェクトまたは COM+ オブジェクトの作成」を参照してください。

新規の Active Server オブジェクトのホストになる単純な ASP ページを生成するようにウィザードに指示できます。生成されたページは，ProgID に基づいて Active Server オブジェクトを作成する（VBScript で書かれた）最低限のスクリプトを提供し，どこでそのメソッドを呼び出せるかを示します。このスクリプトは Server.CreateObject を呼び出して，Active Server オブジェクトを起動します。

メモ 生成されたテストスクリプトは VBScript を使用していますが，ASP は Jscript を使って作成することもできます。

このウィザードを終了すると，Active Server オブジェクトの定義が入っている現在のプロジェクトに，新規ユニットが追加されます。さらに，タイプライブラリプロジェクトが追加され，タイプライブラリエディタが開かれます。これで，41-9 ページの「COM オブジェクトのインターフェースを定義する」で説明するようにタイプライブラリを介してインターフェースのプロパティとメソッドを提供できるようになります。オブジェクトのプロパティとメソッドの実装を書くときに，ASP 組み込みオブジェクト（後述）を利用して，ASP アプリケーションについての情報，およびブラウザと通信するために使用する HTTP メッセージについての情報を取得することができます。

Active Server オブジェクトは，ほかのオートメーションオブジェクトと同様に，**デュアルインターフェース**を実装します。このインターフェースは，VTable を介したアーリー（コンパイル時）バインディングも，IDispatch インターフェースを介したレイト（実行時）バインディングもサポートしています。デュアルインターフェースについての詳細は，41-13 ページの「デュアルインターフェース」を参照してください。

ASP 組み込みオブジェクトの使い方

ASP 組み込みオブジェクトは，ASP 内で実行されるオブジェクトに対して ASP が供給する COM オブジェクトのセットです。ASP 組み込みオブジェクトを使用すると，Active Server オブジェクトは，アプリケーションと Web ブラウザの間で渡されるメッセージを反映する情報，および同じ ASP アプリケーションに属す Active Server オブジェクトの間で共有される情報を格納する場所にアクセスできます。

これらのオブジェクトに簡単にアクセスできるようにするために，Active Server オブジェクトの基本クラスはこれらをプロパティとして公開します。これらのオブジェクトについて完全に理解したい場合は，Microsoft の資料を参照してください。ここでは概要を説明します。

アプリケーション

アプリケーションオブジェクトは，IApplicationObject インターフェースを介してアクセスされます。これは ASP アプリケーション全体（仮想ディレクトリおよびそのサブディレクトリ内のすべての .asp ファイルのセットとして定義される）を表します。アプリケーションオブジェクトは複数のクライアントによって共有できるので，スレッドの衝突を防止するために使用する必要があるロックサポートを含んでいます。

IApplicationObject には以下のものが含まれます。

表 42.1 IApplicationObject インターフェースメンバー

プロパティ，メソッド，またはイベント	説明
Contents プロパティ	スクリプトコマンドを使ってアプリケーションに追加されたすべてのオブジェクトをリストする。このインターフェースには Remove と RemoveAll の 2 つのメソッドがあり，これを使用するとリストから 1 つまたは全部のオブジェクトを削除できる
StaticObjects プロパティ	<OBJECT> タグを使ってアプリケーションに追加されたすべてのオブジェクトをリストする
Lock メソッド	Unlock を呼び出すまでほかのクライアントがアプリケーションオブジェクトをロックしないようにする。すべてのクライアントは，（プロパティなどの）共有メモリにアクセスする前に Lock を呼び出す必要がある
Unlock メソッド	Lock メソッドを使って設定されたロックを解除する
Application_OnEnd イベント	アプリケーションが終了するときに，Session_OnEnd イベントの後に発生する。使用可能な組み込みオブジェクトは Application と Server だけである。このイベントハンドラは VBScript または JScript で作成しなければならない
Application_OnStart イベント	新規のセッションが作成される前（Session_OnStart の前）に発生する。使用可能な組み込みオブジェクトは Application と Server だけである。このイベントハンドラは VBScript または JScript で作成しなければならない

リクエスト

リクエストオブジェクトは，IRequest インターフェースを介してアクセスされます。ASP を開かせた HTTP リクエストメッセージについての情報を提供します。

IRequest には以下のものが含まれます。

表 42.2 IRequest インターフェースメンバー

プロパティ，メソッド，またはイベント	説明
ClientCertificate プロパティ	HTTP メッセージとともに送信されるクライアント証明書の中の全項目の値を示す
Cookies プロパティ	HTTP メッセージ上のすべての Cookie ヘッダーの値を示す
Form プロパティ	HTTP 本体内のフォーム要素の値を示す。これは名前でアクセスできる
QueryString プロパティ	HTTP ヘッダーから問い合わせ文字列内のすべての変数の値を示す
ServerVariables プロパティ	さまざまな環境変数の値を示す。これらの変数は，共通 HTTP ヘッダー変数の大部分を表す

表 42.2 IRequest インターフェースメンバー（つづき）

プロパティ，メソッド，またはイベント	説明
TotalBytes プロパティ	リクエスト本体内のバイト数を示す。これは BinaryRead メソッドが返すバイト数の上限である
BinaryRead メソッド	Post メッセージのコンテンツを取り出す。このメソッドを呼び出して，読み出す最大バイト数を指定する。その結果のコンテンツは Variant 配列のバイトとして返される。BinaryRead を呼び出した後は，Form プロパティを使うことはできない

レスポンス

リクエストオブジェクトは，IResponse インターフェースを介してアクセスされます。このオブジェクトでは，クライアントブラウザに返される HTTP レスポンスメッセージについての情報を指定できます。

IResponse には以下のものが含まれます。

表 42.3 IResponse インターフェースメンバー

プロパティ，メソッド，またはイベント	説明
Cookies プロパティ	HTTP メッセージ上のすべての Cookie ヘッダーの値を決定する
Buffer プロパティ	ページ出力をバッファリングするかどうかを示す。ページ出力をバッファリングするときに，現在のページ上のすべてのサーバースクリプトが処理されるまで，サーバーはクライアントにレスポンスを送信しない
CacheControl プロパティ	プロクシーサーバーがレスポンス内の出力をキャッシュできるかどうかを決定する
Charset プロパティ	Content-Type ヘッダーに文字セットの名前を追加する
ContentType プロパティ	レスポンスメッセージ本体の HTTP コンテンツタイプを指定する
Expires プロパティ	ブラウザがレスポンスをキャッシュできる期限が切れるまでの期間を指定する
ExpiresAbsolute プロパティ	レスポンスが期限切れになる日付と時刻を指定する
IsClientConnected プロパティ	クライアントがサーバーから切断されたかどうかを示す
Pics プロパティ	レスポンスヘッダーの pics-label 項目の値を設定する
Status プロパティ	レスポンスのステータスを示す。これは HTTP ステータスヘッダーの値である
AddHeader メソッド	名前と値を指定して HTTP ヘッダーを追加する
AppendToLog メソッド	このリクエストの Web サーバーログの末尾に文字列を追加する
BinaryWrite メソッド	レスポンスメッセージの本体に生の（解釈されていない）情報を書き込む
Clear メソッド	バッファリングされた HTML 出力を消去する
End メソッド	.asp ファイルの処理を停止し，現在の結果を返す
Flush メソッド	バッファリングされた出力をただちに送信する
Redirect メソッド	リダイレクトレスポンスメッセージを送信し，異なる URL にクライアントブラウザをリダイレクトする
Write メソッド	現在の HTTP 出力に変数を文字列として書き込む

セッション

セッションオブジェクトは，ISessionObject インターフェースを介してアクセスされます。このオブジェクトを使用すると，ASP アプリケーションとのクライアントの対話の持続期間のあいだ持続する変数を格納できます。つまり，ASP アプリケーション内のページからページにクライアントが移動してもこの変数は解放されず，クライアントがアプリケーション全体を終了させてはじめて解放されます。

ISessionObject には以下のものが含まれます。

表 42.4 ISessionObject インターフェースメンバー

プロパティ，メソッド，またはイベント	説明
Contents プロパティ	<OBJECT> タグを使ってセッションに追加されたすべてのオブジェクトをリストする。リスト内の変数を名前でアクセスするか，コンテンツオブジェクトの Remove または RemoveAll メソッドを呼び出して値を削除することができる
StaticObjects プロパティ	<OBJECT> タグを使ってセッションに追加されたすべてのオブジェクトをリストする
CodePage プロパティ	シンボルマッピングに使用するコードページを指定する。ロケールが異なると，使用するコードページが異なる可能性がある
LCID プロパティ	文字列コンテンツを解釈するために使用するロケール ID を指定する
SessionID プロパティ	現在のクライアントのセッション ID を示す
TimeOut プロパティ	アプリケーションが終了するまでクライアントからのリクエスト（または更新）がない状態でセッションが持続する時間を（分単位で）指定する
Abandon メソッド	セッションを破棄し，リソースを解放する
Session_OnEnd イベント	セッションが中止されるかタイムアウトになったときに発生する。使用可能な組み込みオブジェクトは Application，Server，および Session だけである。このイベントハンドラは VBScript または JScript で作成しなければならない
Session_OnStart イベント	サーバーが新規のセッションを作成するときに発生する（Application_OnStart の後，かつ ASP 上のスクリプトを実行する前）。すべての組み込みオブジェクトが使用できる。このイベントハンドラは VBScript または JScript で作成しなければならない

サーバー

サーバーオブジェクトは，IServer インターフェースを介してアクセスされます。ASP アプリケーションを作成するためのさまざまなユーティリティを提供します。

IServer には以下のものが含まれます。

表 42.5 IServer インターフェースメンバー

プロパティ，メソッド，またはイベント	説明
ScriptTimeOut プロパティ	セッションオブジェクトの TimeOut プロパティと同じ
CreateObject メソッド	指定された Active Server オブジェクトをインスタンス化する
Execute メソッド	指定された .asp ファイル内のスクリプトを実行する
GetLastError メソッド	エラー状態を説明する ASPError オブジェクトを返す

表 42.5 IServer インターフェースメンバー（つづき）

プロパティ，メソッド，またはイベント	説明
HTMLEncode メソッド	HTML ヘッダーに使用する文字列をコード化し，予約文字を適切な定数シンボルで置き換える
MapPath メソッド	指定された仮想パス（現在のサーバー上の絶対パスまたは現在のページに対する相対パス）を物理パスにマップする
Transfer メソッド	現在のステート情報のすべてを，別の ASP に送信して処理できるようにする
URLEncode メソッド	指定された文字列に（エスケープ文字などの）URL コード化規則を適用する

インプロセスまたはアウトオブプロセスサーバー用の ASP の作成

ASP ページの **Server.CreateObject** を使って，インプロセスまたはアウトオブプロセスサーバーのどちらか必要な方を起動できます。ただし，インプロセスサーバーを起動するほうがより一般的です。

ほとんどのインプロセスサーバーと違って，インプロセスサーバー内の Active Server オブジェクトはクライアントのプロセス空間内では動作しません。そのかわりに，IIS プロセス空間内では動作します。つまり，（ActiveX オブジェクトを使用する場合のように）クライアントがアプリケーションをダウンロードする必要がありません。アウトオブプロセスサーバーと比べると，インプロセスコンポーネント DLL の方がより高速で安全なので，サーバー側での使用により適しています。

アウトオブプロセスサーバーは安全性が低いので，アウトオブプロセスの実行ファイルを使用できないように IIS を設定するのが普通です。この場合，Active Server オブジェクト用にアウトオブプロセスサーバーを作成すると，たとえば次のようなエラーメッセージが表示されます。

```
Server object error 'ASP 0196'
Cannot launch out of process component
/path/outofprocess_exe.asp, line 11
```

また，アウトオブプロセスコンポーネントは，各オブジェクトインスタンスについて個別のサーバープロセスを作成することがよくあるので，CGI アプリケーションよりも動作が鈍くなります。このコンポーネントには，コンポーネント DLL ほどのスケーラビリティはありません。

サイトにとってパフォーマンスとスケーラビリティを優先すべきなら，インプロセスサーバーを使用することを強くお勧めします。ただし，トラフィックが普通または少なめのイントラネットサイトでなら，サイト全体のパフォーマンスを低下させずにアウトオブプロセスコンポーネントを使用することができます。

インプロセスサーバーとアウトオブプロセスサーバーの一般的情報については，第 38 章-6 の「インプロセス，アウトオブプロセス，およびリモートサーバー」を参照してください。

Active Server オブジェクトの登録

ASP は，インプロセスサーバーまたはアウトオブプロセスサーバーとして登録できます。ただし，インプロセスサーバーのほうがよく使用されます。

メモ ASP オブジェクトをシステムから削除したい場合は，先に，そのエントリを Windows のレジストリから削除して，登録を解除する必要があります。

インプロセスサーバーの登録

インプロセスサーバー（DLL または OCX）を登録する手順は次のとおりです。

- ［実行｜ ActiveX サーバーの登録］を選択します。

インプロセスサーバーを登録解除する手順は次のとおりです。

- ［登録｜ ActiveX サーバーの削除］を選択します。

アウトオブプロセスサーバーの登録

アウトオブプロセスサーバーを登録する手順は次のとおりです。

- **/regserver** コマンドラインオプションでサーバーを実行します（コマンドラインオプションは，［実行｜実行時引数］ダイアログボックスで設定できます）。

  サーバーを実行することによって，登録することもできます。

アウトオブプロセスサーバーを登録解除する手順は次のとおりです。

- **/unregserver** コマンドラインオプションでサーバーを実行します。

ASP アプリケーションのテストとデバッグ

Active Server オブジェクトのようなインプロセスサーバーのデバッグは，DLL のデバッグと似ています。DLL をロードするホストアプリケーションを選択してから，普通にデバッグします。Active Server オブジェクトをテストおよびデバッグする手順は次のとおりです。

1. 必要であれば，［プロジェクト｜オプション］ダイアログボックスの［コンパイラ］タブを使って，デバッグ情報をオンにします。さらに，［ツール｜デバッガオプション］ダイアログの中の［統合開発環境を使う］をオンにします。
2. ［実行｜実行時引数］を選択し，［ホストアプリケーション］ボックスに Web サーバーの名前を入力してから，［OK］を選択します。
3. ［実行｜実行］を選択します。
4. Active Server オブジェクト実装にブレークポイントを設定します。
5. Web ブラウザを使って，ASP と対話します。

ブレークポイントに達すると，デバッガは一時停止します。

第43章

ActiveX コントロールの作成

ActiveX コントロールは，ホストアプリケーションに組み込んでその機能を拡張するためのソフトウェアコンポーネントです。ActiveX コントロールをサポートするアプリケーションには，C++Builder，Delphi，Visual Basic，Internet Explorer，（プラグインを与えられた）Netscape Navigator などがあります。ActiveX コントロールは，この組み込みを可能にする特定のインターフェースのセットを実装します。

たとえば，C++Builder にはチャート作成，スプレッドシート，グラフィックコントロールなど，さまざまな ActiveX コントロールが付属します。これらのコントロールは，IDE のコンポーネントパレットに追加できます。追加されたコントロールは，その他の標準 VCL コンポーネントと同様に，フォーム上にドロップして配置できます。オブジェクトインスペクタを使ってプロパティを設定することもできます。

ActiveX コントロールを Web に配布することもできます。配布した ActiveX コントロールは，HTML ドキュメントや ActiveX 対応の Web ブラウザを通して参照できます。

C++Builder には，以下の 2 種類の ActiveX コントロールを作成できるウィザードが用意されています。

- **VCL クラスをラップする ActiveX コントロール。**VCL クラスをラップすることによって，既存のコンポーネントを ActiveX コントロールに変換するか，新規のコンポーネントを作成し，ローカルでテストしてから ActiveX コントロールに変換することができます。ActiveX コントロールは一般的には，大きいホストアプリケーションに埋め込むことを目的としています。
- **ActiveForm。**ActiveForm では，フォームデザイナを使用して，ダイアログや完全なアプリケーションのように動作する精巧なコントロールを作成することができます。ActiveForm の開発方法は，一般的な C++Builder アプリケーションの開発方法とほぼ同じです。ActiveForm は一般的には，Web で配布することを目的としています。

この章では，C++Builder 環境で ActiveX コントロールを作成する方法を概説します。ウィザードを使用しない ActiveX コントロール作成の実装について完全な詳細を提供するわけではありません。詳細については，Microsoft Developer's Network（MSDN）で提供されている資料を参照するか，Microsoft の Web サイトで ActiveX の情報を検索してください。

ActiveX コントロールの作成の概要

C++Builder を使用して ActiveX コントロールを作成する方法は，通常のコントロールやフォームを作成する方法によく似ています。この点はほかの COM オブジェクトの作成方法と著しく異なります。ほかの COM オブジェクトの場合は，まずオブジェクトのインターフェースを定義してから，実装を完成します。ActiveForm 以外の ActiveX コントロールを作成するには，この手順を逆に行います。つまり，まず VCL コントロールを実装し，コントロールを書けたらインターフェースとタイプライブラリを生成します。ActiveForm を作成する場合は，フォームと同時にインターフェースとタイプライブラリが作成され，次に，フォームデザイナを使用してこのフォームを実装します。

完成した ActiveX コントロールを構成するものは，基底の実装を提供する VCL コントロール，VCL コントロールをラップする COM オブジェクト，COM オブジェクトのプロパティ，メソッド，イベントをリストするタイプライブラリです。

（ActiveForm 以外の）新規の ActiveX コントロールを作成する手順は次のとおりです。

1. ActiveX コントロールの土台になるカスタム VCL コントロールを設計して作成します。
2. ActiveX コントロールウィザードを使って，手順 1 で作成した VCL コントロールから ActiveX コントロールを作成します。
3. ActiveX プロパティページウィザードを使って，ActiveX コントロールのプロパティページを 1 つまたは複数作成します（オプション）。
4. プロパティページを ActiveX コントロールと関連付けます（オプション）。
5. 作成したコントロールを登録します。
6. ターゲットになり得るすべてのアプリケーションで，作成した ActiveX コントロールをテストします。
7. Web で ActiveX コントロールを配布します（オプション）。

新規の ActiveForm を作成する手順は次のとおりです。

1. Active フォームウィザードを使用して，ActiveForm およびそのフォームに関連付けられた ActiveX ラッパーを作成します（このフォームは空白フォームとして IDE に表示されます）。
2. フォームデザイナを使用して通常のフォームを作成および実装する場合と同じ方法で，フォームデザイナを使用して ActiveForm にコンポーネントを追加し，その動作を実装します。
3. 前述の手順 3 ～ 7 を実行して，ActiveForm にプロパティページを与え，これを登録し，Web で配布します。

ActiveX コントロールの構成要素

ActiveX コントロールには多くの要素が含まれており，それぞれが特定の機能を実行します。これらの要素は，VCL コントロール，およびプロパティ，メソッド，イベントをエクスポーズする対応する COM オブジェクトラッパー，そして関連付けられた 1 つまたは複数のタイプライブラリなどです。

VCL コントロール

C++Builder における ActiveX コントロールの基底の実装は，VCL コントロールです。ActiveX コントロールを作成するときには，その土台になる VCL コントロールをまず設計または選択しなければなりません。

基底の VCL コントロールは TWinControl の下位オブジェクトでなければなりません。なぜなら，ホストアプリケーションを親にできるウィンドウを持たなければならないからです。ActiveForm を作成すると，このオブジェクトは TActiveForm の子孫になります。

メモ ActiveX コントロールウィザードは，ActiveX コントロールを作成するために選択可能な TWinControl の子孫をリストします。ただし，TWinControl の子孫がすべてこのリストに表示されるわけではありません。THeaderControl など一部のコントロールは，（RegisterNonActiveX 手続きを使って）ActiveX と互換性がないものとして登録されるので，リストには表示されません。

ActiveX ラッパー

実際の COM オブジェクトは，VCL コントロール用の ActiveX ラッパーオブジェクトです。名前は TVCLClassXImpl の形式になります（TVCLClass は VCL コントロールクラスの名前）。したがって，たとえば TButton 用の ActiveX ラッパーならば，名前は TButtonXImpl になります。

ラッパークラスは，VCLCONTROL_IMPL マクロが宣言したクラスを継承し，このマクロが ActiveX インターフェースのサポートを提供します。ActiveX ラッパーは，このサポートを継承するので，VCL コントロールに Windows メッセージを転送でき，ホストアプリケーション内でそのウィンドウの親になることができます。

ActiveX ラッパーは，デフォルトインターフェースを介してクライアントに VCL コントロールのプロパティとメソッドをエクスポーズします。ラッパークラスのプロパティとメソッドの大部分はウィザードによって自動的に実装され，基底の VCL コントロールにメソッド呼び出しが委任されます。クライアント上で VCL コントロールのイベントを送出するメソッドもラッパークラスに提供され，VCL コントロール上でこれらのメソッドがイベントハンドラとして割り当てられます。

タイプライブラリ

ラッパークラスの型定義，そのデフォルトインターフェース，およびこれらが必要とする型定義が収められたタイプライブラリが，ActiveX コントロールウィザードによって自動的に生成されます。この型情報は，コントロールが自分のサービスをホストアプリケーションに知らせるための手段を提供します。この情報はタイプライブラリエディタを使って表示および編集できます。この情報は別個のバイナリタイプライブラリファイル（.TLB 拡張子）に格納されますが，リソースとして ActiveX コントロール DLL にも自動的にコンパイルされます。

プロパティページ

オプションで ActiveX コントロールにプロパティページを与えることができます。プロパティページを使うと，ホスト（クライアント）アプリケーションのユーザーが，コントロールのプロパティを表示したり編集できるようになります。開発者は複数のプロパティを 1 ページにグループ化することも，1 ページを使ってダイアログボックスに似たインターフェースをプロパティ用に提供することもできます。プロパティページの作成方法についての詳細は，43-13 ページの「ActiveX コントロールのプロパティページの作成」を参照してください。

ActiveX コントロールの設計

ActiveX コントロールを設計するときに最初にすることは，カスタム VCL コントロールの作成です。それによって，ActiveX コントロールの土台ができます。カスタムコントロールの作成についての詳細は，Part V「カスタムコンポーネントの作成」を参照してください。

VCL コントロールを設計するときには，このコントロール自体はアプリケーションではなく，別のアプリケーションに埋め込まれるということを頭に置いてください。したがって，精巧なダイアログボックスやほかの主要ユーザーインターフェースコンポーネントを実装する必要はおそらくないでしょう。一般的に目的は，メインアプリケーションの中でその規則に従って動作する単純なコントロールを作成することです。

さらに，オブジェクトがクライアントにエクスポーズするすべてのプロパティとメソッドはオートメーション互換型にする必要があります。なぜなら，ActiveX コントロールのインターフェースが IDispatch をサポートしなければならないからです。このウィザードは，オートメーション互換でないパラメータを持つラッパークラスのインターフェースにはメソッドを追加しません。オートメーション互換型のリストは，39-10 ページの「有効な型」を参照してください。

ウィザードは COM ラッパークラスを使って，必要な ActiveX インターフェースをすべて実装します。また，ラッパークラスのデフォルトインターフェースを介して，オートメーション互換のプロパティ，メソッド，およびイベントをすべて公開します。いったんウィザードで COM ラッパークラスとそのインターフェースを生成すると，タイプライブラリエディタを使ってデフォルトインターフェースを変更するか，追加のインターフェースを実装してラッパークラスを拡大することができます。

VCL コントロールからの ActiveX コントロールの生成

VCL コントロールから ActiveX コントロールを生成するには，ActiveX コントロールウィザードを使います。このとき VCL コントロールのプロパティ，メソッド，およびイベントが ActiveX コントロールに引き継がれます。

ActiveX コントロールウィザードを使う前に，生成される ActiveX コントロールの基底の実装をどの VCL コントロールが提供するかを決めなければなりません。

ActiveX コントロールウィザードを起動する手順は次のとおりです。

1. ［ファイル｜新規作成｜その他］を選択して［新規作成］ダイアログボックスを開きます。
2. ［ActiveX］タブを選択します。
3. ［ActiveX コントロール］アイコンをダブルクリックします。

このウィザードで，新規の ActiveX コントロールがラップする VCL コントロールの名前を選択します。このダイアログには，選択可能なすべてのコントロールがリストされます。つまり，TWinControl の子孫であり，RegisterNonActiveX 手続きを使って ActiveX と互換性がないものとして登録されていないコントロールがリストされます。

参考 選択したいコントロールがドロップダウンリストにない場合は，IDE にインストールしたかどうか，またはそのユニットをプロジェクトに追加したかどうかを確認してください。

いったん VCL コントロールを選択すると，CoClass の名前，ActiveX ラッパーの実装ユニット，および ActiveX ライブラリプロジェクトが自動的に生成されます（現在開いている ActiveX ライブラリプロジェクトがあり，その中に COM+ イベントオブジェクトが入っていない場合，現在のプロジェクトが自動的に使用されます）。これらはウィザードで変更できます（ただし，すでに開いている ActiveX ライブラリプロジェクトがある場合，プロジェクト名は編集できません）。

このウィザードでは常にアパートメントがスレッドモデルとして指定されます。コントロールが通常は 1 つしか入らない ActiveX プロジェクトの場合は，これは問題ありません。ただし，プロジェクトにオブジェクトを追加する場合は，開発者が責任を持ってスレッドサポートを提供する必要があります。

このウィザードでは，以下のようなさまざまな ActiveX コントロールオプションも設定できます。

- **設計時モードライセンスを追加：**設計時モードライセンスを追加すると，コントロールのユーザーがそのコントロールのライセンスを持っていなければ，設計目的で，または実行時にコントロールを開けないようにできる
- **バージョン情報を追加**：著作権やファイル説明などのバージョン情報を ActiveX コントロールに含めることができる。この情報は，ブラウザで表示できる。Visual Basic 4.0 などのホストクライアントはバージョン情報を必要とし，バージョン情報がないと ActiveX コントロールのホストになろうとしない。バージョン情報を指定するには，［プロジェクト｜オプション］を選択し，［バージョン情報］ページを選択する
- **バージョンダイアログを追加：**コントロールの［バージョン情報］ダイアログボックスを実装する独立したフォームを生成するように指示できる。ホストアプリケーションのユーザーは，開発環境でこの［バージョン情報］ダイアログボックスを表示できる。デフォルトでは，このダイアログボックスには ActiveX コントロールの名前，画像，著作権情報，および［OK］ボタンが表示される。このデフォルトフォームを変更して，プロジェクトに追加することができる

このウィザードを終了すると，以下のものが生成されます。

- ActiveX ライブラリプロジェクトファイル。ActiveX コントロールの開始に必要なコードが含まれる。通常，このファイルの内容を変更することはない
- タイプライブラリファイル。コントロールの CoClass，クライアントにエクスポーズするインターフェース，およびこれらが必要とする型定義を定義する。タイプライブラリについての詳細は，第 39 章「タイプライブラリの操作」を参照
- ActiveX 実装ユニット。VCLCONTROL_IMPL マクロを使って ATL（Microsoft Active Template Library）に適合する ActiveX コントロールを定義して実装する。この ActiveX コントロールは，開発者が何も手を加えなくても完全に機能する実装である。ただし，ActiveX コントロールがクライアントにエクスポーズするプロパティ，メソッド，およびイベントをカスタマイズしたい場合は，このクラスを変更できる
- ATL ユニット。名前は ActiveXControlProj_ATL.cpp (.h) の形式になり，ActiveXControlProj はプロジェクトの名前である。このユニットの主な構成要素は，ATL テンプレートクラスをプロジェクトが使用できるようにする include 文，および生成された ActiveX ラッパーを VCL オブジェクトとともに使用できるようにするクラスを定義する include 文である。このユニットはさらに，ATL クラスに対して ActiveX ライブラリを表す，_Module というグローバル変数も宣言する
- ［バージョン情報］ダイアログボックスのフォームとユニット（要求した場合のみ）
- .LIC ファイル（設計時モードライセンスを追加した場合のみ）

VCL フォームを土台にした ActiveX コントロールの生成

ほかの ActiveX コントロールと違って，ActiveForm はまず設計してから ActiveX ラッパークラスでラップするのではありません。そのかわりに，Active フォームウィザードが空白フォームを生成し，このウィザードが終了すると表示されるフォームデザイナで開発者はこのフォームを設計します。

ActiveForm を Web で配布する場合，ActiveForm への参照を含み，ActiveForm の位置を指定する HTML ページが C++Builder によって作成されます。その後，Web ブラウザから ActiveForm を表示して実行することができます。ブラウザ内では，ActiveForm はスタンドアロンの C++Builder フォームと同じように動作します。ActiveForm には，カスタム VCL コントロールなど，どのような VCL コンポーネントまたは ActiveX コントロールも入れることができます。

Active フォームウィザードを起動する手順は次のとおりです。

1. ［ファイル｜新規作成｜その他］を選択して［新規作成］ダイアログボックスを開きます。
2. ［ActiveX］タブを選択します。
3. ［Active フォーム］アイコンをダブルクリックします。

Active フォームウィザードは，ActiveX コントロールウィザードに似て見えますが，ラップする VCL クラスの名前を指定できない点が異なります。なぜなら，ActiveForm は常に TActiveForm を土台にしているからです。

ActiveX コントロールウィザードと同じように，CoClass，実装ユニット，および ActiveX ライブラリプロジェクトのデフォルト名は変更できます。同様に，このウィザードでも，ActiveForm で設計時モードライセンスを追加するかどうか，バージョン情報を追加するかどうか，およびバージョンダイアログを追加するかどうかを指定できます。

このウィザードを終了すると，以下のものが生成されます。

- ActiveX ライブラリプロジェクトファイル。ActiveX コントロールの開始に必要なコードが含まれる。通常，このファイルの内容を変更することはない
- タイプライブラリファイル。コントロールの CoClass，クライアントにエクスポーズするインターフェース，およびこれらが必要とする型定義を定義する。タイプライブラリについての詳細は，第 39 章「タイプライブラリの操作」を参照
- TActiveForm を継承するフォーム。このフォームがフォームデザイナに表示され，フォームデザイナでは，クライアントに表示される ActiveForm をビジュアルに設計することができる。この実装は，生成された実装ユニットに表示される
- フォームの ActiveX ラッパーの宣言。このラッパークラスは実装ユニット内でも定義され，名前が TActiveFormXImpl の形式になり，TActiveFormX はフォームクラスの名前である。この ActiveX ラッパーは，開発者が何も手を加えなくても完全に機能する実装である。ただし，ActiveForm がクライアントにエクスポーズするプロパティ，メソッド，およびイベントをカスタマイズしたい場合は，このクラスを変更できる
- ATL ユニット。名前は ActiveXControlProj_ATL.cpp (.h) の形式になり，ActiveXControlProj はプロジェクトの名前である。このユニットの主な構成要素は，ATL テンプレートクラスをプロジェク

トが使用できるようにする include 文，および生成された ActiveX ラッパーを VCL フォームとともに使用できるようにするクラスを定義する include 文である。このユニットはさらに，ATL クラスに対して ActiveX ライブラリを表す，_Module というグローバル変数も宣言する

- ［バージョン情報］ダイアログボックスのフォームとユニット（要求した場合のみ）
- .LIC ファイル（設計時モードライセンスを追加した場合のみ）

この時点で，コントロールを付け加えたりしてフォームを自由に設計できます。

ActiveForm プロジェクトを設計し，ActiveX ライブラリ（.OCX 拡張子を持つ）にコンパイルすると，その ActiveForm プロジェクトを Web サーバーで配布できます。このとき，ActiveForm への参照の入ったテスト用 HTML ページが C++Builder によって作成されます。

ActiveX コントロールへのライセンス付与

ActiveX コントロールに対するライセンス付与は，設計時にライセンスキーを提供すること，実行時に作成されるコントロールに対して動的なライセンスの作成をサポートすることから成り立ちます。

設計時のライセンスを提供するために，ActiveX ウィザードはコントロールのキーを作成します。このキーは，プロジェクトファイルと同じ名前で拡張子 LIC の付いたファイルに格納されます。この .LIC ファイルはプロジェクトに追加されます。キーが設定されたコントロールをユーザーが開発環境で開くには，.LIC ファイルのコピーを所有していなければなりません。［設計時モードライセンスを追加］オプションにチェックマークを付けたプロジェクトの中の各コントロールは，.LIC ファイルに個別のキーエントリを持ちます。

実行時ライセンスをサポートするために，ラッパークラスは GetLicenseString と GetLicenseFilename の 2 つのメソッドを実装します。GetLicenseString はコントロールのライセンス文字列を返し，GetLicenseFilename は .LIC ファイルの名前を返します。ホストアプリケーションが ActiveX コントロールを作成しようとすると，コントロールのクラスファクトリがこれらのメソッドを呼び出し，GetLicenseString が返す文字列と，.LIC ファイルに格納された文字列とを比較します。

Internet Explorer の実行時ライセンスには，特別のレベルの間接性が必要です。理由は，ユーザーは任意の Web ページの HTML ソースコードを表示でき，表示される前に ActiveX コントロールがユーザーのコンピュータにコピーされるからです。Internet Explorer で使用されるコントロールの実行時ライセンスを作成するには，まずライセンスパッケージファイル（LPK ファイル）を生成してから，コントロールを含む HTML ページにこのファイルを埋め込みます。LPK ファイルは，基本的には，ActiveX コントロール CLSID とライセンスキーの配列です。

メモ LPK ファイルを生成するには，LPK_TOOL.EXE というユーティリティを使います。このユーティリティは，Microsoft の Web サイト（www.microsoft.com）からダウンロードできます。

LPK ファイルを Web ページに埋め込むには，次のように HTML オブジェクト <OBJECT> と <PARAM> を使います。

```
<OBJECT CLASSID="clsid:6980CB99-f75D-84cf-B254-55CA55A69452">
    <PARAM NAME="LPKPath" VALUE="ctrllic.lpk">
</OBJECT>
```

CLSID は，オブジェクトをライセンスパッケージとして識別し，PARAM はライセンスパッケージファイルの位置を HTML ページを基準にした相対位置で指定します。

Internet Explorer は，コントロールを含む Web ページを表示しようとするときに，LPK ファイルを構文解析し，ライセンスキーを抽出します。そして，このライセンスキーと（GetLicenseString が返す）コントロールライセンスとが一致すれば，ページ上のコントロールを描画します。Web ページに複数の LPK が含まれている場合，Internet Explorer は，最初の LPK 以外は無視します。

詳細については，Microsoft の Web サイトで「Licensing ActiveX Controls」を探してください。

ActiveX コントロールのインターフェースのカスタマイズ

ActiveX コントロールウィザードと Active フォームウィザードは，ActiveX ラッパークラスのデフォルトインターフェースを生成します。このデフォルトインターフェースは，単純にオリジナルの VCL コントロールまたはフォームのプロパティ，メソッド，およびイベントをエクスポーズしますが，以下のものは例外です。

- データベース対応のプロパティは表示されない。ActiveX コントロールがコントロールをデータベース対応にする仕組みは VCL コントロールのものとは異なるので，データに関連するプロパティは変換されない。ActiveX コントロールをデータベース対応にする方法については，43-11 ページの「タイプライブラリを使って単純なデータバインディングを有効にする」を参照
- オートメーション互換でない型のプロパティ，メソッド，またはイベントは表示されない。これらを ActiveX コントロールのインターフェースに追加したい場合は，ウィザードを終了した後で行う

第 39 章「タイプライブラリの操作」に示すとおり，タイプライブラリエディタを使ってタイプライブラリを編集することにより，ActiveX コントロール内のプロパティ，メソッド，およびイベントの追加，編集，削除が行えます。

メモ パブリッシュに設定されていないプロパティを ActiveX コントロールのインターフェースに追加することができます。このようなプロパティは実行時に設定でき，開発環境に表示されますが，加えられた変更内容は持続しません。つまり，設計時にコントロールのユーザーがプロパティの値を変更しても，その変更はコントロールの実行時には反映されません。なぜなら，ActiveX コントロールは，ATL ストリーミングシステムではなく VCL ストリーミングシステムを使用するからです。ソースが VCL オブジェクトであり，プロパティがパブリッシュに設定されていない場合は，その VCL オブジェクトの子孫を作成し，その子孫の中でプロパティをパブリッシュに設定することで，プロパティを持続させることができます。

VCL コントロールのプロパティ，メソッド，およびイベントをすべてホストアプリケーションにエクスポーズしないように選択することもできます。タイプライブラリエディタを使用すれば，ウィザードが生成したインターフェースからこれらを除去できます。タイプライブラリエディタを使ってインターフェースからプロパティとメソッドを除去しても，対応する実装クラスからは除去されません。これらを除去するには，タイプライブラリエディタでインターフェースを変更し終えてから，実装ユニット内で ActiveX ラッパークラスを編集します。

警告 オリジナルの VCL コントロールまたは VCL フォームから ActiveX コントロールを再び生成する場合，タイプライブラリに対して行った変更はすべて無効になります。

参考 ウィザードが ActiveX ラッパークラスに追加するメソッドをチェックすることをお勧めします。そうすれば，オートメーション互換でなかったデータベース対応のプロパティまたはメソッドがどこで省略されたかがわかるだけでなく，ウィザードが実装を生成できなかったメソッドを検出することもできます。このようなメソッドは，問題を示す実装の中にコメントとともに表示されます。

プロパティ，メソッド，イベントを追加する

ActiveX コントロールのインターフェースへのプロパティとメソッドの追加は，COM インターフェースへのプロパティ，メソッド，イベントの追加と同じように働きます。イベントの追加はメソッド（イベントハンドラ）の追加とほぼ同じですが，オブジェクトのデフォルトインターフェースではなくイベントインターフェースに追加します。タイプライブラリを使ってプロパティとイベントを追加する方法については，41-9 ページの「COM オブジェクトのインターフェースを定義する」を参照してください。

ActiveX コントロールのインターフェースに追加するときに，基底の VCL コントロール内のプロパティ，メソッド，またはイベントだけを公開することも珍しくありません。以下の節では，その方法について説明します。

プロパティとメソッドの追加

ActiveX ラッパークラスは，アクセスメソッド read および write を使ってインターフェース内のプロパティを実装します。つまり，ActiveX ラッパークラスには COM プロパティがあり，このプロパティは取得メソッドと設定メソッドの両方またはいずれかとしてインターフェース上に表示されます。VCL プロパティと違って，COM プロパティのインターフェース上に「property」宣言は表示されません。そのかわりに，プロパティアクセスメソッドとしてフラグが付いたメソッドが表示されます。ActiveX コントロールのデフォルトインターフェースにプロパティを追加すると，（タイプライブラリエディタが更新する _TLB ユニットに表示される）ラッパークラス定義が，実装しなければならない 1 ～ 2 個の新規メソッド（取得メソッドと設定メソッドの両方またはいずれか）を取得し，インターフェースにメソッドを追加する場合と同じように，このラッパークラスは実装すべき対応するメソッドを取得します。したがって，ラッパークラスのインターフェースへのプロパティの追加は，本質的にはメソッドの追加と同じであり，ラッパークラス定義は，中身を埋めるだけで完成できる新規のスケルトンメソッド実装を取得します。

メモ 生成された _TLB ユニットに表示されるものについての詳細は，40-5 ページの「タイプライブラリ情報のインポート時に生成されるコード」を参照してください。

たとえば，基底の VCL オブジェクト内の AnsiString 型の Caption プロパティの例で説明します。タイプライブラリエディタでこのプロパティを追加すると，C++Builder は，ラッパークラスに以下の宣言を追加します。

```
STDMETHOD(get_Caption(BSTR* Value));
STDMETHOD(set_Caption(BSTR Value));
```

さらに，中身を埋めるだけで完成できる，次のようなスケルトンメソッド実装も追加されます。

```
STDMETHODIMP TButtonXImpl::get_Caption(BSTR* Value)
{
  try
  {
```

```
  }
  catch(Exception &e)
  {
    return Error(e.Message.c_str(), IID_IButtonX);
  }
  return S_OK;
};
STDMETHODIMP TButtonXImpl::set_Caption(BSTR Value)
{
  try
  {
  }
  catch(Exception &e)
  {
    return Error(e.Message.c_str(), IID_IButtonX);
  }
  return S_OK;
};
```

通常は，関連付けられた VCL コントロールに委任するだけでこれらのメソッドを実装でき，この VCL コントロールにアクセスするにはラッパークラスの m_VclCtl メンバーを使います。

```
STDMETHODIMP TButtonXImpl::get_Caption(BSTR* Value)
{
  try
  {
    *Value = WideString(m_VclCtl->Caption).Copy();
  }
  catch(Exception &e)
  {
    return Error(e.Message.c_str(), IID_IButtonX);
  }
  return S_OK;
};
STDMETHODIMP TButtonXImpl::set_Caption(BSTR Value)
{
  try
  {
    m_VclCtl->Caption = AnsiString(Value);
  }
  catch(Exception &e)
  {
    return Error(e.Message.c_str(), IID_IButtonX);
  }
  return S_OK;
};
```

COM のデータ型を C++ のネイティブな型に変換するためのコードを追加する必要がある場合もあります。このコード例では，型キャストすることによって対処しています。

イベントの追加

ActiveX コントロールは，オートメーションオブジェクトがクライアントにイベントを送出するのと同じように，イベントをコンテナに送出することができます。この仕組みについては，41-11 ページの「クライアントにイベントをエクスポーズする」を参照してください。

ActiveX コントロールの土台として使用する VCL コントロールがパブリッシュに設定されたイベントを持っている場合，クライアントイベントシンクのリストを管理するために必要なサポートが ActiveX ラッパークラスに自動的に追加され，イベントに応答するためにクライアントが実装しなければならない外向きディスパッチインターフェースが定義されます。

コンテナにイベントを送出するには，ActiveX ラッパークラスが VCL オブジェクト上のイベントのイベントハンドラを実装しなければなりません。このイベントハンドラは，_TLB ユニット内で定義された TEvents_CoClassName クラスによって実装される Fire_EventName メソッドを呼び出します。

```
void __fastcall TButtonXImpl::KeyPressEvent(TObject *Sender, char &Key)
{
  short TempKey;
  TempKey = (short)Key;
  Fire_OnKeyPress(&TempKey);
  Key = (short)TempKey;
}
```

次に，このイベントハンドラを VCL コントロールに割り当てて，イベントが発生したときに呼び出されるようにしなければなりません。それには，イベントを InitializeControl メソッドに追加します。このメソッドは，（実装ユニットのヘッダー内の）ラッパークラス修飾子の中に表示されます。

```
void InitializeControl()
{
  m_VclCtl->OnClick = ClickEvent;
  m_VclCtl->OnKeyPress = KeyPressEvent;
}
```

タイプライブラリを使って単純なデータバインディングを有効にする

単純なデータバインディングによって，ActiveX コントロールのプロパティをデータベース内のフィールドにバインドすることができます。それには，どの値がフィールドデータを表し，いつ変更されるかについて，ActiveX コントロールがホストアプリケーションと通信しなければなりません。この通信を可能にするには，タイプライブラリエディタを使ってプロパティのバインディングフラグを設定します。

プロパティにバインド可能（Bindable）としてマークすると，プロパティ（データベース内のフィールドなど）にユーザーが変更を加えたときに，コントロールは値が変更されたことをコンテナ（クライアントホストアプリケーション）に通知し，データベースレコードの更新を要求するようになります。コンテナはデータベースと対話してから，レコードの更新に成功したか失敗したかをコントロールに通知します。

メモ ActiveX コントロールのホストになるコンテナアプリケーションは，タイプライブラリで有効にしたデータベース対応プロパティをデータベースに接続する責任を負います。C++Builder を使ってこのようなコンテナを作成する方法については，40-9 ページの「データベース対応の ActiveX コントロールの使い方」を参照してください。

単純なデータバインディングを有効にするには，タイプライブラリを使います。

1. ツールバーで，バインドしたいプロパティをクリックします。

2. ［フラグ］ページを選択します。
3. 以下のバインディング属性を選択します。

バインディング属性	内容
Bindable	プロパティがデータのバインディングをサポートすることを示す。Bindable（バインド可能）とマークされたプロパティは，プロパティの値が変更されたときにそれをコンテナに通知する
Request Edit	プロパティが OnRequestEdit 通知をサポートすることを示す。これによってコントロールは，その値をユーザーが編集できるかどうかをコンテナに問い合わせられるようになる
Display Bindable	このプロパティがバインド可能であることをコンテナがユーザーに対して表示できることを示す
Default Bindable	オブジェクトをもっともよく表す単一のバインド可能なプロパティであることを示す。Default Bindable 属性を持つプロパティは Bindable 属性も持たなければならない。dispinterface 内の複数のプロパティに指定することはできない
Immediate Bindable	フォーム上の個々のバインド可能なプロパティがこの動作を指定できる。このビットが設定されている場合，すべての変更が通知される。この新しいビットを有効にするには，bindable と requestedit の両方の属性ビットが設定されなければならない

4. ツールバーの［ソースコードの更新］ボタンをクリックしてタイプライブラリを更新します。

コントロールのデータバインディングをテストするには，まずコントロールを登録しなければなりません。

たとえば，TEdit コントロールをデータベース対応 ActiveX コントロールに変換するには，TEdit から ActiveX コントロールを作成し，次に Text プロパティフラグを Bindable，Display Bindable，Default Bindable，および Immediate Bindable に変更します。

ActiveX ラッパー内のデータの編集を可能にするには，編集を許可する前に，値を変更できるかどうかをコンテナに尋ねなければなりません。そのために，キーが押されたというメッセージを VCL コントロールがユーザーから受け取ったときに，TVclComControl 基本クラスから継承された FireOnRequestEdit が呼び出されます。ActiveX ラッパーはすでに，OnKeyPress イベントハンドラを VCL コントロールに割り当てています。これは次のように変更できます。

```
void __fastcall TMyAwareEditImpl::KeyPressEvent(TObject *Sender, char &Key)
{
   short TempKey;
   const DISPID dispid = -517; // これはデータにバインドされたプロパティのディスパッチ ID
   if (FireOnRequestEdit(dispid) == S_FALSE) // 編集できるどうかをコンテナに尋ねる
   {
      Key = 0; // 編集できなければ，キーが押されたというメッセージを取り消す
      return;
   }
   TempKey = (short)Key;
   Fire_OnKeyPress(&TempKey);  // コンテナ内で送出する OnKeyPress イベントを転送する
   Key = (short)TempKey;
}
```

ActiveX ラッパーは，データが変更されたときのコンテナへの通知もする必要があります。そのために，編集コントロールの値が変わると，ActiveX ラッパーは（TVclComControl から継承された）

FireOnChanged メソッドを呼び出します。FireOnChanged は，何かが変わったことをコントロールに通知します。コントロール内のデータが編集されると，コンテナは，関連付けられたデータセットを編集モードにすることができます。FireOnChanged は，フィールドデータのリアルタイムの変更も可能にします。次のコードに，変更された OnChange イベントハンドラを示します。

```
void __fastcall TMyAwareEditImpl::ChangeEvent(TObject *Sender)
{
  const DISPID dispid = -517; // データにバインドされたプロパティのディスパッチ ID
  FireOnChanged(dispid); // コンテナに変更を通知するためにこの行を追加する
  Fire_OnChange();   // これはコンテナ内でイベントを送出するためのオリジナルの呼び出しだった
}
```

コントロールは，登録およびインポートした後，データの表示に使えます。

ActiveX コントロールのプロパティページの作成

プロパティページは，C++Builder のオブジェクトインスペクタに似たダイアログボックスです。ユーザーは，ここで ActiveX コントロールのプロパティを変更できます。プロパティページダイアログでは，コントロールのいくつものプロパティを同時に編集するためにグループ化できます。また，より複雑なプロパティを設定できるダイアログボックスを提供することができます。

プロパティページにアクセスするには，通常 ActiveX コントロールを右クリックして［プロパティ］を選択します。

プロパティページ作成の手順は，フォーム作成の手順とよく似ています。

1. 新しいプロパティページを作成します。
2. プロパティページにコントロールを追加します。
3. プロパティページに配置したコントロールを，ActiveX コントロールのプロパティに関連付けます。
4. プロパティページを ActiveX コントロールに接続します。

メモ ActiveX コントロールまたは ActiveForm にプロパティを追加するときは，持続させたいプロパティをパブリッシュに設定しなければなりません。基底の VCL コントロール内でパブリッシュに設定しない場合は，パブリッシュに設定したものとしてそのプロパティを再宣言する VCL コントロールのカスタムの子孫を作成してから，ActiveX コントロールウィザードを使用して子孫クラスから ActiveX コントロールを作成しなければなりません。

新しいプロパティページの作成

新しいプロパティページの作成には，プロパティページウィザードを使用します。

新しいプロパティページを作成する手順は次のとおりです。

1. ［ファイル｜新規作成｜その他］を選択します。
2. ［ActiveX］タブを選択します。
3. ［プロパティページ］アイコンをダブルクリックします。

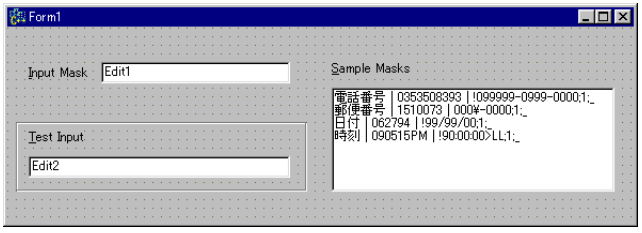
ウィザードによって，新しいフォームとプロパティページの実装ユニットが作成されます。このフォームは TPropertyPage の子孫であり，プロパティが編集される ActiveX コントロールと，このフォームとを関連付けることができます。さらに，この実装ユニットは（PROPERTYPAGE_IMPL マクロを使って）実装オブジェクトを宣言します。この実装オブジェクトは，プロパティページインターフェースを実装し，適切な呼び出しをフォームに渡します。

プロパティページへのコントロールの追加

ActiveX コントロールの各プロパティにユーザーがアクセスできるようにするには，プロパティページにコントロールを追加する必要があります。

たとえば次の図は，ActiveX コントロールの MaskEdit プロパティを設定するためのプロパティページを示しています。

図 43.1　設計モードのマスク編集プロパティページ



ユーザーはリストボックスでサンプルマスクのリストから選択できます。マスク編集コントロールを使うと，ActiveX コントロールに適用する前にマスクをテストできます。コントロールは，フォームに追加する場合と同じようにプロパティページに追加します。

プロパティページコントロールの ActiveX コントロールプロパティへの関連付け

必要なコントロールをプロパティページに追加したら，それぞれのコントロールを対応するプロパティに関連付ける必要があります。それには，プロパティページの UpdatePropertyPage および UpdateObject メソッドにコードを追加します。

プロパティページを更新する

ActiveX コントロールのプロパティが変化したときにプロパティページ上のコントロールを更新するために，UpdatePropertyPage メソッドにコードを追加します。プロパティページの表示を ActiveX コントロールのプロパティの現在の値に更新するには，UpdatePropertyPage メソッドにコードを追加します。

プロパティページの OleObject プロパティを使うと ActiveX コントロールにアクセスできます。このプロパティは，ActiveX コントロールのインターフェースが入った OleVariant です。

次のコードでは，プロパティページの編集コントロール（InputMask）の表示を，ActiveX コントロールの EditMask プロパティの現在の値に更新します。

```
void __fastcall TPropertyPage1::UpdatePropertyPage(void)
{
   InputMask->Text = OleObject.OlePropertyGet("EditMask");
}
```

メモ 複数の ActiveX コントロールを表すプロパティページを作成することも可能です。この場合，OleObject プロパティは使用しません。そのかわり，OleObjects プロパティが保守するインターフェースリストを反復処理しなければなりません。

オブジェクトを更新する

ユーザーがプロパティページの上のコントロールを変更したときにプロパティを更新するために，UpdateObject メソッドにコードを追加します。ActiveX コントロールのプロパティを，コントロールの側で設定された新しい値に更新するには，UpdateObject メソッドにコードを追加します。

この場合も OleObject プロパティを使用して ActiveX コントロールにアクセスします。

次のコードでは，ActiveX コントロールの EditMask プロパティの値を，プロパティページの編集ボックスコントロール（InputMask）で設定された値に更新します。

```
void __fastcall TPropertyPage1::UpdateObject(void)
{
  // コントロールから値を取得して OleObject を更新する
  OleObject.OlePropertySet<WideString>("EditMask", WideString(InputMast->Text).Copy());
}
```

プロパティページの ActiveX コントロールへの接続

プロパティページを ActiveX コントロールに接続する手順は次のとおりです。

1. ActiveX コントロールの実装ユニットのファイルの BEGIN_PROPERTY_MAP から END_PROPERTY_MAP の文の間に，PROP_PAGE マクロ呼び出しを追加し，それをプロパティページの GUID に渡します。このとき引数として，プロパティページの GUID を指定します。GUID はプロパティページウィザードによって自動的に生成され，プロパティページの実装ユニットに定義されています。

   たとえば，プロパティページの GUID が CLSID_PropertyPage1（デフォルト）と定義されている場合，VCL フォームに基づく ActiveX コントロールのプロパティマップセクションは次のように記述できます。

   ```
   BEGIN_PROPERTY_MAP(TActiveFormXImpl)
      // ここでプロパティページを定義する。プロパティページを定義するには,
      // PROP_PAGE マクロとそのページの class id を使う。たとえば,
      // PROP_PAGE(CLSID_ActiveFormXPage)
      PROP_PAGE(CLSID_PropertyPage1)
   END_PROPERTY_MAP()
   ```

2. プロパティページユニットを ActiveX コントロールのユニットにインクルードします。

ActiveX コントロールの登録

作成した ActiveX コントロールは，ほかのアプリケーションからその場所を特定して利用できるように，システムに登録しなければなりません。

ActiveX コントロールを登録する手順は次のとおりです。

- ［実行｜ ActiveX サーバーの登録］を選択します。

メモ ActiveX コントロールをシステムから削除する場合は，削除を行う前にシステムへの登録を解除してください。

ActiveX コントロールの登録を解除する手順は次のとおりです。

- ［登録｜ ActiveX サーバーの削除］を選択します。

コマンドラインから **tregsvr** コマンドを使うか，オペレーティングシステムから regsvr32.exe を実行するという方法もあります。

ActiveX コントロールのテスト

コントロールをテストするには，コントロールをパッケージに追加し，ActiveX コントロールとしてインポートします。この手続きにより，ActiveX コントロールが C++Builder のコンポーネントパレットに追加されます。テストしたいコントロールを，コンポーネントパレットからフォームにドロップして，テストを実行します。

作成したコントロールは，それを使うすべてのターゲットアプリケーション上でテストすることをお勧めします。

ActiveX コントロールのデバッグを行うには，［実行｜実行時引数］を選択して，［ホストアプリケーション］編集ボックスにクライアント名を入力します。

パラメータはホストアプリケーションに渡されます。［実行｜実行］を選択すると，ホストアプリケーションまたはクライアントアプリケーションが実行され，コントロール内にブレークポイントを設定できるようになります。

ActiveX コントロールの Web での配布

作成した ActiveX コントロールを Web クライアントによって使用可能にするためには，Web サーバー上に配布しなければなりません。ActiveX コントロールに変更を加えるたびに，再コンパイルと再配布を行わなければ，クライアントアプリケーションからは変更が見えません。

ActiveX コントロールを配布する前に，クライアントメッセージに応答する Web サーバーを設置しなければなりません。

ActiveX コントロールを配布する手順は次のとおりです。

1. ［プロジェクト｜ Web 配布オプション］を選択します。

2. ［プロジェクト］ページで［配布先ディレクトリ］に Web サーバー上での ActiveX コントロールの .DLL の位置を示すパスを設定します。ローカルパス名でも UNC パスでもかまいません。たとえば，C: \ INETPUB \ wwwroot を設定します。
3. ［配布元 URL］を Web サーバー上での ActiveX コントロールの DLL の位置を示す URL に設定します（ファイル名を除く）。たとえば http://mymachine.borland.com/ のように指定します。その方法についての詳細は，Web サーバーのマニュアルを参照してください。
4. ［HTML ディレクトリ］に ActiveX コントロールへの参照の入った HTML ファイルの位置を示すパスを設定します。たとえば，C: \ INETPUB \ wwwroot と設定します。標準パス名または UNC パスを指定できます。
5. 43-17 ページの「オプションの設定」の説明に従って Web 配布用オプションを設定します。
6. ［OK］を押します。
7. ［プロジェクト｜ Web 配布］を選択します。

   これによって，ActiveX コントロールを含んだ配布コードベースが ActiveX ライブラリ（拡張子は OCX）の中に作成されます。指定されたオプションに応じて，この配布コードベースにさらにキャビネット（CAB 拡張子）や情報（INF 拡張子）が含まれることもあります。

   ActiveX ライブラリ（.OCX）は，手順 2 で指定した［転送先ディレクトリ］の位置に置かれます。HTML ファイルはプロジェクトファイルと同じ名前ですが，拡張子が .HTM になります。それは，手順 4 で指定した［HTML ディレクトリ］の位置に作成されます。HTML ファイルには手順 3 で指定した位置にある ActiveX ライブラリへの URL による参照が入っています。

メモ　これらのファイルを Web サーバー上に置きたい場合は，ftp などの外部ユーティリティを使ってください。

8. ActiveX 対応 Web ブラウザを呼び出し，作成した HTML ページを表示します。

この HTML ページを Web ブラウザで表示すると，作成したフォームまたはコントロールがブラウザ内で組み込みのアプリケーションとして表示され，実行されます。つまり，ライブラリはブラウザと同じプロセスの中で実行されます。

オプションの設定

ActiveX ライブラリを作成したら，ActiveX コントロールを配布する前に，Web 配布用のオプションを設定します。

Web 配布用のオプションには，次のことを設定するためのものがあります。

- **追加ファイルの配布**：ActiveX コントロールがパッケージやその他の追加ファイルに依存している場合には，これらをプロジェクトとともに配布する必要があることを指示できます。デフォルトでは，これらのファイルはプロジェクト全体に対して指定されたオプションと同じものを使用しますが，［パッケージ］ページまたは［追加ファイル］ページを使ってこれらの設定をオーバーライドすることができます。パッケージまたは追加ファイルを含めると，（information の最初の 3 文字をとった）.INF 拡張子を持つファイルが C++Builder によって作成されます。このファイルは，ActiveX ライブラリを実行するためにダウンロードおよび設定する必要がある種々のファイ

ルを指定します。.INF ファイルの構文では，ダウンロードするパッケージまたは追加ファイルを示す URL を指定できます。

- **CAB ファイル圧縮：**キャビネットは単独のファイルで，通常はファイル拡張子 **CAB** が付き，圧縮ファイルをファイルライブラリに格納します。キャビネット圧縮によってファイルのダウンロード時間を大幅に削減できます（70% まで）。ブラウザはインストール時にキャビネットに格納されたファイルを圧縮解除し，ユーザーのシステムにコピーします。配布する各ファイルは，圧縮された .CAB ファイルにもできます。ActiveX ライブラリが CAB ファイル圧縮を使用するように指定するには，［Web 配布オプション］ダイアログの［プロジェクト］ページを使用します。
- **バージョン情報**：ActiveX コントロールにバージョン情報を含めるように指定できます。このバージョン情報は，［プロジェクトオプション］ダイアログの［バージョン情報］ページで設定されています。この情報の中のリリース番号は，ActiveX コントロールを配布するたびに自動的に更新することができます。追加のパッケージやファイルを含める場合，これらのバージョン情報リソースも INF ファイルに追加できます。

追加ファイルを含めるかどうか，および CAB ファイル圧縮を使用するかどうかに応じて，結果の ActiveX ライブラリは，OCX ファイル，OCX ファイルが入った CAB ファイル，または INF ファイルになります。次の表は，さまざまな組み合わせを選択した場合の結果をまとめたものです。

パッケージや追加ファイルの選択	CAB ファイル圧縮	結果
なし	なし	ActiveX ライブラリ（OCX）ファイル
なし	あり	ActiveX ライブラリファイルを含む CAB ファイル
あり	なし	.INF ファイル，ActiveX ライブラリファイル，追加ファイルおよびパッケージ
あり	あり	.INF ファイル，ActiveX ライブラリファイルを含む .CAB ファイル，追加ファイルおよびパッケージのそれぞれに対する CAB ファイル

第44章

MTS オブジェクトまたは COM+ オブジェクトの作成

C++Builder では，（Windows 2000 より前のバージョンの Windows の場合は）MTS（Microsoft Transaction Server），または（Windows 2000 以降の場合は）COM+ が提供するトランザクションサービス，セキュリティ，およびリソース管理を利用するオブジェクトのことを，トランザクションオブジェクトと呼びます。このオブジェクトは大規模な分散環境において機能するように設計されています。トランザクションオブジェクトは Windows 固有の技術に依存するため，クロスプラットフォームアプリケーションでは使用できません。

C++Builder にはトランザクションオブジェクトを作成するウィザードがあるので，COM+ 属性または MTS 環境の利点を活用できます。これらの機能は，COM クライアントとサーバー（特にリモートサーバー）の実装を作成しやすくします。

メモ データベースアプリケーションの場合，C++Builder はトランザクションデータモジュールも提供します。詳細については，第 29 章「多層アプリケーションの作成」を参照してください。

トランザクションオブジェクトは，多くの低レベルサービスを利用します。たとえば以下のようなものがあります。

- サーバーアプリケーションが多くの同時ユーザーを処理できるように，プロセス，スレッド，データベース接続などのシステムリソースを管理する
- アプリケーションの信頼性を保つために，トランザクションを自動的に起動して制御する
- 必要なときにサーバーコンポーネントを作成，実行，削除する
- 許可されたユーザーしかアプリケーションにアクセスできないようにするロールベースのセキュリティを提供する
- サーバー上で発生する条件にクライアントが応答できるようにイベントを管理する（COM+ のみ）

MTS または COM+ がこれらの基本サービスを提供することにより，開発者は分散アプリケーションの細部の開発に専念できます。（MTS または COM+ の）どちらを選択するかは，アプリケーションを実行するように選択したサーバーによって決まります。クライアントにとっては，両者の違い（さ

らに詳しく言えば，サーバーオブジェクトがこれらのサービスのいずれかを使うという事実）は透過です（ただし，クライアントが特殊なインターフェースを介して明示的にトランザクションサービスを操作する場合を除きます）。

トランザクションオブジェクトとは

一般に，トランザクションオブジェクトは小規模であり，別々の業務に使用されます。トランザクションオブジェクトは，アプリケーションのビジネスルールを実装し，アプリケーションの状態の表示と変更を行うことができます。例として，医療アプリケーションの場合を考えてみましょう。さまざまなデータベースに保存されている医療記録は，アプリケーションの持続的状態（患者の健康履歴など）を示します。トランザクションオブジェクトはその状態を更新し，新しい患者，検査結果，X線ファイルなどの変更を反映させます。

トランザクションオブジェクトがその他の COM オブジェクトと違う点は，分散コンピューティング環境で発生する問題を処理するために，MTS または COM+ が供給する属性のセットを使用することです。この属性の中には，トランザクションオブジェクトが IObjectControl インターフェースを実装する必要があるものもあります。IObjectControl は，オブジェクトが起動または停止するときに呼び出されるメソッドを定義し，データベース接続などのリソースを管理できます。IObjectControl は，44-9 ページの「オブジェクトのプール」で説明されているオブジェクトのプールにも必要です。

メモ MTS を使用する場合，トランザクションオブジェクトは IObjectControl を実装しなければなりません。COM+ では IObjectControl は必須ではありませんが，実装することを強くお勧めします。トランザクションオブジェクトウィザードを使うと，IObjectControl から派生したオブジェクトが作成されます。

トランザクションオブジェクトのクライアントは**ベースクライアント**と呼ばれます。ベースクライアントの観点からは，トランザクションオブジェクトは他の COM オブジェクトと同じように見えます。

MTS では，トランザクションオブジェクトをライブラリ（DLL）に組み込み，このライブラリを MTS 実行時環境（MTS の実行形式 mtxex.exe）にインストールしなければなりません。つまり，サーバーオブジェクトは MTS 実行時プロセス空間内で実行します。MTS 実行形式は，ベースクライアントと同じプロセスで実行するか，ベースクライアントと同じマシン上の別のプロセスとして実行するか，別のマシン上のリモートサーバープロセスとして実行することができます。

COM+ では，サーバーアプリケーションがインプロセスサーバーである必要はありません。さまざまなサービスが COM ライブラリに統合されているので，独立した MTS プロセスでサーバーの呼び出しをインターセプトする必要はありません。そのかわりに，COM 自身（正確には COM+）が，リソース管理やトランザクションサポートなどを提供します。ただし，それでもサーバーアプリケーションはインストールしなければならず，この場合は COM+ アプリケーションにインストールします。

ベースクライアントとトランザクションオブジェクトの間の接続は，アウトオブプロセスサーバーの場合と同様に，クライアント上のプロクシーとサーバー上のスタブによって処理されます。接続情報はプロクシーによって保持されます。ベースクライアントとプロクシーの間の接続は，クライアントがサーバーへの接続を要求している限り開いているので，クライアントにはサーバーへの接続を続行しているように見えます。実際には，プロクシーは他のクライアントが接続を使用できるようにリソースを節約するために，オブジェクトを停止してから再起動する場合があります。オブジェクトの起動と停止については，44-4 ページの「即時アクティブ化」を参照してください。

トランザクションオブジェクトの要件

COM の要件に加えて，トランザクションオブジェクトは以下の要件を満たさなければなりません。

- オブジェクトは標準クラスファクトリを持っていなければならない。クラスファクトリは，オブジェクトを作成するときにウィザードによって自動的に供給される
- サーバーは，標準の DllGetClassObject メソッドをエクスポートすることによって自分のクラスオブジェクトを提供しなければならない。これを行うコードはウィザードによって供給される
- すべてのオブジェクトインターフェースと CoClass がタイプライブラリによって記述されていなければならない。タイプライブラリはウィザードによって自動的に作成される。タイプライブラリエディタを使えば，タイプライブラリの中のインターフェースにメソッドとプロパティを追加できる。タイプライブラリの中の情報は，実行時にインストールされたコンポーネントに関する情報を抽出するために MTS エクスプローラまたはコンポーネントサービスによって使用される
- サーバーは標準 COM マーシャリングを使用するインターフェースだけをエクスポートしなければならない。これはトランザクションオブジェクトウィザードによって自動的に供給される。C++Builder バージョンのトランザクションオブジェクトは，カスタムインターフェース用の手動マーシャリングをサポートしない。したがって，すべてのインターフェースを COM の自動マーシャリングサポートを使用するデュアルインターフェースとして実装しなければならない
- サーバーは DllRegisterServer 関数をエクスポートして，このルーチンの中で CLSID，ProgID，インターフェース，およびタイプライブラリの自己登録を実行しなければならない。これはトランザクションオブジェクトウィザードによって提供される

COM+ ではなく MTS を使用する場合は，以下の条件も適用されます。

- MTS ではサーバーがダイナミックリンクライブラリ（DLL）でなければならない。実行ファイル（.EXE ファイル）として実装されたサーバーは，MTS 実行時環境では実行できない
- オブジェクトが IObjectControl インターフェースを実装しなければならない。このインターフェースのサポートは，トランザクションオブジェクトウィザードによって自動的に追加される
- MTS プロセス空間で実行されているサーバーは，MTS 内で実行されていない COM オブジェクトと一緒に集合体にすることはできない

リソースの管理

トランザクションオブジェクトは，アプリケーションが使用するリソースをよりよく管理できます。この場合のリソースとは，オブジェクトインスタンス自身のメモリをはじめとして，オブジェクトインスタンスが使用するあらゆるリソース（データベース接続など）を指します。

一般に，アプリケーションがどのようにリソースを管理するかの設定は，オブジェクトをインストールおよび設定する方法によって行います。トランザクションオブジェクトが以下の機能を利用するように設定します。

- 即時アクティブ化
- リソースのプール
- オブジェクトのプール（COM+ のみ）

ただし，オブジェクトがこれらのサービスをフル活用するようにしたい場合は，IObjectContext インターフェースを使用して，リソースを安全に解放できる時点を示さなければなりません。

オブジェクトのコンテキストへのアクセス

他の COM オブジェクトの場合と同様に，トランザクションオブジェクトも使用する前にまず作成する必要があります。COM クライアントは，COM ライブラリ関数 CoCreateInstance を呼び出してオブジェクトを作成します。

各トランザクションオブジェクトには，対応するコンテキストオブジェクトがなければなりません。このコンテキストオブジェクトは MTS または COM+ によって自動的に実装され，トランザクションオブジェクトの管理に使用されます。コンテキストオブジェクトのインターフェースは IObjectContext です。トランザクションオブジェクトウィザードが作成したトランザクションオブジェクトは，起動されるときに自動的にオブジェクトコンテキストをフェッチします。オブジェクトコンテキストは，メンバー変数 m_spObjectContext に格納されます。このオブジェクトコンテキストポインタは，たとえば次のように使います。

```
BOOL flags;
m_spObjectContext->IsCallerInRole(OLESTR("Manager"), &flags);
```

TMtsDll の Get_ObjectContext メソッドを呼び出しても，オブジェクトコンテキストへのポインタを取得できます。TMtsDll オブジェクトは，MTS または COM+ のどちらでアプリケーションが実行されていても機能するように，オブジェクトコンテキストをフェッチする呼び出しを適合させます。

```
IObjectContext* IAmWatchingYou = NULL;
TMtsDll MTSDLL;
HRESULT hr = MTSDLL.Get_ObjectContext(&IAmWatchingYou);
if (! (SUCCEEDED(hr)) )
{
  // エラー処理のコードをここに記述
}
BOOL flags;
IAmWatchingYou->IsCallerInRole(OLESTR("Manager"), &flags);
```

警告 comsvcs.h で定義された GetObjectContext マクロではなく，TMtsDll の Get_ObjectContext メソッドを使用してください。GetObjectContext マクロは，TDatabase の GetObjectContext メソッドの再定義を防止するために，VCL ヘッダーによって定義されません。

メモ m_spObjectContext メンバーは TMtsDll メソッドを使用して割り当てられるので，MTS と COM+ の両方で機能します。

即時アクティブ化

クライアントがオブジェクトへの参照を維持している間にそのオブジェクトを停止させて再起動する機能を**即時アクティブ化**と呼びます。クライアントの観点からは，クライアントがオブジェクトのインスタンスを作成してから最終的に解放するまでの間，たった 1 つのインスタンスしか存在しません。実際には，オブジェクトが何回も停止されては再起動された可能性があります。オブジェクトを停止させることにより，クライアントがシステムリソースに影響を与えずに，オブジェクトへの参照を長時間保持し続けることができます。オブジェクトが停止されると，リソースをすべて解放できま

す。たとえば，オブジェクトが停止されると，データベース接続を解放して他のオブジェクトが使用できるようにすることができます。

トランザクションオブジェクトは作成時には停止した状態で，クライアントからの要求を受信したときに起動します。トランザクションオブジェクトが作成されると，対応するコンテキストオブジェクトも作成されます。このコンテキストオブジェクトは，トランザクションオブジェクトが何回停止されて再起動されても，それに関係なく存在し続けます。このコンテキストオブジェクトは，IObjectContext インターフェースによってアクセスされ，停止中のオブジェクトを追跡し，トランザクション間の調整を行います。

トランザクションオブジェクトは，安全に停止できるようになると，ただちに停止されます。これを**即時非アクティブ化**と呼びます。トランザクションオブジェクトは，以下のいずれかが発生すると停止されます。

- **オブジェクトが SetComplete または SetAbort を使って停止を要求する。**オブジェクトは，操作が正常に完了し，クライアントからの次の呼び出しに備えてオブジェクトの内部状態を保存する必要がない場合には，IObjectContext の SetComplete メソッドを呼び出します。操作を正常に完了することができず，オブジェクトの状態を保存する必要がない場合は，SetAbort を呼び出します。つまり，オブジェクトの状態は現在のトランザクションが始まる前の状態にロールバックします。オブジェクトはよく，**ステートレス**として設計されます。これは，各メソッドから復帰したときにオブジェクトが停止されることを意味します。
- **トランザクションがコミットされるかアボートされる。**オブジェクトのトランザクションがコミットされるかアボートされると，そのオブジェクトは停止されます。これらの停止されたオブジェクトのうち存在し続けるのは，そのトランザクションの外のクライアントから参照されているものだけです。その後，存在し続けているオブジェクトへの呼び出しが実行されると，それらのオブジェクトは再起動され，新しいトランザクションで実行されます。
- **最後のクライアントがオブジェクトを解放する。**もちろん，クライアントがオブジェクトを解放するとそのオブジェクトは停止され，そのオブジェクトのコンテキストも解放されます。

メモ IDE から COM+ でトランザクションオブジェクトをインストールする場合は，タイプライブラリエディタの［COM+］ページを使用して，オブジェクトが即時アクティブ化をサポートするかどうかを指定できます。それには，タイプライブラリエディタでオブジェクト（CoClass）を選択し，［COM+］ページを表示し，［ジャストインタイムアクティベーション］ボックスにチェックマークを付けるかはずします。または，システム管理者がコンポーネントサービスまたは MTS エクスプローラを使って，この属性を指定します（システム管理者は，開発者がタイプライブラリエディタを使って指定した設定をオーバーライドすることもできます）。

リソースのプール

オブジェクトの停止中にアイドル状態のシステムリソースが解放されるので，解放されたリソースを他のサーバーオブジェクトが利用できます。たとえば，あるサーバーオブジェクトが使わなくなったデータベース接続を別のクライアントが再利用できるようになります。これを**リソースのプール**と呼びます。プールされたリソースは，リソースディスペンサによって管理されます。

リソースディスペンサは，リソースをキャッシュし，一緒にインストールされるトランザクションオブジェクトがリソースを共有できるようにします。リソースディスペンサは，非持続的共有状態の情

報も管理します。このように，リソースディスペンサは SQL Server などのリソースマネージャに似ていますが，持続性は保証しません。

トランザクションオブジェクトを作成する際に，すでに用意されている以下の 2 種類のリソースディスペンサを利用できます。

- データベースリソースディスペンサ
- 共有プロパティマネージャ

プールされたリソースを他のオブジェクトが使用できるようにするには，そのリソースを明示的に解放しなければなりません。

データベースリソースディスペンサ

データベースへの接続を開いたり閉じたりするには時間がかかります。リソースディスペンサを使ってデータベース接続をプールすると，オブジェクトが新しいデータベース接続を作成せずに，既存の接続を再利用できます。たとえば，顧客メンテナンスアプリケーションの中でデータベースルックアップコンポーネントとデータベース更新コンポーネントが動作している場合，2 つのコンポーネントを一緒にインストールすれば，データベース接続を共有できるようになります。したがって，すでに開いているが使用されていない接続を使用することで，アプリケーションはさほど多くの接続を必要としなくなり，新しいオブジェクトインスタンスがデータにすばやくアクセスできます。

- BDE コンポーネントを使用してデータに接続する場合，リソースディスペンサはボーランドデータベースエンジン（BDE: Borland Database Engine）である。このリソースディスペンサは，MTS を使ってトランザクションオブジェクトをインストールした場合にしか使用できない。リソースディスペンサが使用できるようにするには，BDE Administrator を使用して環境設定の［System｜Init］領域で［MTS POOLING］をオンにする
- ADO データベースコンポーネントを使用してデータに接続する場合，リソースディスペンサは ADO によって提供される

メモ データベースアクセスに InterbaseExpress コンポーネントを使用する場合，組み込みのリソースプールはありません。

リモートトランザクションデータモジュールの場合，接続は自動的にオブジェクトのトランザクションに登録され，リソースディスペンサが自動的に接続を回収して再利用できるようになります。

共有プロパティマネージャ

共有プロパティマネージャは，サーバープロセス内の複数のオブジェクトの間で状態を共有できるようにするリソースディスペンサです。共有プロパティマネージャを使用すると，共有データを管理するためにアプリケーションに大量のコードを追加する必要がなくなります。つまり，共有プロパティマネージャは，共有プロパティを同時アクセスから保護するロックとセマフォを実装することにより，共有データを自動的に管理します。共有プロパティマネージャは，**共有プロパティグループ**を提供することで名前の衝突を排除します。共有プロパティグループは，その中の共有プロパティ用にユニークな名前空間を確立します。

共有プロパティマネージャリソースを使うには，まず CreateSharedPropertyGroup ヘルパー関数を使って共有プロパティグループを作成します。すると，すべてのプロパティをそのグループに書き込んだり，そのグループから読み取れるようになります。共有プロパティグループを使うことにより，1 つ

のトランザクションオブジェクトが何回停止されても常に状態情報が保存されるようになります。さらに，同じ MTS パッケージまたは COM+ アプリケーションにインストールされたすべてのトランザクションオブジェクトの間で状態情報を共有することもできます。44-27 ページの「トランザクションオブジェクトのインストール」で説明しているように，トランザクションオブジェクトはパッケージにインストールすることができます。

オブジェクトが状態を共有するためには，それらのオブジェクトがすべて同じプロセスの中で実行していなければなりません。複数の異なるコンポーネントのインスタンスの間でプロパティを共有したい場合は，それらのコンポーネントを同じ MTS パッケージまたは COM+ アプリケーションにインストールしなければなりません。管理者がコンポーネントを 1 つのパッケージから別のパッケージに移す可能性があるので，共有プロパティグループの使用は，同じ DLL または EXE の中で定義されたオブジェクトのインスタンス間に限るのが安全です。

プロパティを共有するオブジェクトは同じ起動属性を持っていなければなりません。同一パッケージ内の 2 つのコンポーネントの起動属性が異なっている場合，通常それらのコンポーネントはプロパティを共有できません。たとえば，一方のコンポーネントがクライアントのプロセスの中で動作するように設定され，もう一方のコンポーネントがサーバープロセスの中で動作するように設定されている場合，通常これらのコンポーネントのオブジェクトは（たとえ同じ MTS パッケージまたは COM+ アプリケーション中にあっても）別々のプロセスの中で動作します。

次の例は，トランザクションオブジェクトの中で共有プロパティマネージャをサポートするためのコードを追加する方法を示しています。

例：トランザクションオブジェクトインスタンス間でのプロパティの共有

この例は，オブジェクトおよびオブジェクトインスタンスの間で共有するプロパティを収める MyGroup というプロパティグループを作成します。この例では，Counter プロパティを共有します。CreateSharedPropertyGroup ヘルパー関数を使ってプロパティグループマネージャとプロパティグループを作成した後，Group オブジェクトの CreateProperty メソッドを使って Counter というプロパティを作成します。

プロパティの値を取得するには，以下に示したように Group オブジェクトの PropertyByName メソッドを使います。PropertyByPosition メソッドを使うこともできます。

```
#include "Project1_TLB.H"
#define _MTX_NOFORCE_LIBS
#include <vcl\mtshlpr.h>

class ATL_NO_VTABLE TSharedPropertyExampleImpl :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<TSharedPropertyExampleImpl, &CLSID_SharedPropertyExample>,
  public IObjectControl,
  public IDispatchImpl<ISharedPropertyExample, &IID_ISharedPropertyExample,
                       &LIBID_Project1>
{
private:
  ISharedPropertyGroupManager* manager;
  ISharedPropertyGroup* PG13;
  ISharedProperty* Counter;
public:
  TSharedPropertyExampleImpl()
  {
  }
```

```
  DECLARE_THREADING_MODEL(otApartment);
  DECLARE_PROGID("Project1.SharedPropertyExample");
  DECLARE_DESCRIPTION("");
static HRESULT WINAPI UpdateRegistry(BOOL bRegister)
{
  TTypedComServerRegistrarT<TSharedPropertyExampleImpl>
  regObj(GetObjectCLSID(), GetProgID(), GetDescription());
  return regObj.UpdateRegistry(bRegister);
}
DECLARE_NOT_AGGREGATABLE(TSharedPropertyExampleImpl)
BEGIN_COM_MAP(TSharedPropertyExampleImpl)
  COM_INTERFACE_ENTRY(ISharedPropertyExample)
  COM_INTERFACE_ENTRY(IObjectControl)
  COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()
public:
  STDMETHOD(Activate)();
  STDMETHOD(IncrementCounter());
  STDMETHOD_(BOOL, CanBePooled)();
  STDMETHOD_(void, Deactivate)();
CComPtr<IObjectContext> m_spObjectContext;
public:
};
// SHAREDPROPERTYEXAMPLEIMPL : TSharedPropertyExampleImpl の実装
#include <vcl.h>
#pragma hdrstop
#include "SHAREDPROPERTYEXAMPLEIMPL.H"
STDMETHODIMP TSharedPropertyExampleImpl::Activate()
{
  static TMtsDll Mts;
  HRESULT hr = E_FAIL;
  hr = Mts.Get_ObjectContext(&m_spObjectContext);
  if (SUCCEEDED(hr))
  {
    VARIANT_BOOL _false = VARIANT_FALSE;
    VARIANT_BOOL _true = VARIANT_TRUE;
    CoCreateInstance( CLSID_SharedPropertyGroupManager, NULL, CLSCTX_INPROC_SERVER,
                      IID_ISharedPropertyGroupManager, (void**)manager);
    manager->CreatePropertyGroup(L"Test Group", LockSetGet, Standard, &_false, &PG13);
    if ( (PG13->CreateProperty(L"Counter", &_true, &Counter)) == S_OK)
    {
      Counter->put_Value(TVariant(0));
    }
    return S_OK;
  }
  return hr;
}
STDMETHODIMP_(BOOL) TSharedPropertyExampleImpl::CanBePooled()
{
  return FALSE;
}
STDMETHODIMP_(void) TSharedPropertyExampleImpl::Deactivate()
{
```

```
  PG13->Release();
  manager->Release();
  m_spObjectContext.Release();
}
STDMETHODIMP TSharedPropertyExampleImpl::IncrementCounter()
{
  try
  {
    TVariant temp;
    Counter->get_Value(&temp);
    temp=(int)temp+1;;
    Counter->put_Value(temp);
  }
  catch(Exception &e)
  {
    return Error(e.Message.c_str(), IID_ISharedPropertyExample);
  }
  return S_OK;
}
```

リソースの解放

開発者はオブジェクトのリソースを解放する責任があります。そのための一般的な方法は，クライアント要求にサービスを提供した後で IObjectContext の SetComplete および SetAbort メソッドを呼び出すことです。これらのメソッドは，リソースディスペンサが割り当てたリソースを解放します。

リソースの解放と同時に，他のオブジェクト（トランザクションオブジェクトとコンテキストオブジェクトも含む）への参照も含めた他のすべてのリソースへの参照と，コンポーネントのインスタンスが占めていたメモリも解放し（コンポーネントを解放し）なければなりません。

これらの呼び出しを含めなくていいのは，クライアントからの呼び出しと呼び出しの間でオブジェクトの状態を維持したい場合だけです。詳細については，44-12 ページの「ステートフルオブジェクトとステートレスオブジェクト」を参照してください。

オブジェクトのプール

リソースをプールできるのと同じように，COM+ ではオブジェクトもプールできます。オブジェクトが停止されると，COM+ は IObjectControl インターフェースの CanBePooled メソッドを呼び出します。このメソッドは，そのオブジェクトを再利用のためにプールできることを示します。CanBePooled が **true** を返す場合は，オブジェクトは停止時に破棄されずに，オブジェクトプールに移されます。オブジェクトプール内のオブジェクトは指定されたタイムアウト時間が経過するまで保持され，それまではどのクライアントの要求にも使用できるようになっています。オブジェクトプールが空の場合にだけ，オブジェクトの新しいインスタンスが作成されます。**false** を返したオブジェクトや，IObjectControl インターフェースをサポートしていないオブジェクトは，停止時に破棄されます。

オブジェクトのプールは MTS では提供されていません。MTS は前述のように CanBePooled を呼び出しますが，プーリングは行われません。このため，ウィザードがトランザクションオブジェクトを作成すると，生成された CanBePooled メソッドは常に **false** を返します。オブジェクトを COM+ でのみ実行し，オブジェクトプーリングを使用できるようにしたい場合は，実装ユニット内でこのメソッドを探し出し，**true** を返すように編集します。

オブジェクトの CanBePooled メソッドが **true** を返す場合でも，COM+ がオブジェクトをオブジェクトプールに移動しないように設定できます。IDE から COM+ でトランザクションオブジェクトをインストールする場合は，タイプライブラリエディタの［COM+］ページを使用して，COM+ がオブジェクトをプールしようとするかどうかを指定できます。それには，タイプライブラリエディタでオブジェクト（CoClass）を選択し，［COM+］ページを表示し，［オブジェクトのプール］ボックスにチェックマークを付けるかはずします。または，システム管理者がコンポーネントサービスまたは MTS エクスプローラを使って，この属性を指定します

同様に，停止されたオブジェクトを解放するまでオブジェクトプール内に保持する時間を設定できます。IDE からインストールする場合は，タイプライブラリエディタの［COM+］ページの［作成時タイムアウト］設定を使用して，この時間を指定できます。または，システム管理者がコンポーネントサービスを使って，この属性を指定します。

MTS および COM+ のトランザクションサポート

トランザクションオブジェクトに名前を与えるトランザクションサポートは，複数のアクションをまとめてトランザクションにすることができます。たとえば医療記録アプリケーションの場合，記録を 1 人の医師から別の医師へ転送するための Transfer コンポーネントがあるとすると，同じトランザクションの中に Add メソッドと Delete メソッドを含めることになるでしょう。これによって，転送の全体が成功するか，トランザクション前の状態にロールバックできるようになります。トランザクションは，複数のデータベースにアクセスする必要のあるアプリケーションのエラー復旧を簡単にします。

トランザクションは以下のことを保証します。

- 1 つのトランザクションで行われた更新は，すべてコミットされるか，アボートされて直前の状態にロールバックされる。これは**原子性**と呼ばれる
- トランザクションはシステムの状態の正しい変化であり，状態の不変量を維持する。これは**一貫性**と呼ばれる
- 同時トランザクションは，互いの部分的でコミットされていない結果を見ることはない。それを見るとアプリケーションの状態に矛盾が生じる可能性があるからである。これは**分離性**と呼ばれる。リソースマネージャはトランザクションベースの同期プロトコルを使って，アクティブなトランザクションのコミットされていない作業内容を分離する
- 管理対象のリソース（データベースレコードなど）にコミットされた更新は，障害（通信障害，プロセス障害，サーバーシステム障害など）が起きても保持される。これは**持続性**と呼ばれる。トランザクションログ機能により，ディスク媒体の障害の後でも持続状態を回復することができる

オブジェクトに関連付けられたコンテキストオブジェクトは，そのオブジェクトがトランザクションの中で動作するかどうかを示し，動作する場合はそのトランザクションを識別します。オブジェクトがトランザクションの一部である場合，リソースマネージャとリソースディスペンサがそのオブジェクトに代わって提供するサービスが，トランザクションの下でも実行されます。リソースディスペンサは，コンテキストオブジェクトを使ってトランザクションベースのサービスを提供します。たとえば，トランザクション内で動作しているオブジェクトが ADO または BDE リソースディスペンサを使ってデータベース接続を割り当てると，その接続は自動的にそのトランザクションのものとして登

録されます。この接続を使用するデータベース更新はすべてそのトランザクションの一部になり，コミットされるかアボートされます。

複数のオブジェクトからの作業を 1 つのトランザクションに構成することができます。オブジェクトが自分のトランザクションの中で，または単一のトランザクションに所属するより大きなオブジェクトグループの一部として動作できることは，MTS および COM+ の大きな利点の 1 つです。これによって，さまざまな方法でオブジェクトを使用できるので，アプリケーション開発者はアプリケーションの論理を変更せずに，別のアプリケーションの中でアプリケーションコードを再利用できます。実際，開発者はトランザクションオブジェクトをインストールするときに，そのオブジェクトがトランザクションの中でどのように使用されるかを決定することができます。オブジェクトを別の MTS パッケージまたは COM+ アプリケーションに追加するだけで，トランザクションの動作を変更できます。トランザクションオブジェクトのインストール方法についての詳細は，44-27 ページの「トランザクションオブジェクトのインストール」を参照してください。

トランザクション属性

トランザクションオブジェクトは，MTS カタログに記録されるか COM+ に登録されるトランザクション属性を持っています。

C++Builder では，トランザクションオブジェクトウィザードまたはタイプライブラリエディタを使って，設計時にトランザクション属性を設定できます。

各トランザクション属性は次のように設定することができます。

トランザクションが必要 オブジェクトは 1 つのトランザクションのスコープ内で動作しなければならない。新しいオブジェクトが作成されると，そのオブジェクトコンテキストはクライアントのコンテキストからトランザクションを継承する。クライアントがトランザクションコンテキストを持っていない場合は，新しいコンテキストが自動的に作成される

新しいトランザクションが必要 オブジェクトは自分自身のトランザクションの中で動作しなければならない。新しいオブジェクトが作成されると，そのオブジェクトのクライアントがトランザクションを持っているかどうかに関係なく，そのオブジェクトのための新しいトランザクションが自動的に作成される。オブジェクトがクライアントのトランザクションの中で動作することはない。かわりに，システムが新しいオブジェクトのために独立したトランザクションを作成する

トランザクションのサポート オブジェクトはクライアントのトランザクションの中で動作できる。新しいオブジェクトが作成されると，そのオブジェクトコンテキストはクライアントのコンテキストからトランザクションを継承する。これによって，複数のオブジェクトを 1 つのトランザクションに構成できるようになる。クライアントがトランザクションを持っていない場合は，新しいコンテキストもトランザクションなしで作成される

未サポート オブジェクトはトランザクションのスコープ内では動作しない。新しいオブジェクトが作成されると，そのオブジェクトのクライアントがトランザクションを持っているかどうかに関係なく，オブジェクトコンテキストがトランザクションなしで作成される。この設定は COM+ でしか使用できない

サポートしない この設定の意味は，MTS または COM+ のどちらでオブジェクトをインストールするかによって異なる。MTS では，COM+ での［未サポート］と同じ意味。COM+ では，オブジェクトコンテキストがトランザクションなしで作成されるだけでなく，クライアントがトランザクションを持っている場合に，オブジェクトが起動されないようにする

トランザクション属性の設定

トランザクション属性は，トランザクションオブジェクトウィザードを使ってトランザクションオブジェクトを最初に作成するときに設定できます。

タイプライブラリエディタを使ってもトランザクション属性を設定（または変更）できます。タイプライブラリエディタでトランザクション属性を変更する手順は次のとおりです。

1. ［表示｜タイプライブラリ］を選択してタイプライブラリエディタを開きます。
2. トランザクションオブジェクトに対応するクラスを選択します。
3. ［COM+］タブをクリックし，使いたいトランザクション属性を選択します。

警告 トランザクション属性を設定するときに，C++Builder は指定された属性に対する特殊な GUID をカスタムデータとしてタイプライブラリに挿入します。この値は C++Builder の外部では認識されません。したがって，IDE からトランザクションオブジェクトをインストールする場合にしか効果がありません。それ以外の場合には，システム管理者が MTS エクスプローラまたはコンポーネントサービスを使ってこの値を設定しなければなりません。

メモ トランザクションオブジェクトがすでにインストールされている場合は，トランザクション属性を変更するときに，まずそのオブジェクトをアンインストールしてから再インストールしなければなりません。それには［実行｜ MTS オブジェクトのインストール］または［実行｜ COM+ オブジェクトのインストール］を使います。

ステートフルオブジェクトとステートレスオブジェクト

他の COM オブジェクトと同様に，トランザクションオブジェクトはクライアントとの複数のやりとりを通じて内部状態を維持することができます。たとえば，クライアントが 1 つの呼び出し内でプロパティ値を設定し，次の呼び出しのときにそのプロパティ値が変わっていないことを予想できます。このようなオブジェクトは**ステートフルオブジェクト**と呼ばれます。トランザクションオブジェクトは**ステートレス**にもできます。これは，オブジェクトがクライアントからの次の呼び出しを待つ間，中間状態を保持しないという意味です。

トランザクションがコミットされるかアボートされると，そのトランザクションに関係したオブジェクトはすべて停止され，トランザクションの最中に獲得した状態を失います。これによって，トランザクションの隔離とデータベースの一貫性が確保され，他のトランザクションが利用できるようにサーバーリソースが解放されます。トランザクションが完了すると，オブジェクトが停止したときに，そのオブジェクトが保持していたリソースを回収できるようになります。オブジェクトの状態が解放されるタイミングをコントロールする方法については，次のセクションを参照してください。

オブジェクトの状態を維持するためには，そのオブジェクトがアクティブなままでいる必要があるので，データベース接続などの潜在的に貴重なリソースを保持したままになります。

トランザクションの完了方法の決定

トランザクションをどのように完了させるかに影響を与えるために，トランザクションオブジェクトは，次の表に示すように IObjectContext のメソッドを使います。これらのメソッドをオブジェクトのトランザクション属性とともに使えば，1 つまたは複数のオブジェクトを 1 つのトランザクションに含めることができます。

表 44.1 トランザクションサポート用の IObjectContext のメソッド

メソッド	内容
SetComplete	オブジェクトがそのトランザクションのための作業を正常に完了したことを示す。オブジェクトは，そのコンテキストに最初に入ったメソッドから復帰したときに停止される。そのオブジェクトの実行を要求する次の呼び出しが発生すると，オブジェクトが再起動される
SetAbort	オブジェクトの作業をコミットできず，トランザクションをロールバックする必要があることを示す。オブジェクトは，そのコンテキストに最初に入ったメソッドから復帰したときに停止される。そのオブジェクトの実行を要求する次の呼び出しが発生すると，オブジェクトが再起動される
EnableCommit	オブジェクトの作業が完了しているとは限らないが，トランザクションによる更新を現在の形のままコミットできるということを示す。このメソッドは，クライアントからの複数の呼び出しにわたって状態を維持しながら，トランザクションを完了できるようにするために使用する。SetComplete または SetAbort が呼び出されるまで，オブジェクトが停止されない EnableCommit は，オブジェクトが起動されたときのデフォルト状態である。これは，クライアントからの次の呼び出しのためにオブジェクトの内部状態を維持したいのでない限り，オブジェクトがメソッドから復帰する前に常に SetComplete か SetAbort を呼び出さなければならないためである
DisableCommit	オブジェクトの作業が一貫しておらず，クライアントからさらにメソッドが呼び出されるまでは作業を完了できないことを示す。このメソッドは，クライアントからの複数の呼び出しにわたってオブジェクトの状態を維持しながら，現在のトランザクションをアクティブに保つために，クライアントに制御を返す前に呼び出す DisableCommit は，メソッド呼び出しから復帰するときに，オブジェクトの停止とそのリソースの解放を防止する。オブジェクトが DisableCommit を呼び出した場合，オブジェクトが EnableCommit か SetComplete を呼び出す前にクライアントがトランザクションをコミットしようとすると，トランザクションはアボートされる

トランザクションの起動

トランザクションは以下の 3 つの方法で制御できます。

- クライアントによって制御できる

  クライアントは，トランザクションコンテキストオブジェクトを使って（ITransactionContext インターフェースを使って）トランザクションを直接制御することができます。

- サーバーによって制御できる

  サーバーはオブジェクトコンテキストを明示的に作成してトランザクションを制御できます。この方法でサーバーがオブジェクトを作成すると，作成されたオブジェクトは現在のトランザクションに自動的に登録されます。

- トランザクションはオブジェクトのトランザクション属性の結果として自動的に発生することができる

  トランザクションオブジェクトは，そのオブジェクトがどのようにして作成されたかに関係なく，オブジェクトが常にトランザクションの中で動作するように宣言することができます。この方法をとれば，トランザクションを処理する論理をオブジェクトに含める必要がありません。これによって，クライアントアプリケーションの負担も減ります。クライアントは，単に自分が使用するコンポーネントがそれを必要としているという理由でトランザクションを起動する必要はありません。

クライアント側でのトランザクションオブジェクトのセットアップ

クライアントアプリケーションは，ITransactionContextEx インターフェースを介してトランザクションコンテキストを制御することができます。以下のコード例は，クライアントアプリケーションが CreateTransactionContextEx をどのように使ってトランザクションコンテキストを作成するかを示しています。このメソッドは，このオブジェクトへのインターフェースを返します。

```
#include <vcl \ mtshlpr.h>
int main(int argc, char* argv[])
{
  // まずトランザクションオブジェクトを作成する [MTS をローカルにインストールするか COM+ が必要 ]
  TCOMITransactionClientExample first_client = CoTransactionClientExample::Create();
  // 次に，これがトランザクション状態にあるかどうかを調べる
  ITransactionContext* TCTX;
  HRESULT happily_transacting = first_client->QueryInterface(IID_ITransactionContext,
    (void**)&TCTX);
  // トランザクション状態のときは，ここからデータアクセスの呼び出しが可能。
  // 次に，この呼び出しを実行するトランザクションオブジェクトを待たずに
  // Commit を呼び出すかアボートする
  if (happily_transacting)
  {
    TVariant database = "name";
    TVariant record = "data";
    TVariant flag;
    first_client.UpdateData(&database, &record, &flag);
    flag ?TCTX->Commit() : TCTX->Abort();
  }
  else
  {
    // トランザクション状態でない場合は，トランザクションを作成できる
    ITransactionContextEx* TXCTX  = CreateTransactionContextEx();
    // さらに 1 つのオブジェクトを作成。このように作成されるオブジェクトはすべて
    // このオブジェクトで表されるトランザクションにリストされる
    TCOMITransactionClientExample* second_client;
    TXCTX->CreateInstance(CLSID_TransactionClientExample, IID_ITransactionClientExample,
      (void**)&second_client);
    // 次にデータアクセスを実行し，コミットまたはアボートする
    TVariant database = "name";
    TVariant record = "data";
    TVariant flag;
    second_client->UpdateData(&database, &record, &flag);
    flag ?TXCTX->Commit() : TXCTX->Abort();
  }
```

```
  return 0;
}
```

サーバー側でのトランザクションオブジェクトのセットアップ

サーバー側からトランザクションコンテキストを制御するには，ObjectContext のインスタンスを作成します。以下の例では，Transfer メソッドがトランザクションオブジェクトの中にあります。ObjectContext をこのように使用すると，作成されたオブジェクトのインスタンスは，それを作成したオブジェクトのすべてのトランザクション属性を継承します。

```
#include <vcl\mtshlpr.h>
STDMETHODIMP TTransactionServerExampleImpl::DoTransactionContext(long execflag)
{
  if (m_spObjectContext->IsInTransaction())
  {
    // これは現在のオブジェクトがトランザクションを持ち，
    // そのトランザクション情報を子に渡せることを意味する。
    // 簡単にいうと，このオブジェクトは単に自分自身と同じ型の別のオブジェクトを
    // 同じトランザクション内に作成する。
    // 注記：それでも，必要であれば開発者が集約を行う責任がある
    if (execflag)
    {
      TCOMITransactionServerExample* inner;
      m_spObjectContext->CreateInstance(CLSID_TransactionServerExample,
                                        IID_ITransactionServerExample, (void**)&inner);
      inner->DoTransactionContext(false);
      // ここにデータアクセスコードを追加する。次の data_access_succeeded() は
      // 実装されていないプレースホルダである
      data_access_succeeded() ? m_spObjectContext->EnableCommit()
                              : m_spObjectContext->DisableCommit();
    }
  }
  else
  {
    // これは，現在のオブジェクトにトランザクションがないため，
    // クライアントと同じやり方でトランザクションを作成する必要があることを意味する
    ITransactionContextEx* TCTX = CreateTransactionContextEx();
    TCOMITransactionServerExample* inner;
    // 後で同じ手順に従う
    TCTX->CreateInstance(CLSID_TransactionServerExample,
                         IID_ITransactionServerExample, (void**)&inner);
    inner->DoTransactionContext(true);
    // ここにデータアクセスコードを追加する。次の data_access_succeeded() は
    // 実装されていないプレースホルダである
    data_access_succeeded() ? TCTX->Commit() : TCTX->Abort();
  }
}
```

トランザクションのタイムアウト

トランザクションタイムアウトは，トランザクションがアクティブでいられる時間の長さ（秒数）を設定します。タイムアウト後もアクティブのままになっているトランザクションは，システムによっ

て自動的にアボートされます。デフォルトのタイムアウト値は 60 秒です。値として 0 を指定すると，トランザクションのタイムアウトを無効にすることができます。これは，トランザクションオブジェクトをデバッグするときに役立ちます。

コンピュータでタイムアウト値を設定する手順は次のとおりです。

1. MTS エクスプローラまたはコンポーネントサービスで，［コンピュータ\マイコンピュータ］を選択します。

   デフォルトでは，［マイコンピュータ］はローカルコンピュータに対応します。

2. 右クリックして［プロパティ］を選択し，［オプション］タブを選択します。

   ［オプション］タブは，コンピュータのトランザクションタイムアウトプロパティを設定するために使用されます。

3. トランザクションのタイムアウトを無効にするために，タイムアウト値を 0 に変更します。

4. ［OK］をクリックして設定を保存します。

MTS アプリケーションのデバッグについての詳細は，44-26 ページの「トランザクションオブジェクトのデバッグとテスト」を参照してください。

ロールベースのセキュリティ

MTS と COM+ は，ユーザーの論理グループにロール（役割）を割り当てるというロールベースのセキュリティを提供します。たとえば医療情報アプリケーションでは，医師，X 線技師，患者などのロールを定義することになるでしょう。

それぞれのオブジェクトとインターフェースについて，ロールを割り当てることによって権限を定義します。たとえば医師の医療アプリケーションの場合，医師だけがすべての医療記録を見ることができ，X 線技師は X 線写真だけを見ることができ，患者は自分の医療記録しか見られないということになるでしょう。

一般に，ロールはアプリケーションの開発中に定義し，各 MTS パッケージまたは COM+ アプリケーションにロールを割り当てます。これらのロールはその後，アプリケーションが配布されたときに特定のユーザーに割り当てられます。管理者は，MTS エクスプローラまたはコンポーネントサービスを使ってロールを設定することができます。

オブジェクト全体ではなくコードのブロックへのアクセスを制御したい場合は，IObjectContext メソッドの IsCallerInRole を使ってきめ細かいセキュリティを提供できます。このメソッドはセキュリティが有効になっている場合のみ機能します。セキュリティが有効になっているかどうかを調べるには，IObjectContext メソッドの IsSecurityEnabled を呼び出します。たとえば，次のようにします。

```
if (m_spObjectContext.IsSecurityEnabled()) // セキュリティが有効になっているかどうかを調べる
{
  if (!m_spObjectContext.IsCallerInRole("Physician")) // 呼び出し側のロールを調べる
  { // 医師でない場合は，ここで適切な処理を行う
  }
  else
  { // 普通に呼び出しを実行する
  }
```

```
}
else // セキュリティが有効になっていない
{ // 適切な処理を行う
}
```

メモ より強力なセキュリティが必要なアプリケーションでは，コンテキストオブジェクトに ISecurityProperty インターフェースを実装します。このインターフェースのメソッドにより，オブジェクトの直接呼び出し側やオブジェクトの作成者用に Windows セキュリティ識別子（SID），およびオブジェクトを使用するクライアント用に SID を取得できます。

トランザクションオブジェクト作成の概要

トランザクションオブジェクトを作成する手順は次のとおりです。

1. トランザクションオブジェクトウィザードを使ってトランザクションオブジェクトを作成します。
2. タイプライブラリエディタを使ってオブジェクトのインターフェースにメソッドとプロパティを追加します。タイプライブラリエディタを使ってメソッドとプロパティを追加する方法については，第 39 章「タイプライブラリの操作」を参照してください。
3. オブジェクトのメソッドを実装するときに，IObjectContext インターフェースを使ってトランザクション，持続的ステート，およびセキュリティを管理することができます。さらに，オブジェクト参照を渡す場合は，オブジェクト参照が正しく処理されるように特別な注意を払う必要があります（44-25 ページの「オブジェクト参照を渡す」を参照してください）。
4. トランザクションオブジェクトをデバッグしてテストします。
5. 作成したトランザクションオブジェクトを MTS パッケージまたは COM+ アプリケーションにインストールします。
6. MTS エクスプローラまたはコンポーネントサービスを使ってオブジェクトを管理します。

トランザクションオブジェクトウィザードの使い方

MTS または COM+ が提供するリソース管理，トランザクション処理，ロールベースのセキュリティを利用できる COM オブジェクトを作成するには，トランザクションオブジェクトウィザードを使います。

トランザクションオブジェクトウィザードを起動する手順は次のとおりです。

1. ［ファイル｜新規作成｜その他］を選択します。
2. ［ActiveX］タブを選択します。
3. ［トランザクションオブジェクト］アイコンをダブルクリックします。

ウィザードでは，以下のものを指定しなければなりません。

- クライアントアプリケーションがオブジェクトのインターフェースを呼び出すために使える方法を示すスレッドモデル。このスレッドモデルは，そのオブジェクトがどのように登録されるかを決定する。選択したモデルに忠実に，オブジェクトを実装しなければならない。スレッドモデル

についての詳細は，44-18 ページの「トランザクションオブジェクトのスレッドモデルの選択」を参照

- トランザクションモデル
- オブジェクトがクライアントにイベントを通知するかどうか。イベントサポートは従来型のイベントでのみ提供され，COM+ イベントでは提供されない

この手順が完了すると，トランザクションオブジェクトの定義が入っている現在のプロジェクトに，新規ユニットが追加されます。さらに，タイプライブラリがプロジェクトに追加され，タイプライブラリエディタで開かれます。これで，タイプライブラリを介してインターフェースのプロパティとメソッドをエクスポーズできるようになります。41-9 ページの「COM オブジェクトのインターフェースを定義する」で説明しているように，インターフェースは他の COM オブジェクトと同じ方法で定義できます。

トランザクションオブジェクトは，仮想テーブルによるアーリー（コンパイル時）バインディングと IDispatch インターフェースによるレイト（実行時）バインディングを両方ともサポートする**デュアルインターフェース**を実装します。

生成されたトランザクションオブジェクトは，IObjectControl インターフェースのメソッド Activate，Deactivate，および CanBePooledObject を実装します。

必ずしもトランザクションオブジェクトウィザードを使用する必要はありません。タイプライブラリエディタの［COM+］ページを使用して MTS パッケージまたは COM+ アプリケーションにオブジェクトをインストールすれば，オートメーションオブジェクトを COM+ トランザクションオブジェクトに変換できます（およびインプロセスのオートメーションオブジェクトを MTS トランザクションオブジェクトに変換できます）。ただし，トランザクションオブジェクトウィザードには以下のような利点があります。

- IObjectControl インターフェースを自動的に実装し，OnActivate および OnDeactivate イベントをオブジェクトに追加するので，オブジェクトが起動または停止されたときに応答するイベントハンドラを作成できる
- m_spObjectContext メンバーを自動的に生成するので，オブジェクトが IObjectContext メソッドにアクセスして起動とトランザクションの制御が簡単にできる

トランザクションオブジェクトのスレッドモデルの選択

MTS 実行時環境または COM+ はスレッドを管理します。トランザクションオブジェクトはスレッドを作成してはなりません。トランザクションオブジェクトは，DLL の中身を呼び出すスレッドを終了させてもなりません。

トランザクションオブジェクトウィザードを使ってスレッドモデルを指定すると，オブジェクトがメソッド実行のためにスレッドにどのように割り当てられるかが指定されます。

表 44.2 トランザクションオブジェクトのスレッドモデル

スレッドモデル	内容	実装の長所と短所
シングル	スレッドはサポートされない。クライアントからの要求は呼び出し機構によってシリアル化される シングルスレッドコンポーネントのオブジェクトはすべてメインスレッド上で実行される これは，Threading Model Registry 属性を持たないコンポーネントや再入可能でない COM コンポーネントに使用されるデフォルトの COM スレッドモデルと互換性がある。メソッドの実行はコンポーネント内のすべてのオブジェクトおよびプロセス内のすべてのコンポーネントにわたってシリアル化される	コンポーネントが再入可能でないライブラリを使える 拡張性が非常に限られている シングルスレッドのステートフルコンポーネントはデッドロックを起こしやすい。この問題は，ステートレスオブジェクトを使用し，メソッドから復帰する前に SetComplete を呼び出すことによって排除できる
アパートメント（またはシングルスレッドアパートメント）	オブジェクトはそれぞれ 1 つのスレッドアパートメントに割り当てられる。アパートメントはそのオブジェクトが生きている限り存続する。複数のオブジェクトに複数のスレッドを使用することもできる。これは COM の標準同時性モデルである。各アパートメントは特定のスレッドに結び付けられ，Windows メッセージポンプを持っている	シングルスレッドモデルと比べて同時性が大きく向上する 2 つのオブジェクトが同じアクティビティに属していなければ，それらは同時に動作できる オブジェクトを複数のプロセスに分散できること以外は，COM アパートメントに似ている
フリー / アパートメント両用	クライアントに対するコールバックがシリアル化される以外はアパートメントと同じ	長所はアパートメントと同じ。さらに，オブジェクトのプールを使用する場合はこのモデルが必要

メモ これらのスレッドモデルは，COM オブジェクトによって定義されるものと似ています。ただし，MTS および COM+ はスレッドについてより多くの基本サポートを提供するので，各スレッドモデルの意味は異なっています。また，アクティビティのサポートが組み込まれているため，フリースレッドモデルはトランザクションオブジェクトには適用されません。

アクティビティ

スレッドモデルのほかにも，トランザクションオブジェクトは**アクティビティ**を通じて同時性を達成します。アクティビティはオブジェクトのコンテキストに記録され，オブジェクトとアクティビティの関連付けは変更できません。アクティビティには，ベースクライアントが作成したトランザクションオブジェクトのほかに，そのオブジェクトと子孫が作成したトランザクションオブジェクトも含まれます。これらのオブジェクトは，1 つまたは複数のプロセスに分散させることができ，1 台または複数のコンピュータで実行できます。

たとえば医師の医療アプリケーションは，それぞれが異なるオブジェクトで表されたさまざまな医療データベースに更新を追加したりレコードを削除するためのトランザクションオブジェクトを持っているでしょう。このレコードオブジェクトは，たとえばトランザクションを記録するための領収証オブジェクトなど，他のオブジェクトを使用するかもしれません。その結果，複数のトランザクションオブジェクトがベースクライアントによって直接または間接的に制御されることになります。これらのオブジェクトはすべて同じアクティビティに属します。

MTS または COM+ は各アクティビティでの実行の流れを追跡し，偶然の並行処理によるアプリケーションの状態の破壊を防ぎます。この機能により，潜在的に分散したオブジェクトの集まりが単一の

論理スレッドで実行されます。1 つの論理スレッドを持つことにより，アプリケーションの作成が非常に簡単になります。

トランザクションコンテキストオブジェクトかオブジェクトコンテキストを使ってトランザクションオブジェクトが既存のコンテキストから作成された場合，その新しいオブジェクトは同じアクティビティのメンバーになります。つまり，新しいコンテキストはそのオブジェクトの作成に使われたコンテキストのアクティビティ識別子を継承します。

1 つのアクティビティの中では 1 つの実行論理スレッドしか許されません。これは，オブジェクトが複数のプロセスにわたって分散できることを除けば，動作の点で COM アパートメントスレッドモデルに似ています。ベースクライアントがアクティビティの中を呼び出すと，そのアクティビティ内の作業に関する他のすべての要求（別のクライアントスレッドからの要求など）は，実行の初期スレッドがクライアントに戻るまでブロックされます。

MTS では，あらゆるトランザクションオブジェクトが 1 つのアクティビティに属します。COM+ では，**呼び出し同期**の設定によって，オブジェクトが複数のアクティビティに参加する方法を設定できます。選択可能なオプションは以下のとおりです。

表 44.3　呼び出し同期のオプション

オプション	説明
使用不可	COM+ はオブジェクトにアクティビティを割り当てないが，呼び出し側のコンテキストを使って継承する場合がある。呼び出し側がトランザクションまたはオブジェクトコンテキストを持たない場合，このオブジェクトはアクティビティに割り当てられない。その結果は，オブジェクトが COM+ アプリケーションにインストールされなかった場合と同じになる。アプリケーション内のオブジェクトがリソースマネージャを使用する場合，またはオブジェクトがトランザクションか即時アクティブ化をサポートする場合は，このオプションを使用してはならない
未サポート	呼び出し側のステータスに関係なく，COM+ はアクティビティにオブジェクトを割り当てない。アプリケーション内のオブジェクトがリソースマネージャを使用する場合，またはオブジェクトがトランザクションか即時アクティブ化をサポートする場合は，このオプションを使用してはならない
サポート	COM+ は呼び出し側と同じアクティビティにオブジェクトを割り当てる。呼び出し側がアクティビティに属していない場合，オブジェクトもアクティビティに属さない。アプリケーション内のオブジェクトがリソースマネージャを使用する場合，またはオブジェクトがトランザクションか即時アクティブ化をサポートする場合は，このオプションを使用してはならない
必要	COM+ は常にアクティビティにオブジェクトを割り当て，必要であればアクティビティを作成する。トランザクション属性が［Supported］または［Required］の場合は，このオプションを使用しなければならない
新しく必要	COM+ は常に新しいアクティビティにオブジェクトを割り当て，このアクティビティは呼び出し側のアクティビティとは異なる

COM+ でのイベントの生成

COM ベースの技術の多く（ActiveX スクリプティングエンジンや ActiveX コントロールなど）は，イベントシンクおよび COM の接続ポイントインターフェースを使用してイベントを生成します。イベントシンクと接続ポイントは，密結合イベントモデルの例です。このようなモデルでは，イベントを生成するアプリケーション（COM+ の用語ではパブリッシャ，以前の COM 技術ではシンク）と，

イベントに応答するアプリケーション（サブスクライバ）とが，相互を認識します。パブリッシャとサブスクライバの存続期間は一致します。両者は同時にアクティブでなければなりません。サブスクライバの集合，およびイベントが発生したときにサブスクライバに通知するメカニズムは，パブリッシャ内で保持および実装されなければなりません。

COM+ はイベントを管理するために新しいシステムを導入します。各サブスクライバの管理と通知の負担を各パブリッシャに負わせずに，基底のシステム（COM+）が間に入ってこのプロセスを引き受けます。COM+ イベントモデルは疎結合なので，パブリッシャとサブスクライバをそれぞれ独立して開発，配布，および起動できます。

COM+ イベントモデルによってパブリッシャとサブスクライバの間の通信が大幅に簡単になりましたが，両者の間に存在するようになった新しいソフトウェア層を管理するタスクも追加されました。イベントとサブスクライバについての情報は，イベントストア（COM+ カタログの一部）の中で保持されます。コンポーネントサービスなどの管理ツールは，これらの管理タスクを実行するために使用されます。コンポーネントサービスツールは完全にスクリプト可能なので，管理の大部分を自動化できます。たとえば，インストールスクリプトの実行中にこれらのタスクを行うことができます。さらに，イベントストアは，TComAdminCatalog オブジェクトを使用してプログラム的に管理できます。COM+ コンポーネントのインストールは，［実行｜ COM+ オブジェクトのインストール］を選択すれば C++Builder から直接行うこともできます。

密結合イベントモデルと同様に，イベントは単にインターフェース内のメソッドです。したがって，まずイベントメソッド用のインターフェースを作成しなければなりません。C++ Builder の COM+ イベントオブジェクトウィザードを使用すると，COM+ イベントオブジェクトが入ったプロジェクトを作成できます。次に，コンポーネントサービス管理ツール（または TComAdminCatalog，または IDE）を使用して，イベントクラスコンポーネントを格納する COM+ アプリケーションを作成します。COM+ アプリケーション内でイベントクラスコンポーネントを作成するときに，イベントオブジェクトを選択します。イベントクラスは，サブスクライバのリストにパブリッシャをバインドするために COM+ が使用する接着剤の役割を果たします。

COM+ イベントオブジェクトの興味深い点は，イベントインターフェースの実装を含まないことです。COM+ イベントオブジェクトは，パブリッシャとサブスクライバが通信のために使用するインターフェースを定義するだけです。C++Builder を使用して COM+ イベントオブジェクトを作成するときは，タイプライブラリエディタを使用してインターフェースを定義します。このインターフェースの実装は，COM+ アプリケーションとそのイベントクラスコンポーネントを作成するときに行われます。すると，COM+ が提供する実装の参照がイベントクラスに含まれ，この実装へのアクセスをイベントクラスが提供します。実行時に，パブリッシャは通常の COM メカニズム（CoCreateInstance など）を使用して，イベントクラスのインスタンスを作成します。インターフェースの COM+ 実装がこのように行われるので，パブリッシャが行わなければならないことは，（イベントクラスのインスタンスによって）インターフェース上でメソッドを呼び出して，すべてのサブスクライバに通知するイベントを生成することだけです。

メモ パブリッシャは，COM コンポーネントそのものである必要はありません。パブリッシャとは，イベントクラスのインスタンスを作成し，イベントインターフェース上でメソッドを呼び出してイベントを生成する単なるアプリケーションです。

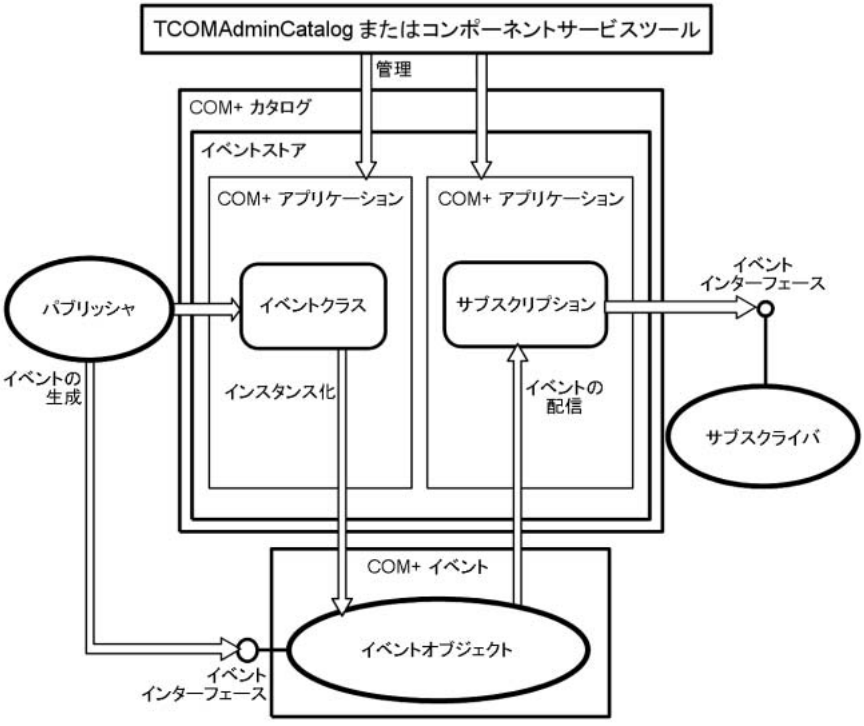
サブスクライバコンポーネントも，COM+ カタログ内にインストールしなければなりません。これも TComAdminCatalog を使用してプログラム的に行うか，IDE またはコンポーネントサービス管理ツールを使用して行うことができます。サブスクライバコンポーネントは，別個の COM+ アプリ

ケーションにインストールするか，イベントクラスコンポーネントを入れるために使用されるアプリケーションと同じアプリケーションにインストールできます。コンポーネントのインストール後に，そのコンポーネントがサポートするイベントインターフェースのそれぞれに対してサブスクリプションを作成しなければなりません。サブスクリプションの作成後に，コンポーネントに監視させたいイベントクラス（すなわちパブリッシャ）を選択します。サブスクライバコンポーネントは，個別のイベントクラスまたはすべてのイベントクラスを選択できます。

COM+ イベントオブジェクトと異なり，COM+ サブスクリプションオブジェクトには独自の実装のイベントインターフェースが含まれます。これは生成されたイベントに応答して実際の作業が行われる場所です。C++ Builder の COM+ イベントサブスクリプションオブジェクトウィザードを使用すると，サブスクライバコンポーネントを含むプロジェクトを作成できます。

パブリッシャ，サブスクライバ，および COM+ カタログの間の対話を次の図に示します。

図 44.1　COM+ イベントシステム



イベントオブジェクトウィザードの使い方

イベントオブジェクトウィザードを使ってイベントオブジェクトを作成できます。ウィザードはまず，現在のプロジェクトに実装コードが入っているかどうかをチェックします。なぜなら，COM+ イベントオブジェクトが入っているプロジェクトには実装が含まれないからです。このようなプロジェクトは，イベントオブジェクト定義しか含むことができません（ただし，1 つのプロジェクトに複数の COM+ イベントオブジェクトを含めることができます）。

イベントオブジェクトウィザードを起動する手順は次のとおりです。

1. ［ファイル｜新規作成｜その他］を選択します。
2. ［ActiveX］タブを選択します。
3. ［COM+ イベントオブジェクト］アイコンをダブルクリックします。

イベントオブジェクトウィザードでは，イベントオブジェクトの名前，イベントハンドラを定義するインターフェースの名前，および（オプションで）イベントの簡単な説明を指定します。

ウィザードを終了すると，イベントオブジェクトとそのインターフェースを定義するタイプライブラリが入ったプロジェクトが作成されます。タイプライブラリエディタを使用してそのインターフェースのメソッドを定義します。このメソッドは，イベントに応答するためにクライアントが実装するイベントハンドラです。

Event オブジェクトプロジェクトには，ATL テンプレートクラスをインポートするためにプロジェクトファイル _ATL ユニットが含まれ，タイプライブラリ情報を定義するために _TLB ユニットが含まれます。しかし，実装ユニットは含まれません。なぜなら，COM+ イベントオブジェクトが実装を持たないからです。インターフェースの実装は，クライアントの責任です。サーバーオブジェクトが COM+ イベントオブジェクトを呼び出すと，COM+ がその呼び出しをインターセプトし，登録されたクライアントにディスパッチします。COM+ イベントオブジェクトは実装オブジェクトを必要としないので，タイプライブラリエディタでオブジェクトのインターフェースを定義した後は，プロジェクトをコンパイルし，COM+ を使ってインストールするだけで済みます。

COM+ はイベントオブジェクトのインターフェースに，特定の制限事項を設けます。イベントオブジェクトに対してタイプライブラリエディタで定義するインターフェースは，以下の規則に従わなければなりません。

- イベントオブジェクトのインターフェースは，IDispatch から派生しなければならない
- イベントオブジェクトの全インターフェースにわたってすべてのメソッド名がユニークでなければならない
- イベントオブジェクトのインターフェース上の全メソッドが HRESULT 値を返さなければならない
- メソッドの全パラメータの修飾子が [in] でなければならない

COM+ イベントサブスクリプションオブジェクトウィザードの使い方

C++Builder の COM+ イベントサブスクリプションオブジェクトウィザードを使用すると，サブスクライバコンポーネントを作成できます。このウィザードを起動する手順は次のとおりです。

1. ［ファイル｜新規作成｜その他］を選択します。
2. ［ActiveX］タブを選択します。
3. ［COM+ サブスクリプションオブジェクト］アイコンをダブルクリックします。

このウィザードダイアログでは，イベントインターフェースを実装するクラスの名前を入力します。コンボボックスからスレッドモデルを選択します。［インターフェース］フィールドにイベントインターフェースの名前を入力するか，［参照］ボタンをクリックして，COM+ カタログに現在インス

トールされているすべてのイベントクラスのリストを表示します。［COM+ イベントインターフェースの選択］ダイアログにも［参照］ボタンがあります。このボタンを使用すると，イベントインターフェースが含まれたタイプライブラリを検索および選択することができます。既存のイベントクラス（またはタイプライブラリ）を選択すると，そのイベントクラスがサポートするインターフェースを自動的に実装するオプションを選択できるようになります。この［既存のインターフェースを実装する］チェックボックスにチェックマークを付けると，自動的にそのインターフェース内の各メソッドのスタブを作成します。［上位インターフェースを実装する］チェックボックスにチェックマークを付けると，継承されたインターフェースをウィザードに実装させることができます。ウィザードが実装しない上位インターフェースは，IUnknown，IDispatch，および IAppServer の 3 つです。このウィザードを完了するには，イベントサブスクライバコンポーネントの簡単な説明を入力し，［OK］をクリックします。

COM+ イベントオブジェクトを使ってのイベントの送出

イベントを送出するために，パブリッシャはまず通常の COM メカニズム（CoCreateInstance など）を使用して，イベントクラスのインスタンスを作成します。イベントクラスには独自の実装のイベントインターフェースが入るので，イベントの生成は，結果的には単にインターフェース上で適切なメソッドを呼び出すことになります。

COM+ イベントシステムはこれを引き受けます。イベントメソッドを呼び出すと，COM+ カタログ内のすべてのサブスクライバがシステムによって参照され，各サブスクライバに対する通知が行われます。サブスクライバ側では，イベントは単なるイベントメソッドの呼び出しに見えます。

パブリッシャがイベントを生成すると，サブスクライバへの通知が一度に 1 つずつ同時に行われます。通知の順序を指定することはできず，イベントが送出されるたびに通知の順序が同じになることも期待できません。イベントクラスを COM+ カタログにインストールするときに，管理者は FireInParallel オプションを選択し，複数のスレッドを使ってイベントを提供するよう要求することができます。これは同時の提供を保証するものではなく，そのようになることを許可するようにシステムに要求しているにすぎません。

パブリッシャに返される値は，各サブスクライバからのすべてのリターンコードの集合体です。どのサブスクライバが失敗したかをパブリッシャが探し出す直接的な方法はありません。それには，パブリッシャがパブリッシャフィルタを実装しなければなりません。その詳細については，Microsoft MSDN の資料を参照してください。返されるリターンコードを次の表に要約します。

表 44.4　イベントパブリッシャのリターンコード

リターンコード	説明
S_OK	すべてのサブスクライバが成功した
EVENT_S_SOME_SUBSCRIBERS_FAILED	一部のサブスクライバが呼び出せなかったか，または失敗コードを返した（これはエラー状態ではないことに注意）
EVENT_E_ALL_SUBSCRIBERS_FAILED	サブスクライバを 1 つも呼び出せなかったか，またはすべてのサブスクライバが失敗コードを返した
EVENT_S_NOSUBSCRIBERS	COM+ カタログ内にサブスクリプションがない（これはエラー状態ではないことに注意）

オブジェクト参照を渡す

メモ オブジェクト参照渡しに関する説明は，MTS だけに関係し，COM+ は該当しません。MTS では，MTS で動作しているオブジェクトを指すすべてのポインタをインターセプタ経由でルーティングする必要があるため，オブジェクト参照を渡すメカニズムが必要です。COM+ にはインターセプタが組み込まれているので，オブジェクト参照を渡す必要はありません。

MTS では，以下の方法でだけ，たとえばコールバックとして使うためにオブジェクト参照を渡すことができます。

- CoCreateInstance（またはそれに相当するインターフェース），ITransactionContext::CreateInstance，IObjectContext::CreateInstance などのオブジェクト作成インターフェースからの復帰を通じて渡す
- QueryInterface の呼び出しを通じて渡す
- オブジェクト参照を取得するために SafeRef を呼び出したメソッドを通じて渡す

上記の方法で取得されたオブジェクト参照は**セーフ参照**と呼ばれます。セーフ参照を使って呼び出されたメソッドは，正しいコンテキストの中で実行されることが保証されます。

MTS 実行時環境は，コンテキストの切り替えを管理できるようにするためにセーフ参照を使用する呼び出しを必要とし，クライアント参照に依存しない存続期間をトランザクションオブジェクトが持てるようにします。セーフ参照は COM+ では必要ありません。

SafeRef メソッドの使い方

オブジェクトは SafeRef 関数を使って，コンテキストの外へ安全に渡せる自分自身への参照を取得することができます。この関数は TMtsDll オブジェクトのメソッドとして使用でき，これはサーバーが MTS または COM+ のどちらで実行しているかを調べ，それに応じて適切なポインタを返します。

SafeRef は入力として以下のものを取ります。

- 現在のオブジェクトが別のオブジェクトまたはクライアントに渡したいインターフェースのインターフェース ID（RIID）への参照
- 現在のオブジェクトの IUnknown インターフェースへの参照

SafeRef は，現在のオブジェクトのコンテキストの外に渡しても安全な，RIID パラメータで指定されたインターフェースへのポインタを返します。オブジェクトが自分以外のオブジェクトへのセーフ参照を要求している場合や，RIID パラメータに指定されたインターフェースが実装されていない場合には，SafeRef は NULL を返します。

MTS オブジェクトが自己参照をクライアントか別のオブジェクトに渡したい場合（たとえばコールバックとして使うために），そのオブジェクトは最初に SafeRef を呼び出して，その呼び出しで返された参照を渡さなければなりません。オブジェクトは，**self** ポインタ，つまり QueryInterface への内部呼び出しによって取得した自己参照を，クライアントや他のオブジェクトに渡してはなりません。このような参照がオブジェクトのコンテキストの外へ渡されると，それは有効な参照ではなくなります。

すでに安全である参照について SafeRef を呼び出すと，インターフェースの参照カウントが増分されるだけで，そのセーフ参照が変更されずにそのまま返されます。

クライアントがセーフ参照について QueryInterface を呼び出すと，クライアントに返される参照もセーフ参照になります。

セーフ参照を取得したオブジェクトは，用が済んだらそのセーフ参照を解放しなければなりません。

SafeRef についての詳細は，Microsoft の資料の SafeRef に関するトピックを参照してください。

コールバック

オブジェクトは，クライアントや他のトランザクションオブジェクトに対してコールバックを行うことができます。たとえば，オブジェクトに別のオブジェクトを作成させることができます。作成する側のオブジェクトは，作成されるオブジェクトに自分自身への参照を渡し，作成されたオブジェクトはその参照を使って自分を作成したオブジェクトを呼び出すことができます。

コールバックを使う場合は，以下の制限事項に注意してください。

- ベースクライアントや別のパッケージに対するコールバックには，クライアント上でのアクセスレベルのセキュリティが必要である。さらに，クライアントは DCOM サーバーでなければならない
- ファイアウォールが介在していると，クライアントへのコールバックがブロックされることがある
- コールバックに対して行われる作業は，呼び出されたオブジェクトの環境内で実行される。それは同じトランザクションの一部である場合も，別のトランザクションである場合も，トランザクションではない場合もある
- MTS では，作成する側のオブジェクトは，自分自身をコールバックするために，SafeRef を呼び出して，返された参照を作成される側のオブジェクトに渡さなければならない

トランザクションオブジェクトのデバッグとテスト

ローカルのトランザクションオブジェクトもリモートのトランザクションオブジェクトもデバッグできます。トランザクションオブジェクトをデバッグするときには，トランザクションタイムアウトをオフにすると便利です。

トランザクションタイムアウトは，トランザクションがアクティブでいられる時間の長さ（秒数）を設定します。タイムアウト後もアクティブのままになっているトランザクションは，システムによって自動的にアボートされます。デフォルトのタイムアウト値は 60 秒です。値として 0 を指定すると，トランザクションのタイムアウトを無効にすることができます。これは，デバッグするときに役立ちます。

リモートデバッグについての詳細は，オンラインヘルプのリモートデバッグに関するトピックを参照してください。

MTS で実行しようとしているトランザクションオブジェクトをテストするときには，テスト環境を単純化するために，最初に MTS 環境の外でオブジェクトをテストするとよいでしょう。

サーバーを開発する場合，サーバーがまだメモリの中にあるときにサーバーを再構築することはできません。そうしようとすると，「実行可能ファイルがロードされているので DLL に書き込めない」のようなコンパイルエラーメッセージが表示されます。これを回避するには，サーバーがアイドル状態

のときにシャットダウンされるように，MTS パッケージまたは COM+ アプリケーションのプロパティを設定します。

サーバーがアイドル状態のときにシャットダウンされるように設定する手順は次のとおりです。

1. MTS エクスプローラまたはコンポーネントサービスで，目的のトランザクションオブジェクトがインストールされている MTS パッケージまたは COM+ アプリケーションを右クリックして，［プロパティ］を選択します。
2. ［詳細設定］タブを選択します。

   ［詳細設定］タブは，パッケージに関連付けられているサーバープロセスが常に動作し続けるか，一定の時間が経過するとシャットダウンするかを決定します。
3. タイムアウト値を 0 に変更します。これで，サービス対象のクライアントがなくなるとすぐにサーバーがシャットダウンするようになります。
4. ［OK］をクリックして設定を保存します。

トランザクションオブジェクトのインストール

MTS アプリケーションは，MTS 実行形式（EXE）の単一のインスタンスの中で動作するインプロセス MTS オブジェクトのグループで構成されます。すべて同じプロセスの中で動作する COM オブジェクトのグループを**パッケージ**と呼びます。単一のマシンが複数の異なるパッケージを実行することもできます。その場合，各パッケージはそれぞれ別個の MTS EXE の中で実行されます。

COM+ では，パッケージに似たグループを使用し，これを COM+ アプリケーションと呼んでいます。**COM+ アプリケーション**では，オブジェクトがインプロセスである必要はなく，別個の実行時環境はありません。

アプリケーションコンポーネントを 1 つの MTS パッケージまたは COM+ アプリケーションにまとめて，単一のプロセスによって管理するようにできます。アプリケーションを複数のプロセスまたはマシンに分けるために，コンポーネントを複数の MTS パッケージまたは COM+ アプリケーションに分散させることもできます。

トランザクションオブジェクトを MTS パッケージまたは COM+ アプリケーションにインストールする手順は次のとおりです。

1. システムが COM+ をサポートする場合は，［実行｜ COM+ オブジェクトのインストール］を選択します。システムが COM+ をサポートせず，システムに MTS をインストールしてある場合は，［実行｜ MTS オブジェクトのインストール］を選択します。システムが MTS も COM+ もサポートしない場合は，トランザクションオブジェクトをインストールするメニュー項目が表示されません。
2. ［オブジェクトのインストール］ダイアログボックスで，インストールしたいオブジェクトにチェックマークを付けます。
3. MTS オブジェクトをインストールする場合は，［パッケージ］ボタンを選択して，システム上の MTS パッケージのリストを表示します。COM+ オブジェクトをインストールする場合は，［アプリケーション］ボタンを選択します。オブジェクトをインストールする MTS パッケージまたは

COM+ アプリケーションを示します。［新規パッケージに追加］または［新規アプリケーションに追加］を選択すると，オブジェクトをインストールする新しい MTS パッケージまたは COM+ アプリケーションを作成できます。［既存のパッケージに追加］または［既存のアプリケーションに追加］を選択すると，リスト内の既存の MTS パッケージまたは COM+ アプリケーションにオブジェクトをインストールできます。

4. ［OK］を選択するとカタログが更新され，インストールしたオブジェクトを実行時に使えるようになります。

MTS パッケージには複数の DLL からのコンポーネントを収めることができ，1 つの DLL からの複数のコンポーネントを複数のパッケージにインストールすることができます。ただし，1 つのコンポーネントを複数のパッケージに分散させることはできません。

同様に，COM+ アプリケーションには複数の実行可能ファイルからのコンポーネントを収めることができ，1 つの実行可能ファイルからの複数のコンポーネントを複数の COM+ アプリケーションにインストールすることができます。

メモ コンポーネントサービスまたは MTS エクスプローラを使ってもトランザクションオブジェクトをインストールできます。これらのツールのいずれかを使ってオブジェクトをインストールする場合は，タイプライブラリエディタの［COM+］ページに表示されるオブジェクトの設定を適用するようにしてください。IDE からインストールしない場合は，この設定は自動的には適用されません。

トランザクションオブジェクトの管理

いったんトランザクションオブジェクトをインストールすると，（MTS パッケージにインストールした場合は）MTS エクスプローラ，または（COM+ アプリケーションにインストールした場合は）コンポーネントサービスを使って，それらの実行時オブジェクトを管理できるようになります。両ツールは同じものですが，MTS エクスプローラは MTS 実行時環境で動作し，コンポーネントサービスは COM+ オブジェクト上で動作します。

コンポーネントサービスと MTS エクスプローラは，トランザクションオブジェクトの管理と配布を行うためのグラフィカルユーザーインターフェースを持っています。これらのツールのいずれかを使うと，以下の操作ができます。

- トランザクションオブジェクト，MTS パッケージまたは COM+ アプリケーション，およびロールを環境設定する
- パッケージまたは COM+ アプリケーション内のコンポーネントのプロパティと，コンピュータにインストールされた MTS パッケージまたは COM+ アプリケーションを表示する
- トランザクションを構成するオブジェクトについて，トランザクションをモニタして管理する
- MTS パッケージまたは COM+ アプリケーションをコンピュータ間で移動する
- リモートトランザクションオブジェクトをローカルクライアントが使用できるようにする

詳細については，Microsoft 発行の適切な管理者ガイドを参照してください。

第 V 部

カスタムコンポーネントの作成

第 V 部「カスタムコンポーネントの作成」の各章では，C++Builder でカスタムコンポーネントを設計し実装するために必要な概念およびテクニックについて説明します。

第45章

コンポーネント作成の概要

この章では，C++Builder アプリケーション用コンポーネントの設計と開発プロセスの概要について説明します。ここでは，C++Builder とその標準コンポーネントについてのある程度の知識を前提として記述されています。この章では以下のことについて説明します。

- クラスライブラリ
- コンポーネントとクラス
- コンポーネントの作成
- コンポーネントのさまざまな側面
- 新しいコンポーネントの作成
- インストール前のコンポーネントのテスト
- インストールしたコンポーネントのテスト
- コンポーネントパレットへのコンポーネントのインストール

新しいコンポーネントのインストールについては，15-5 ページの「コンポーネントパッケージのインストール」を参照してください。

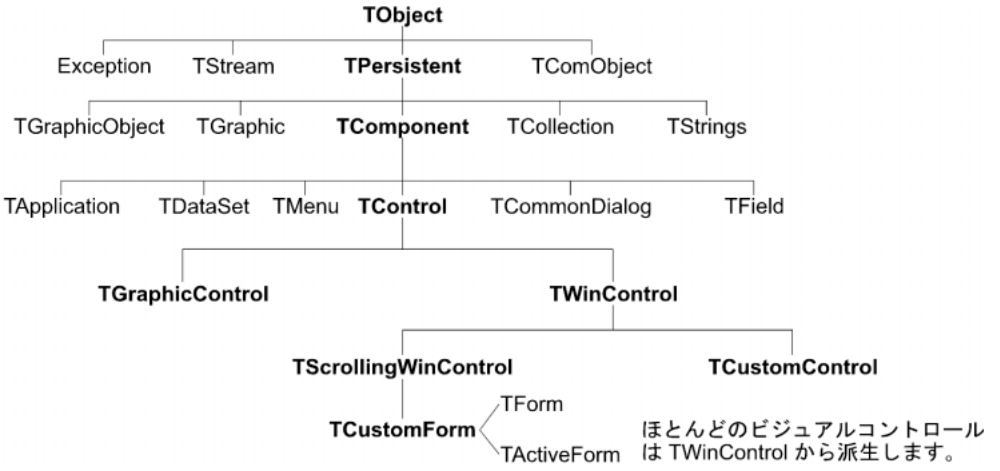
クラスライブラリ

C++Builder のコンポーネント群を構成するクラス階層は 2 つあります。1 つは VCL（ビジュアルコンポーネントライブラリ），もう 1 つは CLX（クロスプラットフォーム用コンポーネントライブラリ）です。図 45.1 に，VCL を構成するクラス間の関係を示します。CLX の階層は VCL の階層と似ていますが，Windows コントロールを「ウィジェット」と呼ぶ（よって TWinControl を TWidgetControl と呼ぶ）など，いくつか違いがあります。クラス階層とクラス間の継承関係についての詳細は，第 46 章「コンポーネント開発者のためのオブジェクト指向プログラミング」を参照してください。VCL と CLX の相違点については，14-5 ページの「CLX と VCL」を参照してください。CLX のコンポーネントに関する詳細は，CLX オンラインリファレンスを参照してください。

VCL と CLX では TComponent クラスがすべてのコンポーネントに共通の上位クラスになっています。TComponent は，コンポーネントが C++Builder で動作するために最小限必要なプロパティとイベ

ントを定義します。そして，このライブラリのさまざまな下位クラスが，それ以外のより特化した機能を実装します。

図 45.1　ビジュアルコンポーネントライブラリのクラス階層



コンポーネントを作成する場合は，このクラス階層内の既存のクラス型から新しいクラスを派生させます。

コンポーネントとクラス

コンポーネントはクラスであり，コンポーネント開発者はアプリケーション開発者とは異なるレベルでオブジェクトを使用します。新しいコンポーネントを作成するには，新しいクラスを派生させる必要があります。

要約すると，コンポーネントの作成とアプリケーションでのコンポーネントの使用との間には，次の 2 つの大きな相違点があります。コンポーネントを開発するときには，

- コンポーネント開発者は，アプリケーションプログラマがアクセスできないクラスの各部分にもアクセスする
- コンポーネント開発者は，コンポーネントに（プロパティなどの）新しい部分を追加する

このため，コンポーネント開発者はコンポーネントについてより詳しく理解しておく必要があります。また，開発したコンポーネントをアプリケーション開発者がどのように使用するのかということについても考慮しなければなりません。

コンポーネントの作成

開発者が設計時に扱うプログラム要素のほとんどはコンポーネントです。コンポーネントを作成するには，既存のクラスから新しいクラスを派生させます。新しいコンポーネントを作成するには，以下の方法があります。

- 既存のコントロールの変更
- ウィンドウコントロールの作成
- グラフィックコントロールの作成
- Windows コントロールのサブクラスの作成
- 非ビジュアルコンポーネントの作成

表 45.1 に，コンポーネントの各作成方法で，派生元として使用するクラスを要約します。

表 45.1　コンポーネントの派生元

種類	派生元となるオブジェクト型
既存のコンポーネントの変更	TButton や TListBox などの既存のコンポーネントか，TCustomListBox などの抽象コンポーネントクラス
ウィンドウコントロール（CLX ではウィジェットベースのコントロール）の作成	TWinControl（CLX では TWidgetControl）
グラフィックコントロールの作成	TGraphicControl
コントロールのサブクラスの作成	ウィンドウコントロール（VCL）またはウィジェットベースのコントロール（CLX）
非ビジュアルコンポーネントの作成	TComponent

また，TRegIniFile や TFont など，コンポーネントではないクラスを派生させることもできますが，そうしたクラスはフォームの中では操作できません。

既存のコントロールの変更

コンポーネントを作成する最も簡単な方法は，既存のコンポーネントをカスタマイズする方法です。C++Builder が提供するどのコンポーネントからでも，新しいコンポーネントを派生させることができます。

リストボックスやグリッドのようなコントロールには，共通の基本的な機能を持ついくつかのバリエーションがあります。VCL と CLX にはこれらに対応した抽象クラスがあります。抽象クラスは，新しくカスタマイズしたコンポーネントを派生させる元として使用できます。なお，VCL の抽象クラスの名前には，TCustomGrid などのように「Custom」という文字列が含まれます。

たとえば，標準の TListBox クラスのプロパティからいくつかを除いて，特別なリストボックスを作成するとします。上位クラスから継承したプロパティを除去する（隠蔽する）ことはできないので，TListBox より上位の階層のクラスからそのコンポーネントを派生させます。この場合，TWinControl（CLX の場合は TWidgetControl）クラスまで戻ってしまうと，リストボックスの機能を再びすべて作成しなければなりません。そのため，ＶＣＬと CLX には，TCustomListBox が提供されています。この TCustomListBox には，リストボックスのためのプロパティがあります。ただし，そのすべてがパブリッシュに設定されているわけではありません。TCustomListBox などのような抽象クラスからコンポーネントを派生させる場合には，開発するコンポーネントで使用するプロパティだけをパブリッシュに設定し，そのほかのプロパティはプロテクト設定のまま残します。

第 47 章「プロパティの作成」では，継承プロパティをパブリッシュに設定する方法を説明しています。第 53 章「既存のコンポーネントの変更」および第 55 章「グリッドのカスタマイズ」には，既存のコントロールの変更例があります。

ウィンドウコントロールの作成

VCL と CLX のウィンドウコントロールは実行時に表示され，ユーザーが対話できるオブジェクトです。各ウィンドウコントロールにはウィンドウハンドルがあり，ウィンドウハンドルには Handle プロパティを通じてアクセスします。ウィンドウハンドルにより，オペレーティングシステム はそのコントロールを識別して操作できます。VCL コントールを使う場合，ウィンドウハンドルがあるためコントロールは入力フォーカスを受け取ることができます。また，ウィンドウハンドルは Windows API 関数に渡すことができます。CLX では，ウィンドウコントールのことを「ウィジェットベースのコントロール」といいます。各ウィジェットベースのコントロールにはハンドルがあり，ハンドルには Handle プロパティを通じてアクセスします。基になるウィジェットがハンドルにより識別されます。

すべてのウィンドウコントロールは TWinControl クラス（CLX では TWidgetControl クラス）から派生します。ウィンドウコントロールにはプッシュボタン，リストボックス，編集ボックスなど，ほとんどの標準ウィンドウコントロールが含まれます。TWinControl（CLX の場合は TWidgetControl）からオリジナルのコントロール（既存のコントロールに関連していないコントロール）を直接派生させることができますが，このような派生のために C++Builder では TCustomControl コンポーネントが用意されています。TCustomControl は，複雑なビジュアルイメージを描画しやすくする専用のコントロールです。

ウィンドウコントロールの作成例については，第 55 章「グリッドのカスタマイズ」を参照してください。

グラフィックコントロールの作成

コントロールが入力フォーカスを受け取る必要がない場合は，そのコントロールをグラフィックコントロールにできます。グラフィックコントロールはウィンドウコントロールに似ていますが，ウィンドウハンドルを持たないので，システムリソースを節約できます。TLabel のような入力フォーカスを受け取らないコンポーネントがグラフィックコントロールです。これらのグラフィックコントロールがフォーカスを受け取らないようにしても，マウスメッセージに反応するように設計することができます。

C++Builder では，グラフィックコントロールの作成のために TGraphicControl コンポーネントを用意しています。TGraphicControl は，TControl から派生した抽象クラスです。作成するコントロールを TControl から直接派生させることもできますが，TGraphicControl から派生させる方がはるかに簡単です。なぜなら，TGraphicControl はペイント用のキャンバスを用意し，Windows 上で WM_PAINT メッセージを処理するからです。開発者に必要な作業は，Paint メソッドをオーバーライドすることだけです。

グラフィックコントロールの作成例については，第 54 章「グラフィックコントロールの作成」を参照してください。

Windows コントロールのサブクラスの作成

従来の Windows プログラミングでカスタムコントロールを作成するには，新しいウィンドウクラスを定義して Windows に登録します。ウィンドウクラスはオブジェクト指向プログラミングのオブジェクトやクラスと似ており，同一種類のコントロールのインスタンス間で共有される情報を含んでいます。新しいウィンドウクラスは既存のウィンドウクラスに基づいて作成できます。この方法をサブクラス化といいます。

C++Builder では，既存のウィンドウクラスの周りにコンポーネントの「ラッパー」を作成できます。このため，C++Builder アプリケーションで使用するカスタムコントロールのライブラリがすでに存在している場合，ほかのコンポーネントと同様に，独自のコントロールのように動作する C++Builder コンポーネントを作成でき，そこから新しいコントロールを派生させることができます。

Windows コントロールをサブクラス化する技法の例については，標準の Windows コントロールを表す StdCtls ヘッダーファイルのコンポーネント（TEdit など）を参照してください。CLX の例については，QStdCtls を参照してください。

非ビジュアルコンポーネントの作成

非ビジュアルコンポーネントは，データベース（TDataSet，TSQLConnection）やシステムタイマー（TTimer）などの要素のインターフェースとして，またダイアログボックス（VCL の TCommonDialog，CLX の TDialog）とその下位コンポーネントのプレースホルダとして使われます。プログラマが開発するコンポーネントのほとんどはビジュアルコントロールでしょう。非ビジュアルコンポーネントは，すべてのコンポーネントの抽象基本クラスである TComponent から直接派生させることができます。

コンポーネントのさまざまな側面

コンポーネントを C++Builder 環境における信頼性の高い要素にするために，コンポーネントの設計時にいくつかの規則に従う必要があります。この節では，以下の事項について説明します。

- 依存関係の除去
- プロパティ，メソッド，イベントの設定
- グラフィックのカプセル化
- コンポーネントの登録

依存関係の除去

コンポーネントの有用性を高める 1 つの特性は，コンポーネントがコード内の任意の場所で実行できる処理への制限がないことです。コンポーネントは性格上，さまざまな組み合わせ，順序，状況において，アプリケーションに組み入れられます。前処理を必要とせず，あらゆる状況で機能するようなコンポーネントを設計しなければなりません。

依存関係除去の好例として，TWinControl の Handle プロパティがあります。Windows アプリケーションを作成したことがあればわかるように，CreateWindow という API 関数を呼び出してウィンドウコントロールを作成する前に，そのウィンドウコントロールにアクセスすることはできません。これは，プログラムを実行する際に最もエラーを起こしやすい点の 1 つです。C++Builder ウィンドウコントロールでは，ユーザーはこの点について考慮する必要はありません。ウィンドウハンドルが必要なときは常にそのハンドルが有効だからです。コントロールは，ウィンドウハンドルを表すプロパティを参照すれば，そのウィンドウが作成済みであるかどうかを検査できます。ハンドルが有効でない場合は，コントロールはウィンドウを作成してハンドルを返します。こうして，アプリケーションのコードが Handle プロパティをアクセスしたときは，常に有効なハンドルを得られることが保証されます。

C++Builder コンポーネントでは，ウィンドウの作成などの処理がバックグラウンドで実行されるため，開発者は本来の作業に専念できます。ウィンドウハンドルを API 関数に渡す前に，あらかじめハンドルが存在するかどうかを確認したり，ウィンドウを作成するというような作業は必要ありません。アプリケーション開発者は，誤った状態にならないように絶えずチェックする必要はなく，意図したとおりに処理が行われることを確信できます。

依存関係を持たないコンポーネントを作成するのは時間がかかるかもしれませんが，そのような時間は一般に有意義なものです。なぜなら，それによってアプリケーション開発者が単調な作業を繰り返し行う必要がなくなるだけでなく，マニュアルの作成やサポートにかかる負担が減少するからです。

プロパティ，メソッド，イベントの設定

フォームデザイナで操作する，目に見えるイメージを除いて，コンポーネントの明白な要素は，プロパティ，イベント，メソッドです。このマニュアルではそれぞれに独立の章が割り当てられています。ここでは，この 3 つの要素の使い方について説明します。

プロパティ

アプリケーション開発者は，値の設定や読み出しができる変数のようなつもりでプロパティを扱います。コンポーネント開発者はプロパティを使用することにより，実際のデータ構造を隠蔽したり，値のアクセスに伴う特殊な処理を定義できます。

プロパティの使用によって得られる利点は以下のとおりです。

- 設計時にプロパティを使用できる。これにより，アプリケーション開発者はコードを記述せずに，プロパティの初期値を設定，変更できる
- プロパティが変更されるたびに，値や形式を検査できる。設計時に入力を検証することで，エラーを防止できる
- 必要なときにコンポーネントが適切な値を構築できる。よくあるエラーは，初期化されていない変数を参照すること。データをプロパティにすれば，必要なときに値が使用可能であることを保証できる
- プロパティを使うと，単純で一定したインターフェースのもとでデータを隠蔽できる。プロパティでデータを扱うと，データの内部構造が変わってもコンポーネントを使う開発者とのインターフェースを変えずに済む

コンポーネントにプロパティを追加する方法については，第 47 章「プロパティの作成」を参照してください。

イベント

イベントとは，実行時の入力やその他の動作に応答してコードを実行する特殊なプロパティです。イベントは，アプリケーション開発者に対し，コードの特定のブロックを実行時の特定の動作（マウス操作やキー入力など）に結び付ける方法を提供します。イベントが発生すると実行されるコードをイベントハンドラといいます。

イベントを使うと，アプリケーション開発者は新しいコンポーネントを定義しなくても，さまざまな入力に対する応答を指定できます。

第 48 章「イベントの作成」では，標準のイベントを実装したり，新しいイベントを追加する方法について説明しています。

メソッド

クラスメソッドは，クラスの特定のインスタンスではなくクラスに作用する関数です。たとえば，あらゆるコンポーネントのコンストラクタメソッドはクラスメソッドです。コンポーネントメソッドは，コンポーネントのインスタンスそのものに作用する関数です。アプリケーション開発者はメソッドを使って，コンポーネントが特定の動作を実行するよう指示したり，どのプロパティにもない値を取得したりします。

メソッドはコードの実行を必要とするので，呼び出せるのは実行時だけです。メソッドは以下のような点で役に立ちます。

- メソッドは，コンポーネントの機能をデータが存在するのと同じオブジェクトにカプセル化する
- メソッドは，一定した単純なインターフェースのもとで複雑な手続きを隠蔽できる。アプリケーション開発者はあるコンポーネントの AlignControls メソッドを，その機能やほかのコンポーネントの AlignControls メソッドとの相違を知らなくても呼び出せる
- メソッドを使うと，複数のプロパティを単一の呼び出しで更新できる

第 49 章「メソッドの作成」では，コンポーネントにメソッドを追加する方法について説明しています。

グラフィックのカプセル化

C++Builder では，多様なグラフィックツールをキャンバス内にカプセル化することにより，Windows のグラフィックを簡単にします。キャンバスはウィンドウやコントロールの描画面で，ペン，ブラシ，フォントなどのほかのクラスも含んでいます。キャンバスは Windows のデバイスコンテキストと似ていますが，Windows のデバイスコンテキストと違う点は，キャンバスでは後始末が自動的に行われるということです。

Windows グラフィックアプリケーションを開発したことがあれば，Windows のグラフィックデバイスインターフェース（GDI）が課す規則についてご存知でしょう。規則にはたとえば，利用可能なデバイスコンテキストの数の制限や，グラフィックオブジェクトを破棄する前には，オブジェクトを初期の状態に戻さなければならないことなどがあります。

C++Builder では，そうした規則にわずらわされることはありません。フォームやほかのコンポーネントに描画する場合には，コンポーネントの Canvas プロパティを使用します。ペンやブラシをカスタマイズする場合には，色やスタイルを設定します。設定が終了すると，リソースの破棄は C++Builder が自動的に行います。アプリケーションが同じ種類のリソースを頻繁に使用する場合は，C++Builder はリソースが再生成されないようにリソースをキャッシュに入れておきます。

もちろん今までのように Windows GDI にも全面的にアクセスできます。しかし，C++Builder コンポーネントのキャンバスを使用する方が，コードが簡単になり実行速度も上がります。グラフィック機能についての詳細は，第 50 章「コンポーネントにおけるグラフィックの使い方」を参照してください。

CLX は違った方法でグラフィックをカプセル化します。CLX ではペインタがキャンバスです。フォームやほかのコンポーネントに描画する場合には，コンポーネントの Canvas プロパティを使用します。Canvas は，プロパティであると同時に，TCanvas というオブジェクトでもあります。TCanvas は，Handle プロパティによってアクセスできる Qt ペインタのラッパーです。このハンドルを使用すると，低レベルの Qt グラフィックライブラリ関数にアクセスできます。

ペンやブラシをカスタマイズする場合には，色やスタイルを設定します。設定が終了すると，リソースの破棄は C++Builder が自動的に行います。CLX もリソースをキャッシュに入れます。

CLX コンポーネントを継承することによって，CLX コンポーネントに組み込まれたキャンバスを使用することができます。グラフィックイメージがコンポーネント内でどのように機能するかは，コンポーネントが継承するオブジェクトのキャンバスに応じて異なります。

コンポーネントの登録

コンポーネントを C++Builder の IDE にインストールするには，あらかじめそのコンポーネントを登録する必要があります。登録とは，コンポーネントパレット上にそのコンポーネントを配置する位置を C++Builder に指示することです。また，C++Builder がコンポーネントをフォームファイルに格納する方法もカスタマイズできます。新しいコンポーネントの登録についての詳細は，第 52 章「コンポーネントを設計時に利用できるようにする」を参照してください。

新しいコンポーネントの作成

新しいコンポーネントを作成する方法には，以下の 2 つがあります。

- コンポーネントウィザードによるコンポーネントの作成
- コンポーネントを手作業で作成する

いずれかの方法で，最小限の機能を持つコンポーネントが作成されます。作成されたコンポーネントは，コンポーネントパレットにインストールできます。インストール後，新しいコンポーネントをフォームに追加して，設計時と実行時の両方についてテストできます。そのコンポーネントにさらに多くの機能を追加して，コンポーネントパレットを更新し，再びテストすることもできます。

新しいコンポーネントを作成する場合に必要な基本手順がいくつかあります。以下に手順を示します。このマニュアルのほかの例はすべて，この手順についての知識を前提にして記述されています。

1. 新しいコンポーネント用のユニットを作成します。

2. 既存のコンポーネント型からコンポーネントを派生させます。
3. プロパティ，メソッド，イベントを追加します。
4. C++Builder にコンポーネントを登録します。
5. 開発したコンポーネントと，そのコンポーネントのプロパティ，メソッド，イベントについてヘルプファイルを作成します。

メモ コンポーネントの使用方法をユーザーに説明するヘルプファイルをオプションで作成できます。

6. コンポーネントを C++Builder IDE にインストールできるように，パッケージ（特殊なダイナミックリンクライブラリ）を作成する

作成プロセスが終了したとき，完成したコンポーネントには以下のファイルが含まれています。

- パッケージファイル（.BPL）またはパッケージコレクションファイル（.PCE）
- パッケージのライブラリ（.LIB）
- パッケージの Borland インポートライブラリファイル（.BPI）
- コンパイル済みユニットファイル（.OBJ）
- パレットマップのコンパイル済みリソースファイル（.RES）
- ヘルプファイル（.HLP）

新しいコンポーネントを表すビットマップを作成することもできます。45-15 ページの「コンポーネント用のビットマップの作成」を参照してください。

第 V 部の各章では，コンポーネントの構築に関するすべての側面について説明し，各種コンポーネントを作成する例を示します。

コンポーネントウィザードによるコンポーネントの作成

コンポーネントウィザードは，コンポーネントの作成の初期段階を簡略化します。コンポーネントウィザードを使う場合，以下の項目を指定する必要があります。

- コンポーネントの派生元のクラス
- 新しいコンポーネントのクラス名
- コンポーネントを表示するコンポーネントパレットのページ
- コンポーネントを作成するユニット名
- ユニットがある検索パス
- コンポーネントを入れるパッケージ名

コンポーネントウィザードは，コンポーネントを手作業で作成する場合と同様に，以下の 3 つの作業を行います。

- ユニット（.CPP ファイルと対応するヘッダーファイル）の作成 .
- コンポーネントの派生
- 新しいコンストラクタの宣言
- コンポーネントの登録

ただし，コンポーネントウィザードでは，既存のユニット（.CPP ファイルと対応するヘッダーファイルから構成）にコンポーネントを追加することはできません。新しいコンポーネントを追加する場合は，手作業でユニットに追加する必要があります。

1. コンポーネントウィザードを開始するには，以下のどちらかを行います。
   - ［コンポーネント｜コンポーネントの新規作成］を選択する
   - ［ファイル｜新規作成｜その他］を選択し，［新規作成］ページで［コンポーネント］アイコンをダブルクリックする

図 45.2 コンポーネントウィザード

コンポーネントウィザードで各項目に値を入力します。

2. ［上位クラス］項目で，新しいコンポーネントの派生元のクラスを指定します。

メモ ドロップダウンリストに表示されるコンポーネントの多くは，VCL と CLX の 2 通りのユニット名が表示されます。CLX 固有のユニット名は Q で始まります（たとえば，Graphics でなく QGraphics になる）。派生元のコンポーネントを間違えないように注意してください。

3. ［クラス名］項目で，新しいコンポーネントクラスの名前を指定します。
4. ［パレットページ名］項目で，新しいコンポーネントをインストールするコンポーネントパレットのページを指定します。
5. ［ユニットファイル名］項目に，コンポーネントクラスを宣言するユニットの名前を指定します。
6. ユニットが検索パスにない場合，［検索パス］項目の検索パスを必要に応じて編集します。
7. ［インストール］ボタンを押すと，作成したコンポーネントのインストール先パッケージを指定するためのダイアログが開きます。［OK］ボタンを押したときは，コンポーネントのソースコードが作成されるだけです。パッケージにインストールする場合にはメニューから［コンポーネント｜インストール］を選択しパッケージを選びます。

注意 名前が「Custom」で始まる VCL クラスまたは CLX クラス（TCustomControl など）からコンポーネントを派生させる場合，オリジナルのコンポーネントで抽象メソッドをオーバーライドするまで，新しいコンポーネントをフォームに配置しないでください。C++Builder では，抽象プロパティまたは抽象メソッドを含むクラスのインスタンスオブジェクトは作成できません。

8. コンポーネントウィザードで各項目に値を入力した後［OK］を選択してください。.cpp ファイルとそれに対応するヘッダーファイルから構成される新しいユニットが作成されます。

.cpp ファイルの内容がコードエディタに表示されます。このファイルにはコンポーネントのコンストラクタとコンポーネントを登録する Register 関数が含まれています。Register 関数は，コンポーネントライブラリに追加するコンポーネントの名前と，そのコンポーネントをコンポーネントパレットのどのページに置くかを指定して，コンポーネントを登録します。冒頭の include 文では，作成されたヘッダーファイルが指定されます。例を示します。

```
#include <vcl.h>
#pragma hdrstop
#include "NewComponent.h"
#pragma package(smart_init);
//---------------------------------------------------------------------------
// ValidCtrCheck を使用して，作成済みコンポーネントに
// 純粋仮想関数がないことを確認
//

static inline void ValidCtrCheck(TNewComponent *)
{
        new TNewComponent(NULL);
}
//---------------------------------------------------------------------------
__fastcall TNewComponent::TNewComponent(TComponent* Owner)
   : TComponent(Owner)
{
}
//---------------------------------------------------------------------------
namespace Newcomponent
{
   void __fastcall PACKAGE Register()
   {
      TComponentClass classes[1] = {__classid(TNewComponent)};
      RegisterComponents("Samples", classes, 0); //CLX では Samples とは別のページを使用
   }
}
```

CLX　CLX アプリケーションでは，ヘッダーファイルの名前と場所が異なります。たとえば，<vcl \ controls.hpp> は CLX では <clx \ qcontrols.hpp> です。

コードエディタ内でヘッダーファイルを開くには，ヘッダーファイル名の部分にカーソルを置き，マウスを右クリックし，コンテキストメニューから［カーソル位置のファイルを開く］を選択します。

ヘッダーファイルには，コンストラクタ宣言などの新しいクラス宣言とそのクラスをサポートする **#include** 文が複数含まれます。たとえば次のようにします。

```
#ifndef NewComponentH
#define NewComponentH
//---------------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
class PACKAGE TNewComponent : public TComponent
{
private:
protected:
public:
```

```
    __fastcall TNewComponent(TComponent* Owner);
__published:
};
//---------------------------------------------------------------------------
#endif
```

次の作業に進む前に .cpp ファイルに識別しやすい名前を付けて保存します。

コンポーネントを手作業で作成する

新しいコンポーネントを作成する最も簡単な方法は，コンポーネントウィザードを使用することです。同じ作業を手作業でも行えます。

コンポーネントを手作業で作成するには，以下の 3 つの段階があります。

1. ユニットファイルの作成
2. コンポーネントの派生
3. 新しいコンストラクタの宣言
4. コンポーネントの登録

ユニットファイルの作成

C++Builder のユニットは，.CPP ファイルと，.H ファイルから構成されます。コンパイル時には，この 2 つのファイルをもとに .OBJ ファイルが作成されます。C++Builder では，ユニットをさまざまな目的で使用します。すべてのフォームと，大半のコンポーネント（またはその論理グループ）がそれぞれ独自のユニットを持っています。

コンポーネントの作成に当たっては，そのコンポーネント用の新しいユニットを作成するか，または既存のユニットにその新しいコンポーネントを追加します。

1. コンポーネントのユニットを作成するには次のどちらかの操作を行います。
   - ［ファイル｜新規作成｜ユニット］を選択します。
   - ［ファイル｜新規作成｜その他］を選択して［新規作成］ダイアログボックスを表示し，［ユニット］を選択して［OK］をクリックします。

   .CPP ファイルとヘッダーファイルが作成され，コードエディタに .CPP ファイルの内容が表示されます。ここでは，識別しやすい名前を付けて保存します。
2. ヘッダーファイルを開くには，コードエディタでヘッダーファイル名の部分にカーソルを置き，右クリックしてポップアップメニューから［カーソル位置のファイルを開く］を選択します。
3. 既存のユニットを開くには，［ファイル｜開く］を選択して，コンポーネントを追加したいソースコードユニットを選びます。

メモ 既存のユニットにコンポーネントを追加する場合には，ユニットにはコンポーネントコードだけが入っていることを確認してください。たとえば，フォームが入っているユニットにコンポーネントコードを追加すると，コンポーネントパレットでエラーが発生します。

4. コンポーネント用の新しいユニットか既存のユニットがあれば，コンポーネントクラスを派生させることができます。

コンポーネントの派生

すべてのコンポーネントは，TComponent，あるいは TControl や TGraphicControl などの TComponent を特殊化した下位クラスから派生したクラスです。また，他の既存のコンポーネントクラスから派生したものもあります。45-2 ページの「コンポーネントの作成」で，さまざまな種類のコンポーネントとクラスの派生関係について説明しています。

クラスを派生させる方法についての詳細は，46-1 ページの「新しいクラスの定義」を参照してください。

コンポーネントクラスを派生させるには，そのコンポーネントを入れるヘッダーファイルにクラス宣言を記述します。

単純なコンポーネントクラスとは，TComponent から直接継承した非ビジュアルコンポーネントクラスです。

単純なコンポーネントクラスを作成するには，ヘッダーファイルに次のクラス宣言を記述します。

```
class PACKAGE TNewComponent : public TComponent
{
};
```

PACKAGE マクロが展開されると，クラスのインポートおよびエクスポートを許可する文になります。また，新しいコンポーネントに必要な .HPP ファイルを指定するために include 文を追加しなければなりません。以下は，必要とされる include 文の一般的な例です。

```
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
```

この新しいコンポーネントは，このままでは TComponent と同じことしか行いません。ただ，そこに新しいコンポーネントを構築するためのフレームワークが作成されたことになります。

新しいコンストラクタの宣言

新しいコンポーネントには，クラスのコンストラクタをその派生元からオーバーライドするコンストラクタが必要です。新しいコンポーネントにコンストラクタを記述する場合は常に，継承されたコンストラクタを呼び出さなければなりません。

クラス宣言の内部では，そのクラスの public 部で仮想コンストラクタを宣言します。**public** 部についての詳細は，46-4 ページの「アクセスの制御」を参照してください。たとえば次のようにします。

```
class PACKAGE TNewComponent : public TComponent
{
public:
  virtual __fastcall TNewComponent(TComponent* AOwner);
};
```

ヘッダーファイルでの定義に合わせて，.CPP ファイルでは次のコンストラクタを実装します。

```
__fastcall TNewComponent::TNewComponent(TComponent* AOwner): TComponent(AOwner)
{
}
```

コンストラクタ内では，コンポーネントが作成されるときに実行したいコードを追加します。

コンポーネントの登録

登録は，C++Builder のコンポーネントライブラリに追加するコンポーネントと，そのコンポーネントを表示するコンポーネントパレットのページを C++Builder に知らせるだけです。登録についての詳細は，第 52 章「コンポーネントを設計時に利用できるようにする」を参照してください。

コンポーネントを登録する手順は次のとおりです。

1. ユニットの .CPP ファイルに Register という名前の関数を記述し，それを特定の名前空間に配置します。名前空間の名前は，コンポーネントが含まれるファイルの名前から拡張子を取り除いたもので，先頭の文字以外はすべて小文字で表されます。

   たとえば，.CPP ファイルの名前が Newcomp であるとき，以下のコードが名前空間 Newcomp の内部に記述されます。

   ```
   namespace Newcomp
   {
     void __fastcall PACKAGE Register()
     {
     }
   }
   ```

2. Register 関数の中で TComponentClass 型のオープン配列を宣言して，登録しようとするコンポーネントを格納します。構文は次のようになります。

   ```
   TComponentClass classes[1] = {__classid(TNewComponent)};
   ```

   この例では，クラスの配列格納されているコンポーネントは 1 つだけですが，登録したいコンポーネントすべてを配列に格納することもできます。

3. Register 関数の内部で，登録したいコンポーネントのそれぞれに対して RegisterComponents 関数を呼び出します。

   RegisterComponents 関数は 3 つのパラメータをとります。そのパラメータは，コンポーネントパレットのページ名，コンポーネントクラスの配列，およびコンポーネントクラスの配列の最後のクラスのインデックスです。既存の登録プロセスに新しいコンポーネントを追加する場合には，既存の文中のコンポーネントの集合に新しいコンポーネントを追加するか，RegisterComponents を呼び出す新しい文を追加します。

   コンポーネントパレットの同一ページに複数のコンポーネントを登録する場合は，RegisterComponents を 1 回呼び出すだけでそれらをまとめて登録できます。

TNewComponent という名前でコンポーネントを登録し，コンポーネントパレットの［Samples］ページに配置するには，TNewComponent クラスが宣言されるユニットの .CPP ファイルに以下のコードを記述します。

```
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TNewComponent)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

この Register 呼び出しは TNewComponent をコンポーネントパレットの［Samples］ページに配置します。

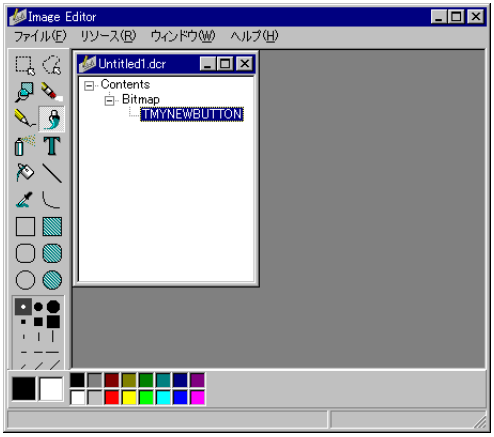
コンポーネントの登録が済んだら，最終的にコンポーネントをコンポーネントパレットにインストールする前に，コンポーネントをテストできます。これについては，45-19 ページの「コンポーネントパレットへのコンポーネントのインストール」で詳しく説明しています。

コンポーネント用のビットマップの作成

新しいコンポーネントを作成すると，カスタムコンポーネント用の独自のビットマップを定義できます。

1. ［ツール｜イメージエディタ］を選択します。
2. イメージエディタで，［ファイル｜新規作成｜コンポーネントリソースファイル（.dcr)］を選択します。
3. ［untitled1.dcr］ダイアログボックスで，［Contents］を右クリックします。［新規作成｜ビットマップ］を選択します。
4. ［ビットマッププロパティ］ダイアログボックスで，［幅］と［高さ］の両方を 24 ピクセルに変更します。［VGA（16 色)］がチェックされていることを確認します。［OK］をクリックします。
5. ［Contents］の下に，［Bitmap］と［Bitmap1］が表示されます。［Bitmap1］を選択し，右クリックして［名称変更］を選択します。ビットマップに，T を含めてすべて大文字で新しいコンポーネントのクラス名と同じ名前を付けます。たとえば，新しいクラス名が TMyNewButton の場合は，ビットマップの名前を TMYNEWBUTTON とします。

メモ　［コンポーネントの新規作成］ダイアログボックスの指定にかかわらず，すべて大文字の名前を付ける必要があります。



6. TMYNEWBUTTON をダブルクリックすると，空のビットマップのダイアログボックスが表示されます。
7. イメージエディタの一番下のカラーパレットを使用してアイコンを設計します。

8. ［ファイル｜名前を付けて保存］を選択し，リソースファイル（.dcr または .res）に，コンポーネントクラスを宣言するユニットと同じ基本名を付けます。たとえば，リソースファイルに MyNewButton.dcr という名前を付けます。
9. ［コンポーネント｜コンポーネントの新規作成］を選択します。45-9 ページの手順に従って，コンポーネントウィザードで新しいコンポーネントを作成します。コンポーネントのソースファイル MyNewButton.cpp が MyNewButton.dcr と同じディレクトリにあることを確認します。

   コンポーネントウィザードでは，クラス名が TMyNewButton の場合，コンポーネントのソースファイルまたはユニットには MyNewButton.cpp という名前が付き，デフォルトで LIB ディレクトリに配置されます。［参照］ボタンをクリックし，生成されたコンポーネントユニットの新しい場所を検出します。

メモ .dcr ファイルでなく .res ファイルを使用してビットマップを作成する場合は，コンポーネントのソースファイルに参照を追加してリソースをバインドします。たとえば，.res ファイルの名前が MyNewButton.res の場合は，.cpp ファイルと .res ファイルが同じディレクトリにあることを確認してから，MyNewButton.cpp に以下を追加します。

```
#pragma resource "*.res"
```

10. ［コンポーネント｜コンポーネントのインストール］を選択し，新しいパッケージまたは既存のパッケージにコンポーネントをインストールします。［OK］をクリックします。

    新しいパッケージが作成され，インストールされます。新しいコンポーネントを表すビットマップが，コンポーネントウィザードで指定したコンポーネントパレットのページに表示されます。

インストール前のコンポーネントのテスト

コンポーネントをコンポーネントパレットにインストールする前に，そのコンポーネントの実行時の動作をテストできます。このテストは新たに作成したコンポーネントのデバッグに役立ちます。このテクニックは，コンポーネントパレットに表示するかどうかにかかわらず，すべてのコンポーネントに対して使用できます。インストールしたコンポーネントのテストについては，45-18 ページの「インストールしたコンポーネントのテスト」を参照してください。

インストール前にコンポーネントをテストすることの利点は，通常，クラスをインスタンス化したときにしか見られないコンパイル時エラーを生成できることです。たとえば，抽象クラスのインスタンスを作成しようとすると，エラーが発生し，純粋仮想関数をオーバーロードしなければならないことを示します。

コンポーネントをパレットから選択しフォームに配置したときに，C++Builder が実行するアクションをエミュレートすることにより，インストールされていないコンポーネントをテストします。

インストールされていないコンポーネントをテストする手順は次のとおりです。

1. 新規にアプリケーションを作成するか，既存のアプリケーションを開きます。
2. コンポーネントが含まれているユニットをプロジェクトに追加するには，［プロジェクト｜プロジェクトに追加］を選択します。

3. フォームユニットのヘッダーファイルにコンポーネントユニットの .H ファイルをインクルードします。
4. そのコンポーネントを表すフォームに，データメンバーを追加します。

   これは，コンポーネントを開発者が手作業で追加する場合と，C++Builder が追加する場合との大きな相違点です。開発者は，フォームのクラス宣言の一番下の **public** 部にデータメンバーを追加します。これに対して C++Builder は，クラス宣言の上方の C++Builder 自身で管理している **published** 部に追加します。

   フォームのクラス宣言の C++Builder が管理する部分には，決してデータメンバーを追加しないでください。型宣言のこの部分の項目は，フォームファイルに格納されている項目に対応しています。フォーム上に存在しないコンポーネントの名前が追加されると，フォームファイルが無効になってしまう恐れがあります。
5. フォームのコンストラクタでコンポーネントを構築します。

   コンポーネントのコンストラクタを呼び出した場合には，そのコンポーネントのオーナー（適切な時期にそのコンポーネントを破棄する責任を持つコンポーネント）を指定するパラメータを渡す必要があります。これについては，ほとんどすべての場合に，**this** をオーナーとして渡すことになります。メソッドでは，**this** はそのメソッドが所属するクラスへの参照を表します。フォームの OnCreate ハンドラでは，**this** はフォームへの参照です。
6. コンポーネントの Parent プロパティを指定します。

   Parent プロパティの設定は，必ずコントロールを構築した直後に行います。親は，そのコントロールをビジュアルに含んでいるようなコンポーネントです。多くの場合，コンポーネントの親はフォーム自身になりますが，グループボックスやパネルの場合もあります。通常，Parent は **this**，つまりフォームに設定します。必ず，Parent はコントロールのほかのプロパティを設定する前に設定します。
7. 必要に応じて，ほかのコンポーネントプロパティを設定します。

NewCtrl という名前のユニットで，TNewControl クラスの新しいコンポーネントをテストすることを考えます。新しいプロジェクトを作成し，前述の手順にしたがって，次のようなフォーム用のヘッダーファイルを作成します。

```
//---------------------------------------------------------------------------
#ifndef TestFormH
#define TestFormH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include "NewCtrl.h"
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:          // IDE が管理するコンポーネント
private:              // ユーザー宣言
public:               // ユーザー宣言
    TNewControl* NewControl1;
    __fastcall TForm1(TComponent* Owner);
```

```
};
//---------------------------------------------------------------------------
extern TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

NEWCTRL.H ファイルをインクルードしている #include 文は，現在開いているプロジェクトの作業ディレクトリか，プロジェクトのインクルードパスが示すディレクトリのどちらかにコンポーネントが格納されていることを前提にしています。

次の例は，フォームユニットの .CPP ファイルです。

```
#include <vcl.h>
#pragma hdrstop
#include "TestForm.h"
#include "NewCtrl.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
static inline TNewControl *ValidCtrCheck()
{
    reutrn new TNewControl(NULL);
}
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
    NewControl1 = new TNewControl(this);
    NewControl1->Parent = this;
    NewControl1->Left = 12;
}
//---------------------------------------------------------------------------
namespace Newctrl
{
    void __fastcall PACKAGE Register()
    {
        TComponentClass classes[1] = {__classid(TNewControl)};
        RegisterComponents("Samples", classes, 0);
    }
}
```

インストールしたコンポーネントのテスト

コンポーネントをコンポーネントパレットにインストールする前に，そのコンポーネントの実行時の動作をテストできます。このテストは新たに作成したコンポーネントのデバッグに役立ちます。このテクニックは，コンポーネントパレットに表示するかどうかにかかわらず，すべてのコンポーネントに対して使用できます。インストールしていないコンポーネントのテストについては，45-16 ページの「インストール前のコンポーネントのテスト」を参照してください。

インストール後にコンポーネントをテストすることにより，フォームにドロップされたときに設計時例外を生成するだけのコンポーネントをデバッグすることができます。

C++Builder の 2 番目の実行中インスタンスを使って，インストールされたコンポーネントをテストします。

1. C++Builder IDE のメニューから［プロジェクト｜オプション］を選択し，［ディレクトリ / 条件］ページの［デバッグ用ソースパス］で，コンポーネントのソースファイルへのパスを設定します。
2. 次に，［ツール｜デバッガオプション］を選択します。［言語固有の例外］ページで，追跡する例外を有効にします。
3. コンポーネントのソースファイルを開いて，ブレークポイントを設定します。
4. ［実行｜実行時引数］を選択し，［ホストアプリケーション］フィールドに C++Builder 実行可能ファイルの名前と格納場所を設定します。
5. ［実行時の引数］ダイアログで［読み込み］ボタンをクリックして，C++Builder の 2 番目のインスタンスを起動します。
6. 次に，テストするコンポーネントをフォームにドロップします。これによって，ソース内のブレークポイントで一時停止します。

コンポーネントパレットへのコンポーネントのインストール

コンポーネントパレットにコンポーネントを追加するには以下の 2 つの手順を実行します。

- ソースファイルの有効化
- コンポーネントの追加

ソースファイルの有効化

1 つのコンポーネントが使用するすべてのソースファイルは同じディレクトリに置きます。これらのファイルには，ソースコードファイル（.CPP および .PAS）とバイナリファイル（.DFM，.RES，.RC，および .DCR）があります。ヘッダーファイル（.H および .HPP）は，インクルードディレクトリに配置する必要があります（IDE またはプロジェクトの検索パスの場所の場合もあります）。

コンポーネントを追加すると，その結果としていくつかのファイルが作成されます。これらのファイルは，IDE 環境オプション（［ツール｜環境オプション］を選択し，［ライブラリ］タブページへ移動します）で指定されたディレクトリに自動的に置かれます。.LIB ファイルは BPI/LIB 出力ディレクトリに置かれます。コンポーネントを追加した結果，新しいコンポーネントが既存のパッケージではなく新しいパッケージにインストールされる場合，.BPL ファイルは BPL 出力ディレクトリに，.BPI ファイルは BPI/LIB 出力ディレクトリに配置されます。

コンポーネントの追加

コンポーネントをコンポーネントライブラリに追加する手順は次のとおりです。

1. ［コンポーネント｜コンポーネントのインストール］を選択します。

［コンポーネントのインストール］ダイアログボックスが表示されます。

2. 該当するページを選択して，新しいコンポーネントを既存のパッケージにインストールするか新しいパッケージにインストールするかを指定します。
3. 新しいコンポーネントを含む .CPP ファイルの名前を入力するか，または［参照］を選んでそのユニットを検索指定します。
4. 新しいコンポーネントの .CPP ファイルが表示されない場合は，正しい検索パスを入力します。
5. コンポーネントをインストールするパッケージの名前を入力するか，または［参照］を選んでそのパッケージを検索します。
6. コンポーネントを新しいパッケージにインストールした場合，必要であればパッケージの説明を入力します。
7. ［OK］を選択して，［コンポーネントのインストール］ダイアログボックスを閉じます。これにより，パッケージがコンパイル後に再構築され，コンポーネントパレットにコンポーネントがインストールされます。

メモ 新しくインストールされたコンポーネントは，最初はコンポーネント開発者が指定したコンポーネントパレットのページに表示されます。コンポーネントがコンポーネントパレットにインストールされた後，［コンポーネント｜パレットの設定］ダイアログボックスを使用して，コンポーネントを別のページに移動できます。

第46章

コンポーネント開発者のための オブジェクト指向プログラミング

クラスはデータとコードから構成されます。コンポーネント開発者だけでなく，普通の C++Builder アプリケーションの開発者もオブジェクトを常に扱っています。その意味においては，開発者もコンポーネントユーザーとなります。

新しいコンポーネントを作成する場合，アプリケーション開発者には必要のない方法でクラスを取り扱います。また，コンポーネントの内部機構を，そのコンポーネントを使う開発者から隠蔽するようにします。コンポーネントに適した上位クラスを選択し，開発者が必要とするプロパティとメソッドだけを提供するインターフェースを設計して，このセクションで示すその他のガイドラインに従うことにより，用途が広く，繰り返し使えるコンポーネントを作成できます。

コンポーネントの作成を開始する前に，オブジェクト指向プログラミング（OOP）一般について，次の項目を理解しておく必要があります。

- 新しいクラスの定義
- 上位クラス，下位クラス，クラス階層
- アクセスの制御
- メソッドのディスパッチ
- 抽象クラスメンバー
- クラスとポインタ

新しいクラスの定義

コンポーネント開発者は新しいクラスを作成します。それに対して，アプリケーション開発者はクラスのインスタンスを操作します。ここに，コンポーネント開発者とアプリケーション開発者との相違点があります。

クラスは基本的に 1 つの型です。プログラマは常に型とインスタンスを使っています。ただ，こうした用語を一般には使わないだけです。たとえば，**int** のような型の変数を作成します。一般にクラスは，単純なデータ型よりも複雑ですが，使い方は同じです。同じ型に属する各インスタンスにさまざまな値を指定することにより，きわめて多様な処理を実現できます。

たとえば，よくあることですが，一方は OK という文字列を持ち，他方は Cancel という文字列を持つ 2 つのボタンをあるフォーム上に作成するとします。この場合，2 つのボタンは両方とも TButton クラスのインスタンスになります。しかし，Caption プロパティに異なる値を指定し，さらに OnClick イベントに異なるハンドラを設定することにより，この 2 つのインスタンスにまったく異なった機能を与えます。

新しいクラスの派生

新しいクラスを派生させる理由は 2 つあります。

- クラスのデフォルトを変更する
- クラスに新しい機能を追加する

どちらの場合も，目標は再利用可能なオブジェクトの作成です。再利用を考慮してコンポーネントを設計すれば，後々，多くの作業を省けます。開発するクラスには，有用なデフォルト値を与えて，同時にカスタマイズも可能にしておきます。

クラスのデフォルトを変更する

たいていのプログラマは反復作業は避けたいものです。たとえば，同じコードを何度も繰り返し記述していることに気付いた場合には，そのコードを関数として独立させるか，またはさらにルーチンをライブラリ化して多くのプログラムで使用できるようにします。コンポーネントについても同じ考え方がそのまま当てはまります。同じプロパティを繰り返し変更したり，同じメソッドを繰り返し呼び出しているような場合には，それをデフォルトで行う新しいコンポーネントを作成すべきです。

たとえば，アプリケーションを作成するたびに，特定の操作をするためのダイアログボックスを追加していることがあります。同じダイアログボックスを毎回作成するのはそれほど困難なことではありませんが，それは不必要な作業です。なぜなら，最初に 1 度だけダイアログボックスを設計してプロパティを設定し，それに関連付けたラッパーコンポーネントをコンポーネントパレットにインストールすることができるからです。ダイアログボックスを再使用可能なコンポーネントにすれば，反復作業を減少できるだけでなく，標準化が推進され，ダイアログボックスを再作成することによりエラーが発生する可能性が減少します。

第 53 章「既存のコンポーネントの変更」では，コンポーネントのデフォルトプロパティの変更例を示しています。

メモ 既存のコンポーネントのパブリッシュプロパティだけを変更する場合や，コンポーネントまたはコンポーネントグループの特定のイベントハンドラを保存する場合は，**コンポーネントテンプレート**を作成すると作業が簡単になります。

クラスに新しい機能を追加する

新しいコンポーネントを作成する一般的な理由の１つは，既存のコンポーネントが備えていない機能の追加です。この場合には，新しいコンポーネントを既存のコンポーネントから派生させるか，TComponent や TControl などの抽象基本クラスから派生させるかのどちらかの方法を採ります。

新しいコンポーネントは，必要な機能に最も近い機能のサブセットを備えたクラスから派生させます。クラスに対して機能を追加することはできますが，クラスから機能を取り去ることはできません。したがって，既存のコンポーネントクラスに，開発するコンポーネントに含めたくないプロパティが入っている場合には，そのコンポーネントの上位クラスから派生させる必要があります。

たとえば，リストボックスに機能を追加する場合には，TListBox からコンポーネントを派生させることができます。ただし，ある新しい機能を追加するだけでなく，標準リストボックスのある種の機能は除去したいという場合には，TListBox の上位クラスである TCustomListBox からコンポーネントを派生させます。この場合，リストボックスの機能のうち必要なものだけ再作成します。そして，新しい機能を追加します。

抽象コンポーネントクラスのカスタマイズの例については，第 55 章「グリッドのカスタマイズ」を参照してください。

新しいコンポーネントクラスの宣言

C++Builder では，標準コンポーネントのほかに，新しいコンポーネントを派生させるための基礎として設計された抽象クラスが数多く用意されています。45-3 ページの 表 45.1 に，コンポーネント開発で派生元として使用できるクラスが示されています。

新しいコンポーネントクラスを宣言するには，そのコンポーネントのヘッダファイルにクラス宣言を記述します。

簡単なグラフィックコンポーネントの宣言を次に示します。

```
class PACKAGE TSampleShape : public TGraphicControl
{
public:
    virtual __fastcall TSampleShape(TComponent* Owner);
};
```

SysDefs.h ファイルで定義されている PACKAGE マクロを追加することを忘れないでください。このマクロにより，クラスのインポートおよびエクスポートが可能になります。

最終的には，このコンポーネント宣言の最後の } の前にプロパティ，データメンバー，メソッドの各宣言が通常は記述されることになります。しかし，上の例のような宣言でもかまいません。当初は空のままにしておき，後で追加できます。

上位クラス，下位クラス，クラス階層

アプリケーション開発者から見ると，すべてのコントロールはプロパティ Top および Left を持っています。このプロパティは，そのコントロールのフォームでの位置を示します。この 2 つのプロパティは，共通の上位クラスである TControl から継承されていますが，アプリケーション開発者はそ

のようなことを意識する必要はありません。しかし，コンポーネントを作成する場合には，コンポーネントにどのクラスを継承させるか決めなければなりません。また，継承した機能を再作成することなく利用できるように，開発するコントロールが継承する機能について把握しておく必要があります。

コンポーネントの派生元となるクラスを，そのコンポーネントの直接上位クラスといいます。各コンポーネントの継承元は，そのコンポーネントの直接上位クラスや，直接上位クラスの上位クラスという具合になります。コンポーネントの継承元となるすべてのクラスを，そのコンポーネントの上位クラスといいます。この場合コンポーネントは，その上位クラスから見て下位クラスになります。

1 つのアプリケーションでは，すべての上位下位の関係が 1 つのクラス階層を構成します。階層がより下位のクラスほど多くの機能が入っています。あるクラスはその上位クラスの機能をすべて継承したうえで，新しいプロパティやメソッドを追加したり，既存のプロパティやメソッドを再定義するからです。

C++Builder では，直接上位クラスを指定しなかったときは，デフォルトの上位クラスである TObject から派生します。TObject はオブジェクト階層のすべてのクラスの最上位クラスです。

どのオブジェクトからフォームを派生させるかを決める基本ルールは単純です。新しいオブジェクトに入れたい内容がもっとも多く含まれていて，なおかつ新しいオブジェクトに必要のない内容はいっさい入っていないオブジェクトを選ぶようにします。オブジェクトの内容は，いつでも追加することはできますが，継承した内容を取り除くことはできないからです。

アクセスの制御

プロパティ，メソッド，データメンバーへのアクセス制御には５つのレベルがあります。このアクセス制御は可視性ともいいます。可視性により，どのコードがクラスのどの部分にアクセスできるかが決まります。可視性を指定することにより，コンポーネントへのインターフェースを定義します。

表 46.1 に，可視性のレベルを，アクセスの制約がもっとも厳しいレベルからもっとも緩いレベルの順に示します。

表 46.1　オブジェクトの可視性のレベル

可視性	意味	使用目的
private	そのクラスが定義されているクラスからのみアクセスできる	実装の詳細の隠蔽
protected	そのクラスが定義されているクラスとその下位クラスからアクセスできる	コンポーネント開発者用インターフェースの定義
public	すべてのコードからアクセスできる	実行時インターフェースの定義
__automated	すべてのコードからアクセスできるオートメーション型情報が生成される	OLE オートメーションのみ
__published	すべてのコード，およびオブジェクトインスペクタからアクセスできる。フォームファイルに保存される	設計時インターフェースの定義

実装の詳細の隠蔽

クラスのある部分を **private** として宣言すると，その部分はそのクラスの外側からはフレンド関数を使わない限り見えなくなります。クラスの **private** 部は，クラスのユーザーから実装の詳細を隠すのに最も効果的な方法です。クラスのユーザーは **private** 部にアクセスできないので，ユーザーが記述するコードに影響を与えることなく，内部の実装を変更できます。

データメンバー，メソッド，またはプロパティにアクセス制御を指定しない場合，その部分は **private** 扱いになります。

あるデータメンバーを **private** として宣言して，アプリケーション開発者による情報へのアクセスを防ぐ方法の例を 2 つの部分に分けて示します。

この例の前半は，ヘッダーファイルと，フォームの OnCreate イベントハンドラによって **private** データメンバーに値を代入する .CPP ファイルによってできています。イベントハンドラは TSecretForm クラスの中で定義されているので，ユニットのコンパイルはエラーなしで終了します。

```
#ifndef HideInfoH
#define HideInfoH
//---------------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
class PACKAGE TSecretForm : public TForm
{
__published:          // IDE が管理するコンポーネント
    void __fastcall FormCreate(TObject *Sender);
private:
    int FSecretCode;                                    // プライベートデータメンバーの宣言
public:               // ユーザー宣言
  __fastcall TSecretForm(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern TSecretForm *SecretForm;
//---------------------------------------------------------------------------
#endif
```

対応する HideInfo.CPP ファイルを次に示します。

```
#include <vcl.h>
#pragma hdrstop
#include "hideInfo.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
#pragma resource "*.dfm"
TSecretForm *SecretForm;
//---------------------------------------------------------------------------
__fastcall TSecretForm(TComponent* Owner);
  : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TSecretForm::FormCreate(TObject *Sender);
{
```

```
  FSecretCode = 42;                              // コンパイル可能
}
```

この例の後半は，SecretForm フォームのデータメンバー FSecretCode に値を代入しようとする別のフォームユニットです。次にユニットのヘッダーファイルの内容を示します。

```
#ifndef TestHideH
#define TestHideH
//---------------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
class PACKAGE TTestForm : public TForm
{
__published:          // IDE が管理するコンポーネント
    void __fastcall FormCreate(TObject *Sender);
public:               // ユーザー宣言
    __fastcall TTestForm(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern TTestForm *TestForm;
//---------------------------------------------------------------------------
#endif
```

以下に示すのは，このヘッダーファイルに対応する .CPP ファイルの内容です。TTestForm の OnCreate イベントハンドラは SecretForm フォームに **private** データメンバーへの値を代入しようとするため，「'Secretform::FSecretCode' にアクセスできない」というエラーメッセージが表示され，コンパイルは失敗します。

```
#include <vcl.h>
#pragma hdrstop
#include "testHide.h"
#include "hideInfo.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
#pragma resource "*.dfm"
TTestForm *TestForm;
//---------------------------------------------------------------------------
__fastcall TTestForm(TComponent* Owner);
  : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TTestForm::FormCreate(TObject *Sender);
{
  SecretForm->FSecretCode = 13;          // エラーメッセージが表示され，コンパイルは中断する
}
```

HideInfo ユニットを使用するプログラムは TSecretForm 型のクラスを使用できますが，そのクラスの FSecretCode データメンバーにはアクセスできません。

CLX　CLX アプリケーションでは，ヘッダーファイルの名前と場所が異なります。たとえば，<vcl \ controls.hpp> は CLX では <clx \ qcontrols.hpp> です。

コンポーネント開発者用インターフェースの定義

クラスのある部分を **protected** として宣言すると，その部分は，そのクラスとその下位クラスにしか見えません。

protected 宣言を使うと，クラスに対してコンポーネント開発者用のインターフェースを定義できます。**protected** の部分には，アプリケーションユニットはアクセスできませんが，派生クラスはアクセスできます。つまり，コンポーネント開発者は，アプリケーション開発者に詳細を公開しなくても，クラスの動作を変更できます。

メモ よくある間違いは，イベントハンドラから **protected** のメソッドにアクセスしようとすることです。イベントハンドラは通常はフォームのメソッドであり，イベントを受け取るコンポーネントではありません。したがって，（コンポーネントがフォームと同じユニット内で宣言されている場合以外は）イベントハンドラはコンポーネントの **protected** のメソッドにはアクセスできません。

実行時インターフェースの定義

クラスで **public** として宣言した部分は，そのクラス全体にアクセスするすべてのコードから見えるようになります。

public 部は，実行時にすべてのコードで使用できます。したがって，クラスはその **public** 部によって実行時インターフェースが定義されることになります。実行時の入力に依存するプロパティや読み出し専用のプロパティなど，設計時には無効な項目は，実行時インターフェースに宣言します。また，アプリケーション開発者によって呼び出されるメソッドも **public** として宣言しなければなりません。

次の例は，コンポーネントの実行時インターフェースの部分に宣言された 2 つの読み出し専用プロパティを示します。

```
class PACKAGE TSampleComponent : public TComponent
{
private:
    int FTempCelsius;                                    // 実装の詳細を private 部に記述
    int GetTempFahrenheit();
public:
    ...
    __property int TempCelsius = {read=FTempCelsius};    // プロパティを public 部に記述
    __property int TempFahrenheit = {read=GetTempFahrenheit};
};
```

次に，.CPP ファイルの GetTempFahrenheit メソッドを示します。

```
int TSampleComponent::GetTempFahrenheit()
{
  return FTempCelsius * (9 / 5) + 32;
}
```

設計時インターフェースの定義

クラスで **published** として宣言した部分は，パブリックになり，さらに実行時型情報が生成されます。この実行時型情報により，オブジェクトインスペクタでプロパティやイベントにアクセスできるようになります。

published 部はオブジェクトインスペクタに表示されるため，クラスの **published** 部によって，そのクラスの設計時インターフェースが定義されることになります。クラスの各側面のうち，アプリケーション開発者が設計時にカスタマイズする可能性のある部分はすべて設計時インターフェースに含めます。これに対して，実行時環境の特定の情報に依存するすべてのプロパティは，設計時インターフェースには含めません。

読み出し専用プロパティはアプリケーション開発者が直接値を代入できないので，設計時インターフェースに含めません。読み出し専用プロパティは published ではなく public と宣言します。

次の例は，パブリッシュに設定された Temperature プロパティを示します。このプロパティはパブリッシュに設定されているため，設計時にオブジェクトインスペクタに表示されます。

```
class PACKAGE TSampleComponent : public TComponent
{
private:
    int FTemperature;
  ...
__published:
    __property int Temperature = {read=FTemperature, write=FTemperature};
};
```

メソッドのディスパッチ

ディスパッチは，クラスメソッド呼び出しを検出したとき，どのクラスメソッドを呼び出すかをアプリケーションに記述するときに使われる用語です。クラスメソッドを呼び出すコードを記述すると，そのコードは他の関数呼び出しと同じように見えます。しかし，クラスにはメソッドをディスパッチするための方法が

2 通りあります。

- 通常の（仮想でない）メソッド
- 仮想メソッド

通常のメソッド

メソッドを **virtual** 宣言したり，基本クラスの **virtual** メソッドをオーバーライドしない限り，すべてのメソッドは通常の，非仮想メソッドになります。コンパイラは通常のクラスメンバーの正確なアドレスを決定します。これをコンパイル時バインドといいます。

基本クラスのレギュラーメソッドは，派生クラスによって継承されます。次の例では，Derived 型のオブジェクトは，Regular() メソッドを自分自身に属するメソッドであるかのように呼び出せます。派生クラス内で，その上位クラス内のレギュラーメソッドと同じ名前およびパラメータを持つメソッ

ドを宣言して，上位クラスのメソッドを置換します。d->AnotherRegular() メソッドが呼び出されると，Derived クラスの AnotherRegular() メソッドの置換がディスパッチされます。

```
class Base
{
public:
    void Regular();
    void AnotherRegular();
    virtual void Virtual();
};
class Derived : public Base
{
public:
    void AnotherRegular();          // Base::AnotherRegular() を置換する
    void Virtual();                 // Base::Virtual() をオーバーライドする
};
void FunctionOne()
{
  Derived *d;
  d = new Derived;
  d->Regular();             // Derived クラスのメンバーであるかのように Regular() を呼び出す
                            // d->Base::Regular() の呼び出しと同じ
  d->AnotherRegular();      // ... 再定義された AnotherRegular() を呼び出す
                            // ... Base::AnotherRegular() を置換する
  delete d;
}
void FunctionTwo(Base *b)
{
  b->Virtual();
  b->AnotherRegular();
}
```

仮想メソッド

コンパイル時に結合されるレギュラーメソッドと異なり，仮想メソッドは実行時に結合されます。C++ の仮想メカニズムにより，呼び出し元であるクラスの型に応じたメソッドの呼び出しが可能になります。

前の例では，Derived オブジェクトへのポインタを引数として FunctionTwo() 関数を呼び出すと，Derived::Virtual() 関数が呼び出されます。仮想メカニズムは，実行時に渡したオブジェクトのクラス型を動的に調べ，適切なメソッドをディスパッチします。しかし，レギュラー関数 b->AnotherRegular() を呼び出すと，必ず Base::AnotherRegular() が呼び出されます。これは，AnotherRegular() のアドレスが，コンパイル時に決定されるためです。

新しい仮想メソッドを宣言するときは，メソッドの宣言の前にキーワード **virtual** を付加します。

コンパイラがキーワード **virtual** を見つけると，クラスの仮想メソッドテーブル（VMT）にエントリを作成します。VMT は，クラス内のすべての仮想メソッドのアドレスを保持します。この参照テーブルは，実行時に b->Virtual が Base::Virtual() でなく Derrived::Virtual() を呼び出すことを決定するために使われます。

既存のクラスから新しいクラスを派生させると，その新しいクラスにはそれ自身の VMT が与えられます。この VMT には，上位クラスから継承したエントリに加えて，新しいクラスで新たに宣言されたすべての仮想メソッドのエントリが格納されます。下位クラスは，すべての継承した仮想メソッドをオーバーライドできます。

メソッドのオーバーライド

メソッドのオーバーライドは，上位メソッドの置換ではありません。メソッドのオーバーライドとは，そのメソッドを拡張したり，詳細化することを意味します。下位クラスのメソッドをオーバーライドするには，派生クラスのメソッドを再定義します。このとき，引数の数と型が同じであることを確認してください。

次の例では，2 つの簡単なコンポーネントの宣言を示します。第 1 のコンポーネントは，ディスパッチ方法が異なる 2 つのメソッドを宣言しています。第 1 のコンポーネントから派生した第 2 のコンポーネントは，非仮想メソッドを置換し，仮想メソッドをオーバーライドします。

```
class PACKAGE TFirstComponent : public TComponent
{
public:
  void Move();                                  // 通常のメソッド
  virtual void Flash();                         // 仮想メソッド
};
class PACKAGE TSecondComponent : public TFirstComponent
{
public:
  void Move();              // TFirstComponent::Move() を隠蔽する新しいメソッドの宣言
  void Flash();             // TFirstComponent の 仮想メソッド TFirstComponent::Flash を
                            // オーバーライドする
};
```

抽象クラスメンバー

上位クラスでメソッドを **abstract** として宣言した場合，下位コンポーネントでそのメソッドを再宣言して実装することにより明示しなければ，その新しいコンポーネントはアプリケーションで使えません。C++Builder では，抽象メンバーを含むクラスのインスタンスは作成できません。クラスの継承部分の明示についての詳細は，第 47 章「プロパティの作成」と第 49 章「メソッドの作成」を参照してください。

クラスとポインタ

クラスは実際にはポインタであり，コンポーネントも同様です。しかし，クラスをパラメータとして渡すときは，クラスがポインタであることが重要になります。一般に，クラスは参照渡しではなく値渡しで渡します。なぜなら，クラスはもともとポインタ，つまり参照であるため，あるクラスを参照渡しで渡すことは，参照の参照を渡すことになり，あまり意味を持たないからです。

第47章

プロパティの作成

プロパティは，コンポーネントで最も可視的な構成要素です。アプリケーション開発者は設計時に画面でプロパティを操作でき，その操作の結果はすぐにフォームデザイナでコンポーネントに反映されます。プロパティを適切に設計すれば，ユーザーにとっては使いやすく，開発者にとっては保守しやすいコンポーネントが作成できます。

このセクションでは，開発するコンポーネントでプロパティを有効に使用するため，以下のことを説明します。

- プロパティを作成する理由
- プロパティの型
- 継承プロパティの公開
- プロパティの定義
- 配列プロパティの作成
- プロパティの保存と読み込み

プロパティを作成する理由

アプリケーション開発者の立場からは，プロパティは変数のようなものです。開発者は，データメンバーと同じように，プロパティの値を設定または読み出します。開発者にとって，変数では可能でプロパティでは不可能なことは，プロパティをメソッドに引数として参照渡しできないということだけです。

以下の理由でプロパティは単純なデータメンバーより大きな能力を持っています。

- プロパティはアプリケーション開発者が設計時に設定できる。実行時にのみ処理可能なメソッドと違って，開発者はプロパティによってアプリケーションを実行する前にコンポーネントをカスタマイズできる。プロパティはオブジェクトインスペクタに表示できるので，プログラマの作業が簡単になる。つまり，いくつものパラメータを処理してオブジェクトを作成するかわりに，C++Builder でオブジェクトインスペクタの値を設定できる。またオブジェクトインスペクタでは，プロパティへの代入が行われるとすぐにその代入が検証される

- プロパティは実装の詳細を隠蔽できる。たとえば，内部に暗号化形式で格納されているデータを，プロパティの値としては暗号化されていないように見せることが可能である。プロパティの値が単純な数値であっても，コンポーネントはその値を得るためにデータベースを検索したり，複雑な計算を行うことがある。プロパティを使えば，単純な代入のように見える操作に複雑な働きをさせることができる。つまりデータメンバーへの代入のように見える操作を，複雑な処理を実行するメソッドの呼び出しにできる
- プロパティは仮想メソッドであり得る。アプリケーション開発者から見て単一のプロパティでも，コンポーネントによっては実装が異なる場合がある

簡単な例は，すべてのコントロールが持つ Top プロパティです。Top への新しい値の代入は，単にその値を変更するだけの操作ではありません。コントロールを再配置して再描画します。また，プロパティの設定が及ぼす影響は，必ずしも 1 つのコンポーネントに限定されるわけではありません。あるスピードボタンの Down プロパティを **true** に設定すると，グループ内のほかのすべてのスピードボタンの Down プロパティは **false** になります。

プロパティの型

プロパティはどの型にでもできます。型によってオブジェクトインスペクタでの表示方法は異なります。オブジェクトインスペクタでは，設計時に行われたプロパティへの代入が検証されます。

表 47.1　オブジェクトインスペクタでのプロパティの型による表示方法の違い

プロパティの型	オブジェクトインスペクタでの扱われ方
単純型	数値，文字，文字列の各プロパティは，それぞれ数値，文字，文字列として表示される。アプリケーション開発者はプロパティの値を直接入力して編集できる
列挙型	論理型を含めて，列挙型のプロパティでは，ソースコードで定義された値が表示される。開発者は，値の列をダブルクリックすることで，使用可能な値を循環的に切り替えられる。また，ある列挙型プロパティのすべての値を示すドロップダウンリストも用意されている
集合型	集合型のプロパティは，1 つの集合として表示される。ユーザーは，プロパティをダブルクリックして集合を展開すれば，集合の各要素を論理値として扱える（その要素が集合に含まれている場合に **true** となる）
オブジェクト型	それ自身クラスであるプロパティは，コンポーネントの登録手続きで指定した独自のプロパティエディタを持つ場合がある。ただし，そのオブジェクトがその中のプロパティをパブリッシュに設定している場合は，ユーザーはオブジェクトインスペクタでオブジェクトプロパティのリストを展開して（ダブルクリックして），それらを個別に編集できる。オブジェクトプロパティは必ず TPersistent を継承する
インターフェース型	インターフェースであるプロパティは，（TComponent の下位の）コンポーネントによって実装されるインターフェースが値である限り，オブジェクトインスペクタに表示できる。インターフェースプロパティは，多くの場合，独自のプロパティエディタを備えている
配列型	配列プロパティは，独自のプロパティエディタを備えている必要がある。オブジェクトインスペクタでは編集はサポートされていない。コンポーネントを登録するときにプロパティエディタを指定できる

継承プロパティの公開

すべてのコンポーネントは，それぞれの上位クラスからプロパティを継承しています。既存のコンポーネントから新しいコンポーネントを派生させると，その新しいコンポーネントは直接上位コンポーネントのすべてのプロパティを継承します。抽象クラスの 1 つから派生させた場合には，継承したプロパティの多くは **protected** か **public** のいずれかになりますが，**published** にはなりません。

protected または **public** のプロパティをオブジェクトインスペクタで設計時に使えるようにするためには，そのプロパティを **published** として再宣言します。再宣言するためには，下位クラスの宣言部に継承したプロパティの宣言を追加します。

たとえば，TWinControl から VCL コンポーネントを派生させると，このコンポーネントは **protected** の Ctl3D プロパティを継承します。新しいコンポーネントで DockSite を再宣言することで，保護のレベルを **public** か **published** のいずれかに変更できます。

次のコードは，DockSite を **published** として再宣言し，設計時に処理可能にする方法を示しています。

```
class PACKAGE TSampleComponent : public TWinControl
{
__published:
    __property DockSite;
};
```

プロパティを再宣言する際には，プロパティ名だけを指定します。「プロパティの定義」で説明するような情報や型を指定する必要はありません。また，新しいデフォルト値を宣言でき，そのプロパティを保存するかどうかも指定できます。

再宣言では，あるプロパティに対する制約を減らすことはできますが，制約の強化はできません。つまり，**protected** プロパティを **public** プロパティにすることはできますが，**public** プロパティを **protected** として再宣言し，このプロパティを隠蔽することはできないということです。

プロパティの定義

ここでは，新しいプロパティの宣言方法を示し，標準コンポーネントに関する規則を説明します。以下のトピックについて説明します。

- プロパティの宣言
- 内部的なデータ記憶
- 直接アクセス
- アクセスメソッド
- デフォルトのプロパティ値

プロパティの宣言

プロパティはコンポーネントクラスの宣言内で宣言されます。プロパティを宣言するには次の 3 つの項目を指定します。

- プロパティの名前

- プロパティの型
- プロパティの値を読み書きするメソッド。**write** メソッドが宣言されなければ，プロパティは読み出し専用

コンポーネントのクラス宣言の **__published** セクションで宣言したプロパティは，設計時にオブジェクトインスペクタで編集できます。パブリッシュプロパティの値は，コンポーネントと一緒にフォームファイルに保存されます。**public** セクションで宣言したプロパティは実行時に使用可能で，プログラムコードで読み出したり設定したりできます。

Count プロパティの代表的な宣言の例を次に示します。

```
class PACKAGE TYourComponent : public TComponent
{
private:
    int FCount;                                                  // 格納するデータメンバー
    int __fastcall GetCount();                                   // read メソッド
    void __fastcall SetCount( int ACount );                      // write メソッド
public:
    __property int Count = {read=GetCount, write=SetCount};      // プロパティ宣言
  ...
};
```

内部的なデータ記憶

プロパティのデータを格納する方法には，なんの制約もありません。ただし一般的には，C++Builder コンポーネントには次の規則があります。

- プロパティのデータはクラスデータメンバーに格納される
- プロパティデータの格納に使うデータメンバーはプライベートなので，コンポーネント自身の中からのみアクセスする。派生コンポーネントは継承プロパティを使わなければならず，プロパティの内部的なデータ記憶域に直接アクセスする必要はない
- クラスデータメンバーの識別子は，プロパティの名前の先頭に文字 F を加えたものとする。たとえば，TControl で定義された Width プロパティのデータ自体は，FWidth というデータメンバーに格納される

以上の背景となる基本原則は，プロパティの実装メソッドだけがプロパティの背後にあるデータにアクセスする，ということです。このため，メソッドやほかのプロパティがそのようなデータを操作する場合は，記憶域のデータに直接アクセスするのではなく，プロパティを通して操作を行います。そうすれば，継承したプロパティの実装方法を変更しても，派生したコンポーネントが無効になるようなことはありません。

直接アクセス

プロパティデータを処理する最も簡単な方法は，データに直接アクセスすることです。それには，プロパティ宣言の **read** 部と **write** 部で，プロパティ値の代入と読み出しは，アクセスメソッド の呼び出しではなく，内部記憶データメンバーへの直接アクセスで行う，ということを指定します。プロパティをオブジェクトインスペクタで操作したいが，プロパティ値の変更で即時処理が行われない場合には，直接アクセスは便利です。

一般によく行われる方法では，プロパティ宣言の **read** 部では直接アクセスを指定し，**write** 部ではメソッドの使用を指定します。書き込みではメソッドを使用するので，プロパティ値を変更するとコンポーネントの状態が更新されます。

次の例では，コンポーネント型宣言の **read** 部と **write** 部の両方でプロパティに直接アクセスを指定しています。

```
class PACKAGE TSampleComponent : public TComponent
{
private:                                          // 内部記憶はプライベート
    bool FReadOnly;                               // 値を入れておくデータメンバーを宣言
   ...
__published:                                      // 設計時にプロパティを使用可能にする
    __property bool ReadOnly = {read=FReadOnly, write=FReadOnly};
};
```

アクセスメソッド

プロパティ宣言の **read** 部と **write** 部には，データメンバーのかわりにアクセスメソッドを指定できます。アクセスメソッドはプロテクトにしなければならず，通常は **virtual** と宣言します。これにより，下位コンポーネントはプロパティの実装をオーバーライドできます。

アクセスメソッドをパブリックにしないでください。アクセスメソッドをプロテクトに保てば，アプリケーション開発者がアクセスメソッドのいずれかを呼び出し，プロパティを誤って変更してしまうことはありません。

index 指定子を使って 3 つのプロパティを宣言したクラスの例を示します。**index** 指定子によって，3 つのプロパティすべてが同じ **read** と **write** のアクセスメソッドを持つことができます。

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
  int __fastcall GetDateElement(int Index);    // Index パラメータをメモする
  void __fastcall SetDateElement(int Index, int Value);
public:
  __property int Day = {read=GetDateElement, write=SetDateElement, index=3, nodefault};
  __property int Month = {read=GetDateElement, write=SetDateElement, index=2,
                          nodefault};
  __property int Year = {read=GetDateElement, write=SetDateElement, index=1, nodefault};
};
```

日付の各要素（年，月，日）はどれも整数であり，日付を設定するときには各要素を日付に符号化する必要があるので，ここではコードの重複を避けるために 3 つのプロパティで **read** および **write** メソッドを共有しています。日付要素を読み出すためのメソッドと日付要素を書き込むためのメソッドがそれぞれ 1 つずつ必要です。

日付から要素を取得する read メソッドは次のようになります。

```
int __fastcall TSampleCalendar::GetDateElement(int Index)
{
  unsigned short AYear, AMonth, ADay;
  int result;
  FDate.DecodeDate(&AYear, &AMonth, &ADay); // 日付を要素に分解
  switch (Index)
```

```
  {
    case 1: result = AYear;  break;
    case 2: result = AMonth; break;
    case 3: result = ADay;   break;
    default: result = -1;
  }
  return result;
}
```

日付の要素を正しく設定する **write** メソッドは次のようになります。

```
void __fastcall TSampleCalendar::SetDateElement(int Index, int Value);
{
  unsigned short AYear, AMonth, ADay;
  if (Value > 0) // 要素はすべて正の値でなければならない
  {
    FDate.DecodeDate(&AYear, &AMonth, &ADay); // 日付要素を取得
    switch (Index)
    {
      case 1: AYear = Value;  break;
      case 2: AMonth = Value; break;
      case 3: ADay = Value;   break;
      default: return;
    }
  }
  FDate = TDateTime(AYear, AMonth, ADay); // 変更した日付を符号化
  Refresh(); // カレンダーの表示を更新
}
```

read メソッド

プロパティの read メソッドは，パラメータを 1 つも取らない（後述する例外を除く）関数で，そのプロパティと同じ型の値を返します。命名規則によって，この関数の名前はプロパティ名の先頭に「Get」を加えた名前にします。たとえば，Count という名前のプロパティの **read** メソッドは GetCount となります。**read** メソッドは内部記憶データに対して必要な処理を行って，適切な型のプロパティ値を生成します。

パラメータを取らないという規則の例外は配列プロパティとインデックス指定子（47-8 ページの「配列プロパティの作成」を参照）を使うプロパティだけで，どちらもインデックス値をパラメータとして渡します。インデックス指定子は，いくつかのプロパティで共有する単一の **read** メソッドを作成するために使います。

read メソッドを宣言しなければ，そのプロパティは書き込み専用になります。書き込み専用プロパティが使用されることはほとんどありません。

write メソッド

プロパティの **write** メソッドは，そのプロパティと同じ型の 1 つのパラメータを取る（後述する例外を除く）メンバー関数です。このパラメータは，参照渡しと値渡しのいずれの方法でも受け取ることができ，またどのような名前でも付けられます。命名規則によって，**write** メソッドの名前はプロパティ名の先頭に「Set」を加えた名前にします。たとえば，Count という名前のプロパティの **write** メソッドは SetCount となります。パラメータに渡された値は，そのプロパティの新しい値になります。**write** メソッドは，データを適切にプロパティの内部記憶に格納するために必要なすべての処理を行います。

1 つのパラメータを取るという規則の唯一の例外は配列プロパティとインデックス指定子を使うプロパティの場合で，どちらもインデックス値をもう 1 つのパラメータとして渡します。インデックス指定子は，いくつかのプロパティで共有する単一の **write** メソッドを作成するために使います。

write メソッドを宣言しなければ，そのプロパティは読み出し専用になります。

write メソッドでは，プロパティの変更の前に，新しく設定する値が現在値と異なっているかどうかを検査することがよくあります。次の例では，整数プロパティ Count の **write** メソッドがデータメンバー FCount に値を格納しています。

```
void __fastcall TMyComponent::SetCount( int Value )
{
  if ( Value != FCount )
  {
    FCount = Value;
    Update();
  }
}
```

デフォルトのプロパティ値

プロパティを宣言する場合，そのプロパティのデフォルト値を指定することができます。C++Builder ではプロパティの値とデフォルト値とを比較することで，そのプロパティをフォームファイルに保存するかどうかが決定されます。C++Builder では，デフォルト値を指定しないプロパティは常にフォームファイルに保存されます。

プロパティのデフォルト値を宣言するには，プロパティの名前の後ろに = を書き，そのあとに { } で囲んで **default** キーワードとデフォルト値を記述します。たとえば次のようにします。

```
__property bool IsTrue = {default=true};
```

メモ プロパティの宣言時にデフォルト値を指定しても，オブジェクト作成時にプロパティがその値で初期化されるわけではありません。特別な場合を除いて，コンポーネントのコンストラクタメソッド内に，プロパティの値をデフォルト値で初期化するコードを追加する必要があります。オブジェクトを初期化するとフィールドが 0 に設定されるため，整数プロパティを 0 に，文字列プロパティをヌルに，論理型プロパティを false に設定する場合にはコードの追加は必要ありません。

デフォルト値を指定しない方法

継承元のプロパティがデフォルト値を持っていた場合に，プロパティを再宣言することで，デフォルト値を変更することができます。デフォルト値そのものを持たないようにすることもできます。

プロパティがデフォルト値を持たないように指定するには，プロパティの名前の後ろに = を書き，そのあとに { } で囲んで **nodefault** キーワードを記述します。たとえば次のようにします。

```
__property int NewInteger = {nodefault};
```

あるプロパティを初めて宣言する場合は **nodefault** を指定する必要はありません。デフォルト値を宣言しなければ，デフォルト値はないことになります。

次のコンポーネント宣言の例は，デフォルト値 **true** を持つ論理型プロパティ IsTrue を持っています。コンポーネント宣言にはプロパティを初期化するコンストラクタがあります。

```
class PACKAGE TSampleComponent : public TComponent
{
private:
     bool FIsTrue;
public:
    virtual __fastcall TSampleComponent( TComponent* Owner );
__published:
    __property bool IsTrue = {read=FIsTrue, write=FIsTrue, default=true};
};
__fastcall TSampleComponent::TSampleComponent ( TComponent* Owner )
  : TComponent ( Owner )
{
  FIsTrue = true;
}
```

配列プロパティの作成

プロパティの中には，配列と同様にインデックス付けされたプロパティがあります。たとえば，TMemo の Lines プロパティはメモのテキストを構成するインデックス付けされた文字列のリストで，文字列の配列として扱えます。この Lines によって，大きなデータ（メモテキスト）の中の特定の要素（文字列）に簡単にアクセスできます。

配列プロパティはほかのプロパティと同じように宣言できますが，以下の点が異なります。

- 宣言に，型を持つインデックスが 1 つまたは複数個含まれる。インデックスには任意の型が指定できる
- プロパティの宣言に **read** 部と **write** 部を記述する場合は，メソッドを指定しなければならない。データメンバーは指定できない

配列プロパティの **read** メソッドと **write** メソッドは，インデックスを示す追加パラメータを受け取ります。このパラメータの順序は，宣言で指定されたインデックスと同じ順序，同じ型でなければなりません。

配列プロパティと配列にはいくつか相違点があります。配列プロパティのインデックスは，配列のインデックスと違って，整数型にする必要がありません。たとえば，プロパティを文字列によってインデックス付けをすることができます。また，配列プロパティでは個々の要素が参照できるだけで，プロパティの全範囲は参照できません。

次の例では，整数インデックスによって文字列を返すプロパティの宣言を示します。

```
class PACKAGE TDemoComponent : public TComponent
{
private:
    System::AnsiString __fastcall GetNumberSize(int Index);
public:
    __property System::AnsiString NumberSize[int Index] = {read=GetNumberSize};
  ...
};
```

次に示すのは，.CPP ファイル内の GetNumberSize メソッドです。

```
System::AnsiString __fastcall TDemoComponent::GetNumberSize(int Index)
{
  System::AnsiString Result;
  switch (Index)
  {
    case 0:
      Result = "Zero";
     break;
    case 100:
      Result = "Medium";
     break;
    case 1000:
      Result = "Large";
     break;
    default: Result = "Unknown size";
  }
  return Result;
}
```

サブコンポーネントのプロパティの作成

デフォルトでは，プロパティの値がほかのコンポーネントである場合にそのプロパティに値を代入するには，ほかのコンポーネントのインスタンスをフォームまたはデータモジュールに追加してから，そのコンポーネントをプロパティの値として代入します。ただし，プロパティ値を実装するオブジェクトのインスタンスを，コンポーネントが独自に作成することもできます。このような専用のコンポーネントをサブコンポーネントといいます。

サブコンポーネントにすることができるのは，持続的オブジェクト（TPersistent の下位オブジェクト）です。別個のコンポーネントがたまたまプロパティの値として代入される場合と異なり，サブコンポーネントのパブリッシュプロパティは，サブコンポーネントを作成したコンポーネントとともに保存されます。ただし，これを機能させるには，以下の条件を満たさなければなりません。

- サブコンポーネントのオーナーが，サブコンポーネントを作成したコンポーネントでなければならず，パブリッシュプロパティの値としてサブコンポーネントを使用する。サブコンポーネントが TComponent の下位である場合は，サブコンポーネントの Owner プロパティを設定することで可能になる。それ以外のサブコンポーネントの場合は，持続的オブジェクトの GetOwner メソッドをオーバーライドして，作成元のコンポーネントを返すようにしなければならない
- サブコンポーネントが TComponent の下位である場合は，自分がサブコンポーネントであることを示すために SetSubComponent メソッドを呼び出さなければならない。通常，この呼び出しは，サブコンポーネントを作成するときにオーナーが行うか，サブコンポーネントのコンストラクタが行う

通常，値がサブコンポーネントのプロパティは読み出し専用です。値がサブコンポーネントのプロパティを変更できるようにする場合は，ほかのコンポーネントがプロパティ値として代入されるときに，プロパティセッターがサブコンポーネントを解放しなければなりません。さらに，プロパティが NULL に設定されると，コンポーネントは多くの場合，サブコンポーネントを再インスタンス化します。それ以外の場合は，いったんプロパティがほかのコンポーネントに変更されると，そのサブコンポーネントは設計時に復元できなくなります。次の例は，値が TTimer であるプロパティのプロパティセッターを示しています。

```
void __fastcall TDemoComponent::SetTimerProp(ExtCtrls::TTimer *Value)
{
  if (Value != FTimer)
  {
    if (Value)
    {
      if (FTimer && FTimer->Owner == this)
        delete FTimer;
      FTimer = Value;
      FTimer->FreeNotification(this);
    }
    else // NULL 値
    {
      if (FTimer && FTimer->Owner != this)
      {
        FTimer = new ExtCtrls::TTimer(this);
        FTimer.SetSubComponent(true);
        FTimer->FreeNotification(this);
      }
    }
  }
}
```

上記のプロパティセッターは，プロパティ値として設定されているコンポーネントの FreeNotification メソッドを呼び出しています。この呼び出しによって，プロパティの値であるコンポーネントが破棄される場合に通知を送信するようになります。コンポーネントは，Notification メソッドを呼び出すことでこの通知を送信します。この呼び出しを処理するには，次のように Notification メソッドをオーバーライドします。

```
void __fastcall TDemoComponent::Notification(Classes::TComponent *AComponent,
    Classes::TOperation Operation)
{
  TComponent::Notification(AComponent, Operation); { 継承したメソッドを呼び出す }
  if ((Operation == opRemove) && (AComponent == (TComponent *)FTimer))
    FTimer = NULL;
}
```

プロパティの保存と読み込み

C++Builder では，フォームおよびフォーム内のコンポーネントはフォームファイル（VCL は .dfm ファイル，CLX は .xfm ファイル）に保存されます。フォームファイルには，フォームとそのコンポーネントのプロパティが保存されています。コンポーネントを新しく作成する場合，別の開発者がフォームにそのコンポーネントを追加し，保存するときに，コンポーネントがそのプロパティをフォームファイルに書き込むように設定しておかなければなりません。同様に，C++Builder に読み込まれるときや，アプリケーションの一部として実行されるときには，コンポーネントは自身をフォームファイルから復元します。

コンポーネントは自分自身の保存と読み込みの機能を継承しているので，ほとんどの場合，フォームファイルを操作するコンポーネントに対して何かをする必要はありません。しかし，コンポーネントが自身を保存する方法や，読み込まれたときの初期化方法を変更したい場合があります。そのためには，基礎的なメカニズムについて理解する必要があります。

プロパティの保存と読み込みについて以下のことを説明します。

- 保存と読み込みのメカニズムの使用
- デフォルト値の指定
- 保存対象の決定
- 読み込み後の初期化
- パブリッシュ以外のプロパティの保存と読み込み

保存と読み込みのメカニズムの使用

フォームの記述には，フォームのプロパティのリストと，フォーム内の各コンポーネントの同様の記述があります。フォームやコンポーネントは，自身の記述の保存と読み込みを行う機能を備えています。

デフォルトでは，コンポーネントが自身の記述を保存する場合は，パブリッシュに宣言されているプロパティのうち，値がデフォルトと異なるものを宣言順に書き込みます。コンポーネントが自身の記述を読み込む場合は，まず自身を作成し，すべてのプロパティをデフォルト値に設定してから，保存されている（デフォルト値ではない）プロパティ値を読み込みます。

このデフォルトのメカニズムは大半のコンポーネントの必要を満たすため，通常，コンポーネント開発者が操作を行う必要はありません。しかし，特定のコンポーネントの保存と読み込みの処理をカスタマイズする方法もいくつか用意されています。

デフォルト値の指定

C++Builder コンポーネントは，デフォルトに指定された値と異なっているプロパティだけを保存します。特に指定しなければ，C++Builder はプロパティがデフォルト値を持たないとみなし，コンポーネントがそのプロパティを値にかかわらず常に保存します。

プロパティのデフォルト値を指定する手順は次のとおりです。

1. プロパティ名の後に等号（=）を追加します。
2. 等号の後に中カッコ（{}）を追加します。
3. 中カッコ内にキーワード **default** を記述し，もう 1 つ等号を続けます。
4. 新しいデフォルト値を入力します。

たとえば次のようにします。

```
__property Alignment = {default=taCenter};
```

プロパティを再宣言するときも，デフォルト値を指定できます。実際，異なったデフォルト値を指定するためにプロパティを再宣言することもよくあります。

メモ デフォルト値を指定しても，オブジェクト作成時にプロパティがその値で初期化されるわけではありません。特別な場合を除いて，コンポーネントのコンストラクタでプロパティの値をデフォルト値で初期化するコードを追加する必要があります。オブジェクトを初期化すると，コンポーネントのコンストラクタで値が設定されないプロパティは 0 の値に設定されます。0 の値とは，格納メモリが 0 に設定された場合にそのプロパティがとる値のことです。したがって，数値を 0 に，論理値を **false** に，ポインタを NULL に，といったように設定する場合にはコードの追加は必要ありません。この値の解釈に自信が持てない場合は，コンストラクタのメソッドで値を代入してください。

次のコード例では，コンポーネント宣言で Align プロパティにデフォルト値を指定し，コンポーネントコンストラクタでそのデフォルト値を実際に代入しています。この場合，新しいコンポーネントは，標準パネルコンポーネントを変更して，ウィンドウのステータスバーとして使用します。そのため，ウィンドウのデフォルトの配置をオーナーの下部として指定します。

```
class PACKAGE TMyStatusBar : public TPanel
{
public:
  virtual __fastcall TMyStatusBar(TComponent* AOwner);
__published:
  __property Align = {default=alBottom};
};
```

TMyStatusBar コンポーネントのコンストラクタは .CPP ファイルに記述されています。

```
__fastcall TMyStatusBar::TMyStatusBar (TComponent* AOwner) : TPanel(AOwner)
{
   Align = alBottom;
}
```

保存対象の決定

C++Builder では，コンポーネントのプロパティのそれぞれについて，保存するかどうかを制御できます。デフォルトでは，クラス宣言の **published** 部のすべてのプロパティが保存されます。ただし，特定のプロパティを保存しないように指定したり，プロパティを保存するかどうかを動的に決定する関数を指定することができます。

C++Builder でプロパティを保存するかどうかを制御する手順は次のとおりです。

1. プロパティ名の後に等号（=）を追加します。
2. 等号の後に中カッコ（{}）を追加します。
3. 中カッコ内にキーワード **stored** を記述し，その後に **true**，**false**，または論理関数の名前を記述します。

次のコード例では，コンポーネントで 3 つの新しいプロパティを宣言しています。最初のプロパティは必ず保存され，2 番目のプロパティは常に保存されず，3 番目のプロパティは関数が返す論理値に応じて保存されます。

```
class PACKAGE TSampleComponent : public TComponent
{
protected:
  bool __fastcall StoreIt();
public:
   ...
__published:
  __property int Important = {stored=true};          // 常に保存される
  __property int Unimportant = {stored=false};       // 保存されない
  __property int Sometimes = {stored=StoreIt};       // 関数の戻り値に応じて保存される
};
```

読み込み後の初期化

コンポーネントは，保存されている記述からすべてのプロパティを読み込んだのち，必要な初期化を実行する仮想メソッド Loaded を呼び出します。Loaded はフォームやコントロールが表示される前に呼び出されるので，初期化処理によって画面にちらつきが生じることはありません。

プロパティが読み込まれた後でコンポーネントを初期化する場合には，Loaded メソッドをオーバーライドします。

メモ 新しい Loaded メソッドでは，まず継承した Loaded メソッドを呼び出します。そうすれば，独自のコンポーネントの初期化を行う前に，すべての継承プロパティを適切に初期化できます。

パブリッシュ以外のプロパティの保存と読み込み

デフォルトでは，パブリッシュプロパティだけがコンポーネントと一緒に保存され，読み込まれます。しかし，パブリッシュに設定されていないプロパティの保存と読み込みもできます。その場合，オブジェクトインスペクタに表示されない持続的プロパティを持つことが可能になります。また，プロパティ値が複雑であるために C++Builder が読み書きの方法を知らない場合でも，コンポーネントはプロパティ値の保存と読み込みを行うことができます。たとえば TStrings オブジェクトが自身の文字列を読み書きする場合，C++Builder による自動動作には依存できず，以下のメカニズムを使わなければなりません。

パブリッシュ以外のプロパティを保存するには，プロパティ値の読み込み方法と保存方法を C++Builder に伝えるコードを記述します。

プロパティの読み込みと保存を行う独自のコードを記述する手順は次のとおりです。

1. プロパティ値を保存するメソッドと読み込むメソッドを作成します。
2. そのメソッドをファイラオブジェクトに渡して，DefineProperties メソッドをオーバーライドします。

プロパティ値の保存と読み込みを行うメソッドを作成する

パブリッシュに設定されていないプロパティの保存と読み込みをするには，まず，プロパティ値を保存するメソッドと，プロパティ値を読み込むメソッドを作成する必要があります。プロパティ値の保存と読み込みを行うメソッドは，次の 2 通りの方法で作成できます。

- プロパティ値の保存には TWriterProc 型のメソッド，プロパティ値の読み込みには TReaderProc 型のメソッドをそれぞれ作成する。この方法を使うと，単純型を読み書きする C++Builder 組み込みの機能を利用できる。C++Builder 側で読み書きの方法が分かっている型をもとにプロパティ値を指定した場合は，この方法を使用する
- TStreamProc 型のメソッドを 2 つ作成する（プロパティ値を保存するメソッドと，プロパティ値を読み込むメソッド）。TStreamProc はストリームを引数に取るため，そのストリームのメソッドを使ってプロパティ値の読み書きができる

例として，実行時に作成されるコンポーネントを表すプロパティを考えてみます。C++Builder はこのプロパティ値を書き込む方法を知っていますが，コンポーネントの作成場所がフォームデザイナでないため，自動的に書き込むことはしません。ストリームシステムによりすでにコンポーネントの読み込みと保存ができるので，1 番目の方法を使用できます。以下に示すメソッドは，MyCompProperty プロパティの値として動的に作成されるコンポーネントの読み込みと保存を行っています。

```
void __fastcall TSampleComponent::LoadCompProperty(TReader *Reader)
{
  if (Reader->ReadBoolean())
    MyCompProperty = Reader->ReadComponent(NULL);
}
void __fastcall TSampleComponent::StoreCompProperty(TWriter *Writer)
{
  if (MyCompProperty)
  {
    Writer->WriteBoolean(true);
    Writer->WriteComponent(MyCompProperty);
  }
  else
    Writer->WriteBoolean(false);
}
```

DefineProperties メソッドのオーバーライド

プロパティ値の保存と読み込みを行うメソッドを作成したら，コンポーネントの DefineProperties メソッドをオーバーライドできます。C++Builder は，コンポーネントの読み込み時と保存時に DefineProperties メソッドを呼び出します。DefineProperties メソッドで，現在のファイラの DefineProperty メソッドか DefineBinaryProperty メソッドを呼び出し，そのメソッドに対して，プロパティ値の保存または読み込みに使用するメソッドを渡します。TWriterProc 型と TReaderProc 型の読み込みメソッドと保存メソッドを作成した場合は，ファイラの DefineProperty メソッドを呼び出します。TStreamProc 型のメソッドを作成した場合は，DefineBinaryProperty を呼び出します。

どちらのメソッドでプロパティを定義するにせよ，そのメソッドに対し，プロパティ値の保存と読み込みを行うメソッドと，プロパティ値を書き込む必要があるかどうかを示す論理値を渡します。プロパティ値が継承可能であるか，またはプロパティ値がデフォルト値を持っていれば，書き込む必要はありません。

たとえば，TReaderProc 型の LoadCompProperty メソッドと TWriterProc 型の StoreCompProperty メソッドを作成したとすると，次の例のように DefineProperties をオーバーライドします。

```
void __fastcall TSampleComponent::DefineProperties(TFiler *Filer)
{
  // 何かを行う前に，基本クラスにプロパティを宣言させる
  // この例では，TSampleComponent が TComponent から直接派生しているものとする
  TComponent::DefineProperties(Filer);
  bool WriteValue;
  if (Filer->Ancestor) // 継承した値かどうかをチェック
  {
    if ((TSampleComponent *)Filer->Ancestor)->MyCompProperty == NULL)
      WriteValue = (MyCompProperty != NULL);
    else if ((MyCompProperty == NULL) ||
             (((TSampleComponent *)Filer->Ancestor)->MyCompProperty->Name !=
                                                    MyCompProperty->Name))
      WriteValue = true;
    else WriteValue = false;
  }
  else // 継承した値ではない。ヌルでなければプロパティに書き込む
    WriteValue = (MyCompProperty != NULL);
  Filer->DefineProperty("MyCompProperty ",LoadCompProperty,StoreCompProperty, WriteValue);
}
```

第48章

イベントの作成

イベントとは，ユーザーアクションやフォーカスの変更などのシステムの出来事と，その出来事に応答するコードとの間の関連付けのことです。この応答コードがイベントハンドラで，ほとんどの場合アプリケーション開発者によって記述されます。アプリケーション開発者は，イベントを使ってクラスそのものを変更することなくコンポーネントの動作をカスタマイズできます。これを委任といいます。

マウスアクションなど，よく発生するユーザーアクションに対するイベントは，すべての標準コンポーネントに組み込まれています。しかし，独自のイベントも作成できます。このセクションでは，コンポーネントのイベントの作成について以下のことを説明します。

- イベントとはなにか
- 標準イベントの実装
- 独自のイベントの定義

イベントはプロパティとして実装されます。このため，コンポーネントのイベントを作成 / 変更するには，第 47 章「プロパティの作成」にまず目を通してください。

イベントとはなにか

イベントとは，ある出来事とコードを関連付けるメカニズムのことです。より具体的には，イベントはクロージャです。

アプリケーション開発者にとってのイベントは，システムの出来事に関連付けられた OnClick などの名前で示され，特定のコードを結び付けられます。たとえば，Button1 という名前の押しボタンは，OnClick イベントを伴っています。デフォルトでは C++Builder は，このボタンのあるフォームに Button1Click という名前のイベントハンドラを生成し，OnClick に関連付けます。このボタンでクリックイベントが発生すると，ボタンは OnClick のメソッド，つまりこの場合は Button1Click を呼び出します。

イベントを記述するためには，以下のことについて知っておく必要があります。

- イベントはクロージャである
- イベントはプロパティである
- イベントの型はクロージャ型である
- イベントハンドラの戻り型は **void** である
- イベントハンドラはオプションである

イベントはクロージャである

C++Builder は，クロージャを使ってイベントを実装します。クロージャとは，特定のクラスインスタンスの特定のメソッドを指すポインタ型です。コンポーネント開発者はクロージャをプレースホルダとして扱えます。つまり，コンポーネントのコードがイベントを検出した場合，ユーザーがそのイベントに対して指定しているメソッドがあれば，そのメソッドを呼び出します。

クロージャはクラスインスタンスへの隠されたポインタを保持しています。ユーザーがコンポーネントのイベントにハンドラを指定した場合，この指定は特定のクラスインスタンスの特定のメソッドに対する指定であって，単に特定の名前のメソッドに対して指定したわけではありません。このインスタンスは通常，そのコンポーネントが所属するフォームですが，そうでない場合もあります。

たとえば，すべてのコントロールは，クリックイベントを処理する仮想メソッド Click を継承します。

```
virtual void __fastcall Click(void);
```

Click は，ユーザーのクリックイベントハンドラが存在する場合はそれを呼び出すように実装されています。コントロールがクリックされると，ユーザーがコントロールの OnClick イベントにハンドラを指定している場合には，そのメソッドが呼び出されます。ハンドラを指定していなかった場合には何も起こりません。

イベントはプロパティである

コンポーネントはプロパティによってイベントを実装します。ほかのプロパティとは異なり，イベントはメソッドを使用して **read** 部と **write** 部を実装するわけではありません。その代わりに，イベントプロパティは同じ型のプライベートなデータメンバーをプロパティとして使用します。

このデータメンバーの名前は，通常，プロパティの名前の先頭に文字 F を加えた名前と同じです。たとえば，OnClick クロージャは TNotifyEvent 型の FOnClick という名前のデータメンバーに格納されます。OnClick イベントプロパティの宣言の例を次に示します。

```
class PACKAGE TControl : public TComponent
{
private:
    TNotifyEvent FOnClick;
  ...
protected:
    __property TNotifyEvent OnClick = {read=FOnClick, write=FOnClick};
  ...
};
```

TNotifyEvent とその他のイベント型については，次の節「イベントの型はクロージャ型である」を参照してください。

ほかのプロパティの場合と同じように，イベントの値は実行時に設定または変更できます。ただし，イベントをプロパティとする主要な利点は，コンポーネントユーザーが設計時にオブジェクトインスペクタでイベントにハンドラを指定できることです。

イベントの型はクロージャ型である

イベントはイベントハンドラに対するポインタなので，イベントプロパティの型はクロージャ型でなければなりません。同様に，イベントハンドラとして使用されるすべてのコードは，適切な型を持つクラスメソッドでなければなりません。

互換性のため，イベントハンドラのメソッドはデフォルトのイベントハンドラと同じ数，同じ型のパラメータを，同じ順序，同じ方法で受け取らなければなりません。

C++Builder はすべての標準イベントに対してクロージャ型を定義しています。独自のイベントを作成する場合は，既存のクロージャが適切ならばそれを使用できます。そうでなければ独自の型を定義します。

イベントハンドラの戻り型は void である

イベントハンドラ の戻り値として許されるのは **void** のみです。イベントハンドラの戻り型が **void** であっても，引数を参照で渡せばユーザーのコードから情報を取得できます。この場合は，必ず引数に有効な値を代入してからそのハンドラを呼び出すようにしてください。そうしないと，ユーザーが常に値を設定しなければなりません。

TKeyPressEvent 型のキーを押すイベントでは，参照渡しの引数を使用します。次のように TKeyPressEvent では，どのオブジェクトがそのイベントを生成したかを示すパラメータと，どのキーが押されたかを示す引数の 2 つが定義されています。

```
typedef void __fastcall (__closure *TKeyPressEvent)(TObject *Sender, Char &Key);
```

通常，Key パラメータはユーザーが押した文字を示します。ただし，場合によってはコンポーネントユーザーがその文字を変更することがあります。たとえば，編集コントロールの入力ですべての文字を大文字に変換する場合です。それには，次のようなキー入力用のハンドラを定義できます。

```
void __fastcall TForm1::Edit1KeyPress(TObject *Sender, Char &Key)
{
  Key = UpCase(Key);
}
```

また参照渡しの引数によってユーザーがデフォルトの処理をオーバーライドできるように設計することも可能です。

イベントハンドラはオプションである

イベントを作成する場合は，コンポーネントを使う開発者がイベントにハンドラを結び付けないことがあるので注意してください。つまり，特定のイベントにハンドラが結び付けられていなくても，それだけでコンポーネントが障害を起こしたりエラーを出したりしないようにします（ハンドラの呼び出しと，ハンドラが結び付けられていないイベントを処理するための機構については，48-8 ページの「イベントの呼び出し」を参照してください）。

GUI アプリケーションでは常にイベントが発生しています。ビジュアルコンポーネントを横切ってマウスポインタを移動させるだけで多くのマウス移動メッセージを送り出され，コンポーネントはそのメッセージを OnMouseMove イベントに変換します。しかし多くの場合，開発者はマウス移動イベントを処理したくはなく，それで問題が生じてはなりません。したがって，作成するコンポーネントでは，イベント用のハンドラを必要としてはなりません。

さらに，アプリケーション開発者がイベントハンドラに記述できるコードに制約はありません。VCL/CLX のコンポーネントのイベントハンドラは，エラーが発生するような事態がなるべく起きないように記述されています。確かに，アプリケーションコードの論理エラーは回避できません。しかし，イベントが呼び出される前にデータ構造が初期化されるように記述し，アプリケーション開発者が無効なデータを参照しないように配慮することは可能です。

標準イベントの実装

C++Builder のコントロールは，よく起こる出来事に対するイベントを継承しています。そのようなイベントを標準イベントと呼びます。標準イベントのすべてはコントロールに組み込まれていますが，標準イベントは通常は **protected** になっているので，開発者は標準イベントにハンドラを結び付けられません。しかしコントロールを作成する場合は，そのコントロールのユーザーがアクセスできるようにすることは可能です。

標準イベントをコントロールで処理する場合は次の 3 点について考慮します。

- 標準イベントとは
- イベントの公開
- 標準イベントの処理方法の変更

標準イベントとは

標準イベントには，すべてのコントロール用に定義されたイベントと標準ウィンドウコントロールに対してのみ定義されているイベントの 2 種類があります。

すべてのコントロール用の標準イベント

最も基本的なイベントは，TControl クラスで定義されています。ウィンドウ，グラフィック，カスタムのどの種類のコントロールも，そのイベントを継承します。以下のイベントは，すべてのコントロールで使用できます。

OnClick	OnDragDrop	OnEndDrag	OnMouseMove
OnDblClick	OnDragOver	OnMouseDown	OnMouseUp

標準イベントに対応する仮想プロテクトメソッドが TControl に宣言されており，各仮想メソッド名はイベント名に対応しています。たとえば，OnClick イベントは Click というメソッドを呼び出し，OnEndDrag イベントは DoEndDrag というメソッドを呼び出します。

標準コントロール用の標準イベント

標準ウィンドウコントロールは，すべてのコントロールに共通のイベントに加えて，以下に示すイベントを処理できます。標準ウィンドウコントロールは TWinControl（VCL）または TWidgetControl（CLX）から派生したものです。

OnEnter	OnKeyDown	OnKeyPress
OnKeyUp	OnExit	

TControl の標準コントロールと同様に，ウィンドウコントロールイベントにも対応するメソッドがあります。上に挙げた標準のキーイベントは，通常のキーストロークに応答します。

VCL 【Alt】キーなどの特殊なキーストロークに応答するには，Windows からの WM_GETDLGCODE または CM_WANTSPECIALKEYS メッセージに応答しなければなりません。メッセージハンドラの作成についての詳細は，第 51 章「メッセージとシステム通知の処理」を参照してください。

イベントの公開

TControl と TWinControl（CLX の場合は TWidgetControl）での標準イベントの宣言は，それに対応するメソッドと同様に **protected** になっています。これらの抽象クラスから継承していて実行時または設計時に標準イベントにアクセスできるようにするには，そのイベントを **public** または **published** として再宣言します。

プロパティを実装の指定なしで再宣言すると，メソッドの保護レベル（アクセス制御）だけが変更されます。この方法で，TControl で定義されているが表示されないイベントを **public** または **published** として宣言し，アクセスできるようにします。

コンポーネントの OnClick イベントを設計時にアクセスできるようにするには，コンポーネントのクラス宣言に次のコードを記述します。

```
class PACKAGE TMyControl : public TCustomControl
{
  ...
__published:
    __property OnClick;            // オブジェクトインスペクタで使用可能な OnClick を使用する
};
```

標準イベントの処理方法の変更

コンポーネントが一定の種類のイベントに応答する方法を変更する場合は，コードを記述してそのイベントに割り当てたいでしょう。それこそがアプリケーション開発者の仕事です。しかし，コンポーネントの作成中は，そのコンポーネントを使う開発者がそのイベントを使えるようにしておかなければなりません。

標準イベントのそれぞれがプロテクトメソッドで実装されているのはこのためです。このメソッドをオーバーライドすれば，内部のイベント処理方法を変更できます。そこで継承したメソッドを呼び出せば，アプリケーション開発者のコードで記述されたイベントと標準イベント処理を実行できます。

メソッドを呼び出す順序が重要です。一般には，継承したメソッドを最初に呼び出して，カスタマイズの前に（また，ときにはカスタマイズが実行されないように）アプリケーション開発者のイベントハンドラを実行します。ただし，継承メソッドを呼び出す前にカスタマイズ処理を実行したい場合もあります。たとえば，継承したコードがコンポーネントの状態に依存している場合に，カスタマイズしたコードで状態をあらかじめ変更しておくことがあります。このような場合は，状態を変更した後にユーザーコードを実行して，ユーザーコードが変更済みの状態を参照できるようにします。

開発中のコンポーネントで，クリックに応答する方法を変更するとします。アプリケーションを開発する場合の方法で OnClick イベントにハンドラを結び付けるかわりに，次のように Click プロテクトメソッドをオーバーライドします。

```
void __fastcall TMyControl::Click()
{
  TWinControl::Click();         // ハンドラの呼び出しなど標準的な処理を実行
  // ここにコンポーネント開発者のカスタマイズコードを記述
}
```

独自のイベントの定義

まったく新しいイベントを定義することは，それほど多くはありません。しかし，あるコンポーネントにほかのコンポーネントとはまったく異なる動作を与える場合もあり，その場合にはコンポーネントのイベントを新しく定義します。

イベントを定義するときに考慮すべき事項は以下のとおりです。

- イベントの発生
- ハンドラ型の定義
- イベントの宣言
- イベントの呼び出し

イベントの発生

まず，何がイベントを発生させるのかという問題があります。イベントによっては答えは明白です。たとえば，マウスボタンが押されたことを示すイベントは，ユーザーがマウスの左ボタンを押して，Windows がアプリケーションに WM_LBUTTONDOWN メッセージを送ったときに発生します。コンポーネントはこのメッセージを受け取ると MouseDown メソッドを呼び出します。MouseDown メソッドは，ユーザーが OnMouseDown イベントに結び付けたコードを呼び出します。

しかし，それほど明確に外部の出来事と結び付いていないイベントもあります。たとえば，スクロールバーの OnChange イベントは，キー入力，マウスクリック，またはほかのコントロールの変更などさまざまな出来事によって発生します。イベントを定義する際には，それぞれの出来事が適切なイベントを呼び出すように配慮します。

CLX　CLX アプリケーションについては，51-10 ページの「CLX によるシステム通知への応答」を参照してください。

2 種類のイベント

ユーザーの操作と状態の変更という 2 種類の出来事に対してイベントを用意します。ユーザー操作によるイベントは，Windows からのメッセージによって発生します。このイベントは多くの場合，コンポーネントが応答しなければならないようなユーザー操作が行われたことを示します。これに対して，状態の変更によるイベントは，フォーカスの変更のような Windows からのメッセージによっても発生しますが，プロパティやコードによる変更によっても発生します。

どのようなイベントを定義するかは，全面的に開発者にまかされています。開発者がイベントの使用方法をよく理解できるようにイベントを定義してください。

ハンドラ型の定義

イベントの発生方法について決定したら，次にそのイベントの処理方法を定義します。つまり，イベントハンドラの型を決定します。多くの場合，開発者が定義したイベントのハンドラは，簡単な通知を行うものかイベント独自の型を持つものです。また，ハンドラから情報を取得することもできます。

単純な通知

通知イベントとは，特定のイベントが発生したことを告げるだけのイベントで，発生した時期や場所についての具体的な情報は伴いません。通知イベントには TNotifyEvent 型を使用します。
TNotifyEvent は，唯一のパラメータとしてイベントの発生元を持つだけです。このため，通知イベントのハンドラがそのイベントから得られる情報は，発生したイベントの種類と，そのイベントはどのコンポーネントで発生したのかということだけです。たとえば，クリックイベントは通知イベントです。クリックイベントのハンドラで利用できる情報は，クリックが発生したということと，どのコンポーネントがクリックされたかということだけです。

通知は一方通行です。たとえば，フィードバックを提供したり，通知イベントのそれ以上の処理を防止させるようなメカニズムは備えていません。

イベント独自のハンドラ

場合によっては，どのイベントがどのコンポーネントに対して発生したかという情報では十分ではないことがあります。たとえば，キーが押されたことを示すイベントの場合，おそらくハンドラはユーザーがどのキーを押したのか知りたいはずです。このような場合には，パラメータとして必要な追加情報を含めるハンドラが必要です。

メッセージによってイベントが発生した場合には，多くの場合，メッセージのパラメータがそのままイベントのハンドラに渡されます。

ハンドラからの情報を返す方法

すべてのイベントハンドラの戻り型は void なので，ハンドラから情報を返す場合は必ず参照渡しの引数を使用します。ユーザーのコンポーネントは，ユーザーのハンドラを実行した後，返された情報によって，イベントを処理する方法やイベントを処理するかどうかを決定できます。

たとえば，すべてのキーイベント（OnKeyDown，OnKeyUp，OnKeyPress）は，Key パラメータにキーを示す値を参照渡しします。イベントハンドラで Key を変更すると，アプリケーションはその変更された値を受け取ります。このような方法で，たとえば入力された文字を大文字に変換できます。

イベントの宣言

イベントハンドラの型を決定したら，次にそのイベントのクロージャとプロパティを宣言します。ユーザーにわかりやすいように，イベントにはその機能を表すような説明的な名前を付けます。なるべく，同じようなイベントには一貫性のある名前を付けます。

「On」で始まるイベント名

C++Builder の標準イベントの名前はほとんど「On」で始まります。これは単なる慣習で，コンパイラの規定ではありません。オブジェクトインスペクタは，プロパティの型によってそのプロパティがイベントかどうかを判断します。すべてのクロージャプロパティはイベントとみなされ，［イベント］ページに表示されます。

ただし，開発者はイベントに「On」で始まるアルファベットの名前が付いていることに慣れています。それ以外の名前を使用すると混乱する可能性があります。

メモ この規則の例外として，なんらかの出来事の前および後に発生するイベントの多くが「Before」および「After」で始まっています。

イベントの呼び出し

イベントの呼び出しはできるだけ集中させます。つまり，仮想メソッドを 1 つ作成して，アプリケーションのイベントハンドラが存在する場合はそれを呼び出し，必要なデフォルトの処理を行います。

イベントの呼び出しが 1 箇所で行われていれば，開発したコンポーネントから新しいコンポーネントを派生させる場合，イベントを呼び出す場所を探す負担が軽減されます。1 つのメソッドをオーバーライドすれば，イベントの処理をカスタマイズできるからです。

イベントを呼び出す場合は，次の 2 つのことについても考慮します。

- 空のハンドラも有効であること
- ユーザーがオーバーライドできるデフォルトの処理方法

空のハンドラも有効であること

空のイベントハンドラによってエラーが発生することのないようにしてください。また，コンポーネントの動作がアプリケーションのイベント処理コードからの特定の応答に依存しないようにしてください。

空のハンドラは，ハンドラがない場合と同じ結果をもたらすようにします。アプリケーションのイベントハンドラを呼び出すコードは，次のようになります。

```
if (OnClick)
  OnClick(this);
// デフォルトの処理を実行
```

次のようなコードは記述しないでください。

```
if (OnClick)
  OnClick(this);
else
  // デフォルトの処理を実行
```

ユーザーがオーバーライドできるデフォルトの処理方法

ある種のイベントでは，開発者がデフォルトの処理方法を置き換えたり，すべての応答を抑止する場合があります。それを可能にするためには，ハンドラに引数を参照で渡して，ハンドラから返された値を検査します。

この方法は，空のハンドラはハンドラがない場合と同じ結果をもたらすようにしなければならないという前節の規則と矛盾しません。空のハンドラでは参照によって渡された引数の値は変更されないので，空のハンドラが呼び出された後は，常にデフォルトの処理が行われます。

キーが押されたことを示すイベントの処理では，ユーザーは Key パラメータにヌル文字を設定することで，コンポーネントのデフォルトのキー入力処理を抑止できます。これを実現するコードは次のとおりです。

```
if (OnKeyPress)
  OnKeyPress(this, &Key);
if (Key != NULL)
  // デフォルトの処理を実行
```

Windows メッセージを処理する必要があるため，実際のコードはこれとは少し異なりますが，基本的な論理は同じです。デフォルトでは，コンポーネントはユーザーが割り当てたハンドラを呼び出してから，標準の処理を実行します。ユーザーのハンドラが Key にヌル文字を設定した場合には，コンポーネントはそのデフォルト処理を行いません。

第49章

メソッドの作成

コンポーネントメソッドは，他のクラスのメソッドと同じで，コンポーネントクラスの構造に組み込まれたメンバー関数です。基本的には，コンポーネントメソッドで行えることには何の制約もありません。ただし，C++Builder では標準的なメソッドを作成するための方法がいくつか定められています。このセクションでは，以下のことを説明します。

- 依存を避ける
- メソッドの命名
- メソッドの保護の設定
- 仮想メソッドの作成
- メソッドの宣言

一般に，コンポーネント内のメソッドの数も，アプリケーションで呼び出さなければならないメソッドの数もできるだけ少なくします。また，開発者がメソッドとして実装しようと考える機能の多くは，たいていの場合プロパティにカプセル化できます。プロパティを使用すれば C++Builder 環境に適したインターフェースが作成され，設計時にプロパティを操作できます。

依存を避ける

コンポーネントを使用する際の前提条件を少なくして，開発者の負担を最小限にします。開発者がコンポーネントに対して，いつでもどのようなことでも行えるようにコンポーネントを設計します。この原則に従えない場合もありますが，できるだけこの原則に近づくことを目標とします。

以下に，避けた方がよい依存とはどのようなものかを示します。

- ユーザーがコンポーネントを使用するときに必ず呼び出さなければならないメソッド
- 特定の順序で実行しなければならないメソッド
- コンポーネントのあるイベントやメソッドが無効になる状態やモードを招くメソッド

相互依存がある場合，不適切な状態から確実に抜け出せる方法を用意します。たとえば，あるメソッドの呼び出しがコンポーネントの状態を変化させ，その状態ではほかのメソッドの呼び出しが無効と

なるような場合には，アプリケーションが呼び出したときにメソッドがメインコードを実行する前に状態を修正するための別のメソッドを用意しておきます。少なくとも，ユーザーが無効なメソッドを呼び出した場合に例外が生成されるようにしておきます。

つまり，コードの各部分が相互に依存する場合には，そのコードが間違った方法で使用されても，問題が生じないように配慮します。ユーザーがコンポーネントの依存関係に配慮しなかった場合でも，システム障害にするのではなく警告メッセージを表示するなどの方法を採ります。

メソッドの命名

C++Builder では，メソッドやパラメータに付ける名前に制約はありません。しかし，いくつかの命名規則に従えば，アプリケーション開発者にとってわかりやすいメソッドを作成できます。開発したコンポーネントが広く使用されるかどうかは，コンポーネントの設計にかかっています。

自分自身や小さなグループ内で使用するコードを開発する場合は，命名方法についてはそれほど考慮しないかもしれません。しかし，そのコードに不案内な，またはそもそもコーディング自体に不案内なユーザーによって使用されるコンポーネントを開発する場合は，メソッドに明確なわかりやすい名前を付けることが大切です。

メソッドにわかりやすい名前を付ける方法を以下に示します。

- 説明的な名前にする。意味を表す動詞を使う

  PasteFromClipboard のような名前は，Paste や PFC のような短い名前よりもはるかにわかりやすい名前です。

- 関数には返す値を表す名前を付ける

  プログラマ本人にとっては，X という名前の関数が水平方向の位置を返すことは明白かもしれませんが，GetHorizontalPosition のような名前にすれば，すべてのユーザーが理解できます。

- 関数の戻り型が **void** の場合，関数には能動態の動詞の名前を付ける

  関数名には意味の明確な能動態の動詞を使うとわかりやすい名前になります。たとえば，DoFiles よりも ReadFileNames という名前の方が動作を予測できます。

最後に，そのメソッドが本当に必要かどうか再考します。メソッド名に動詞が入っているかどうかで，その判断が可能です。名前に動詞が入っていないメソッドを数多く作成した場合は，そのメソッドをプロパティで実現できないかどうか考慮します。

メソッドの保護の設定

データメンバー，メソッド，プロパティなどを含め，クラスの各部分には保護レベル，つまり「可視性」を設定できます。この点については，46-4 ページの「アクセスの制御」で説明しています。あるメソッドに対して可視性を選択するのは容易です。

コンポーネントに記述するほとんどのメソッドは，**public** または **protected** です。メソッドを **private** にすることはほとんどありません。**private** にするのは，メソッドがそのコンポーネントに特有のもので，派生したコンポーネントからのアクセスさえ禁止するような場合です。

メモ 一般的に，イベントハンドラ以外のメソッドを **__published** で定義する意味はありません。こうしてもユーザーにとってはそのメソッドが **public** であるのと同じことです。

public にするメソッド

アプリケーション開発者が呼び出すメソッドはすべて **public** として宣言します。メソッド呼び出しの多くはイベントハンドラで発生します。したがって，メソッドがシステムリソースを独占したり，オペレーティングシステムをユーザーに応答できないような状態にすることは適切ではありません。

メモ コンストラクタとデストラクタは，常に **public** にします。

protected にするメソッド

コンポーネントの内部で使用するメソッドは，すべて **protected** にします。そうすれば，アプリケーションが誤ってそのメソッドを呼び出すことを防止できます。**protected** として宣言したメソッドは，アプリケーションコードからは呼び出せませんが，派生クラスからは呼び出せます。

たとえば，一定のデータが設定されていることを前提とするメソッドがあるとします。メソッドを **public** に設定した場合は，データが設定される前にアプリケーションがそのメソッドを呼び出す可能性があります。これに対して，メソッドを **protected** に設定した場合は，アプリケーションはそのメソッドを直接呼び出せません。特定の状態を前提とするメソッドはプロテクトとして宣言し，ほかの **public** メソッドであらかじめデータを設定してからその **protected** メソッドを呼び出すようにします。

プロパティを実装するメソッドは，仮想 **protected** メソッドとして宣言します。メソッドをそのように宣言すれば，アプリケーション開発者はプロパティの実装をオーバーライドして，その機能の強化または全面的な置き換えができます。このようなプロパティには，非常に多態性があります。アクセスメソッドを **protected** に保てば，開発者が誤ってアクセスメソッドを呼び出し，プロパティを変更してしまうことはありません。

仮想メソッドの作成

同一のメソッド呼び出しによって，さまざまな型に固有のコードを実行させたいときには，メソッドを **virtual** にします。

アプリケーション開発者によって直接使用されるコンポーネントでは，たいていの場合すべてのメソッドは非仮想メソッドとして実装できます。しかし，ほかのコンポーネントの派生元として使用される抽象コンポーネントでは，追加メソッドを **virtual** メソッドとして実装することを考慮します。この方法では，派生したコンポーネントで，継承した **virtual** メソッドをオーバーライドできます。

メソッドの宣言

コンポーネントメソッドの宣言方法は，一般のクラスメソッドの宣言方法と同じです。

コンポーネントに新しいメソッドを宣言する手順は次のとおりです。

- コンポーネントのヘッダーファイルのコンポーネントクラス宣言に宣言を追加する
- ユニットの .CPP ファイルにメソッドを実装するコードを記述する。

2 つの新しいメソッド（**protected** メソッドと仮想 **public** メソッド）を定義するコンポーネントのコード例を次に示します。これは .H ファイルの中のインターフェース定義です。

```
class PACKAGE TSampleComponent : public TControl
{
protected:
    void __fastcall MakeBigger();
public:
    virtual int __fastcall CalculateArea();
  ...
};
```

メソッドを実装するユニットの .CPP ファイルのコードは次のとおりです。

```
void __fastcall TSampleComponent::MakeBigger()
{
  Height = Height + 5;
  Width = Width + 5;
}
int __fastcall TSampleComponent::CalculateArea()
{
  return Width * Height;
}
```

第50章

コンポーネントにおけるグラフィックの使い方

Windows では，デバイスに依存しないグラフィックを作成するために，強力なグラフィックデバイスインターフェース（GDI）が提供されています。ただし，プログラマが GDI を使用する場合は，グラフィックリソースの管理など数多くの作業が伴います。C++Builder は，GDI の使用に伴う退屈な作業をすべて行います。そのため開発者は，見失ったハンドルや未解放のリソースなどにわずらわされることもなく，生産的な作業に専念できます。

C++Builder アプリケーションは Windows API のどの GDI 関数でも直接呼び出せます。しかし，C++Builder はグラフィック関数をカプセル化し，より速くより簡単なグラフィック作成方法を提供します。

CLX　GDI は Windows 固有の関数なので，CLX アプリケーションおよびクロスプラットフォームアプリケーションには適用できません。ただし，CLX コンポーネントは Qt ライブラリを使用します。

このセクションでは，以下のことを説明します。

- グラフィックの概要
- キャンバスの使い方
- 画像の使い方
- オフスクリーンビットマップ
- 変更への応答

グラフィックの概要

C++Builder では，Windows GDI がいくつかのレベルでカプセル化されています。コンポーネント開発者にとって重要なのは，コンポーネントがどのようにして画面にイメージを表示するかということです。GDI 関数を直接呼び出す場合には，まずデバイスコンテキストのハンドルが必要です。デバイスコンテキストには，ペン，ブラシ，フォントなどの多様な描画ツールを選択しておきます。グラフィックイメージをレンダリングした後は，破棄する前にデバイスコンテキストを元の状態に復元しておかなければなりません。

C++Builder では，開発者が細かいグラフィック処理をする必要がなく，単純ですが完全な機能を備えたインターフェースである Canvas プロパティがコンポーネントに用意されています。キャンバスは，有効なデバイスコンテキストを用意し，使用後にはそのデバイスコンテキストを解放します。またキャンバスは，現在のペン，ブラシ，フォントを示す独自のプロパティを持っています。

キャンバスがすべてのリソースを開発者にかわって管理するので，開発者はペンハンドルなどの作成，選択，解放に気を使う必要がなくなります。使用するペンの種類をキャンバスに指示しておけば，その後の作業はキャンバスが処理します。

C++Builder はリソースを後の使用に備えてキャッシュに入れておくので，繰り返し行う処理のスピードが向上します。これも，グラフィックリソースの管理を C++Builder に任せることの効果です。たとえば，あるペンツールを繰り返し作成し，使用し，破棄するプログラムの場合，そのツールを使用するたびに同じ手順を繰り返します。一方，C++Builder では，繰り返し使用されるツールはキャッシュにある公算が高いため，C++Builder はツールを再作成するかわりに既存のツールを使います。

この例として，アプリケーションが数十個のフォームを開き，数百個のコントロールを持つ場合を想定します。これらのコントロールは，それぞれ 1 つまたは複数の TFont プロパティを持ちます。この場合，TFont オブジェクトのインスタンスが数百，数千と作成される可能性がありますが，大半のアプリケーションでは，フォントキャッシュを利用することにより 2 つか 3 つのフォントハンドルが使用されるだけで済みます。

以下の 2 つのコード例は，C++Builder のグラフィックコードがどれほど単純なものかを示しています。最初のコードは，他の開発ツールを使う場合と同様に，Windows API を直接使用して開発したアプリケーションです。標準の GDI 関数によって，青色の枠を持つ黄色の楕円をウィンドウに描画しています。2 番目のコードは，C++Builder を使用して開発したアプリケーションです。キャンバスを使用して，同じ楕円を描画しています。

GDI 関数を使用するコードを次に示します。

```
void TMyWindow::Paint(TDC& PaintDC, bool erase, TRect& rect)
{
  HPEN PenHandle, OldPenHandle;
  HBRUSH BrushHandle, OldBrushHandle;
  PenHandle = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
  OldPenHandle = SelectObject(PaintDC, PenHandle);
  BrushHandle = CreateSolidBrush(RGB(255, 255, 0));
  OldBrushHandle = SelectObject(PaintDC, BrushHandle);
  Ellipse(10, 20, 50, 50);
  SelectObject(OldBrushHandle);
  DeleteObject(BrushHandle);
  SelectObject(OldPenHandle);
  DeleteObject(PenHandle);
)
```

同じことを行う C++Builder コードは次のようになります。

```
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  Canvas->Pen->Color = clBlue;
  Canvas->Brush->Color = clYellow;
  Canvas->Ellipse(10, 20, 50, 50);
}
```

キャンバスの使い方

キャンバスクラスは，いくつかのレベルでグラフィックコントロールをカプセル化しています。線，図形，テキストを描画するための高レベルな関数，キャンバスの描画機能を設定するための中間レベルのプロパティなどをカプセル化します。さらに VCL では，Windows GDI への低レベルなアクセス方法もカプセル化しています。

表 50.1 に，キャンバスの描画機能の概要を示します。

表 50.1　キャンバスの機能の概要

レベル	動作	ツール
高レベル	線や図形の描画	MoveTo，LineTo，Rectangle，Ellipse などのメソッド
	テキストの表示と寸法の取得	TextOut，TextHeight，TextWidth，TextRect の各メソッド
	領域の塗りつぶし	FillRect メソッドと FloodFill メソッド
中間レベル	テキストとグラフィックのカスタマイズ	Pen，Brush，Font の各プロパティ
	ピクセルの操作	Pixels プロパティ
	イメージのコピーとマージ	Draw，StretchDraw，BrushCopy，CopyRect の各メソッド，CopyMode プロパティ
低レベル	Windows GDI 関数の呼び出し	Handle プロパティ

キャンバスクラスおよびそのメソッドとプロパティについての詳細は，ヘルプを参照してください。

画像の使い方

C++Builder で行うグラフィック作業のほとんどは，コンポーネントやフォームのキャンバスへ直接的に描画する作業に限定されています。C++Builder では，ビットマップ，メタファイル，アイコンなどのスタンドアロングラフィックイメージも処理できます。この場合，パレットは自動的に管理できます。

C++Builder の画像処理について以下の重要な 3 つの項目を説明します。

- 画像，グラフィック，キャンバスの使い方
- グラフィックの読み込みと保存
- パレットの使い方

画像，グラフィック，キャンバスの使い方

以下に示すように，C++Builder にはグラフィックを扱う 3 種類のクラスがあります。

- キャンバス（TCanvas）は，フォーム，グラフィックコントロール，ビットマップ，プリンタの描画表面を表します。キャンバスはスタンドアロンのクラスではなく，常にほかのクラスのプロパティとして存在します。

- グラフィック（TGraphic）は，ビットマップ，アイコン，メタファイルなど，通常はファイルかリソースに格納されているグラフィックイメージを表します。C++Builder は，TBitmap，TIcon，TMetafile（VCL のみ）の各クラスを定義します。この 3 つのクラスはともに TGraphic クラスから派生しています。また，開発者は独自のグラフィッククラスを定義できます。グラフィックに最低限必要な共通の標準的なインターフェースが TGraphic に定義されているので，アプリケーションからはさまざまな種類のグラフィックを容易に使用できます。
- 画像（TPicture）は，グラフィックのコンテナです。グラフィックのコンテナは，すべてのグラフィッククラスをその中に格納できます。ビットマップ，アイコン，メタファイル，ユーザー定義グラフィック型は，すべて TPicture 型のフィールドに格納できます。アプリケーションは，画像クラスに格納されたさまざまなグラフィック型を，同一の方法で操作できます。たとえば，イメージコントロールは Picture という TPicture 型のプロパティを持っており，それによってさまざまなグラフィック型のイメージを表示します。

画像クラスは常にグラフィッククラスを含んでおり，また，グラフィッククラスはキャンバスクラスを含む場合があります（TBitmap は，キャンバスクラスを含む唯一の標準グラフィッククラスです）。通常，画像を扱う場合は，グラフィッククラスのうち TPicture を通して操作できる部分だけを使用します。グラフィッククラス固有の細部を操作する場合には，その図形オブジェクトの Graphic プロパティを使用します。

グラフィックの読み込みと保存

C++Builder のすべての画像とグラフィックは，ファイルからイメージを読み込んだり，再度そのファイル（またはほかのファイル）に保存することができます。画像のイメージの読み込みと保存はいつでも行えます。

CLX　CLX コンポーネントを作成する場合，イメージの読み込みと保存は，Qt MIME ソースやストリームオブジェクトでも行うことができます。

イメージをファイルから画像に読み込むには，その画像オブジェクトの LoadFromFile メソッドを呼び出します。イメージを画像からファイルに保存するには，その画像オブジェクトの SaveToFile メソッドを呼び出します。

LoadFromFile と SaveToFile は，唯一のパラメータとしてファイル名を受け取ります。LoadFromFile は，ファイル名の拡張子によって読み込むグラフィックオブジェクトの種類を判定します。SaveToFile は，保存するグラフィックオブジェクト型に合わせてファイルの種類を決定します。

ビットマップをイメージコントロールの画像に読み込むには，次のように，ビットマップファイルの名前を画像の LoadFromFile メソッドに渡します。

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Image1->Picture->LoadFromFile("c: \\ windows \\ athena.bmp");
}
```

画像オブジェクトは，.bmp をビットマップファイルの標準拡張子として認識し，グラフィックを TBitmap として作成してから，そのグラフィックの LoadFromFile メソッドを呼び出します。このグラフィックはビットマップなので，LoadFromFile メソッドはイメージをビットマップとしてファイルから読み込みます。

パレットの使い方

VCL コンポーネントと CLX コンポーネントの場合，C++Builder は，パレットを使用するデバイス（一般的に 256 色モード）では，自動的にパレットの実現をサポートします。パレットを持つコントロールでは，TControl から継承した 2 つのメソッドを使用して，Windows がそのパレットを扱う方法を制御できます。

コントロールのパレットについて，以下の 2 つの項目を説明します。

- コントロール用のパレットの指定
- パレットの変更への応答

ほとんどのコントロールはパレットとは関係ありません。しかし，イメージコントロールなど，ある種のグラフィックイメージを持つコントロールは，確実に適切な外観で表示されるように，Windows やスクリーンデバイスドライバと対話する場合があります。Windows では，この処理をパレットの「実現」と呼びます。

パレットの実現では，最前面のウィンドウのすべてのパレット色はそのまま表示されます。背面のウィンドウのパレット色も可能な限りそのまま表示されますが，表示できなかった色については，「実際の」パレットの最も近い色にマッピングされます。Windows は，ウィンドウが互いに前後に移動するのに応じて，継続的にパレットの実現処理を行っています。

メモ C++Builder 自身は，ビットマップ以外については，パレットの作成や操作に対するサポートを提供していません。しかし，パレットハンドルを取得すれば C++Builder コントロールがそれを管理できます。

コントロール用のパレットの指定

VCL CLX と のコントロールにパレットを指定するには，そのコントロールの GetPalette メソッドをオーバーライドして，パレットのハンドルを返します。

コントロールにパレットを指定すると，以下の 2 つの処理が行われます。

- アプリケーションに，コントロールのパレットを実現する必要があることを通知する
- そのパレットを実際に使用するパレットとして指定する

パレットの変更への応答

VCL コントロールが GetPalette をオーバーライドしてパレットを指定した場合，C++Builder は自動的に，Windows からのパレットメッセージに対して応答します。パレットメッセージを処理するメソッドは PaletteChanged です。

PaletteChanged は，コントロールのパレットを前景と背景のどちらで実現するかを決定します。Windows によるパレットの実現では，前景パレットを最前面のウィンドウが使用し，背景パレットはほかのウィンドウが使用します。しかし，C++Builder はそのうえに，コントロールのパレットを 1 つのウィンドウ内でタブ順に実現します。タブ順の先頭にないコントロールに前景パレットを使用させる場合だけ，デフォルトの動作をオーバーライドします。

オフスクリーンビットマップ

複雑なグラフィックイメージを描く際にグラフィックプログラミングでよく使用されるテクニックとして，オフスクリーンビットマップを作成し，そのビットマップ上にイメージを描画してから，完成したイメージを最終目標である画面にコピーする方法があります。画面に直接描画すると画面がちらつく場合がありますが，オフスクリーンイメージを使用すればこのちらつきは減少します。

また，C++Builder のビットマップクラスは，通常リソースやファイルのビットマップイメージを表しますが，オフスクリーンイメージとしても使用できます。

オフスクリーンビットマップの操作について，次の 2 つの項目を説明します。

- オフスクリーンビットマップの作成と管理
- ビットマップイメージのコピー

オフスクリーンビットマップの作成と管理

複雑なグラフィックイメージを作成する場合は，画面に表示されているキャンバスに直接描画する方法は採りません。フォームやコントロールのキャンバスに描画するかわりに，ビットマップオブジェクトを作成し，そのキャンバスに描画してから，仕上がったイメージをオンスクリーンのキャンバスにコピーします。オフスクリーンビットマップは，グラフィックコントロールの Paint メソッドでよく使用されます。

オフスクリーンビットマップに複雑な画像を描画する例については，コンポーネントパレットの［Samples］ページにあるゲージコントロールのソースコードを参照してください。ゲージは，まずオフスクリーンビットマップにさまざまな図形やテキストを描画してから画面にそれをコピーします。ゲージのソースコードは，Examples \ Controls \ Source サブディレクトリにあるファイル Cgauges.cpp に入っています。

ビットマップイメージのコピー

C++Builder では，あるキャンバスからほかのキャンバスへイメージをコピーする場合，4 つの方法があります。各方法に応じて，それぞれのメソッドを呼び出します。

表 50.2 にキャンバスオブジェクトのイメージコピーメソッドの要約を示します。

表 50.2　イメージコピーメソッド

処理	呼び出すメソッド
グラフィック全体をコピー	Draw
グラフィックをコピーしてサイズ変更	StretchDraw
キャンバスの一部をコピー	CopyRect
ラスター演算を適用してビットマップをコピー	BrushCopy（VCL）
グラフィックを繰り返しコピーし，領域に並べて配置	TiledDraw（CLX）

変更への応答

キャンバスや，キャンバスが所有するオブジェクト（ペン，ブラシ，フォント）など，すべてのグラフィックオブジェクトには，オブジェクトの変更に応答するためのイベントがあります。これらのイベントを使用すれば，コンポーネント（またはコンポーネントを使用しているアプリケーション）の変更に対してイメージの再描画によって応答させることができます。

グラフィックオブジェクトがコンポーネントの設計時インターフェースの一部としてパブリッシュに宣言されている場合は，そのオブジェクトの変更に応答することが特に重要です。そうしないと，コンポーネントの設計時の外観がオブジェクトインスペクタのプロパティと一致しません。

グラフィックオブジェクトの変更に応答するには，オブジェクトの OnChange イベントにメソッドを指定します。

次に示す図形コンポーネントでは，図形を描画するペンやブラシを表すプロパティは，パブリッシュに設定されています。コンポーネントのコンストラクタは，各プロパティの OnChange イベントにメソッドを指定し，ペンやブラシが変更されたときにイメージを更新するようにしています。図形コンポーネントは Object Pascal で記述されていますが，新しい名前 TMyShape を付けて C++ に翻訳した図形コンポーネントを次に示します。

ヘッダーファイルのクラス宣言は次のとおりです。

```
class PACKAGE TMyShape : public TGraphicControl
{
private:
protected:
public:
    virtual __fastcall TMyShape(TComponent* Owner);
__published:
    TPen *FPen;
    TBrush *FBrush;
    void __fastcall StyleChanged(TObject *Sender);
};
```

.CPP ファイルのコードは次のとおりです。

```
__fastcall TMyShape::TMyShape(TComponent* Owner)
  : TGraphicControl(Owner)
{
  Height = 65;
  Width = 65;
  FPen = new TPen;
  FPen->OnChange = StyleChanged;
  FBrush = new TBrush;
  FBrush->OnChange = StyleChanged;
}
void __fastcall TMyShape::StyleChanged(TObject *Sender)
{
  Invalidate();
}
```

第51章

メッセージとシステム通知の処理

多くの場合，コンポーネントはオペレーティングシステムからの通知に応答する必要があります。オペレーティングシステムは，ユーザーがマウスとキーボードで行った動作などの出来事をアプリケーションに通知します。コントロールの中にも通知を生成するものがあります。たとえば，ユーザーがリストボックスの項目を選択するというアクションの結果などがあります。VCL と CLX は，一般的な通知のほとんどを処理します。しかし，このような通知を処理する独自のコードを開発者が記述しなければならないこともあります。

VCL では，メッセージの形で通知が行われます。これらのメッセージは，Windows，VCL コンポーネント，開発者が定義したコンポーネントなど，あらゆるソースによって生成されます。このセクションでは，メッセージについて以下の 3 項目を説明します。

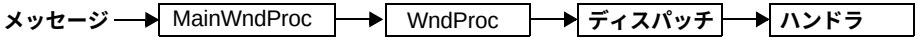
- メッセージ処理機構
- メッセージ処理方法の変更
- 新しいメッセージハンドラの作成

CLX では，Windows メッセージでなくシグナルとシステムイベントの形で通知が行われます。CLX でシステム通知がどう機能するかについての詳細は，51-10 ページの「CLX によるシステム通知への応答」を参照してください。

メッセージ処理機構

すべての VCL クラスは，メッセージ処理メソッドまたは メッセージハンドラと呼ばれる，メッセージ処理のためのメカニズムを備えています。メッセージハンドラの基本的な構造は次のとおりです。つまり，クラスがメッセージを受け取ると，そのメッセージに対応するメソッドが呼び出されることで，メッセージがディスパッチされます。対応するメソッドが存在しないメッセージについては，デフォルトのハンドラが使用されます。

次の図は，メッセージディスパッチ機構を示しています。



ビジュアルコンポーネントライブラリのメッセージディスパッチ機構は，特定のクラスに向けられたすべての Windows メッセージ（ユーザー定義メッセージを含む）をメソッド呼び出しに変換します。開発者はメッセージディスパッチ機構自体を変更する必要はありません。必要なのは，メッセージ処理メソッドを作成することだけです。詳細については，51-7 ページの「新しいメッセージ処理メソッドの宣言」を参照してください。

Windows のメッセージの内容

Windows のメッセージは，いくつかのデータメンバーを持つデータ構造体と考えることができます。最も重要なデータメンバーは，メッセージを識別するための整数値のフィールドです。Windows では多くのメッセージが定義されていますが，MESSAGES.HPP ファイルにはそのすべての識別子が宣言されています。

Windows プログラマは，WM_COMMAND や WM_PAINT などのメッセージ識別用の Windows の定義の扱いに慣れています。従来の Windows のプログラムには，システムが生成したメッセージに対するコールバックとして機能するウィンドウプロシージャが含まれています。このウィンドウプロシージャは通常，このウィンドウで処理しようとするメッセージごとに case ラベルの付いた大規模な switch 文が使用されます。

メッセージの処理に役立つ付加情報は，wParam（「Word パラメータ」に対して）および lParam（「Long パラメータ」に対して）の 2 つのパラメータとして，このウィンドウプロシージャに渡されます。通常，それぞれのパラメータには 2 つ以上の情報が含まれています。したがって，LOWORD や HIWORD などの Windows マクロを使って，適切な部分を抽出する必要があります。たとえば，HIWORD(lParam) を呼び出すと，このパラメータの上位ワードを抽出できます。

従来，Windows プログラマには Windows API においてパラメータが持つ情報についての知識が必要でした。現在，Windows ではメッセージクラッカを使用しているので，Windows メッセージおよびそのパラメータの処理に関連する構文が単純になっています。このメッセージクラッカを使用すると，大規模な switch 文を使用して，すべての情報をパラメータに分解する代わりに，ハンドラ関数とメッセージを簡単に関連付けることができます。標準的な Windows プログラムに WINDOWSX.H をインクルードすると，プログラム中で HANDLE_MSG マクロが使用できます。これを使って次のようにコードを書くことができます。

```
void MyKeyDownHandler(  HWND hwnd. UINT nVirtKey, BOOL fDown, int CRepeat, UINT flags )
{
  ...
}

LRESULT MyWndProc( HWND hwnd, UINT Message, WPARAM wParam, LPARAM lParam )
{
  switch(  Message )
  {
    HANDLE_MSG( hwnd, WM_KEYDOWN, MyKeyDownHandler );
    ...
}
```

この形式でメッセージクラッキングを使用すると，メッセージが特定のハンドラにディスパッチされることがより明確になります。また，使用するハンドラ関数のパラメータリストに意味のある名前を

付けることもできます。たとえば，WM_KEYDOWN メッセージの wParam の値として，nVirtKey という名前のパラメータを取るようにした方が関数が理解しやすくなります。

メッセージをディスパッチする方法

アプリケーションがウィンドウを作成する場合は，Windows カーネルにウィンドウプロシージャを登録します。このウィンドウプロシージャは，ウィンドウに対して送られるメッセージを処理するルーチンです。ウィンドウプロシージャは通常，そのウィンドウが処理する各メッセージをエントリとする巨大な **switch** 文になります。なお，ここでは「ウィンドウ」という用語は，ウィンドウ，コントロールなど，画面に表示されるさまざまなものを表しています。新しい種類のウィンドウを作成するたびに，完全なウィンドウプロシージャを作成しなければなりません。

VCL では，以下に示す方法でメッセージのディスパッチが単純化されています。

- 各コンポーネントは，完全なメッセージディスパッチ機構を継承する
- ディスパッチ機構にはデフォルトの処理方法がある。独自の処理を行うメッセージに対してのみ新しいハンドラを定義する
- メッセージ処理を少しだけ変更して残りの処理を継承メソッドにまかせることができる

このメッセージディスパッチ機構では，任意のメッセージを任意のコンポーネントに任意の時点で安全に送ることができます。コンポーネントがハンドラを持っていないメッセージはデフォルトの方法で処理されます。

メッセージ経路のトレース

VCL では，アプリケーションの各コンポーネント型のウィンドウプロシージャとして，MainWndProc という名前のメソッドが登録されます。MainWndProc には例外処理ブロックがあります。ここで，Windows からのメッセージ構造体を WndProc という仮想メソッドに渡します。また，アプリケーションクラスの HandleException メソッドを呼び出すことですべての例外を処理します。

MainWndProc は非仮想メソッドで，特定のメッセージに対する処理を行いません。各コンポーネント型はこのメソッドをオーバーライドできるので，カスタマイズは WndProc で行われます。

WndProc メソッドは，処理に影響するような状態について検査し，不要なメッセージを「トラップ」します。たとえば，コンポーネントはマウスのドラッグ中，キーボードイベントを無視します。そのため，TWinControl の WndProc メソッドはコンポーネントがドラッグされていないときにのみ，キーボードイベントを次に渡します。最終的に WndProc は，TObject から継承した非仮想メソッド Dispatch を呼び出します。Dispatch メソッドは，メッセージを処理するメソッドを決定して呼び出します。

Dispatch は，メッセージをディスパッチする方法を決定するのに，メッセージ構造体の Msg データメンバーを使用します。コンポーネントがそのメッセージに対するハンドラを定義している場合には，Dispatch はそのメソッドを呼び出します。コンポーネントがそのメッセージに対するハンドラを定義していない場合には，Dispatch は DefaultHandler を呼び出します。

メッセージ処理方法の変更

コンポーネントのメッセージ処理方法を変更する前に，本当に変更する必要があるかどうかを確認してください。VCL は大部分の Windows メッセージを，コンポーネント開発者とコンポーネントユーザーの両方が処理できるイベントに変換します。たいていの場合は，メッセージ処理の動作を変更するよりも，イベント処理の動作を変更する方が適切です。

VCL コンポーネントのメッセージ処理方法を変更するには，メッセージ処理メソッドをオーバーライドします。また，メッセージをトラップすることにより，特定の状況下ではコンポーネントがメッセージを処理しないようにすることもできます。

ハンドラメソッドのオーバーライド

コンポーネントがメッセージを処理する方法を変更するには，そのメッセージに対応するメッセージ処理メソッドをオーバーライドします。コンポーネントがそのメッセージに対応するメソッドを持っていない場合には，メッセージ処理メソッドを新しく宣言します。

メッセージ処理メソッドをオーバーライドする手順は次のとおりです。

1. コンポーネント宣言の **protected** 部でオーバーライドするメソッドと同じ名前の新しいメソッドをコンポーネントで宣言します。
2. 次の 3 つのマクロを使って，オーバーライドするメソッドをメッセージにマップします。

   マクロは次の形式で記述します。

   ```
   BEGIN_MESSAGE_MAP
     VCL_MESSAGE_HANDLER(parameter1, parameter2, parameter3)
   END_MESSAGE_MAP
   ```

   parameter1 は Windows で定義されているメッセージインデックスです。parameter2 はメッセージ構造体型，parameter3 はメッセージメソッドの名前です。

   BEGIN_MESSAGE_MAP と END_MESSAGE_MAP の間には，VCL_MESSAGE_HANDLER マクロを必要なだけ記述できます。

たとえば，コンポーネントの WM_PAINT メッセージの処理をオーバーライドするには，次のように WMPaint メソッドを再宣言し，3 つのマクロを使って，このメソッドを WM_PAINT メッセージにマップます。

```
class PACKAGE TMyComponent : public TComponent
{
protected:
    void __fastcall WMPaint(TWMPaint* Message);
BEGIN_MESSAGE_MAP
  VCL_MESSAGE_HANDLER(WM_PAINT, TWMPaint, WMPaint)
END_MESSAGE_MAP(TComponent)
};
```

メッセージパラメータ

メッセージ処理メソッドの実行中，コンポーネントはそのメッセージ構造体のすべてのパラメータにアクセスできます。メッセージハンドラに渡されるパラメータはポインタなので，ハンドラは必要な場合にパラメータの値を変更できます。パラメータの中では，メッセージの戻り値は頻繁に変更されます。Result は，メッセージを送った SendMessage から返される値です。

メッセージ処理メソッドの Message パラメータの型は，メッセージによって変化します。個々のパラメータの名前と意味については，Windows メッセージに関するマニュアルを参照してください。なんらかの理由で，メッセージパラメータを旧式の名前（WParam，LParam など）で参照する場合は，Message を TMessage に型キャストします。TMessage 型では，メッセージパラメータは旧式の名前で参照されます。

メッセージのトラップ

あるコンポーネントについて，特定の状況下ではメッセージを無視させたい場合があります。つまり，コンポーネントがそのメッセージをハンドラにディスパッチしないようにするわけです。メッセージをトラップするには，WndProc という仮想メソッドをオーバーライドします。

VCL コンポーネントの場合，WndProc メソッドは，メッセージをフィルタにかけて選別してから Dispatch メソッドに渡します。その後，Dispatch メソッドがメッセージを処理するメソッドを決定します。WndProc をオーバーライドすれば，ディスパッチされる前にメッセージを選別できます。TWinControl から派生したコントロールの WndProc をオーバーライドするコードは次のとおりです。

```
void __fastcall TMyControl::WndProc(TMessage& Message)
{
   // 処理の継続を決定するための条件
   if(Message.Msg != WM_LBUTTONDOWN)
      TWinControl::WndProc(Message);
}
```

TControl コンポーネントでは，ユーザーがコントロールをドラッグアンドドロップしているときは，すべてのマウスメッセージにフィルタをかけて特別な処理を行います。WndProc をオーバーライドする方法には，以下の 2 つの利点があります。

- ハンドラを個別に指定するかわりに，メッセージのある範囲全体に対してフィルタをかけられる
- メッセージがディスパッチされないので，ハンドラはまったく呼び出されない

次のコードは，VCL に Object Pascal で実装されている TControl の WndProc メソッドの一部分です。

```
procedure TControl.WndProc(var Message: TMessage);
begin
  ...
  if (Message.Msg >= WM_MOUSEFIRST) and (Message.Msg <= WM_MOUSELAST) then
    if Dragging then                                { ドラッグを特別な方法で処理 }
      DragMouseMsg(TWMMouse(Message))
    else
      ...                                           {... 必要に応じてほかのプロパティも設定 }
    end;
  ...                                               { それ以外は普通に処理 }
end;
```

新しいメッセージハンドラの作成

VCL では，よく使用されるメッセージのほとんどに対してハンドラが用意されています。しかし，独自のメッセージを定義したときは，開発者自身が新しいメッセージハンドラを作成しなければなりません。ユーザー定義メッセージについて，以下の 3 項目を説明します。

- 独自のメッセージの定義
- 新しいメッセージ処理メソッドの宣言
- メッセージの送信

独自のメッセージの定義

多くの標準コンポーネントでは，内部で使用するためのメッセージが定義されています。メッセージを定義する一般的な目的は，標準のメッセージでは扱われない情報や，状態の変更を通知することです。VCL では，開発者が独自のメッセージを定義することができます。

メッセージの定義には，以下の 2 つのステップが必要です。メッセージを定義する手順は次のとおりです。

1. メッセージ識別子の宣言
2. メッセージ構造体型の宣言

メッセージ識別子の宣言

メッセージ識別子は整数の定数です。Windows ではシステム用に 1024 未満のメッセージ識別子が予約されているので，開発者が独自のメッセージを宣言する場合は，それ以上の番号を付けます。

定数 WM_APP は，ユーザー定義メッセージの開始番号を表しています。メッセージ識別子を定義する場合は，WM_APP 以上の番号を使用します。

ただし，ある種の標準 Windows コントロールは，ユーザー定義用の範囲の番号を使用することがあります。そのようなコントロールとしては，リストボックス，コンボボックス，編集ボックス，コマンドボタンがあります。このようなコントロールから派生させたコンポーネントに新しいメッセージを定義する場合には，必ず MESSAGES.HPP ファイルで Windows がそのコントロールに対して定義しているメッセージの番号を確認してください。

次のコード例は，2 つのユーザー定義メッセージの定義を示しています。

```
#define MY_MYFIRSTMESSAGE  (WM_APP + 400)
#define MY_MYSECONDMESSAGE (WM_APP + 401)
```

メッセージ構造体型の宣言

メッセージのパラメータに名前を付ける場合には，そのメッセージの構造型を宣言します。メッセージ構造体型は，メッセージ処理メソッドに渡されるパラメータの型です。メッセージのパラメータを使用しない場合や，旧式のパラメータ表記方法（WParam，LParam など）を使用する場合には，デフォルトのメッセージ構造体である TMessage を使用します。

メッセージ構造体型を宣言するときは次の命名規則に従います。

1. 構造型の名前は，そのメッセージの名前の先頭に T を加えたものにします。
2. まず，最初のデータメンバーとして TMsgParam 型の構造体 Msg を定義する
3. 次の 2 バイトは Word パラメータに対応し，その次の 2 バイトは使わないものとして定義します。この 4 バイトは wParam に対応します。
4. 次の 4 バイトは lParam に対応します。多くの場合 Longint 型を定義します。
5. 最後に Longint 型の Result を定義します。

マウスメッセージ用のメッセージ構造体 TWMKey の例を次に示します。

```
struct TWMKey
{
  Cardinal Msg;                    // 最初のパラメータはメッセージ ID
  Word CharCode;                   // 最初の wParam
  Word Unused;
  Longint KeyData;                 // lParam
  Longint Result;                  // 結果を格納するデータメンバー
};
```

新しいメッセージ処理メソッドの宣言

新しいメッセージ処理メソッドの宣言が必要になるのは以下の 2 つの場合です。

- 標準コンポーネントが処理できない Windows メッセージを，処理できるコンポーネントを開発する場合
- 独自のメッセージを定義し，それを使うコンポーネントを開発する場合

メッセージ処理メソッドを宣言する手順は以下のとおりです。

1. コンポーネントのクラス宣言の **protected** 部で，BEGIN_MESSAGE_MAP ... END_MESSAGE_MAP マクロを使ってメソッドを宣言します。
2. メソッドの戻り型を必ず **void** にします。
3. 処理するメッセージの名前に基づいてメソッドを命名します。なお, 名前に下線記号は使えません。
4. メッセージ構造体型の Message をポインタとして渡します。
5. マクロを使用してメソッドをメッセージにマッピングします。
6. メッセージメソッド実装内で，コンポーネント独自の処理コードを記述します。
7. 継承メッセージハンドラを呼び出します。

ユーザー定義メッセージ CM_CHANGECOLOR のためのメッセージハンドラの宣言例を次に示します。

```
#define CM_CHANGECOLOR (WM_APP + 400)
class TMyControl : public TControl
{
protected:
    void __fastcall CMChangeColor(TMessage &Message);
BEGIN_MESSAGE_MAP
  VCL_MESSAGE_HANDLER(CM_CHANGECOLOR, TMessage, CMChangeColor)
```

```
END_MESSAGE_MAP(TControl)
};
void __fastcall TMyControl::CMChangeColor(TMessage &Message)
{
  Color = Message.LParam;                 // long パラメータに渡された値を Color に代入する
  TControl::CMChangeColor(Message);       // 親クラスのメッセージハンドラを呼び出す
}
```

メッセージの送信

通常，アプリケーションは状態変更の通知や情報のブロードキャストを行うためにメッセージを送信します。コンポーネントは，フォーム内のすべてのコントロールにメッセージを通知することも，特定のコントロール（またはアプリケーション自体）にメッセージを送信することも，コンポーネント自体にメッセージを送ることもできます。

Windows メッセージの送信には，さまざまな方法があります。どの方法をとるかは，なぜメッセージを送信するのかによります。以下の項では，Windows メッセージを送信するさまざまな方法について説明します。

フォーム内のすべてのコントロールにメッセージをブロードキャストする

コンポーネントがフォームなどのコンテナのすべてのコントロールに影響するグローバルな設定を変更する場合は，該当するコントロールにメッセージを送信し，各コントロールがそれ自体を適切に更新できるようにする必要があります。すべてのコントロールが通知に応答するわけではありませんが，メッセージをブロードキャストすると応答の方法を指定されているすべてのコントロールに通知できます。それ以外のコントロールは，メッセージを無視します。

別のコントロール内のすべてのコントロールにメッセージをブロードキャストするには，Broadcast メソッドを使用します。メッセージをブロードキャストする前に，伝えたい情報をメッセージ構造体に挿入します（メッセージ構造体についての詳細は，51-6 ページの「メッセージ構造体型の宣言」を参照してください）。

```
TMessage Msg;
Msg.Msg = MY_MYCUSTOMMESSAGE;
Msg.WParam = 0;
Msg.LParam = (int)(this);
Msg.Result = 0;
```

次に，通知するすべてのコントロールの親にこのメッセージ構造体を渡します。これは，アプリケーション内のどんなコントロールでもかまいません。たとえば，次のようにプログラマが開発するコントロールの親でもかまいません。

```
Parent->Broadcast(Msg);
```

次のように，コントロールが属するフォームでもかまいません。

```
GetParentForm(this)->Broadcast(Msg);
```

次のように，アクティブフォームでもかまいません。

```
Screen->ActiveForm->Broadcast(Msg);
```

次のように，アプリケーションのすべてのフォームでもかまいません。

```
for (int i = 0; i < Screen->FormCount; i++)
  Screen->Forms[i]->Broadcast(Msg);
```

コントロールのメッセージハンドラを直接呼び出す

メッセージへの応答が必要なコントロールが唯一の場合もあります。メッセージを受信しなければならないコントロールがわかっている場合は，もっとも単純で直接的なメッセージの送信方法として，コントロールの Perform メソッドを呼び出すことでができます。

コントロールの Perform メソッドを呼び出す理由は 2 つあります。

- コントロールは，1 つの標準の Windows メッセージ（または他のメッセージ）に対しては同じ応答を生成する必要があります。たとえば，グリッドコントロールは，キー入力処理メッセージを受け取るとインライン編集コントロールを作成し，次にこの編集コントロールにキー入力処理メッセージを送信します。
- どのコントロールに通知するかはわかっていても，そのコントロールの型がわからない場合があります。目的のコントロールの型がわからないので，その専用のメソッドは指定できませんが，すべてのコントロールはメッセージ処理機能を備えているので，いつでもメッセージを送信できます。そのコントロールに送信したメッセージに対するメッセージハンドラがある場合は，適切な応答が返されます。それ以外の場合はメッセージが無視され，0 が返されます。

Perform メソッドを呼び出す場合は，メッセージ構造体を作成する必要がありません。パラメータとして，メッセージ識別子 WParam と LParam を渡すだけです。Perform メソッドはメッセージの結果を返します。

Windows のメッセージキューを使用してメッセージを送信する

マルチスレッドアプリケーションでは，実行中のスレッドに目的のコントロールが存在するわけではないので，Perform メソッドを呼び出すことはできません。しかし，Windows メッセージキューを使用すると，実行されていないスレッドと安全に通信できます。メッセージ処理は常にメインの VCL スレッドで発生しますが，Windows メッセージキューを使用するとアプリケーションの任意のスレッドからメッセージを送信できます。SendMessage の呼び出しは同期的です。すなわち，SendMessage は目的のコントロールが別のスレッドにある場合でも，そのコントロールがメッセージを処理するまでは応答がありません。

Windows API 呼び出し SendMessage を使用すると，Windows メッセージキューを使用してコントロールにメッセージを送信できます。SendMessage では，Perform メソッドと同じパラメータをとりますが，Window ハンドルを渡すことによって目的のコントロールを識別する必要があります。したがって，次のコードのかわりに，

```
MsgResult = TargetControl->Perform(MY_MYMESSAGE, 0, 0);
```

次のコードを記述します。

```
MsgResult = SendMessage(TargetControl->Handle, MYMESSAGE, 0, 0);
```

SendMessage についての詳細は，Microsoft MSDN のマニュアルを参照してください。同時に実行できる複数のスレッドを記述する方法については，11-7 ページの「スレッドの調整」を参照してください。

すぐに実行しないメッセージを送信する

メッセージを送信したいが，メッセージの送信先で直ちに実行するのが安全かどうかわからないことがあります。たとえば，メッセージを送信するコードが目的のコントロールのイベントハンドラから呼び出される場合は，そのコントロールがメッセージを実行する前にイベントハンドラが完了していることを確認する必要があります。メッセージの結果を確認する必要がない限り，この状況を処理できます。

Windows API 呼び出し PostMessage を使用すると，コントロールにメッセージを送信しても，そのメッセージはコントロールが他のメッセージを完了するまでは処理されません。PostMessage のパラメータは SendMessage とまったく同じです。

PostMessage についての詳細は，Microsoft MSDN のマニュアルを参照してください。

CLX によるシステム通知への応答

Windows を使用すると，オペレーティングシステムからアプリケーションとそのコントロールに Windows メッセージを使用して直接通知が送信されます。しかし，この方法は CLX には適しません。CLX はクロスプラットフォームライブラリであり，Linux 上では Windows メッセージを使用できないためです。かわりに，CLX ではプラットフォームに依存しない方法でシステム通知に応答します。

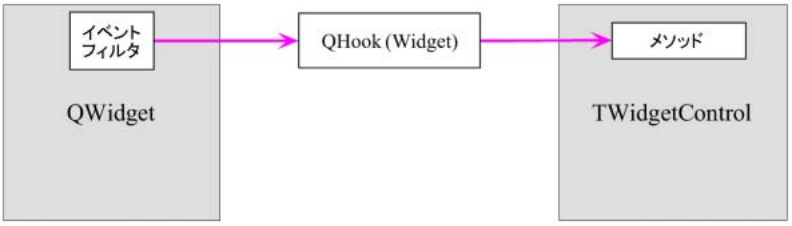
CLX で Windows メッセージに相当するものは，基になるウィジェット層で生成されるシグナルのシステムです。VCL の Windows メッセージには，オペレーティングシステムで生成されるものと，または VCL がラップする Windows 特有のコントロールによって生成されるものがありますが，CLX で使用するウィジェット層ではこの 2 つを区別します。ウィジェットで生成された通知はシグナルと呼ばれます。オペレーティングシステムで生成された通知はシステムイベントと呼ばれます。ウィジェット層は CLX コンポーネントにイベント型のシグナルとしてシステムイベントを送信します。

シグナルへの応答

基になるウィジェット層ではさまざまなシグナルが生成され，それぞれが別の型の通知を表します。これらのシグナルには，システムイベント（イベントシグナル）とシグナルを生成するウィジェットに固有の通知があります。たとえば，すべてのウィジェットはウィジェットが解放されるときに破棄シグナルを生成します。トラックバー ウィジェットは valueChanged シグナルを生成し，ヘッダーコントロールは sectionClicked シグナルを生成します。

個々の CLX コンポーネントは，シグナルハンドラとしてメソッドを割り当てることで，基になるウィジェットからのシグナルに応答します。このために，基になるウィジェットに固有のフックオブジェクトを使用します。フックオブジェクトはメソッドポインタを集めただけの小規模なオブジェクトであり，個々のメソッドポインタは特定のシグナルに固有です。CLX コンポーネントのメソッドが特定のシグナルのハンドラとしてフックオブジェクトに割り当てられると，ウィジェットがそのシグナルを生成するたびにこの CLX コンポーネントのメソッドが呼び出されます。概要を図 51.1 に示します。

図 51.1　シグナルの経路



メモ　個々のフックオブジェクトに割り当てるメソッドは，Qt ユニットで宣言されます。特定のフックオブジェクトで使用するメソッドは，qt.hpp で確認してください。メソッドは，割り当てられているフックオブジェクトを反映する名前でグローバルルーチンに展開されます。たとえば，アプリケーションウィジェット（QApplication）に関連するフックオブジェクトに割り当てられたメソッドは，すべて「QApplication_hook」で始まります。このような展開は，Object Pascal CLX オブジェクトが C++ フックオブジェクトのメソッドにアクセスするために必要です。

カスタムシグナルハンドラの割り当て

CLX コントロールの多くには，基になるウィジェットからのシグナルを処理するメソッドが割り当てられています。通常，これらのメソッドはプライベートであり，仮想メソッドではありません。したがって，独自のメソッドを記述する場合は，ウィジェットに関連するフックオブジェクトに独自のメソッドを割り当てる必要があります。この場合は，HookEvents メソッドをオーバーライドします。

メモ　応答を必要とするシグナルがシステムイベント通知の場合，HookEvents メソッドをオーバーライドすることはできません。システムイベントに応答する方法についての詳細は，後述の「システムイベントへの応答」を参照してください。

HookEvents メソッドをオーバーライドする場合は，TMethod 型の変数を宣言します。次に，シグナルハンドラとしてフックオブジェクトに割り当てる個々のメソッドについて，以下の作業を行います。

1. TMethod 型の変数を初期化し，シグナルのメソッドハンドラを表すようにします。
2. この変数をフックオブジェクトに割り当てます。フックオブジェクトには，コンポーネントが THandleComponent または TWidgetControl から継承した Hooks プロパティを使用してアクセスします。

オーバーライドする場合は，基本クラスが割り当てるシグナルハンドラもフックされるように，必ず HookEvents メソッドを呼び出します。

次のコードは，TTrackBar の HookEvents メソッドの翻訳です。このコードは，HookEvents メソッドをオーバーライドする方法を示しています。

```
virtual void __fastcall TTrackBar::HookEvents(void)
{
  TMethod Method;
  // QSlider valueChanged シグナルのハンドラを表すように Method を初期化
  // ValueChangedHook はこのシグナルに応答する TTrackBar のメソッド
  QSlider_valueChanged_Event(Method) = @ValueChangedHook;
  // Method をフックオブジェクトに割り当てる。ただし，Hooks は基になるウィジェットに
  // 関連するフックオブジェクトの型に変換できる
  QSlider_hook_hook_valueChanged(dynamic_cast<QSlider_hookH>(Hooks), Method);
  // 次に sliderMoved イベントの処理を繰り返す
```

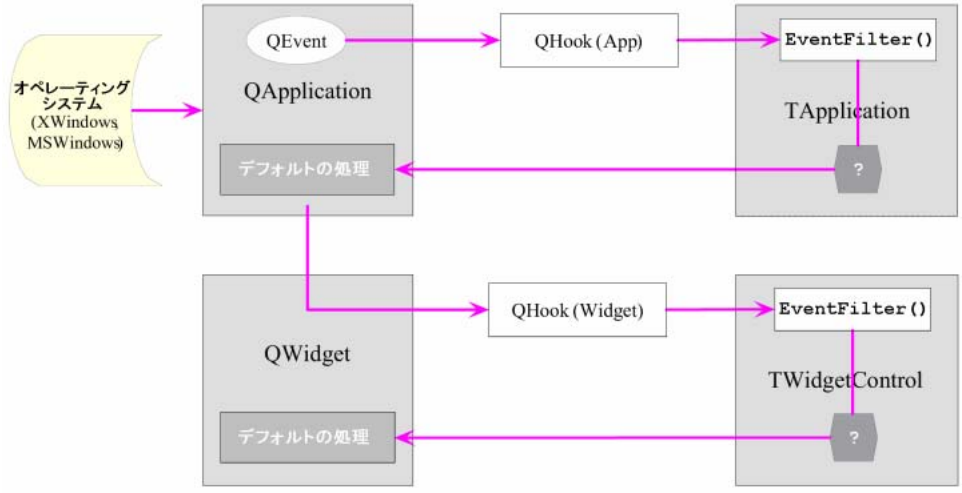
```
    QSlider_sliderMoved_Event(Method) := @ValueChangedHook;
    QSlider_hook_hook_valueChanged(dynamic_cast<QSlider_hookH>(Hooks), Method);
    // 継承したシグナルハンドラがフックされるように，継承メソッドを呼び出す
    TWidgetControl::HookEvents();
  }
```

システムイベントへの応答

ウィジェット層は，オペレーティングシステムからのイベント通知を受け取ると，そのイベントを表す専用のイベントオブジェクト（QEvent またはその下位クラス）を生成します。イベントオブジェクトには，発生したイベントに関する読み取り専用の情報が含まれています。イベントオブジェクトの型は，発生したイベントの型を表します。

ウィジェット層は，イベント型の特殊なシグナルを使用して CLX コンポーネントにシステムイベントを通知します。そして，イベントのシグナルハンドラに QEvent オブジェクトを渡します。イベントシグナルはまずアプリケーションオブジェクトに移動するので，その処理は他のシグナルの処理よりも少し複雑です。つまり，アプリケーションがシステムイベントに応答する機会は，アプリケーションレベル（TApplication）と個々のコンポーネントレベル（TWidgetControl または THandleComponent の下位クラス）の 2 つがあります。, これらのすべてのクラス（TApplication, TWidgetControl，THandleComponent）には，すでにウィジェット層からのイベントシグナル用のシグナルハンドラが割り当てられています。つまり，すべてのシステムイベントは，VCL コントロールの WndProc メソッドと同様の役割を果たす EventFilter メソッドに転送されます。概要は，図 51.2 を参照してください。

図 51.2　システムイベントの経路



EventFilter は，非常によく使われるシステム通知を処理し，コンポーネントの基本クラスのイベントに翻訳します。, たとえば，TWidgetControl の EventFilter メソッドは OnMouseDown イベント，OnMouseMove イベント，OnMouseUp イベントを生成してマウスイベント（QMouseEvent）に応答し，OnKeyDown イベント，OnKeyPress イベント，OnKeyString イベント，OnKeyUp イベントを生成してキーボードイベント（QKeyEvent）に応答します。

よく使われるイベント

TWidgetControl の EventFilter メソッドは，TControl または TWidgetControl のプロテクトメソッドを呼び出すことで一般的なシステム通知の多くを処理します。これらの多くは仮想メソッドなので，独自のコンポーネントを記述すればこれをオーバーライドし，システムイベントに対する独自の応答を実装できます。これらのメソッドをオーバーライドするときは，ほとんどの場合，基になるウィジェット層のオブジェクト（イベントオブジェクトなど）を操作する必要はありません。

独自の コンポーネントを使ってシステム通知に応答する場合は，すでにその通知に応答しているプロテクトメソッドがあるかどうかをまず確認することが大切です。該当するイベントに応答するプロテクトメソッドがあるかどうかを調べるには，オンラインリファレンスで TControl または TWidgetControl（およびコンポーネントを派生させる他の基本クラス）を参照してください。表 51.1 に，もっともよく使われる TControl と TWidgetControl プロテクトメソッドで使用できるものを示します。

表 51.1　システム通知に応答する TWidgetControl プロテクトメソッド

メソッド	説明
BeginAutoDrag	コントロールに dmAutomatic の DragMode がある場合にユーザーが左のマウスボタンをクリックするときに呼び出される
Click	ユーザーが操作しているマウスボタンを放すときに呼び出される
DblClick	ユーザーが操作しているマウスでダブルクリックするときに呼び出される
DoMouseWheel	ユーザーがマウスホイールを回転させるときに呼び出される
DragOver	ユーザーが操作しているマウスカーソルをドラッグするときに呼び出される
KeyDown	コントロールにフォーカスがある場合に，ユーザーがキーを押すときに呼び出される
KeyPress	KeyDown でキー入力が処理されない場合に KeyDown の後に呼び出される
KeyString	システムがマルチバイト文字システムを使用する場合にユーザーがキー入力を行うときに呼び出される
KeyUp	コントロールにフォーカスがある場合に，ユーザーがキーを放すときに呼び出される
MouseDown	ユーザーが操作しているマウスボタンをクリックするときに呼び出される
MouseMove	ユーザーが操作しているマウスカーソルを移動するときに呼び出される
MouseUp	ユーザーが操作しているマウスボタンを放すときに呼び出される
PaintRequest	システムがコントロールの再描画を要求するときに呼び出される
WidgetDestroyed	コントロールの基になるウィジェットが破棄されるときに呼び出される

オーバーライドでは，デフォルトの処理によってシグナルに応答できるように，継承メソッドを呼び出します。

メモ　システムイベントに応答するメソッドのほかに，コントロールには TControl または TWidgetControl から派生した同様のメソッドがたくさん含まれており，コントロールにさまざまなイベントを通知します。これらはシステムイベントに応答しなくても，VCL コントロールに送信される多くの Windows メッセージと同じ作業を行います。これらのメソッドの例を表 51.1 に示します。

表 51.2 システム通知に応答する TWidgetControl プロテクトメソッド

メソッド	説明
BoundsChanged	コントロールのサイズが変更されるときに呼び出される
ColorChanged	コントロールの色が変更されるときに呼び出される
CursorChanged	カーソルの形状が変更されるときに呼び出される。マウスカーソルは特定のウィジェット上では特定の形状となること仮定される
EnabledChanged	アプリケーションがウィンドウまたはコントロールの有効化された状態を変更するときに呼び出される
FontChanged	フォントリソースの集合が変更されるときに呼び出される
PaletteChanged	ウィジェットのパレットが変更されるときに呼び出される
ShowHintChanged	コントロール上でヘルプのヒントが表示されるときまたは非表示になるときに呼び出される
StyleChanged	ウィンドウまたはコントロールの GUI スタイルが変更されるときに呼び出される
TabStopChanged	フォームのタブ順が変更されるときに呼び出される
TextChanged	コントロールのテキストが変更されるときに呼び出される
VisibleChanged	コントロールが表示されるときまたは非表示になるときに呼び出される

EventFilter メソッドのオーバーライド

イベント通知に応答しようとしてもそのイベント用のオーバーライドできるプロテクトメソッドがない場合は，EventFilter メソッド自体をオーバーライドできます。オーバーライドでは，EventFilter メソッドの Event パラメータの型をチェックし，応答する通知の型と一致する場合に独自の処理を行います。EventFilter メソッドに **true** を返させることで，イベント通知がそれ以上処理されないようにします。

メモ さまざまな型の QEvent オブジェクトについての詳細は，TrollTech の Qt のドキュメントを参照してください。

次のコードは，TCustomControl の EventFilter メソッドの翻訳です。このコードは，EventFilter をオーバーライドするときに QEvent からイベント型を取得する方法を示しています。ここでは説明を省きますが，QEvent オブジェクトは，イベント型がわかれば，その型専用の QEvent の下位クラス（QMouseEvent など）に変換することができます。

```
virtual bool __fastcall TCustomControl::EventFilter(Qt::QObjectH* Sender,
  Qt::QEventH* Event)
{
  bool retval = TWidgetControl::EventFilter(Sender, Event);
  switch (QEvent_type(Event))
  {
    case QEventType_Resize:
    case QEventType_FocusIn:
    case QEventType_FocusOut:
      UpdateMask();
  }
  return retval;
}
```

Qt イベントの生成

VCL コントロールがカスタム Windows メッセージを定義し，送信できるのと同様に，CLX コントロールでも Qt システムイベントを定義し，生成することができます。最初の手順はイベントのユニークな ID を定義することです（カスタム Windows メッセージを定義するときのメッセージ ID と同様）。

```
static const MyEvent_ID = (int) QCLXEventType_ClxUser + 50;
```

イベントを生成するコードで，QCustomEvent_create 関数（Qt.hpp で宣言されている）を使って新しいイベント ID のイベントオブジェクトを作成します。オブジェクトの第 2 パラメータを使用すると，イベントに関連付ける情報へのポインタを表すデータ値をイベントオブジェクトに提供できます。

```
QCustomEventH *MyEvent = QCustomEvent_create(MyEvent_ID, this);
```

イベント オブジェクトを作成すると，QApplication_postEvent メソッドを呼び出してこれを送信できます。

```
QApplication_postEvent(Application->Handle, MyEvent);
```

この通知に応答する任意のコンポーネントについて，単に EventFilter メソッドをオーバーライドし，MyEvent_ID のイベント型をチェックします。EventFilter メソッドは，Qt.hpp で宣言されている QCustomEvent_data メソッドを呼び出し，コンストラクタに提供したデータを取得できます。

第52章

コンポーネントを設計時に利用できるようにする

この章では，ユーザーが作成したコンポーネントを IDE で利用できるようにする手順について説明します。設計時にコンポーネントを操作できるようにするには，以下のステップが必要です。

- コンポーネントの登録
- パレットビットマップの追加
- コンポーネント用のヘルプの作成
- プロパティエディタの追加
- コンポーネントエディタの追加
- コンポーネントのコンパイルとパッケージ化

これらすべてのステップは，すべてのコンポーネントに必要なわけではありません。たとえば，新しいプロパティやイベントを定義しなかった場合は，それについてのヘルプを作成する必要はありません。必ず必要な手順は，登録とコンパイルです。

一度独自のコンポーネントを登録し，コンパイルしてパッケージ化すると，それをほかの開発者に配布したり，IDE にインストールできます。IDE にパッケージをインストールする方法についての詳細は，15-5 ページの「コンポーネントパッケージのインストール」を参照してください。

コンポーネントの登録

登録はコンパイルユニット別に行います。1 つのコンパイルユニットに複数のコンポーネントを作成した場合は，それらのコンポーネントは一度に登録できます。

コンポーネントを登録するには，コンポーネントのユニットの .CPP ファイルに Register 関数を記述します。Register 関数の中でコンポーネントを登録し，コンポーネントパレットのどこのページにそのコンポーネントをインストールするかを決めます。

メモ IDE で［コンポーネント｜コンポーネントの新規作成］を選択して，コンポーネントを作成した場合は，コンポーネントの登録に必要なコードが自動的に追加されます。

手作業でコンポーネントを登録する手順は，次のとおりです。

- Register 関数の宣言
- Register 関数の記述

Register 関数の宣言

コンポーネントを登録するには，コンポーネントのユニットの .CPP ファイルに Register という名前の関数を記述します。Register 関数は特定の名前空間内に記述しなければなりません。その名前空間は，コンポーネントを含むファイルの名前にします。また，先頭の文字以外はすべて小文字にします。

次のコードは，Register 関数を名前空間で実装する方法を示しています。ファイル名が Newcomp.CPP なので，名前空間の名前は Newcomp になります。

```
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {
  }
}
```

この Register 関数の中で，コンポーネントパレットに登録する各コンポーネントについて RegisterComponents を呼び出します。ヘッダーファイルと .CPP ファイルの組み合わせの中に複数のコンポーネントが含まれる場合は，それらを同時に登録できます。PACKAGE マクロが展開されると，クラスのインポートおよびエクスポートを許可する文になります。

Register 関数の記述

コンポーネントが所属するユニットの Register 関数の中で，コンポーネントパレットに追加する各コンポーネントを登録します。そのユニットに複数のコンポーネントが含まれる場合は，それらを同時に登録できます。

コンポーネントを登録するには，コンポーネントを追加するコンポーネントパレットのそれぞれのページに対して RegisterComponents 関数 を呼び出します。RegisterComponents では，以下の 3 つの重要な情報を扱います。

1. コンポーネントを指定する
2. パレットページを指定する
3. RegisterComponents 関数を使用する

コンポーネントを指定する

Register 関数の中で TComponentClass 型のオープン配列を宣言して，登録しようとするコンポーネントを格納します。構文は次のようになります。

```
TMetaClass classes[1] = {__classid(TNewComponent)};
```

この例では，クラスの配列格納されているコンポーネントは 1 つだけですが，登録したいコンポーネントすべてを配列に格納することもできます。たとえば，次のコードは 2 つのコンポーネントを配列に記述しています。

```
TMetaClass classes[2] = {__classid(TNewComponent), __classid(TAnotherComponent)};
```

コンポーネントを配列に格納するには，独立した文でコンポーネントを配列に代入する方法もあります。次の文は，前述の例と同じ 2 つのコンポーネントを配列に格納しています。

```
TMetaClass classes[2];
classes[0] = __classid(TNewComponent);
classes[1] = __classid(TAnotherComponent);
```

パレットページを指定する

パレットページの名前は AnsiString 型です。指定した名前のパレットページが存在しない場合は，Delphi はその名前で新しいページを作成します。Delphi では，標準ページの名前を文字列リストリソースに格納します。したがって，国際化対応バージョンの製品では各国の言語でページの名前を付けることができます。標準ページの 1 つにコンポーネントをインストールする場合は，LoadStr を呼び出して，そのページ名の文字列を取得します。その際，そのページの文字列リソースを表す定数を渡します。たとえば，［System］ページの場合は srSystem を渡します。

RegisterComponents 関数を使用する

Register 関数の中では，RegisterComponents を呼び出して，クラス配列内のコンポーネントを登録します。,RegisterComponents 関数は 3 つのパラメータをとります。それは，コンポーネントパレットページの名前，コンポーネントクラスの配列，およびその配列の最後の要素のインデックスです。

NEWCOMP.CPP ファイル内の Register 関数を次に示します。ここでは，TMyComponent という名前のコンポーネントを登録し，コンポーネントパレットページに配置します。

```
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {
      TMetaClass classes[1] = {__classid(TMyComponent)};
      RegisterComponents("Miscellaneous", classes, 0);
  }
}
```

RegisterComponents を呼び出すときの 3 番目の引数は 0 であることに注意してください。これは，クラス配列の最後の要素のインデックス（配列のサイズ－ 1）を意味します。

また，複数のコンポーネントを同じページに一度に登録したり，複数のコンポーネントを複数のページに登録することもできます。次にそのような例を示します。

```
namespace Mycomps
{
 void __fastcall PACKAGE Register()
  {
      // 2 つのコンポーネントを格納する配列を宣言する
      TMetaClass classes1[2] = {__classid(TFirst), __classid(TSecond)};
      // 配列 classes1 に格納されている 2 つのコンポーネントを新しいパレットページに追加する
      RegisterComponents("Miscellaneous", classes1, 1);
      // 2 番目の配列を宣言する
      TMetaClass classes2[1];
```

```
        // 1 つのコンポーネントを配列の最初の要素として割り当てる
        classes2[0]  = __classid(TThird);
        // classes2 配列のコンポーネントを Samples ページに追加する
        RegisterComponents("Samples", classes2, 0);
    }
}
```

この例では，2 つの配列，classes1 と classes2 が宣言されています。最初の RegisterComponents 呼び出しでは，配列 classes1 の要素数が 2 なので，3 番目の引数は配列の 2 番目の要素のインデックス，すなわち 1 になります。2 回目の RegisterComponents 呼び出しでは，配列 classes2 の要素数が 1 なので，3 番目の引数は 0 になります。

パレットビットマップの追加

すべてのコンポーネントには，そのコンポーネントをコンポーネントパレットで表示するためのビットマップが必要です。独自のビットマップを指定しなければ，C++Builder ではデフォルトのビットマップが使用されます。

パレットビットマップが必要なのは設計時だけなので，パレットビットマップをコンポーネントのコンパイルユニットの中に含めてコンパイルする必要はありません。パレットビットマップは Windows リソースファイルで提供します。ファイルの名前は，.CPP ファイルと同じ名前に拡張子 DCR（Dynamic Component Resource）を付けたものにします。リソースファイルは C++Builder のイメージエディタで作成できます。ビットマップの大きさは 24 ピクセル四方にします。

インストールする各コンポーネントに 1 つずつパレットビットマップファイルを作成し，そのファイル内に，登録する各コンポーネントに 1 つずつビットマップを作成します。このビットマップイメージにはコンポーネントクラスと同じ名前を付けます。パレットビットマップファイルは，コンパイル済みのファイルと同じディレクトリに入れておきます。こうすれば，コンポーネントをコンポーネントパレットにインストールする際に C++Builder がビットマップを探し出せます。

たとえば，TMyControl という名前のコンポーネントを作成した場合は，TMYCONTROL という名前のビットマップを含む .DCR または .RES リソースファイルを作成する必要があります。リソース名では大文字と小文字は区別されませんが，通常は大文字を使用します。

コンポーネント用のヘルプの作成

一般に，フォームで標準コンポーネントを選択したときや，オブジェクトインスペクタでプロパティやイベントを選択したときに【F1】を押すと，その項目についてのヘルプが呼び出されます。開発したコンポーネントにヘルプファイルを用意しておけば，開発者は同じようにヘルプを利用できます。

独自のコンポーネントについて説明した小さなヘルプファイルを提供できます。作成したヘルプファイルは総合的な C++Builder ヘルプシステムの一部に組み込まれます。

コンポーネントとともに使用するヘルプファイルの作成方法についての詳細は，52-5 ページの「ヘルプファイルの作成」を参照してください。

ヘルプファイルの作成

Windows のヘルプファイルのソースファイル（.rtf 形式）は，どのツールを使って作成してもかまいません。C++Builder には，ヘルプファイルをコンパイルし，オンラインヘルプのオーサリングガイドを提供する Microsoft Help Workshop が用意されています。ヘルプファイルの作成についての完全な情報は，Help Workshop のオンラインガイドを参照してください。

コンポーネント用のヘルプファイルを作成するには，次の 3 つの手順を行います。

- 項目を作成する
- コンポーネントのヘルプを状況感知型にする
- コンポーネントのヘルプファイルを追加する

項目を作成する

独自のコンポーネントのヘルプを，ライブラリ中のほかのコンポーネントのヘルプとシームレスに統合するには，次の規則を守ってください。

1. 各コンポーネントには 1 つのヘルプトピックを割り当てる

コンポーネントトピックでは，そのコンポーネントがどのユニット内で宣言されているかを示し，そのコンポーネントを簡単に説明します。コンポーネントトピックからリンクされた関連項目のウィンドウでは，オブジェクト階層におけるそのコンポーネントの位置についての説明や，そのコンポーネントのすべてのプロパティ，イベント，およびメソッドの一覧を表示します。アプリケーション開発者は，フォーム上でこのコンポーネントを選択し【F1】を押すと，このトピックにアクセスできます。たとえば，任意のコンポーネントをフォームに配置して【F1】を押します。

コンポーネントトピックには，そのトピックに固有の値を持つ # 脚注がなければなりません。# 脚注は，ヘルプシステムで各トピックをユニークに識別します。

コンポーネントトピックには，ヘルプシステムインデックス内でのキーワード検索用に K 脚注を付加し，キーワードには，そのコンポーネントクラス名を含めます。たとえば，TMemo コンポーネント用のキーワード脚注は「TMemo」になります。

コンポーネントトピックには，トピックタイトルを提供する文字 $ の脚注も必要です。このタイトルは，［トピックの検索］ダイアログボックス，［しおりの設定］ダイアログボックス，および［ヒストリ］ウィンドウに表示されます。

2. 各コンポーネントに，二次的なジャンプトピックを用意する

- 階層トピック。コンポーネント階層中のそのコンポーネントのすべての上位コンポーネントへのリンクを含む
- そのコンポーネントで利用可能なすべてのプロパティの一覧。それらのプロパティを説明する項目へのリンクを含む
- そのコンポーネントで利用可能なすべてのイベントの一覧。それらのイベントを説明する項目へのリンクを含む
- そのコンポーネントで利用可能なメソッドの一覧。それらのメソッドを説明する項目へのリンクを含む

C++Builder のヘルプシステム内のオブジェクトクラス，プロパティ，メソッド，またはイベントへのリンクは，Alink を使って作成します。オブジェクトクラスへのリンクを作成する場合は，Alink でそのオブジェクトのクラス名を使います。その際，クラス名の後に下線文字（_）と文字列「object」を付けます。たとえば，TCustomPanel オブジェクトへのリンクは次のようになります。

```
!AL(TCustomPanel_object,1)
```

プロパティ，メソッド，またはイベントへのリンクを作成する場合は，プロパティ，メソッド，またはイベント名の前にオブジェクト名と下線文字を置きます。たとえば，TControl で実装されている Text プロパティへのリンクを作成するには，次のようにします。

```
!AL(TControl_Text,1)
```

試しに関連項目を表示するには，任意のコンポーネントのヘルプを表示して，階層，プロパティ，メソッド，またはイベントというラベルの付いたリンクをクリックします。

3. **コンポーネント内で宣言されている各プロパティ，メソッド，およびイベントは，1 つのトピックを割り当てる**

   プロパティ，イベント，またはメソッドに関するトピックでは，その項目の宣言を表示し，その使い方を説明します。アプリケーション開発者は，オブジェクトインスペクタでその項目を強調表示させて【F1】を押すか，またはコードエディタでその項目の名前にカーソルを移動させて【F1】を押すことにより，その項目の画面を表示します。試しにプロパティに関するトピックを参照するには，オブジェクトインスペクタ内で任意のプロパティを選択して【F1】を押します。

   プロパティ，イベント，およびメソッドに関するトピックには，K 脚注を付加し，プロパティ，メソッド，またはイベントの名前と，その名前とコンポーネント名との組み合わせのリストを記述します。たとえば，TControl の Text プロパティの K 脚注は次のようになります。

   ```
   Text,TControl;TControl,Text;Text,
   ```

   プロパティ，メソッド，およびイベントに関するトピックには，$ 脚注も付加する必要があります。これは，TControl::Text のようなトピックタイトルを表します。

これらすべてのトピックには，トピックを一意に識別するためのトピック ID を付加します。これは文字 # の脚注として入力します。

コンポーネントのヘルプを状況感知型にする

各コンポーネント，プロパティ，メソッド，およびイベントに関するトピックには，文字 A の脚注を付けなければなりません。A 脚注が使われるのは，ユーザーがコンポーネントを選択して【F1】を押した場合，またはオブジェクトインスペクタ内でプロパティまたはイベントを選択して【F1】を押した場合です。A 脚注を記述する場合は，以下のような命名規則に従わなければなりません。

コンポーネントのヘルプトピックの場合は，A 脚注は，次の構文のように 2 つの項目で構成し，項目の間をセミコロンで区切ります。

```
ComponentClass_Object;ComponentClass
```

ComponentClass はコンポーネントクラス名を表します。

プロパティまたはイベントのヘルプトピックの場合は，A 脚注は，次の構文のように 3 つの項目で構成し，項目の間をセミコロンで区切ります。

```
ComponentClass_Element;Element_Type;Element
```

ComponentClass はコンポーネントクラス名を，Element はプロパティ，メソッド，またはイベント名を表します。Type は Property，Method，または Event のいずれかになります。

たとえば，TMyGrid というコンポーネントの BackgroundColor プロパティ用の A 脚注は次のようになります。

```
TMyGrid_BackgroundColor;BackgroundColor_Property;BackgroundColor
```

コンポーネントのヘルプファイルを追加する

独自のヘルプファイルを C++Builder のヘルプに追加するには，bin ディレクトリの OpenHelp ユーティリティ（oh.exe ファイル）を使用します。また，このユーティリティへは，IDE で［ヘルプ｜カスタマイズ］を使用してアクセスできます。

OpenHelp.hlp ファイルには，ヘルプシステムに新しいヘルプファイルを追加する方法も含めて，OpenHelp の使用方法が説明されています。

プロパティエディタの追加

オブジェクトインスペクタでは，すべての型のプロパティに対応したデフォルトの編集機能が用意されています。しかし，プロパティエディタを記述して登録すれば，特定のプロパティに対応したエディタを使用できます。開発したコンポーネントのプロパティのみに適用するプロパティエディタを登録することができ，また，ある型のすべてのプロパティに適用するエディタを作成することもできます。

もっとも簡単なレベルでは，プロパティエディタは 2 つの方法の一方，または両方で使用されます。つまり，現在値をテキスト文字列として表示してユーザーに編集させる方法と，その他の編集機能を備えたダイアログボックスを表示する方法です。開発者は，編集するプロパティにとってどの方法が便利であるか判断して，どちらの方法を使用するか，両方を使用するかを決定します。

プロパティエディタの開発には，以下の 5 つのステップが必要です。

1. プロパティエディタクラスを派生させる
2. プロパティをテキストとして編集する
3. プロパティ全体を編集する
4. エディタの属性を指定する
5. プロパティエディタを登録する

プロパティエディタクラスを派生させる

VCL と CLX には数種類のプロパティエディタが定義されていますが，これらのエディタはすべて，TPropertyEditor を継承するクラスです。プロパティエディタを作成する場合は，TPropertyEditor から直接プロパティエディタクラスを派生させる方法と，表 52.1 に示すプロパティエディタクラスの 1 つから派生させる方法のいずれかを使用します。DesignEditors ユニットにあるクラスは，VCL アプリケーションと CLX アプリケーションのどちらにも使用できます。ただし，専用ダイアログを提供するプロパティエディタクラスがいくつかあります。この場合は VCL と CLX のどちらかにしか対応し

ません。このようなプロパティエディタクラスは，それぞれ VCLEditors ユニットと CLXEditors ユニットにあります。

メモ プロパティエディタに絶対に必要なことは，TBasePropertyEditor を継承すること，および IProperty インターフェースをサポートすることです。ただし，TPropertyEditor は IProperty インターフェースのデフォルトの実装を提供します。

表 52.1 のリストは完全ではありません。VCLEditors ユニットと DsgnIntf ユニットには，コンポーネント名などの特殊のプロパティに対して使用するいくつかの特殊なプロパティエディタも定義されています。次の表に，ユーザー定義プロパティにとってもっとも有用なプロパティエディタを示します。

表 52.1 定義済みのプロパティエディタの型

型	編集対象のプロパティ
TOrdinalProperty	すべての順序型（整数型，文字型，列挙型）のプロパティのためのプロパティエディタは，TOrdinalProperty を継承する
TIntegerProperty	すべての整数型。定義済みの部分範囲型とユーザー定義の部分範囲型を含む
TCharProperty	Char 型と，Char の部分範囲型（'A'..'Z' など）
TEnumProperty	すべての列挙型
TFloatProperty	すべての浮動小数点数
TStringProperty	AnsiStrings 型
TSetElementProperty	集合の個々の要素。論理値として表示される
TSetProperty	すべての集合。集合は直接には編集できないが，集合要素プロパティのリストに展開できる
TClassProperty	クラス。クラスの名前を表示し，そのクラスのプロパティを展開できる
TMethodProperty	メソッドポインタ。特に，イベントを扱う
TComponentProperty	同じフォーム上のコンポーネント。ユーザーはコンポーネントのプロパティを編集できないが，互換性のある型の特定のコンポーネントを指示できる
TColorProperty	コンポーネントの色。色定数に該当すればそれを表示し，そうでない場合は 16 進数で表示する。ドロップダウンリストには色定数が入っている。ダブルクリックすると色選択ダイアログボックスが開く
TFontNameProperty	フォント名。ドロップダウンリストには現在インストールされているすべてのフォントが表示される
TFontProperty	フォント。フォントダイアログボックスにアクセスでき，また，個々のフォントプロパティを展開できる

次の例は，TMyPropertyEditor という名前の簡単なプロパティエディタの宣言を示しています。

```
class PACKAGE TMyPropertyEditor : public TPropertyEditor
{
  public:
    virtual bool __fastcall AllEqual(void);
    virtual System::AnsiString __fastcall GetValue(void);
    virtual void __fastcall SetValue(const System::AnsiString Value);
   __fastcall virtual ~TMyPropertyEditor(void) { }
    __fastcall TMyPropertyEditor(void) : Dsgnintf::TPropertyEditor() { }
};
```

プロパティをテキストとして編集する

すべてのプロパティには，オブジェクトインスペクタで表示するために，値の文字列表現を用意しなければなりません。また，ほとんどのプロパティでは，ユーザーはそのプロパティに新しい値を入力できます。プロパティエディタクラスの仮想メソッドをオーバーライドすることで，文字列表現と実際の値とを変換します。

オーバーライドするメソッドの名前は，GetValue と SetValue です。プロパティエディタはまた，表 52.2 で示すように，さまざまな型の値の代入と読み出しに使用されるメソッド群を継承します。.

表 52.2　プロパティの値を読み書きするためのメソッド

プロパティの型	Get メソッド	Set メソッド
浮動小数点数	GetFloatValue	SetFloatValue
クロージャ（イベント）	GetMethodValue	SetMethodValue
序数型	GetOrdValue	SetOrdValue
文字列	GetStrValue	SetStrValue

GetValue メソッドをオーバーライドする場合は，Get メソッドの 1 つを呼び出します。SetValue メソッドをオーバーライドする場合は，Set メソッドの 1 つを呼び出します。

プロパティの値を表示する

プロパティエディタの GetValue メソッドは，そのプロパティの現在値を表す文字列を返します。オブジェクトインスペクタは，この文字列をそのプロパティの値列に表示します。デフォルトでは，GetValue は「unknown」を返します。

独自のプロパティの文字列表現を用意するには，プロパティエディタの GetValue メソッドをオーバーライドします。

プロパティが文字列値でない場合は，GetValue はその値を文字列表現に変換しなければなりません。

プロパティの値を設定する

プロパティエディタの SetValue メソッドは，ユーザーがオブジェクトインスペクタで入力した文字列を受け取り，その文字列を適切な型に変換し，プロパティの値を設定します。受け取った文字列がプロパティの適正な値を表していない場合は，SetValue は例外を送って，誤った値を使用しないようにしなければなりません。

文字列値をプロパティに読み込むには，プロパティエディタの SetValue メソッドをオーバーライドします。

SetValue は文字列を値に変換し，検証してから Set メソッドの 1 つを呼び出します。

プロパティ全体を編集する

オプションとして，ユーザーが視覚的にプロパティを編集できるダイアログボックスを用意できます。プロパティエディタがもっともよく使用されるのは，それ自身がクラスであるプロパティに対し

てです。たとえば，Font プロパティの場合，ユーザーはフォントダイアログボックスを開いて，特定のフォントのすべての属性を一度に選択できます。

プロパティ全体を編集するエディタダイアログボックスを用意するには，プロパティエディタクラスの Edit メソッドをオーバーライドします。

Edit メソッドは，GetValue メソッドと SetValue メソッドを記述する際に使用したのと同じ Get メソッドと Set メソッドを使用します。実際は，Edit メソッドは Get メソッドと Set メソッドの両方を呼び出します。このエディタは特定の型を扱うので，通常はプロパティ値を文字列に変換する必要はありません。このエディタは一般に，値を「取り出された状態のまま」処理します。

ユーザーがプロパティの隣の［...］ボタンをクリックするか，値列をダブルクリックすると，オブジェクトインスペクタがプロパティエディタの Edit メソッドを呼び出します。

Edit メソッドを実装する手順は次のとおりです。

1. プロパティに使用するエディタダイアログを構築します。
2. Get メソッドを使って，プロパティの現在の値を読み出します。
3. ユーザーが新しい値を選択した場合は，Set メソッドを使ってその値をプロパティに代入します。
4. エディタダイアログを破棄します。

エディタの属性を指定する

オブジェクトインスペクタが適切なツールを表示できるように，プロパティエディタはいくつかの情報を通知しなければなりません。たとえば，オブジェクトインスペクタは，そのプロパティにサブプロパティがあるかどうか，または可能な値のリストを表示できるかどうかを知っていなければなりません。

エディタ属性を指定するには，プロパティエディタの GetAttributes メソッドをオーバーライドします。

GetAttributes は TPropertyAttributes 型の値の集合を返すメソッドです。返される値には，次に示す値のいずれかまたはすべてが含まれます。

表 52.3　プロパティエディタの属性フラグ

フラグ	関連メソッド	含まれている場合の意味
paValueList	GetValues	このエディタは列挙型の値のリストを提供できる
paSubProperties	GetProperties	このプロパティには表示できるサブプロパティがある
paDialog	Edit	このエディタはプロパティ全体を編集するためのダイアログボックスを表示できる
paMultiSelect	なし	このプロパティは，ユーザーが 2 つ以上のコンポーネントを選択したときのみ表示する
paAutoUpdate	SetValue	変更されるたびに，値の承認を待たずにコンポーネントを更新する
paSortList	なし	オブジェクトインスペクタが，値リストをソートする必要がある
paReadOnly	なし	ユーザーはそのプロパティ値を変更できない

表 52.3 プロパティエディタの属性フラグ（つづき）

フラグ	関連メソッド	含まれている場合の意味
paRevertable	なし	オブジェクトインスペクタのコンテキストメニューで［継承元の値に戻す］を使用可能にする。メニュー項目の指定によって，プロパティエディタは現在のプロパティ値を破棄し，以前に設定されていたデフォルト値または標準値に戻す
paFullWidthName	なし	この値は表示する必要がない。その代わりに，オブジェクトインスペクタがその全幅をプロパティ名に使用する
paVolatileSubProperties	GetProperties	オブジェクトインスペクタは，プロパティ値が変更されるたびにすべてのサブプロパティの値を取得し直す
paReference	GetComponentValue	この値はほかのオブジェクトへの参照である。paSubProperties とともに使用すると，参照されたオブジェクトはこのプロパティのサブプロパティとして表示される

Color プロパティは，ほかのプロパティと比べてかなり柔軟に使用されます。ユーザーはオブジェクトインスペクタで，入力，リストからの選択，カスタマイズされたエディタによる指定など，いくつかの方法でこのプロパティを選択できます。このため，TColorProperty の GetAttributes メソッドの戻り値には，次のようにいくつかの属性が入っています。

```
virtual __fastcall TPropertyAttributes TColorProperty::GetAttributes()
{
  return TPropertyAttributes() << paMultiSelect << paDialog << paValueList << paRevertable;
}
```

プロパティエディタを登録する

プロパティエディタを作成したら，次にそれを C++Builder に登録する必要があります。プロパティエディタを登録すると，プロパティの型と特定のプロパティエディタとが関連付けられます。エディタの登録では，特定の型を持つすべてのプロパティを指定することも，特定の型のコンポーネントの特定のプロパティだけを指定することもできます。

プロパティエディタを登録するには，RegisterPropertyEditor 関数を呼び出します。

RegisterPropertyEditor は以下の 4 つのパラメータを受け取ります。

- 編集するプロパティの型を示す型情報ポインタ。型情報を次の方法で指定します。

  ```
  __typeinfo(TMyComponent)
  ```

- エディタが適用されるコンポーネントの型。このパラメータが NULL の場合は，指定された型のプロパティすべてにこのエディタが適用されます。
- プロパティの名前。このパラメータは，直前のパラメータによって特定の型のコンポーネントが指定された場合にのみ有効です。この場合には，このエディタが適用されるコンポーネント型の特定のプロパティの名前を指定できます。
- 指定のプロパティの編集に使用されるプロパティエディタの型

コンポーネントパレットの標準コンポーネント用のエディタを登録する関数の一部を例として次に示します。

```
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {
    RegisterPropertyEditor(__typeinfo(TComponent), 0L, "", __classid(TComponentProperty));
    RegisterPropertyEditor(__typeinfo(TComponentName), __classid(TComponent), "Name",
       __classid(TComponentNameProperty));
    RegisterPropertyEditor(__typeinfo(TMenuItem), __classid(TMenu), "",
       __classid(TMenuItemProperty));
    ...
  }
}
```

この関数の 3 つの文は，以下の 3 通りの RegisterPropertyEditor の使用を記述しています。

- 最初の文は，もっとも一般的な登録を行っています。この文では，TComponent 型（または登録済みの独自のエディタを持たない TComponent の下位型）のプロパティすべてに対して，プロパティエディタ TComponentProperty を登録しています。一般に，プロパティエディタの登録では，ある特定の型のためのエディタを作成し，そのエディタをその型のすべてのプロパティに対して指定します。このため，2 番目と 3 番目のパラメータがそれぞれ NULL と空文字列になっています。
- 2 番目の文は，もっとも限定的な登録を行っています。この文では，指定の型のコンポーネントの指定のプロパティ用にエディタを登録しています。この例では，このエディタはすべてのコンポーネントに対して TComponentName 型で定義されている Name プロパティに対応しています。
- 3 番目の文は，1 番目の文よりは限定的で，2 番目の文よりは一般的な登録を行っています。この文では，TMenu 型のコンポーネントの TMenuItem 型のすべてのプロパティ用にエディタを登録しています。

プロパティのカテゴリ

IDE では，適宜，オブジェクトインスペクタに表示するプロパティをカテゴリ別に選択できます。新しいカスタムコンポーネントを作成した場合，プロパティをカテゴリに登録することで，同じようにプロパティの表示，非表示ができます。コンポーネントの登録と同時にプロパティのカテゴリを登録するには，RegisterPropertyInCategory または RegisterPropertiesInCategory を呼び出します。1 つのプロパティを登録するには，RegisterPropertyInCategory を使います。複数のプロパティを単一の関数呼び出しで登録するには，RegisterPropertiesInCategory を使います。どちらの関数も，DesignIntf ユニットで定義されています。

ただし，プロパティの登録は必須ではありません。また，同じカスタムコンポーネントのプロパティをいくつか登録する場合も，必ずしもすべてのプロパティを登録する必要はありません。プロパティを何らかのカテゴリと明示的に関連付けていない場合，そのプロパティは TMiscellaneousCategory カテゴリに含まれます。TMiscellaneousCategory に属するプロパティは，デフォルトのカテゴリ設定によってオブジェクトインスペクタに表示されるか，または非表示になります。

プロパティを登録する関数には，上に挙げた 2 つの関数のほかに，IsPropertyInCategory 関数があります。プロパティを一定のカテゴリに登録しなければならないローカライズユーティリティを作成する場合には，IsPropertyInCategory 関数を使うと便利です。

プロパティを 1 つずつ登録する

プロパティを 1 回に 1 つずつ登録してプロパティのカテゴリと関連付けるには，RegisterPropertyInCategory 関数を使います。RegisterPropertyInCategory には 4 つのオーバーロードバリエーションがあります。それぞれ違った基準でカスタムコンポーネントのプロパティを識別し，プロパティカテゴリと結び付けます。

第 1 のバリエーションは，プロパティをプロパティ名で識別します。次の例では，プロパティ名「AutoSize」で識別し，コンポーネントの視覚表示に関連するプロパティとして登録しています。

```
RegisterPropertyInCategory("Visual", "AutoSize");
```

第 2 のバリエーションは第 1 のバリエーションと似ていますが，特定の型のコンポーネントに表示される特定の名前のプロパティだけにカテゴリを限定する点が異なります。次の例は，カスタムクラス TMyButton のコンポーネントの HelpContext プロパティを「Help and Hints」カテゴリに登録しています。

```
RegisterPropertyInCategory("Help and Hints", __classid(TMyButton), "HelpContext");
```

第 3 のバリエーションは，プロパティの名前ではなく型を使ってプロパティを識別します。次の例は，プロパティの型（TArrangement と呼ばれる特殊なクラス）に基づいてプロパティを登録しています。

```
RegisterPropertyInCategory("Visual", typeid(TArrangement));
```

第 4 のバリエーションは，プロパティの型と名前の両方を使ってプロパティを識別します。次の例は，プロパティの型（TBitmap）とプロパティ名（Pattern）の組み合わせに基づいてプロパティを登録しています。

```
RegisterPropertyInCategory("Visual", typeid(TBitmap), "Pattern");
```

プロパティカテゴリの一覧と用途の概略については，プロパティカテゴリを指定するを参照してください。

複数のプロパティを一度に登録する

複数のプロパティを一度に登録してプロパティのカテゴリと関連付けるには，RegisterPropertiesInCategory 関数を使います。RegisterPropertiesInCategory には 3 つのオーバーロードバリエーションがあります。それぞれ違った基準でカスタムコンポーネントのプロパティを識別し，プロパティカテゴリと結び付けます。

第 1 のバリエーションは，プロパティをプロパティの名前または型で識別します。この場合，リストを定数の配列として渡します。次の例は，「Text」という名前か，TEdit 型のクラスに属するプロパティを Localizable カテゴリに登録しています。

```
RegisterPropertiesInCategory("Localizable", ARRAYOFCONST("Text", __typeinfo(TEdit)));
```

第 2 のバリエーションは，登録されるプロパティを，特定のコンポーネントに属するプロパティに限定します。登録されるプロパティのリストには名前だけが含まれ，型は含まれません。たとえば，次のコードでは，すべてのコンポーネントの「Help and Hints」カテゴリにいくつかのプロパティを登録しています。

```
RegisterPropertyInCategory("Help and Hints", __classid(TComponent),
                           ARRAYOFCONST("HelpContext", "Hint", "ParentShowHint"));
```

第 3 のバリエーションは，登録されるプロパティを，特定の型を持つプロパティに限定します。第 2 のバリエーションと同様に，登録されるプロパティのリストには名前だけが含まれます。

```
RegisterPropertiesInCategory("Localizable", __typeinfo(TStrings),
                             ARRAYOFCONST("Lines", "Commands"));
```

プロパティカテゴリの一覧と用途の概略については，プロパティカテゴリを指定するを参照してください。

プロパティカテゴリを指定する

プロパティをカテゴリに登録する場合，カテゴリの名前にはどのような文字列でも使用できます。以前に使用されていない文字列を使用すると，その名前を持つ新しいプロパティカテゴリクラスをオブジェクトインスペクタが生成します。また，組み込みのカテゴリの 1 つにプロパティを登録することもできます。表 52.4 に組み込みのプロパティカテゴリを示します。

表 52.4 プロパティのカテゴリ

カテゴリ	目的
Action（アクション）	実行時動作に関連するプロパティ。TEdit の Enabled プロパティと Hint プロパティはこのカテゴリに属する
Database（データベース）	データベース操作に関連するプロパティ。TQuery の DatabaseName プロパティと DragKind プロパティはこのカテゴリに属する
Drag, Drop, and Docking（ドラッグ / ドロップ / ドッキング）	ドラッグアンドドロップ操作とドッキング操作に関連するプロパティ。TImage の DragCursor プロパティと DragKind プロパティはこのカテゴリに属する
Help and Hints	オンラインヘルプとヘルプヒントに関連するプロパティ。TMemo の HelpContext プロパティと Hint プロパティはこのカテゴリに属する
Layout（レイアウト）	設計時のコントロールの視覚表示に関連するプロパティ。TDBEdit の Top プロパティと Left プロパティはこのカテゴリに属する
Legacy（旧式）	旧式の操作に関連するプロパティ。TComboBox の Ctl3D プロパティと ParentCtl3D プロパティはこのカテゴリに属する
Linkage（リンク）	コンポーネント間の関連付けまたはリンクに関連するプロパティ。TDataSource の DataSet プロパティはこのカテゴリに属する
Locale（ロケール）	国際化対応ロケールに関連するプロパティ。TMainMenu の BiDiMode プロパティと ParentBiDiMode プロパティはこのカテゴリに属する
Localizable（ローカライズ対象）	アプリケーションのローカライズバージョンでは変更する必要がある可能性があるプロパティ。Caption などの文字列プロパティの多くはこのカテゴリに属する。コントロールのサイズと位置を決めるプロパティもこれに含まれる
Visual（表示）	実行時のコントロールの視覚表示に関連するプロパティ。TScrollBox の Align プロパティと Visible プロパティはこのカテゴリに属する
Input（入力）	データ入力に関連するプロパティ（必ずしもデータベース操作に関係しない）。TEdit の Enabled プロパティと ReadOnly プロパティはこのカテゴリに属する
Miscellaneous（その他）	どのカテゴリにも該当しないか，分類する必要のないプロパティ（特定のカテゴリに明示的に登録していないプロパティを含む）。TSpeedButton の AllowAllUp プロパティと Name プロパティはこのカテゴリに属する

IsPropertyInCategory 関数の使い方

あるプロパティが特定のカテゴリに登録済みかどうかを知るために，アプリケーション側で既存の登録プロパティを問い合わせることができます。ローカライズユーティリティがローカライズ操作の前にプロパティの分類をチェックするような場合，この機能を使うと特に便利です。
IsPropertyInCategory 関数には 2 つのオーバーロードバリエーションがあり，プロパティが何らかのカテゴリに属しているかどうかを，それぞれ異なる基準で判断します。

第 1 のバリエーションは，所有コンポーネントのクラス型と，プロパティ名の組み合わせを基準にして比較します。次のコマンドラインで IsPropertyInCategory が **true** を返すには，プロパティが TCustomEdit の下位に属し，プロパティ名が「Text」であり，かつ Localizable カテゴリに属していなければなりません。

```
IsItThere = IsPropertyInCategory("Localizable", __classid(TCustomEdit), "Text");
```

第 2 のバリエーションは，所有コンポーネントのクラス名と，プロパティ名の組み合わせを基準にして比較します。次のコマンドラインで IsPropertyInCategory が **true** を返すには，プロパティが TCustomEdit の下位であり，プロパティ名が「Text」であり，Localizable カテゴリに属していなければなりません。

```
IsItThere = IsPropertyInCategory("Localizable", "TCustomEdit", "Text");
```

コンポーネントエディタの追加

コンポーネントエディタは，デザイナ内でコンポーネントをダブルクリックした場合の動作を決定します。また，コンポーネントを右クリックしたときに表示されるコンテキストメニューにコマンドを追加します。さらに，コンポーネントエディタを使用して，独自のコンポーネントを Windows のクリップボードに独自の形式でコピーできます。

独自のコンポーネント用のコンポーネントエディタを作成しない場合は，C++Builder はデフォルトのコンポーネントエディタを使います。デフォルトのコンポーネントエディタは，TDefaultEditor クラスで実装されています。TDefaultEditor は，コンポーネントのコンテキストメニューに新しい項目を追加しません。コンポーネントがダブルクリックされた場合は，TDefaultEditor はそのコンポーネントのプロパティを検索し，最初に見つかったイベントハンドラを作成します（またはイベントハンドラに移動します）。

コンテキストメニューに項目を追加したり，コンポーネントがダブルクリックされた場合の動作を変更したり，新しいクリップボード形式を追加するには，TComponentEditor から新しいクラスを派生させ，それを独自のコンポーネントで使用できるように登録します。オーバーライドしたメソッドでは，TComponentEditor の Component プロパティを使用して，編集対象のコンポーネントにアクセスできます。

カスタムコンポーネントエディタを追加する手順は，次のとおりです。

- コンテキストメニューに項目を追加する
- ダブルクリック時の動作を変更する
- クリップボード形式を追加する
- コンポーネントエディタを登録する

コンテキストメニューに項目を追加する

ユーザーがコンポーネントを右クリックすると，コンポーネントエディタの GetVerbCount メソッドと GetVerb メソッドが呼び出されて，コンテキストメニューが作られます。これらのメソッドをオーバーライドすれば，コンテキストメニューにコマンド（verb）を追加できます。

コンテキストメニューに項目を追加するには，次の手順が必要です。

- メニュー項目を指定する
- コマンドを実装する

メニュー項目を指定する

GetVerbCount メソッドをオーバーライドして，コンテキストメニューに追加するコマンドの数を返すようにします。GetVerb メソッドをオーバーライドして，これらの各コマンド用に追加する文字列を返すようにします。GetVerb をオーバーライドする場合は，メニュー項目のショートカットキーにしたい文字の前にアンド記号（&）を追加して，そのコンテキストメニューで下線を表示します。そのコマンドによってダイアログボックスが表示される場合は，必ずコマンドの末尾に省略記号（...）を追加してください。GetVerb は，コマンドのインデックスを表すただ 1 つのパラメータを取ります。

次のコードは，GetVerbCount メソッドと GetVerb メソッドをオーバーライドして，コンテキストメニューに 2 つのコマンドを追加します。

```
int __fastcall TMyEditor::GetVerbCount(void)
{
  return 2;
}
System::AnsiString __fastcall TMyEditor::GetVerb(int Index)
{
  switch (Index)
  {
    case 0: return "&DoThis ..."; break;
    case 1: return "Do&That"; break;
  }
}
```

メモ GetVerb メソッドは，GetVerbCount で表されるすべてのインデックスに対して値を返すようにしておく必要があります。

コマンドを実装する

GetVerb によって提供されたコマンドがデザイナ内で選択された場合は，ExecuteVerb メソッドが呼び出されます。ExecuteVerb メソッドでは，GetVerb メソッドで提供したそれぞれのコマンドの動作を実装します。編集中のコンポーネントにアクセスするには，エディタの Component プロパティを使います。

たとえば，次に示す ExecuteVerb メソッドは，前述の例の GetVerb メソッドに対応するコマンドを実装しています。

```
void __fastcall TMyEditor::ExecuteVerb(int Index)
{
  switch (Index)
  {
    case 0:
```

```
      TMyDialog *MySpecialDialog = new TMyDialog();
      MySpecialDialog->Execute();
      ((TMyComponent *)Component)->ThisProperty = MySpecialDialog->ReturnValue;
      delete MySpecialDialog;
      break;
    case 1:
      That();  // That メソッドの呼び出し
      break;
  }
}
```

ダブルクリック時の動作を変更する

コンポーネントがダブルクリックされると，コンポーネントエディタの Edit メソッドが呼び出されます。デフォルトでは，Edit メソッドは，コンテキストメニューに追加されている最初のコマンドを実行します。したがって，前の例では，コンポーネントをダブルクリックすると，MySpecialDialog が実行されます。

最初のコマンドを実行するということで通常は十分ですが，このデフォルトの動作を変更したい場合もあります。たとえば，以下のような場合は，別の動作をさせたいと考えるでしょう。

- コンテキストメニューにまったくコマンドを追加していない場合
- コンポーネントがダブルクリックされた場合に，複数コマンドを組み合わせたダイアログを表示したい場合

コンポーネントがダブルクリックされた場合に新しい動作をさせるには，Edit メソッドをオーバーライドします。たとえば，次に示す Edit メソッドでは，ユーザーがコンポーネントをダブルクリックすると，フォントダイアログを表示します。

```
void __fastcall TMyEditor::Edit(void)
{
  TFontDialog *pFontDlg = new TFontDialog(NULL);
  pFontDlg->Execute();
  ((TMyComponent *)Component)->Font = pFontDlg->Font;
  delete pFontDlg;
}
```

メモ コンポーネントをダブルクリックして，イベントハンドラ用のコードエディタを表示したい場合は，コンポーネントエディタの基本クラスとして，TComponentEditor の代わりに TDefaultEditor を使います。そして，Edit メソッドをオーバーライドする代わりに，TDefaultEditor::EditProperty プロテクトメソッドをオーバーライドします。EditProperty は，そのコンポーネントのイベントハンドラを検索して，最初に発見したものを表示します。この動作を変更して特定のイベントを検索することもできます。例を示します。

```
void __fastcall TMyEditor::EditProperty(TPropertyEditor* PropertyEditor,
                                        bool &Continue, bool &FreeEditor)
{
  if (PropertyEditor->ClassNameIs("TMethodProperty") &&
      CompareText(PropertyEditor->GetName, "OnSpecialEvent") == 0)
  {
    TDefaultEditor::EditProperty(PropertyEditor, Continue, FreeEditor);
  }
}
```

クリップボード形式を追加する

デフォルトでは，IDE でコンポーネントを選択して［コピー］を選択すると，そのコンポーネントは C++Builder の内部形式でコピーされます。その後で，別のフォームやデータモジュールに貼り付けることができます。Copy メソッドをオーバーライドすれば，独自のコンポーネントを別の形式でクリップボードにコピーできます。

たとえば，次に示す Copy メソッドは，TImage コンポーネントの画像をクリップボードにコピーします。この画像は C++Builder の IDE では無視されますが，別のアプリケーションに貼り付けることができます。

```
void __fastcall TMyComponentEditor::Copy(void)
{
  WORD AFormat;
  int AData;
  HPALETTE APalette;
  ((TImage *)Component)->Picture->SaveToClipboardFormat(AFormat, AData, APalette);
  TClipboard *pClip = Clipboard(); // クリップボードはクリアしない！
  pClip->SetAsHandle(AFormat, AData);
}
```

コンポーネントエディタを登録する

コンポーネントエディタを定義したら，それを登録して，特定のコンポーネントクラスに対して動作するようにします。登録されたコンポーネントエディタは，フォームデザイナでそのクラスが選択されたときに，コンポーネントごとに作成されます。

コンポーネントエディタとコンポーネントクラスを関連付けるには，RegisterComponentEditor を呼び出します。RegisterComponentEditor は，そのエディタを使用するコンポーネントクラスの名前と，コンポーネントエディタクラスの名前をパラメータに取ります。たとえば次の文は，TMyEditor という名前のコンポーネントエディタクラスを，TMyComponent 型のすべてのコンポーネントに対して動作するように登録します。

```
RegisterComponentEditor(__classid( TMyComponent), __classid(TMyEditor));
```

RegisterComponentEditor 呼び出しは，コンポーネントを登録する名前空間に置きます。たとえば，TMyComponent という名前の新しいコンポーネントと，コンポーネントエディタ TMyEditor がともに NewComp.cpp 内に実装されている場合は，次のようなコード（NewComp.cpp 内）を使って，そのコンポーネントとそれに関連するコンポーネントエディタを登録します。

```
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {
    TMetaClass classes[1] = {__classid(TMyComponent)};
    RegisterComponents("Miscellaneous", classes, 0);
    RegisterComponentEditor(classes[0], __classid(TMyEditor));
  }
}
```

コンポーネントのコンパイルとパッケージ化

コンポーネントを登録したら，IDE にインストールする前に，それらをパッケージとしてコンパイルしなければなりません。1 つのパッケージには，1 つ以上のコンポーネントと独自のプロパティエディタを含めることができます。パッケージについての詳細は，第 15 章「パッケージとコンポーネントの操作」を参照してください。

パッケージを作成してコンパイルするには，15-6 ページの「パッケージの作成と編集」を参照してください。独自のコンポーネントのソースコードユニットを，パッケージの Contains リストに置きます。このコンポーネントがほかのパッケージに依存する場合は，それらのパッケージを Requires リストに含めます。

IDE に独自のコンポーネントをインストールするには，15-5 ページの「コンポーネントパッケージのインストール」を参照してください。

カスタムコンポーネントのトラブルシューティング

カスタムコンポーネントを登録したり，インストールする場合によく発生する問題は，パッケージが正常にインストールされたにも関わらず，コンポーネントリストにコンポートが表示されないことです。

リストやパレットにコンポーネントが表示されない原因として一般的に考えられることは次のとおりです。

- Register 関数に PACKAGE 修飾子が記述されていない
- クラスに PACKAGE 修飾子が記述されていない
- C++ ソースファイルに `#pragma package(smart_init)` が記述されていない
- Register 関数が，ソースコードモジュール名と同じ名前で名前空間に記述されていない
- Register がうまくエクスポートされていないエクスポートされた関数を検索するには，.BPL に対して tdump を使います。

  ```
  tdump -ebpl mypack.bpl  mypack.dmp
  ```

  ダンプの exports 節に，エクスポートされた Register 関数が名前空間内に記述されているはずです。

第53章

既存のコンポーネントの変更

コンポーネントの最も簡単な作成方法は，必要な機能の大部分を含むコンポーネントを派生元にして，必要な変更を施すことです。この章では，標準のメモコンポーネントを変更して，黄色い背景を持つメモを作成する簡単なサンプルを示します。このヘルプの他のセクションではより複雑なコンポーネントの作成について説明します。基本的な手順は同じですが，作成するコンポーネントが複雑になるほど，新しいクラスをカスタマイズするためにより多くの手順が必要になります。

既存のコンポーネントを変更する手順は以下のとおりです。

- コンポーネントの作成と登録
- コンポーネントクラスの変更

コンポーネントの作成と登録

コンポーネントの作成は常に同じ方法で開始します。ユニットを作成し，コンポーネントクラスを派生させて登録し，それをコンポーネントパレットにインストールします。この処理についての概要は，45-8 ページの「新しいコンポーネントの作成」を参照してください。

このサンプルでは，一般的な手順でコンポーネントを作成します。コンポーネントに固有の点は以下のとおりです。

1. コンポーネントユニットの名前を YelMemo として保存します。この場合のヘッダーファイルは YELMEMO.H で .CPP ファイルは YELMEMO.CPP となります。
2. TMemo から TYellowMemo という新しいコンポーネントクラスを派生させる
3. TYellowMemo をコンポーネントパレットの［Samples］ページ（CLX では別のページ）に登録する

作成したヘッダーファイルは次のようになります。

```
#ifndef YelMemoH
#define YelmemoH
//----------------------------------------------------------------------------
#include <sysutils.hpp>
#include <controls.hpp>
```

```
#include <classes.hpp>
#include <forms.hpp>
#include <StdCtrls.hpp>
//---------------------------------------------------------------------------
class PACKAGE TYellowMemo : public TMemo
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

対応する .CPP ファイルは次のようになります。

```
#include <vcl.h>
#pragma hdrstop
#include "Yelmemo.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
// ValidCtrCheck を使用して作成済みコンポーネントに
// 純粋仮想関数がないことを確認
//
static inline void ValidCtrCheck(TYellowMemo *)
{
  new TYellowMemo(NULL);
}
//---------------------------------------------------------------------------
__fastcall TYellowMemo::TYellowMemo(TComponent* Owner)
        : TMemo(Owner)
{
}
//---------------------------------------------------------------------------
namespace Yelmemo
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TYellowMemo)};
    RegisterComponents("Samples", classes, 0); // CLX アプリケーションでは "Common Controls"
  }
}
```

CLX　CLX アプリケーションでは，ヘッダーファイルの名前と場所が異なります。たとえば，<vcl \ controls.hpp> は CLX では <clx \ qcontrols.hpp> です。

メモ　この例では，コンポーネントウィザードを使わずに，手動でコンポーネントを作成しています。コンポーネントウィザードを利用すると，コンストラクタが TYellowMemo に自動的に追加されます。

コンポーネントクラスの変更

新しく作成したコンポーネントクラスはさまざまな方法で変更できます。しかしこのサンプルでは，メモコンポーネントの 1 つのプロパティの初期値を変更するだけです。これには，以下のようにコンポーネントクラスに 2 つの小さな変更を加えます。

- コンストラクタをオーバーライドする
- プロパティのデフォルト値を宣言する

コンストラクタは実際にプロパティの値を設定します。デフォルト値を宣言することにより，どの値をフォームファイル（VCL は .dfm，CLX は .xfm）に保存すべきかを C++Builder に伝えます。C++Builder はデフォルトと異なる値だけを保存するので，両方の作業を実行することが重要です。

コンストラクタをオーバーライドする

設計時にコンポーネントをフォームに配置した場合，または実行時にアプリケーションでコンポーネントを作成した場合，コンポーネントのコンストラクタがプロパティ値を設定します。コンポーネントをフォームファイルからロードした場合，アプリケーションは設計時に変更されたプロパティを設定します。

メモ コンストラクタをオーバーライドした場合，新しいコンストラクタで独自の処理を行う前に，必ず継承コンストラクタを呼び出します。詳細については，46-10 ページの「メソッドのオーバーライド」を参照してください。

このサンプルのコンポーネントは，TMemo から継承したコンストラクタをオーバーライドし，Color プロパティを clYellow に設定します。これには，クラス宣言にコンストラクタのオーバーライド宣言を追加し，.CPP ファイルに新しいコンストラクタを記述します。この様子を次に示します。

```
class PACKAGE TYellowMemo : public TMemo
{
public:
    virtual __fastcall TYellowMemo(TComponent* Owner); // コンストラクタ宣言
__published:
    __property Color;
};
__fastcall TYellowMemo::TYellowMemo(TComponent* Owner)
  : TMemo(Owner)                                  // コンストラクタの実装ではまず
                                                  // TMemo のコンストラクタを呼び出す
{
  Color = clYellow;                               // コンポーネントの色を黄色に設定する
}
```

メモ コンポーネントウィザードを利用すると，既存のコンストラクタに `Color = clYellow;` を追加するだけで済みます。

これで，この新しいコンポーネントはコンポーネントパレットにインストールでき，さらにフォームに追加できます。Color プロパティのデフォルト値は clYellow に設定されています。

プロパティのデフォルト値を宣言する

C++Builder は，フォームのデータをフォームファイルに保存する際，デフォルト値と異なる値のプロパティしか保存しません。こうすることでフォームファイルの容量が小さくなり，フォームの読み込みも高速化されます。このため，プロパティを作成したりデフォルト値を変更した場合は，そのプロパティの新しいデフォルト値を宣言します。フォームファイル，プロパティの読み込み，デフォルト値についての詳細は，第 52 章「コンポーネントを設計時に利用できるようにする」を参照してください。

プロパティのデフォルト値を変更する手順は次のとおりです。

1. プロパティの名前を宣言します。
2. プロパティ名の後に等号（=）を置きます。
3. **default** キーワード，もう 1 つの等号，デフォルト値をすべて中カッコ（{}）で囲みます。

プロパティ全体を宣言する必要はありません。再宣言するのは，プロパティの名前とデフォルト値だけです。

この黄色いメモコンポーネントの例では，クラス宣言の **__published** 部で Color プロパティを再宣言し，デフォルト値として clYellow を記述します。

```
class PACKAGE TYellowMemo : public TMemo
{
public:
    virtual __fastcall TYellowMemo(TComponent* Owner);
__published:
    __property Color = {default=clYellow};
};
```

ここで再度 TYellowMemo のクラス宣言を示します。ただし，今度は WordWrap という別の published プロパティが追加されています。WordWrap のデフォルト値は **false** です。

```
class PACKAGE TYellowMemo : public TMemo
{
public:
    virtual __fastcall TYellowMemo(TComponent* Owner);
__published:
    __property Color = {default=clYellow};
    __property WordWrap = {default=false};
};
```

プロパティのデフォルト値を指定しただけではコンポーネントの動作はまったく変化しません。コンポーネントのコンストラクタでデフォルト値を明示的に設定する必要があります。これはアプリケーションの内部構造の違いによるものです。C++Builder は，TYellowMemo の WordWrap が **false** の場合はフォームファイルに WordWrap の値を書き込みません。コンストラクタ内でその値を自動的に設定することになっているからです。そのためのコンストラクタを次に示します。

```
__fastcall TYellowMemo::TYellowMemo(TComponent* AOwner) : TMemo(AOwner)
{
  Color = clYellow;
  WordWrap = false;
}
```

第54章

グラフィックコントロールの作成

グラフィックコントロールは単純なコンポーネントです。純粋なグラフィックコントロールはフォーカスを受け取らないので，ウィンドウハンドルは必要ありません。ユーザーはマウスでグラフィックコントロールを操作できますが，キーボードインターフェースは備えていません。

この章でとりあげるグラフィックコントロール TShape は，コンポーネントパレットの［Additional］ページに表示される図形コンポーネントです。作成するコンポーネントは標準の図形コンポーネントと同じものですが，識別子が重複しないように標準コンポーネントとは異なる名前を付けます。このセクションでは，図形コンポーネントを TSampleShape と命名し，そのコンポーネントの作成について説明します。

- コンポーネントの作成と登録
- 継承プロパティの公開
- グラフィック機能の追加

コンポーネントの作成と登録

コンポーネントの作成は常に同じ方法で開始します。コンポーネントクラスを派生させ，コンポーネントの .CPP および .H ファイルを保存し，コンポーネントクラスを派生させて登録し，コンパイルしてコンポーネントパレットにインストールします。この処理についての概要は，45-8 ページの「新しいコンポーネントの作成」を参照してください。

メモ この例では，コンポーネントウィザードを使わずに，手動でコンポーネントを作成しています。

このサンプルでは，一般的な手順でコンポーネントを作成します。コンポーネントに固有の点は以下のとおりです。

1. TGraphicControl から TSampleShape という新しいコンポーネント型を派生させる
2. コンポーネントのヘッダーファイル名を SHAPES.H，.CPP ファイル名を SHAPES.CPP と名前を付ける
3. TSampleShape をコンポーネントパレットの［Samples］ページ（CLX 内の別のページ）に登録する

作成したヘッダーファイルは次のようになります。

```
//---------------------------------------------------------------------------
#ifndef ShapesH
#define ShapesH
//---------------------------------------------------------------------------
#include <sysutils.hpp>
#include <controls.hpp>
#include <classes.hpp>
#include <forms.hpp>
//---------------------------------------------------------------------------
class PACKAGE TSampleShape : public TGraphicControl
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

.CPP ファイルは次のようになります。

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Shapes.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
// ValidCtrCheck を使用して作成済みコンポーネントに
// 純粋仮想関数がないことを確認
//

static inline void ValidCtrCheck(TSampleShape *)
{
  new TSampleShape(NULL);
}
//---------------------------------------------------------------------------
__fastcall TSampleShape::TGraphicControl(TComponent* Owner)
        : TGraphicControl(Owner)
{
}
//---------------------------------------------------------------------------
namespace Shapes
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TSampleShape)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

CLX　CLX アプリケーションでは，ヘッダーファイルの名前と場所が異なります。たとえば，<vcl \ controls.hpp> は CLX では <clx \ qcontrols.hpp> です。

継承プロパティの公開

コンポーネント型を派生させたら，次に，上位コンポーネントクラスの protected 部で宣言されているプロパティとイベントのうち，どれを新しいコンポーネントに公開するかを決定します。すでに TGraphicControl で，コンポーネントがコントロールとして機能するために必要なすべてのプロパティはパブリッシュに設定されているので，開発者が自分でパブリッシュに設定するのは，マウスイベントに応答してドラッグアンドドロップを処理する機能だけです。

継承プロパティと継承イベントの公開については，47-3 ページの「継承プロパティの公開」と 48-5 ページの「イベントの公開」で説明しています。どちらの場合も，クラス宣言の **published** 部でプロパティ名だけを再宣言します。

この図形コントロールの場合には，3 つのマウスイベント，3 つのドラッグアンドドロップイベント，2 つのドラッグアンドドロッププロパティをパブリッシュに設定します。

```
class PACKAGE TSampleShape : public TGraphicControl
{
private:
__published:
    __property DragCursor ;
    __property DragMode ;
    __property OnDragDrop ;
    __property OnDragOver ;
    __property OnEndDrag ;
    __property OnMouseDown ;
    __property OnMouseMove ;
    __property OnMouseUp ;
};
```

これで，マウスとドラッグアンドドロップに関する機能をユーザーに公開しました。

グラフィック機能の追加

グラフィックコンポーネントを宣言して，公開するすべての継承プロパティをパブリッシュに設定したら，次はグラフィック機能を追加します。このグラフィック機能が，開発するコンポーネントを特徴付けます。グラフィックコントロールの作成時には以下の 2 つの作業が必要です。

1. 描画対象を決定する
2. コンポーネントイメージの描画

また，この図形コントロールにはいくつかのプロパティを追加します。このプロパティによって，アプリケーション開発者が設計時に図形の外観をカスタマイズできるようになります。

描画対象を決定する

グラフィックコントロールは，ユーザー入力などの動的な条件によって，外観を変更します。常に同一の画像を表示するものであればコンポーネントを作る必要はありません。静的な画像を表示するにはイメージコンポーネントに読み込ませてください。

一般にグラフィックコントロールの外観は，プロパティ値の組み合わせによって決まります。たとえばゲージコントロールには，形状を表すプロパティ，方向を表すプロパティ，進捗状況をグラフィックだけでなく数値でも示すかどうかを決定するプロパティなどがあります。同様に，図形コントロールには描画する図形の種類を決定するプロパティがあります。

コントロールに描画する図形を決定するプロパティ Shape を追加するには，以下の作業が必要です。

1. プロパティ型の宣言
2. プロパティの宣言
3. メソッドの実装

プロパティの作成についての詳細は，第 47 章「プロパティの作成」を参照してください。

プロパティ型の宣言

ユーザー定義のプロパティ型を使用する場合は，そのプロパティを含むクラスを宣言する前に，その型の宣言を行っておきます。ユーザー定義のプロパティ型で最も多く使われるのは列挙型です。

図形コントロールの場合は，コントロールが描画できる図形の種類を示すために列挙型を定義します。

次の型宣言を，図形コントロールクラスの宣言よりも前に記述します。

```
enum TSampleShapeType { sstRectangle, sstSquare, sstRoundRect, sstRoundSquare,
                        sstEllipse, sstCircle };
class PACKAGE TSampleShape : public TGraphicControl         // 元からある文
```

これで，この型のプロパティをクラスで宣言できるようになりました。

プロパティの宣言

通常，プロパティを宣言する場合は，そのプロパティのデータを格納するプライベートなデータメンバーを宣言し，次にそのプロパティ値の読み出しと書き込みのためのメソッドを指定します。プロパティ値の読み出しでは，メソッドを使用せず，データが格納されているフィールドをそのまま指定することもあります。

この図形コントロールの場合，現在の図形を格納するデータメンバーを宣言し，次にプロパティを宣言します。このプロパティはメソッド呼び出しによってデータメンバーの読み出しや書き込みを行います。

次の宣言を TSampleShape に記述します。

```
class PACKAGE TSampleShape : public TGraphicControl
{
private:
    TSampleShapeType FShape;
    void __fastcall SetShape(TSampleShapeType Value);
__published:
    __property TSampleShapeType Shape = {read=FShape, write=SetShape, nodefault};
};
```

後は，SetShape を実装するだけです。

メソッドの実装

プロパティ定義の **read** または **write** 部で，プロパティデータの操作方法として，直接アクセスではなくメソッドを指定した場合には，そのメソッドを実装する必要があります。

次のように，SHAPES.CPP ファイルに SetShape メソッドの実装を追加します。

```
void __fastcall TSampleShape::SetShape(TSampleShapeType Value)
{
  if (FShape != Value)              // 変更されていない場合は無視
  {
     FShape = Value;                // 新しい値を格納
     Invalidate();                  // 新しい図形を強制的に再描画
  }
}
```

コンストラクタとデストラクタのオーバーライド

コンポーネントのプロパティのデフォルト値を変更し，所有クラスを初期化するには，継承コンストラクタと継承デストラクタをオーバーライドしなければなりません。このとき，新しいコンストラクタやデストラクタから必ず継承メソッドを呼び出します。

プロパティのデフォルト値の変更

グラフィックコントロールのデフォルトのサイズは小さいので，幅と高さをコンストラクタで変更します。プロパティのデフォルト値を変更する方法の詳細については，第 53 章「既存のコンポーネントの変更」を参照してください。

このサンプルでは，図形コントロールを 1 辺が 65 ピクセルの正方形に設定します。

1. コンポーネントクラスの宣言でコンストラクタをオーバーライドします。

   ```
   class PACKAGE TSampleShape : public TGraphicControl
   {
   public:
       virtual __fastcall TSampleShape(TComponent* Owner);
   };
   ```

   コンポーネントウィザードを使用してコンポーネントを作成した場合，この手順は自動的に実行されるので必要ありません。

2. Height プロパティと Width プロパティを新しいデフォルト値で再宣言します。

   ```
   class PACKAGE TSampleShape : public TGraphicControl
   {
     ...
   __published:
       __property Height;
       __property Width;
   }
   ```

3. .CPP ファイルに新しいコンストラクタを記述します。

   ```
   __fastcall TSampleShape::TSampleShape(TComponent* Owner) : TGraphicControl(Owner)
   {
     Height = 65;
     Width = 65;
   }
   ```

   コンポーネントウィザードを使用した場合は，すでに作成済みのコンストラクタに新しいデフォルト値を追加するだけで済みます。

ペンとブラシをパブリッシュに設定する

デフォルトでは，キャンバスは黒色の細いペンと白色のソリッドブラシを使用します。開発者がペンとブラシを変更できるようにするには，開発者が設計時に操作できるクラスを用意し，描画するときにそのクラスをキャンバスにコピーします。ペンやブラシのようなクラスは所有クラスと呼ばれます。コンポーネントによって所有され，コンポーネントによって作成，破棄されるからです。

所有クラスを管理する場合は，以下の作業が必要です。

1. クラスデータメンバーの宣言
2. アクセスプロパティの宣言
3. 所有クラスの初期化
4. 所有クラスのプロパティの設定

データメンバーの宣言

コンポーネントには，それぞれの所有クラスのためにデータメンバーが必要です。このデータメンバーにより，コンポーネントは，そのデータメンバーによって所有オブジェクトに対するポインタを保持し，コンポーネント自身を破棄する前に確実に所有クラスを破棄します。一般に，コンポーネントは所有オブジェクトをコンストラクタで初期化し，デストラクタで破棄します。

所有オブジェクト用のデータメンバーは，ほとんどの場合プライベートに宣言されます。アプリケーションやほかのコンポーネントがその所有オブジェクトにアクセスする場合は，それを可能にするために **published** プロパティまたは **public** プロパティを宣言します。

次のように，ペンとブラシのためのデータメンバーを図形コントロールに追加します。

```
class PACKAGE TSampleShape : public TGraphicControl
{
private:                // データメンバーは常にプライベート
    TPen *FPen;         // ペンオブジェクト用のデータメンバー
    TBrush *FBrush;     // ブラシオブジェクト用のデータメンバー
    ...
};
```

アクセスプロパティの宣言

コンポーネントの所有オブジェクトへのアクセスを用意するには，オブジェクト型のプロパティを宣言します。開発者は，設計時でも実行時でも，そのプロパティをとおして所有オブジェクトにアクセスします。一般に，プロパティの **read** 部ではクラスのデータメンバーを単純に参照し，**write** 部では所有オブジェクトの変更にコンポーネントが反応できるようにメソッドを呼び出します。

図形コントロールに，ペンデータメンバーとブラシデータメンバーへのアクセスを提供するプロパティを追加します。また，ペンやブラシの変更に反応するためのメソッドを宣言します。

```
class PACKAGE TSampleShape : public TGraphicControl
{
  ...
private:
    TPen *FPen;
    TBrush *FBrush;
    void __fastcall SetBrush(TBrush *Value);
    void __fastcall SetPen(TPen *Value);
    ...
```

```
__published:
    __property TBrush* Brush = {read=FBrush, write=SetBrush, nodefault};
    __property TPen* Pen = {read=FPen, write=SetPen, nodefault};
};
```

次に，.CPP ファイルに SetBrush メソッドと SetPen メソッドを記述します。

```
void __fastcall TSampleShape::SetBrush( TBrush* Value)
{
  FBrush->Assign(Value);
}
void __fastcall TSampleShape::SetPen( TPen* Value)
{
  FBrush->Assign(Value);
}
```

Value の内容を直接 FBrush に代入するには，次のように Value の内容を FBrush に直接代入しないでください。

```
    FBrush = Value;
```

こうすると，FBrush の内部ポインタが上書きされてしまい，メモリが失われます。また，メモリ管理の問題が多発する可能性があります。

所有クラスの初期化

コンポーネントにクラスを追加した場合には，そのコンポーネントのコンストラクタにクラスを初期化させ，ユーザーが実行時にそれらのオブジェクトと対話できるようにします。同様に，コンポーネントのデストラクタにはコンポーネント自身を破棄する前に，所有オブジェクトを破棄させるようにします。

図形コントロールにペンとブラシを追加したので，図形コントロールのコンストラクタでそれを初期化し，また図形コントロールのデストラクタでそれを破棄します。この様子を次に示します。

1. 図形コントロールのコンストラクタでペンとブラシを作成します。

```
__fastcall TSampleShape::TSampleShape(TComponent* Owner) : TGraphicControl(Owner)
{
  Height = 65;
  Width = 65;
  FBrush = new TBrush();                                    // ペンを構築
  FPen = new TPen();                                        // ブラシを構築
}
```

2. コンポーネントオブジェクトの宣言でデストラクタをオーバーライドします。

```
class PACKAGE TSampleShape : public TGraphicControl
{
  ...
public:                                                     // デストラクタは常にパブリック
    virtual __fastcall TSampleShape(TComponent* Owner);
    __fastcall ~TSampleShape();                             // デストラクタ
    ...
};
```

3. .CPP ファイルに新しいデストラクタを記述します。

```
__fastcall TSampleShape::~TSampleShape()
{
  delete FPen;                                              // ペンオブジェクトを削除
```

```
  delete FBrush;                                    // ブラシオブジェクトを削除
}
```

所有クラスのプロパティの設定

最後に，ペンとブラシが変更された際に図形コントロールが自分自身を再描画するようにします。ペンクラスとブラシクラスには OnChange イベントがあるので，図形コントロールでメソッドを作成し，両方の OnChange イベントがそのメソッドを指すようにします。

図形コントロールにメソッドを記述します。また，コンポーネントのコンストラクタで，ペンとブラシの OnChange イベントがその新しいメソッドを指すように設定します。この様子を次に示します。

```
class PACKAGE TSampleShape : public TGraphicControl
{
  ...
public:
    void __fastcall StyleChanged(TObject* Owner);
    ...
};
```

また，コンポーネントのコンストラクタで，ペンとブラシの OnChange イベントがその新しいメソッドを指すように設定します。

```
__fastcall TSampleShape::TSampleShape(TComponent* Owner) : TGraphicControl(Owner)
{
  Height = 65;
  Width = 65;
  FBrush = new TBrush();
  FBrush->OnChange = StyleChanged;
  FPen = new TPen();
  FPen->OnChange = StyleChanged;
}
```

StyleChanged メソッドの実装を記述します。

```
void __fastcall TSampleShape::StyleChanged( TObject* Sender)
{
  Invalidate();          // コンポーネントを再描画
}
```

これで，ペンまたはブラシを変更するとコンポーネントが再描画されるようになりました。

コンポーネントイメージの描画

グラフィックコントロールの本質は，画面にイメージを描画する方法で決まります。コントロールにイメージを描画するには，TGraphicControl 抽象型で定義されている Paint というメソッドをオーバーライドします。

図形コントロールの Paint メソッドは，以下の条件を満たすように作成します。

- 選択されたペンとブラシを使用する
- 選択された図形を使用する
- 正方形と円の場合は幅と高さを同じにする

Paint メソッドをオーバーライドする手順は次のとおりです。

1. コンポーネントの宣言に Paint を追加します。
2. .CPP ファイルに Paint メソッドを記述します。

この図形コントロールの場合，クラスの宣言で次のようにメソッドを宣言します。

```
class PACKAGE TSampleShape : public TGraphicControl
{
  ...
protected:
    virtual void __fastcall Paint();
    ...
};
```

そして，.CPP ファイルにこのメソッドを記述します。

```
void __fastcall TSampleShape::Paint()
{
  int X,Y,W,H,S;
  Canvas->Pen = FPen;                                  // コンポーネントのペンをコピー
  Canvas->Brush = FBrush;                              // コンポーネントのブラシをコピー
  W=Width;                                             // コンポーネントの幅を使用
  H=Height;                                            // コンポーネントの高さを使用
  X=Y=0;                                               // 円 / 正方形の最小値を保管
  if( W<H )
    S=W;
  else
    S=H;
  switch(FShape)
  {
    case sstRectangle:                                  // 長方形と正方形を描画
    case sstSquare:
      Canvas->Rectangle(X,Y,X+W,Y+H);
      break;
    case sstRoundRect:                                  // 角が丸い長方形と正方形を描画
    case sstRoundSquare:
      Canvas->RoundRect(X,Y,X+W,Y+H,S/4,S/4);
      break;
    case sstCircle:                                     // 円と楕円を描画
    case sstEllipse:
      Canvas->Ellipse(X,Y,X+W,Y+H);
      break;
    default:
      break;
  }
}
```

コントロールがイメージを更新する必要があるときは常に Paint が呼び出されます。コントロールが最初に表示される場合や，コントロールの前面に表示されているウィンドウが移動した場合は，そのコントロールが描画されます。Invalidate を呼び出して再描画を強制することもできます。これは StyleChanged メソッドでも行っている方法です。

図形の描画方法の改良

標準の図形コントロールは，長方形や楕円だけでなく正方形や円を描画できますが，このサンプルにはまだその機能がありません。正方形や円は長方形や楕円と同様に扱われます。正方形や円を描画する場合は，図形のもっとも短い辺を見つけ，図形を中央に位置付けるコードを記述する必要があります。

正方形と円を正しく描画できるように改良した Paint を次に示します。

```
void __fastcall TSampleShape::Paint(void)
{
  int X,Y,W,H,S;
  Canvas->Pen = FPen;                          // コンポーネントのペンをコピー
  Canvas->Brush = FBrush;                      // コンポーネントのブラシをコピー
  W=Width;                                     // コンポーネントの幅を使用
  H=Height;                                    // コンポーネントの高さを使用
  X=Y=0;                                       // 円 / 正方形の最小値を保管
  if( W<H )
    S=W;
  else
    S=H;
  switch(FShape)                                // 高さ，幅，位置を調整
  {
    case sstRectangle:
    case sstRoundRect:
    case sstEllipse:
      Y=X=0;                                   // 原点を図形の最上部左端に設定
      break;
    case sstSquare:
    case sstRoundSquare:
    case sstCircle:
      X= (W-S)/2;                              // 水平方向の中心合わせ
      Y= (H-S)/2;                              // 垂直方向の中心合わせ
      break;
    default:
    break;
  }
  switch(FShape)
  {
    case sstSquare:                            // 長方形と正方形を描画
      W=H=S;                                   // 幅と高さを最小にする
    case sstRectangle:
      Canvas->Rectangle(X,Y,X+W,Y+H);
      break;
    case sstRoundSquare:                       // 角が丸い長方形と正方形を描画
      W=H=S;
    case sstRoundRect:
      Canvas->RoundRect(X,Y,X+W,Y+H,S/4,S/4);
      break;
    case sstCircle:                            // 円と楕円を描画
      W=H=S;
    case sstEllipse:
      Canvas->Ellipse(X,Y,X+W,Y+H);
      break;
    default:
      break;
  }
}
```

第55章

グリッドのカスタマイズ

C++Builder には，コンポーネントをカスタマイズするときに土台として使用できる抽象コンポーネントがあります。このうちよく使用されるのはグリッドとリストボックスです。この章では，グリッドコンポーネント TCustomGrid を元にして，1 か月分の表示ができる小さなカレンダーを作成します。

このカレンダーを作成する場合は，以下の作業が必要です。

- コンポーネントの作成と登録
- 継承プロパティの公開
- 初期値の変更
- セルのサイズ変更
- セルの内容の表示
- 月と年の移動
- 日付（日）の変更

VCL アプリケーションでは，作成するカレンダーコンポーネントは，コンポーネントパレットの［Samples］ページにある TCalendar に似ています。CLX アプリケーションでは，コンポーネントを別のページに保存するか，新しいパレットページを作成します。52-3 ページの「パレットページを指定する」またはオンラインヘルプで「コンポーネントパレット，ページの追加」を参照してください。

コンポーネントの作成と登録

コンポーネントは常に同じ方法で作成します。コンポーネントクラスを派生させ，コンポーネントの .CPP および .H ファイルを保存し，コンポーネントクラスを派生させて登録し，コンパイルしてコンポーネントパレットにインストールします。この処理についての概要は，45-8 ページの「新しいコンポーネントの作成」を参照してください。

このサンプルでは，一般的な手順でコンポーネントを作成します。コンポーネントに固有の点は以下のとおりです。

1. TCustomGrid から TSampleCalendar という新しいコンポーネント型を派生させる

2. コンポーネントのヘッダーファイルを CALSAMP.H と対応する .CPP ファイルを CALSAMP.CPP と名前を付ける
3. TSampleCalendar をコンポーネントパレットの［Samples］ページ（CLX の場合は別のページ）に登録する。CLX アプリケーションの場合は，別のコンポーネントパレットページを使用します。

作成したヘッダーファイルは次のようになります。

```
#ifndef CalSampH
#define CalSampH
//---------------------------------------------------------------------------
#include <Sysutils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
#include <Grids.hpp>
//---------------------------------------------------------------------------
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

CALSAMP.CPP ファイルは，次のようになります。

```
#include <vcl.h>
#pragma hdrstop
#include "CalSamp.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
namespace Calsamp
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TSampleCalendar)};
    RegisterComponents("Samples", classes, 0);  // CLX では Samples と異なるページを使用
  }
}
```

メモ メモ コンポーネントウィザードを使用してコンポーネントを作成した場合は，ヘッダーファイルに新しいコンストラクタが宣言されます。そして，CALSAMP.CPP ファイルにコンストラクタの最初の部分が含まれます。CALSAMP.CPP ファイルにコンストラクタがないときは，後で追加できます。

CLX　CLX アプリケーションでは，ヘッダーファイルの名前と場所が異なります。たとえば，<vcl \ controls.hpp> は CLX では <clx \ qcontrols.hpp> です。

継承プロパティの公開

抽象グリッドコンポーネント TCustomGrid には，多くの **protected** プロパティがあります。その中から，ユーザーに公開するカレンダーコントロールのプロパティを選択します。

コンポーネントのユーザーに継承したプロテクトプロパティを公開するには，コンポーネント宣言の **__published** 部でそのプロパティを再宣言します。

このカレンダーコントロールでは，次に示すようにいくつかのプロパティとイベントをパブリッシュに設定します。

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
...
__published:
    __property Align ;                    // プロパティをパブリッシュに設定
    __property BorderStyle ;
    __property Color;
    __property Font ;
    __property GridLineWidth ;
    __property ParentColor ;
    __property ParentFont ;
    __property OnClick ;                   // イベントをパブリッシュに設定
    __property OnDblClick ;
    __property OnDragDrop ;
    __property OnDragOver ;
    __property OnEndDrag ;
    __property OnKeyDown ;
    __property OnKeyPress ;
    __property OnKeyUp ;
};
```

ほかにも，描画されるグリッド線を選択する Options プロパティなど，カレンダーには適用されないけれども公開できる多くのプロパティがあります。

完成したカレンダーコンポーネントをコンポーネントパレットにインストールして，アプリケーションから使用すると，上記よりも多くのプロパティとイベントがあります。これから，それらの新しい機能を追加していきます。

初期値の変更

カレンダーは基本的に，行と列の数が固定されているグリッドです（もっとも，常にすべての行に日付が表示されるとは限りません）。カレンダーは常に 1 週間を 7 日で表示するからです。そのため，グリッドのプロパティ ColCount と RowCount は，パブリッシュには設定してありません。ただし，1 週間を 7 日にするためには，プロパティの初期値をそのように設定しなければなりません。

コンポーネントプロパティの初期値を変更するには，コンストラクタをオーバーライドして値を設定します。

コンポーネントのクラス宣言の **public** 部にコンストラクタの宣言を追加し，ヘッダーファイルに新しいコンストラクタを記述する必要があります。

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
protected:
    virtual void __fastcall DrawCell(int ACol, int ARow, const Windows::TRect &Rect,
      TGridDrawState AState);
    ...
```

```
public:
    __fastcall TSampleCalendar(TComponent *Owner);   // 追加したコンストラクタ
    ...
};
```

CALSAMP.CPP ファイルに，次のようなコンストラクタコードを記述します。

```
__fastcall TSampleCalendar::TSampleCalendar(TComponent *Owner) : TCustomGrid(Owner)
{
  ColCount = 7;
  RowCount = 7;
  FixedCols = 0;
  FixedRows = 1;
  ScrollBars = ssNone;
  Options = (Options >> goRangeSelect) << goDrawFocusSelected;
}
void __fastcall TSampleCalendar::DrawCell(int ACol, int ARow, const Windows::TRect
  &ARect, TGridDrawState AState)
{
}
```

メモ DrawCell メソッドもクラス宣言に追加され，.CPP ファイルには，DrawCell メソッドの最初の部分が記述されたことに注意してください。今のところはこれは必ずしも必要ではありませんが，DrawCell をオーバーライドせずに TSampleCalendar をテストしようとすると純粋仮想関数エラーが発生します。この現象は，TCustomGrid が抽象クラスであることから生じるものです。DrawCell のオーバーライドについては，「セルの内容の表示」を参照してください。

これで，7 行 7 列で最初の行が固定行すなわち非スクロールのカレンダーになりました。

セルのサイズ変更

VCL ユーザーやアプリケーションがウィンドウやコントロールのサイズを変更した場合には，Windows はサイズ変更されたウィンドウやコントロールに対して WM_SIZE というメッセージを送り，ウィンドウやコントロールが新しいサイズでイメージを描画するために必要な情報を通知します。VCL コンポーネント側では，このメッセージに応答して，すべてのセルがコントロールの境界内に収まるようにセルのサイズを変更します。WM_SIZE メッセージに応答するには，コンポーネントにメッセージ処理メソッドを追加します。

メッセージ処理メソッドの作成についての詳細は，51-6 ページの「新しいメッセージハンドラの作成」を参照してください。

このサンプルのカレンダーコントロールで WM_SIZE に応答するため，WMSize というプロテクトメソッドをコントロールに追加し WM_SIZE に結び付けます。このメソッドでは，すべてのセルが表示できるようにセルサイズを計算します。次にそのコードを示します。

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
...
protected:
    void __fastcall WMSize(TWMSize &Message);
BEGIN_MESSAGE_MAP
  VCL_MESSAGE_HANDLER(WM_SIZE, TWMSize, WMSize)
```

```
END_MESSAGE_MAP(TCustomGrid)
};
```

CALSAMP.CPP ファイルのメソッドのコードを次に示します。

```
void __fastcall TSampleCalendar::WMSize(TWMSize &Message)
{
  int GridLines;                                        // 一時ローカル変数
  GridLines = 6 * GridLineWidth;                        // すべての線を結合した長さの計算結果
  DefaultColWidth = (Message.Width - GridLines) / 7;    // 新しいデフォルトのセル幅を設定
  DefaultRowHeight = (Message.Height - GridLines) / 7;  // 同様にセルの高さも設定
}
```

これでカレンダーのサイズが変更されたので，すべてのセルはコントロールに合った最大のサイズで表示されます。

CLX　CLX では，ウィンドウやコントロールのサイズを変更した場合，プロテクトメソッド BoundsChanged の呼び出しによって自動的に通知されます。CLX コンポーネント側では，この通知に応答して，すべてのセルがコントロールの境界内に収まるようにセルのサイズを変更します。

このサンプルでは，変更後のサイズですべてのセルが表示される適切なセルサイズを計算するように，カレンダーコントロールが BoundsChanged をオーバーライドする必要があります。

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
...
protected:
    void __fastcall BoundsChanged(void);
};
```

CALSAMP.CPP ファイルのメソッドのコードを次に示します。

```
void __fastcall TSampleCalendar::BoundsChanged(void)
{
  int GridLines;                                    // 一時ローカル変数
  GridLines = 6 * GridLineWidth;                    // すべての線を結合した長さの計算結果
  DefaultColWidth = (Width - GridLines) / 7;        // 新しいデフォルトのセル幅を設定
  DefaultRowHeight = (Height - GridLines) / 7;      // 同様にセルの高さも設定
  TCustomGrid::BoundsChanged();                     // ここで継承メソッドを呼び出す
}
```

セルの内容の表示

グリッドコントロールでは，各セルにそれぞれの内容があります。このカレンダーでは，各セルごとに表示する日付を計算します。ただし，セルによっては表示する日付がないこともあります。デフォルトでは，グリッドのセルの描画は DrawCell という仮想メソッドで行われます。

グリッドセルに内容を表示するには，DrawCell メソッドをオーバーライドします。

固定行にある見出しのセルは，簡単に表示できます。ランタイムライブラリに曜日の略称の配列があるので，カレンダーにはその配列を利用して各列の見出しセルに内容を表示します。この様子を次に示します。

DrawCell メソッドのコードを次に示します。

```
void __fastcall TSampleCalendar::DrawCell(int ACol, int ARow, const Windows::TRect &ARect,
  TGridDrawState AState);
{
  String TheText;
  int TempDay;
  if (ARow == 0) TheText = ShortDayNames[ACol + 1];
  else
  {
    TheText = "";
    TempDay = DayNum(ACol, ARow);                    // DayNum は後で定義する
    if (TempDay != -1) TheText = IntToStr(TempDay);
  }
  Canvas->TextRect(ARect, ARect.Left + (ARect.Right - ARect.Left
    - Canvas->TextWidth(TheText)) / 2,
    ARect.Top + (ARect.Bottom - ARect.Top - Canvas->TextHeight(TheText)) / 2, TheText);
}
```

日付の取得

このカレンダーコントロールを実用的なものにするには，ユーザーやアプリケーションが年，月，日を設定できるようにしなければなりません。C++Builder は日付と時間を TDateTime 型の変数に格納します。TDateTime は日付と時間をコード化した数値表現なので，プログラムで操作するには便利ですが，人間の使用には適していません。

そこで，TDateTime に日付をコード化して格納し，その値への実行時のアクセス方法を用意しますが，同時に Year，Month，Day の各プロパティも提供します。カレンダーコンポーネントのユーザーは，このプロパティを設計時に使用できます。

カレンダー内の日付をトラッキングする手順を以下に示します。

- 日付を内部的に格納する
- 年，月，日にアクセスする
- 日付の表示
- 現在の日付の強調表示

日付を内部的に格納する

カレンダーの日付を格納するために，日付を格納するプライベートなデータメンバーと，その日付をアクセスする実行時用のプロパティを用意します。

1. 日付を格納するデータメンバーをプライベートに宣言します。

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    TDateTime FDate;
    ...
};
```

2. コンストラクタの日付データメンバーを初期化します。

```
__fastcall TSampleCalendar::TSampleCalendar(TComponent *Owner) : TCustomGrid(Owner)
{
  ...
  FDate = FDate.CurrentDate();
}
```

3. コード化された日付にアクセスする実行時用のプロパティを宣言します。

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
public:
    __property TDateTime CalendarDate = {read=FDate, write=SetCalendarDate, nodefault};
    ...
};
```

日付を設定したときは，コントロールに表示されるイメージも更新しなければならないので，日付の設定はメソッドで行います。TSampleCalendar に SetCalendarDate を宣言します。

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    void __fastcall SetCalendarDate(TDateTime Value);
    ...
};
```

次のコードは，SetCalendarDate メソッドです。

```
void __fastcall TSampleCalendar::SetCalendarDate(TDateTime Value)
{
  FDate = Value;                        // 新しい日付の値を設定
  Refresh();                            // イメージの表示を更新
}
```

年，月，日にアクセスする

コード化された数値はプログラムにとっては扱いやすいものですが，人間が操作しやすいのは年，月，日の個別の値です。そこで，コード化された日付の各要素へアクセスするためのプロパティを作成します。

日付の各要素（年，月，日）はすべて整数であり，またどの要素も設定や取得にはコード化された日付との変換が必要です。そこで，3 つのプロパティに共通の実装メソッドを指定して，コードの重複を回避します。つまり，要素を読み出すメソッドと要素を書き込むメソッドの 2 つのメソッドを記述するだけで，3 つのプロパティの取得と設定を実現します。

設計時に年，月，日へアクセスできるようにする手順は以下のとおりです。

1. 3 つのプロパティを宣言し，それぞれにユニークなインデックス番号を割り当てます。

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
  ...
public:
    __property int Day = {read=GetDateElement, write=SetDateElement, index=3, nodefault};
    __property int Month = {read=GetDateElement, write=SetDateElement, index=2, nodefault};
    __property int Year = {read=GetDateElement, write=SetDateElement, index=1, nodefault};
};
```

2. 実装メソッドを宣言して記述し，各インデックス値に対応する要素を設定します。

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    int __fastcall GetDateElement(int Index);           // Index パラメータに注意
    void __fastcall SetDateElement(int Index, int Value);
    ...
};
```

GetDateElement および SetDateElement メソッドは次のようになります。

```
int __fastcall TSampleCalendar::GetDateElement(int Index)
{
  unsigned short AYear, AMonth, ADay;
  int result;
  FDate.DecodeDate(&AYear, &AMonth, &ADay);        // エンコードされた日付を要素に分解
  switch (Index)
  {
    case 1: result = AYear;  break;
    case 2: result = AMonth; break;
    case 3: result = ADay;   break;
    default: result = -1;
  }
  return result;
}
void __fastcall TSampleCalendar::SetDateElement(int Index, int Value);
{
  unsigned short AYear, AMonth, ADay;
  if (Value > 0)                                    // 要素はすべて正の値でなければならない
  {
    FDate.DecodeDate(&AYear, &AMonth, &ADay);      // 現在の日付要素を取得
    switch (Index)
    {
      case 1: AYear = Value;     break;
      case 2: AMonth = Value;    break;
      case 3: ADay = Value;      break;
      default: return;
    }
  }
  FDate = TDateTime(AYear, AMonth, ADay);          // 変更した日付をエンコード
  Refresh();                                       // カレンダーの表示を更新
}
```

これで，カレンダーの年，月，日を設定できるようになりました。年，月，日の設定は，設計時にオブジェクトインスペクタを使用して行うか，実行時にコードで行います。セルに日付を描画するコードはまだ記述していませんが，必要なデータは揃いました。

日付の表示

カレンダーに日付を表示するには，いくつか考慮しなければならない点があります。1 か月の日数は，その月が何月かによって，またその年がうるう年かどうかによって変化します。また，年によって月が始まる曜日はまちまちです。IsLeapYear 関数を使って，その年がうるう年かどうか判断します。SysUtils ヘッダーファイルの MonthDays 配列を使って，その月の日数を取得します。

うるう年と月の日数についての情報を取得したら，次に各日付を表示するグリッド上の場所を計算します。この結果は，その月の最初の曜日によって決まります。

各日付を表示するセルは，年と月によって異なります。そこで，年と月の値を常に反映させたオフセット値を保持し，必要なときにその値を参照するようにします。実際には，オフセット値をクラスデータメンバーに格納しておいて，日付が変更されるたびにデータメンバーを更新します。

日付を適切なセルに表示する手順は次のとおりです。

1. 月のオフセットデータメンバーと，このデータメンバーの値を更新するメソッドをクラスに追加します。

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    int FMonthOffset;                          // オフセット用の記憶域
    ...
protected:
    virtual void __fastcall UpdateCalendar(void);
    ...
};

void __fastcall TSampleCalendar::UpdateCalendar(void)
{
  unsigned short AYear, AMonth, ADay;
  TDateTime FirstDate;                          // 月の最初の日付
  if ((int)FDate != 0)                          // データが有効な場合のみオフセットを計算
  {
    FDate.DecodeDate(&AYear, &AMonth, &ADay);  // 日付要素の取得
    FirstDate = TDateTime(AYear, AMonth, 1);   // 最初の日付
    FMonthOffset = 2 - FirstDate.DayOfWeek();  // グリッド上のオフセット値を計算
  }
  Refresh();                                    // 常にコントロールを再描画
}
```

2. 日付を変更したときに UpdateCalendar を呼び出すために，コンストラクタ，SetCalendarDate メソッド，および SetDateElement メソッドにコードを追加します。

```
__fastcall TSampleCalendar::TSampleCalendar(TComponent *Owner)
  : TCustomGrid(Owner)
{
  ...
  UpdateCalendar();
}
void __fastcall TSampleCalendar::SetCalendarDate(TDateTime Value)
{
  FDate = Value;                                // 元からある文
  UpdateCalendar();                             // 以前はここで Refresh を呼び出していた
}

void __fastcall TSampleCalendar::SetDateElement(int Index, int Value);
{
  ...
  FDate = TDateTime(AYear, AMonth, ADay);       // 元からある文
  UpdateCalendar();                             // 以前はここで Refresh を呼び出していた
}
```

3. 行と列のセル座標を渡すと日付（日の数値）を返すメソッドをカレンダーに追加します。

```
int __fastcall TSampleCalendar::DayNum(int ACol, int ARow)
{
  int result = FMonthOffset + ACol + (ARow - 1) * 7;       // このセルの日付を計算
  if ((result < 1)||(result > MonthDays[IsLeapYear(Year)][Month]))
    result = -1;   // 無効なら -1 を返す
  return result;
}
```

また，コンポーネントの型宣言にこの DayNum の宣言を追加します。

4. これで，日付を表示する場所を計算できるようになりました。そこで DrawCell を改良して，日付を表示するようにします。

```
void __fastcall TSampleCalendar::DrawCell(int ACol, int ARow, const TRect &ARect,
  TGridDrawState AState);
{
  String TheText;
  int TempDay;
  if (ARow == 0)                                    // これは見出し行
    TheText = ShortDayNames[ACol + 1];                  // 日付の名前だけを使用
  else
  {
    TheText = "";                                   // デフォルトは空白のセル
    TempDay = DayNum(ACol, ARow);                   // このセルに表示する数値を取得
    if (TempDay != -1) TheText = IntToStr(TempDay); // 有効な場合はその数値を使用
  }
  Canvas->TextRect(ARect, ARect.Left + (ARect.Right - ARect.Left -
    Canvas->TextWidth(TheText)) / 2,
    ARect.Top + (ARect.Bottom - ARect.Top - Canvas->TextHeight(TheText)) / 2, TheText);
}
```

ここで，このカレンダーコンポーネントを再びインストールしてフォームに配置すると，現在の月が正しく表示されます。

現在の日付の強調表示

カレンダーの各セルに日付が表示されるようになりました。そこで，現在の日付のセルが常に選択され，強調表示されるようにします。デフォルトでは，左上端のセルが選択されていますが，選択するセルは Row プロパティと Column プロパティで指定できます。常に適切なセルが選択されているようにするには，カレンダーを作成したときと，日付が変更されたときの両方の場合に 2 つのプロパティを設定します。

現在の日付のセルを選択するには，UpdateCalendar メソッドが Refresh を呼び出す前に Row と Column を設定します。

```
void __fastcall TSampleCalendar::UpdateCalendar(void)
{
  unsigned short AYear, AMonth, ADay;
  TDateTime FirstDate;
  if ((int) FDate != 0)
  {
  ...                                           // FMonthOffset を設定する既存の文
    Row = (ADay - FMonthOffset) / 7 + 1;
    Col = (ADay - FMonthOffset) % 7;
  }
  Refresh();                                    // 元からある文
}
```

この手続きでは，コード化された日付から抽出しておいた ADay 変数を再利用しています。

月と年の移動

プロパティによるコンポーネントの操作は便利なものです。プロパティは設計時にも操作できます。しかし，一般的で自然な操作については，複数のプロパティを扱うことが多いのでメソッドを用意した方がよい場合もあります。そのような例の 1 つが，カレンダーの「翌月」機能です。月と年を変更するのは単純な処理ですが，コンポーネントを使用する開発者にとっては便利な機能です。

一般的な操作をメソッドにカプセル化する方法の唯一の欠点は，メソッドは実行時にしか処理できない点です。しかし，一般にそのような操作を繰り返すのは繁雑なことであり，通常，設計時に実行することはありません。

カレンダーに次の 4 つのメソッドを追加して，直後と直前の月または年へ移動できるようにします。これらのメソッドでは，IncMonth 関数をちょっと違った方法で使用します。月または年をインクリメントして，CalendarDate をインクリメントまたはデクリメントします。

```
void __fastcall TSampleCalendar::NextMonth()
{
  Word  AYear, AMonth, ADay;
  DecodeDate(IncMonth(CalendarDate, 1), AYear, AMonth, ADay);
  FDate = TDate(AYear, AMonth, ADay);
}
void __fastcall TSampleCalendar::PrevMonth()
{
  Word  AYear, AMonth, ADay;
  DecodeDate(IncMonth(CalendarDate, -1), AYear, AMonth, ADay);
  FDate = TDate(AYear, AMonth, ADay);
}
void __fastcall TSampleCalendar::NextYear()
{
  Word  AYear, AMonth, ADay;
  DecodeDate(IncMonth(CalendarDate, 12), AYear, AMonth, ADay);
  FDate = TDate(AYear, AMonth, ADay);
}
void __fastcall TSampleCalendar::PrevYear()
{
  Word  AYear, AMonth, ADay;
  DecodeDate(IncMonth(CalendarDate, -12), AYear, AMonth, ADay);
  FDate = TDate(AYear, AMonth, ADay);
}
```

この 4 つのメソッドの宣言を必ずクラス宣言の **public** 部に追加してください。

これで，このカレンダーコンポーネントを使用するアプリケーションから，月や年を簡単に変更できます。

日付（日）の変更

特定の月において日付を変更する方法は 2 つあります。矢印キーを使用する方法とマウスのクリックに応答する方法です。標準グリッドコンポーネントでは，この 2 つの操作は両方ともクリックとして処理されます。つまり，矢印キーを押す操作は，隣のセルをクリックする操作として扱われます。

日付の変更手順は次のとおりです。

- 選択セルの変更
- OnChange イベントの提供
- 空白セルの除外

選択セルの変更

矢印キーまたはマウスクリックに応答して，選択されたセルを変更する機能は，グリッドから継承されています。しかし，選択された日付を変更するには，このデフォルトの機能を変更します。

カレンダー内の移動を処理するには，グリッドの Click メソッドをオーバーライドします。

Click のようなユーザーとやり取りするメソッドをオーバーライドする場合は，通常，標準の機能が失われないように継承メソッドの呼び出しを行います。

カレンダーグリッド用にオーバーライドされた Click メソッドの例を次に示します。TSampleCalendar に Click メソッドの宣言を必ず追加してください。

```
void __fastcall TSampleCalendar::Click()
{
  int TempDay = DayNum(Col, Row);            // クリックされたセルの日付を取得
  if (TempDay != -1) Day = TempDay;          // 有効な場合は日付を変更
}
```

OnChange イベントの提供

カレンダーのユーザーが日付を変更できるようになったので，次はアプリケーションがその変更に応答する手段を用意します。

TSampleCalendar に OnChange イベントを追加します。

1. イベント，イベントを格納するデータメンバー，イベントを呼び出す仮想メソッドを宣言します。

```
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    TNotifyEvent FOnChange;
    ...
protected:
    virtual void __fastcall Change();
__published:
    __property TNotifyEvent OnChange = {read=FOnChange, write=FOnChange};
    ...
}
```

2. Change メソッドを記述します。

```
void __fastcall TSampleCalendar::Change()
{
  if(FOnChange != NULL) FOnChange(this);
}
```

3. SetCalendarDate メソッドと SetDateElement メソッドの最後に，Change を呼び出す文を追加します。

```
void __fastcall TSampleCalendar::SetCalendarDate(TDateTime Value)
{
  FDate = Value;
  UpdateCalendar();
  Change();              // 唯一の新しい文
}
void __fastcall TSampleCalendar::SetDateElement(int Index, int Value);
{
  ...   // 必要に応じてほかのプロパティも設定
  FDate = TDateTime(AYear, AMonth, ADay);
  UpdateCalendar();
  Change();              // 新しい文
}
```

カレンダーコンポーネントを使用するアプリケーションは，OnChange イベントにハンドラを結び付ければ，コンポーネントの日付の変更に応答できます。

空白セルの除外

カレンダーの書き込みでは，空白のセルも選択できますが，その場合は日付は変更されません。そこで，空白のセルは選択できないようにします。

セルの選択を制御するには，グリッドの SelectCell メソッドをオーバーライドします。

SelectCell 関数は，列と行をパラメータとして受け取り，指定のセルが選択可能かどうかを論理値で返します。

SelectCell をオーバーライドして，セルに有効な日付が入っていない場合は **false** を返すようにします。その例を次に示します。

```
bool __fastcall TSampleCalendar::SelectCell(int ACol, int ARow)
{
  if (DayNum(ACol,ARow) == -1) return false;       //  -1 は，無効な日付を示す
  else return TCustomGrid::SelectCell(ACol, ARow); // そうでない場合は，継承した値を使用する
}
```

ユーザーが空白のセルをクリックしたり，矢印キーによって空白のセルに移動しようとした場合は，カレンダーコンポーネントは現状を維持し，選択されているセルを変更しません。

第56章

データベース対応コントロールの作成

データベースとの接続を行う場合，データベース対応のコントロールが用意されていると便利です。データベース対応のコントロールは，データベースの特定の部分との間にリンクを確立します。C++Builder には，データベース対応のラベル，編集ボックス，リストボックス，コンボボックス，参照コントロール，グリッドが用意されています。また，独自にデータベース対応のコントロールを作成することもできます。データベース対応コントロールの使用についての詳細は，第 19 章「データコントロールの使い方」を参照してください。

データベースへの対応にはいくつかのレベルがあります。最も単純なレベルは読み出し専用のデータベース対応（データ参照コントロール）です。このレベルでは，データ参照によってデータベースの現在の状態を反映できます。少し複雑なレベルとして，編集可能なデータベース対応（ データ編集コントロール）があります。このレベルでは，ユーザーはコントロールを操作してデータ編集を行うことができます。これとは別に，単一の項目にリンクするだけの場合もあれば，複数のレコードを制御する場合もあります。

この章では，最も単純な例として，データセット内の単一の項目にリンクした読み出し専用コントロールを作成します。ここで使用するコントロールは，第 55 章「グリッドのカスタマイズ」で作成したカレンダー TSampleCalendar です。コンポーネントパレットの［Samples］ページにある標準カレンダー TCalendar も使用することができます（VCL のみ）。

そのあと，新しいデータ参照コントロールとデータ編集コントロールの作成のしかたを説明します。

データ参照コントロールの作成

データベース対応のカレンダーコントロールの作成には，読み出し専用コントロールの場合も，ユーザーがデータセット内の基礎になるデータを変更できるコントロールの場合も，以下の作業が必要です。

- コンポーネントの作成と登録
- データリンクの追加
- データの変更の反映

コンポーネントの作成と登録

コンポーネントは常に同じ方法で作成します。コンポーネントクラスを派生させ，コンポーネントの .CPP および .H ファイルを保存し，コンポーネントクラスを派生させて登録し，コンパイルしてコンポーネントパレットにインストールします。この処理についての概要は，45-8 ページの「新しいコンポーネントの作成」を参照してください。

このサンプルでは，一般的な手順でコンポーネントを作成します。コンポーネントに固有の点は以下のとおりです。

- TSampleCalendar コンポーネントから TDBCalendar という新しいコンポーネントクラスを派生させる。TSampleCalendar コンポーネントの作成方法については，第 55 章「グリッドのカスタマイズ」を参照
- ヘッダーファイルの名前を DBCAL.H とし，.CPP ファイルの名前を DBCAL.CPP とする
- TDBCalendar をコンポーネントパレットの［Samples］ページ（CLX アプリケーション内の別のページ）に登録する

CLX　CLX アプリケーションでは，ヘッダーファイルの名前と場所が異なります。たとえば，<vcl \ controls.hpp> は CLX では <clx \ qcontrols.hpp> です。

作成したヘッダーファイルは次のようになります。

```
#ifndef DBCalH
#define DBCalH
//---------------------------------------------------------------------------
#include <Sysutils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
#include <Grids.hpp>                // Grids ヘッダーをインクルード
#include "calsamp.h"                // TSampleCalendar を宣言するヘッダーをインクルード
//---------------------------------------------------------------------------
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

.CPP ファイルは次のようになります。

```
#pragma link "Calsamp"                          // TSampleCalendar にリンク
#include <vcl.h>
#pragma hdrstop
#include "DBCal.h"
```

```
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
static inline void ValidCtrCheck(TDBCalendar *)
{
   new TDBCalendar(NULL);
}
//---------------------------------------------------------------------------
namespace Dbcal
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TDBCalendar)};
    RegisterComponents("Samples", classes, 0); // CLX アプリケーション内の別のページを使用
  }
}
```

メモ コンポーネントウィザードを使用して TDBCalendar コンポーネントを作成した場合，コンストラクタがすでにヘッダーファイルに宣言されています。また，コンストラクタの定義が .CPP ファイルに含まれています。

これから，この新しいカレンダーをデータ参照コントロールに変更していきます。

コントロールを読み出し専用にする

このカレンダーはデータに対して読み出し専用なので，コントロール自体を読み出し専用にします。データベースは変更されないにもかかわらず，コントロールのデータだけが変更できるようになっているとユーザーが混乱するからです。

カレンダーを読み出し専用にする場合は，以下の作業が必要です。

- ReadOnly プロパティの追加
- 必要な更新を許可

VCL TSampleCalendar ではなく，C++Builder の［Samples］ページにある TCalendar を基にしてコントロールを作成する場合は，すでに ReadOnly プロパティがあるので，このステップは飛ばしてください。

ReadOnly プロパティの追加

ReadOnly プロパティを追加すれば，設計時にコントロールを読み出し専用に設定できます。プロパティを **true** に設定すると，コントロールのすべてのセルが選択不可になります。

1. DBCAL.H ファイルにプロパティ宣言と，プロパティの値を格納するための **private** データメンバーを追加します。

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    bool FReadOnly;                                    // 内部記憶用のフィールド
protected:
public:
    virtual __fastcall TDBCalendar(TComponent* Owner);
__published:
    __property ReadOnly = {read=FReadOnly, write=FReadOnly, default=true};
};
```

2. DBCAL.CPP にコンストラクタを記述します。

```
virtual __fastcall TDBCalendar::TDBCalendar(TComponent* Owner) :
  TSampleCalendar(Owner)
{
  FReadOnly = true;                                    // デフォルト値を設定
}
```

3. SelectCell メソッドをオーバーライドして，コントロールが読み出し専用になっている場合には選択を禁止します。SelectCell については，55-13 ページの「空白セルの除外」を参照してください。

```
bool __fastcall TDBCalendar::SelectCell(int ACol, int ARow)
{
  if (FReadOnly) return false;                    // 読み出し専用の場合には選択不可
  return TSampleCalendar::SelectCell(ACol, ARow); // そうでない場合は，継承メソッドを使用する
}
```

TDBCalendar のクラス宣言に SelectCell の宣言を必ず追加してください。

これで，コントロールはクリックとキー入力を無視するようになりました。日付を変更した場合も選択セルが変更されません。

必要な更新を許可

この読み出し専用カレンダーでは，Row プロパティと Col プロパティの設定など，あらゆる種類の変更は SelectCell メソッドを通して行われます。UpdateCalendar メソッドは日付が変更されるたびに Row プロパティと Col プロパティを設定しようとしますが，SelectCell メソッドが変更を禁止します。そのため，日付が変更されても選択されているセルは変わりません。

そこで，カレンダーに内部的な論理フラグを追加し，このフラグが **true** に設定されている場合には変更が許可されるようにします。この様子を次に示します。

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    ...
    bool FUpdating;                                    // 内部で使用するプライベートなフラグ
protected:
    virtual bool __fastcall SelectCell(long ACol, long ARow);
public:
    ...
    virtual void __fastcall UpdateCalendar();
    ...
};
bool __fastcall TDBCalendar::SelectCell(int ACol, int ARow)
{
    if (!FUpdating && FReadOnly) return false;      // 読み出し専用の場合には選択不可
    return TSampleCalendar::SelectCell(ACol, ARow); // そうでない場合は継承メソッドを使用する
}

void __fastcall TDBCalendar::UpdateCalendar()
{
  FUpdating=true;                                    // 更新を許可するようにフラグを設定
  try
  {
     TSampleCalendar::UpdateCalendar();              // 通常どおり更新する
  }
```

```
    catch(...)
    {
      FUpdating = false;
      throw;
    }
    FUpdating = false;                                    // 常にフラグをクリアする
  }
```

依然としてユーザーの変更は禁止されていますが，日付プロパティの変更が適切に反映されるようになりました。これで，読み出し専用カレンダーが完成しました。次のセクションではデータ参照機能を追加します。

データリンクの追加

コントロールとデータベースとの接続は，データリンクと呼ばれるクラスによって扱われます。コントロールをデータベースの単一のデータメンバーと接続するデータリンククラスは TFieldDataLink です。このほか，テーブル全体を扱うデータリンククラスもあります。

データベース対応のコントロールはデータリンククラスを所有します。つまり，コントロールは所有するデータリンクの構築と破棄を行います。所有クラスの管理についての詳細は，第 54 章「グラフィックコントロールの作成」を参照してください。

所有クラスとしてデータリンクを確立するには，次の 3 つの手順を行います。

1. クラスデータメンバーの宣言
2. アクセスプロパティの宣言
3. データリンクの初期化

データメンバーの宣言

54-6 ページの「データメンバーの宣言」で説明したように，コンポーネントの所有クラスにはデータメンバーが必要です。このサンプルでは，データリンク用の TFieldDataLink 型のデータメンバーを使用します。

次のように，データリンク用のフィールドをカレンダーで宣言します。

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    TFieldDataLink *FDataLink;
    ...
};
```

アプリケーションをコンパイルする前に，DBCAL.H ファイルに DB.HPP ファイルと DBTABLES.HPP ファイルをインクルードする必要があります。

```
#include <DB.hpp>
#include <DBTables.hpp>
```

アクセスプロパティの宣言

すべてのデータベース対応コントロールには DataSource プロパティがあります。このプロパティは，アプリケーションのどのデータソースクラスがコントロールにデータを供給するかを指定します。ま

た，単一の項目にアクセスするコントロールには，データソースの項目を指定する DataField プロパティも必要です。

第 54 章「グラフィックコントロールの作成」の例で示した，所有されるクラスへのアクセスプロパティと異なり，これらのアクセスプロパティは，所有されるクラス自身へのアクセスではなく，そのクラスの対応するプロパティへのアクセスを提供します。これによって，コントロールとそのデータリンクは，同じデータソースとフィールド項目を共有できるプロパティを作成できます。

DataSource プロパティ，DataField プロパティ，およびそれらの実装メソッドを宣言します。メソッドは，データリンククラスのプロパティをアクセスします。

アクセスプロパティ宣言の例

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
   ...
    AnsiString __fastcall GetDataField();                // メソッドはプライベート
    TDataSource *__fastcall GetDataSource();             // データ項目名を返す
    void __fastcall SetDataField(AnsiString Value);      // データソースへの参照を返す
    void __fastcall SetDataSource(TDataSource *Value);   // データ項目に名前を付ける
    ...
__published:                                    // 設計時にプロパティを使用可能にする
    __property AnsiString DataField = {read=GetDataField, write=SetDataField, nodefault};
    __property TDataSource * DataSource = {read=GetDataSource, write=SetDataSource,
      nodefault};
   ...
};
AnsiString __fastcall TDBCalendar::GetDataField()
{
  return FDataLink->FieldName;
}
TDataSource *__fastcall TDBCalendar::GetDataSource()
{
  return FDataLink->DataSource;
}
void __fastcall TDBCalendar::SetDataField(AnsiString Value)
{
  FDataLink->FieldName = Value;
}
void __fastcall TDBCalendar::SetDataSource(TDataSource *Value)
{
  if(Value != NULL)
    Value->FreeNotification(this);
  FDataLink->DataSource = Value;
}
```

これで，カレンダーとそのデータリンクとの間のリンクが確立されました。次に，カレンダーコントロールが作成される際にデータリンククラスを作成し，カレンダーコントロールが破棄される前にデータリンクを破棄するようにします。

データリンクの初期化

データベース対応のコントロールは，その存在の全期間をとおして，データリンクへのアクセスを必要とします。そこで，コントロールのコンストラクタの一部としてデータリンクオブジェクトを作成し，コントロール自身が破棄される前にデータリンクオブジェクトを破棄します。

カレンダーのコンストラクタとデストラクタをオーバーライドします。

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
public:
    virtual __fastcall TDBCalendar(TComponent* Owner);
    __fastcall ~TDBCalendar();
};
__fastcall TDBCalendar::TDBCalendar(TComponent* Owner) : TSampleCalendar(Owner)
{
    FReadOnly = true;
    FDataLink = new TFieldDataLink();
    FDataLink->Control = this;
}
__fastcall TDBCalendar::~TDBCalendar()
{
    FDataLink->Control = NULL;
    FDataLink->OnUpdateData = NULL;
    delete FDataLink;
}
```

これでデータリンクが完成しました。しかし，リンクした項目からデータを読み出す方法が指示されていません。次のセクションではその手順について説明します。

データの変更の反映

現在，コントロールはデータリンクを持ち，データソースおよびデータ項目を指定するプロパティを備えています。次に必要なのは，項目データの変更に対応することです。変更が発生するのは，レコードを移動した場合か項目が変更された場合のいずれかです。

すべてのデータリンククラスには OnDataChange というイベントがあります。データソースからデータの変更を通知されると，データリンクオブジェクトは OnDataChange イベントに結び付けられているイベントハンドラを呼び出します。

データの変更に対応してコントロールを更新するには，データリンクの OnDataChange イベントにハンドラを結び付けます。

このサンプルでは，カレンダーにメソッドを追加して，それをデータリンクの OnDataChange イベントとして指定します。

次に示すように，DataChange メソッドを宣言して実行し，それをコンストラクタでデータリンクの OnDataChange イベントに指定します。デストラクタでは，オブジェクトを破棄する前に OnDataChange イベントハンドラをデタッチします。

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    void __fastcall DataChange(TObject *Sender);
```

```
    ...
};
void __fastcall TDBCalendar::DataChange( TObject* Sender)
{
  if (FDataLink->Field == NULL)                          // 項目が指定されていない場合には ...
    CalendarDate = 0;                                    // ... 無効な日付を設定する
  else CalendarDate = FDataLink->Field->AsDateTime;      // それ以外は新しい日付を設定する
}
__fastcall TDBCalendar::TDBCalendar(TComponent* Owner) :   TSampleCalendar(AOwner)
{
  FReadOnly = true;
  FDataLink = new TFieldDataLink();              // データリンクオブジェクトの構築
  FDataLink->Control = this;
  FDataLink->OnDataChange = DataChange;          // ハンドラをアタッチ
}
__fastcall TDBCalendar::~TDBCalendar()
{
  FDataLink->Control = NULL;
  FDataLink->OnUpdateData = NULL;
  FDataLink->OnDataChange = NULL;                  // ハンドラをまずデタッチしてから ...
  delete FDataLink;                                // ... データリンクオブジェクトを破棄する
}
```

これでデータ参照コントロールができました。

データ編集コントロールの作成

データ編集コントロールを作成する場合は，コンポーネントを作成および登録してから，データ参照コントロールの場合と同様にデータリンクを追加します。基礎となる項目のデータ変更にも同様の方法で応答しますが，まだいくつかの問題を処理しなければなりません。

たとえば，コントロールにキーイベントとマウスイベントの両方に応答させたいとしたら，コントロールは，ユーザーがコントロールの内容を変更した場合に応答しなければなりません。ユーザーがコントロールを終了した場合は，コントロールに加えられた変更をデータセットに反映する必要があります。

ここで説明するデータ編集コントロールは，この章の最初の部分で説明したのと同じカレンダーコントロールです。このコントロールは，リンクした項目のデータの表示も編集もできるように変更されます。

既存のコントロールを変更してデータ編集コントロールにする手順は次のとおりです。

- FReadOnly のデフォルト値の変更
- マウスダウンメッセージとキーダウンメッセージの処理
- 項目のデータリンククラスの更新
- Change メソッドの変更
- データセットの更新

FReadOnly のデフォルト値の変更

これはデータ編集コントロールなので，ReadOnly プロパティはデフォルトで **false** に設定されます。ReadOnly プロパティを **false** にするには，コンストラクタで FReadOnly の値を変更します。

```
__fastcall TDBCalendar::TDBCalendar (TComponent* Owner) : TSampleCalendar(Owner)
{
  FReadOnly = false;              // デフォルト値の設定
  ...
}
```

マウスダウンメッセージとキーダウンメッセージの処理

コントロールのユーザーがコントロールと対話を始めると，コントロールは Windows からマウスダウンメッセージ（WM_LBUTTONDOWN，WM_MBUTTONDOWN，または WM_RBUTTONDOWN）またはキーダウンメッセージ（WM_KEYDOWN）を受け取ります。コントロールがこれらのメッセージに応答できるようにするには，これらのメッセージに応答するハンドラを記述しなければなりません。

- マウスダウンメッセージへの応答
- キーダウンメッセージへの応答

CLX　CLX では，オペレーティングシステムからの通知はシステムイベントの形で行われます。システムイベントとウィジェットイベントに応答するコンポーネントの記述については，51-10 ページの「CLX によるシステム通知への応答」を参照してください。

マウスダウンメッセージへの応答

MouseDown メソッドは，コントロールの OnMouseDown イベント用のプロテクトメソッドです。コントロール自体が Windows のマウスダウンメッセージに応答して MouseDown メソッドを呼び出します。継承メソッド MouseDown をオーバーライドする場合は，OnMouseDown イベントの呼び出しに加えて，ほかの応答をするコードを含めることができます。

MouseDown をオーバーライドし，MouseDown メソッドを TDBCalendar クラスに追加する手順は次のとおりです。

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
...
protected:
    virtual void __fastcall MouseDown(TMouseButton Button,  TShiftState Shift, int X,
      int Y);
    ...
};
```

.CPP ファイルに MouseDown メソッドを次のように記述します。

```
void __fastcall TDBCalendar::MouseDown(TMouseButton Button,  TShiftState Shift, int  X,
  int Y)
{
  TMouseEvent MyMouseDown;                                  // イベント型の宣言
  if (!FReadOnly && FDataLink->Edit())                      // 項目が編集可能なら
    TSampleCalendar::MouseDown(Button, Shift, X, Y);   // 継承した MouseDown を呼び出す
```

```
  else
  {
    MyMouseDown = OnMouseDown;                          // OnMouseDown イベントを割り当てる
    if (MyMouseDown != NULL) MyMouseDown(this, Button, // OnMouseDown イベントハンドラで ...
       Shift, X, Y);                                    // ... コードを実行する
  }
}
```

MouseDown メソッドがマウスダウンメッセージに応答するときに，継承メソッド MouseDown が呼び出されるのは，コントロールの ReadOnly プロパティが **false** でデータリンクオブジェクトが編集モードの場合だけ，つまり項目を編集できる場合だけです。項目を編集できない場合，プログラマが OnMouseDown イベントハンドラに記述したコードがあれば実行されます。

キーダウンメッセージへの応答

KeyDown メソッドは，コントロールの OnKeyDown イベント用のプロテクトメソッドです。コントロール自体が Windows のキーダウンメッセージに応答して KeyDown メソッドを呼び出します。継承メソッド KeyDown をオーバーライドする場合は，OnKeyDown イベントの呼び出しに加えて，ほかの応答をするコードも含めることができます。

KeyDown メソッドをオーバーライドする手順は次のとおりです。

1. KeyDown メソッドを TDBCalendar クラスに追加します。

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
   ...
protected:
    virtual void __fastcall KeyDown(unsigned short &Key,  TShiftState Shift);
   ...
};
```

2. .CPP ファイルに KeyDown メソッドを記述します。

```
void __fastcall TDBCalendar::KeyDown(unsigned short &Key, TShiftState Shift)
{
  TKeyEvent MyKeyDown;                                  // イベント型の宣言
  Set<unsigned short,0,8> keySet;
  keySet = keySet << VK_UP << VK_DOWN << VK_LEFT        // セットに仮想キーを割り当てる
    << VK_RIGHT  << VK_END << VK_HOME << VK_PRIOR << VK_NEXT;
  if (!FReadOnly &&                                    // コントロールが読み出し専用でなく...
      (keySet.Contains(Key)) &&                        // ... そしてキーがセット中にあり
      FDataLink->Edit() )                              // ... かつ項目が編集モードである場合
  {
    TCustomGrid::KeyDown(Key, Shift);           // 継承された KeyDown メソッドを呼び出す
  }
  else
  {
    MyKeyDown = OnKeyDown;                              // OnKeyDown イベントを割り当てる
    if (MyKeyDown != NULL) MyKeyDown(this,Key,Shift); // OnKeyDown イベントハンドラで ...
  }                                                     //  ... コードを実行する
}
```

KeyDown メソッドがマウスダウンメッセージに応答するときに，継承メソッド KeyDown が呼び出されるのは，コントロールの ReadOnly プロパティが **false** であり，押されたキーがカーソルコントロールキーの 1 つであり，しかもデータリンクオブジェクトが編集モードの場合だけ，つまり項目を

編集できる場合だけです。項目を編集できなかったり，ほかのキーが押された場合，プログラマがOnKeyDown イベントハンドラに記述したコードがあれば実行されます。

項目のデータリンククラスの更新

データ変更には以下の 2 種類があります。

- データベース対応コントロールに反映しなければならない項目値の変更
- 項目値に反映しなければならないデータベース対応コントロールの変更

TDBCalendar コンポーネントは，データセット内の項目値が変更されたときにその変更値をCalendarDate プロパティに代入する DataChange メソッドをすでに持っています。DataChange メソッドは，OnDataChange イベント用のハンドラです。したがって，カレンダーコンポーネントは最初の種類のデータ変更を処理できます。

同様に項目のデータリンククラスは，コントロールのユーザーがデータベース対応コントロールの内容を変更したときに発生する OnUpdateData イベントも持っています。カレンダーコントロールは，OnUpdateData イベントのイベントハンドラになる UpdateData メソッドを持っています。UpdateData メソッドはデータベース対応コントロールの変更値を項目のデータリンクに代入します。

1. カレンダーの値に加えられた変更を項目値に反映するには，UpdateData メソッドをカレンダーコンポーネントの **private** セクションに追加します。

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    void __fastcall UpdateData(TObject *Sender);
};
```

2. .CPP ファイルに UpdateData メソッドを記述します。

```
void __fastcall TDBCalendar::UpdateData( TObject* Sender)
{
    FDataLink->Field->AsDateTime = CalendarDate;  // 項目をカレンダー日付へのリンクに設定する
}
```

3. TDBCalendar のコンストラクタ内で，OnUpdateData イベントに UpdateData メソッドを代入します。

```
__fastcall TDBCalendar::TDBCalendar(TComponent* Owner)
  : TSampleCalendar(Owner)
{
  FDataLink = new TFieldDataLink();       // 元からある文
  FDataLink->OnUpdateData = UpdateData;  // OnUpdateData イベントに UpdateData を割り当てる
}
```

Change メソッドの変更

TDBCalendar の Change メソッドは，新しい日付値が設定されるたびに呼び出されます。Change メソッドは，OnChange イベントハンドラがあれば呼び出します。コンポーネントユーザーは，日付の変更に応答するコードを OnChange イベントハンドラに記述できます。

カレンダー日付が変わると，変更が起きたことをデータセットに通知しなければなりません。この通知をするには，Change メソッドをオーバーライドし，コードをもう 1 行追加します。実行する手順は次のとおりです。

1. 新しい Change メソッドを TDBCalendar コンポーネントに追加します。

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
protected:
    virtual void __fastcall Change();
    ...
};
```

2. データセットにデータの変更を通知する Modified メソッドを呼び出し，次に継承メソッド Change を呼び出すよう Change メソッドを記述します。

```
void __fastcall TDBCalendar::Change()
{
  if (FDataLink != NULL)
    FDataLink->Modified();                // Modified メソッドを呼び出す
  TSampleCalendar::Change();              // 継承した Change メソッドを呼び出す
}
```

データセットの更新

これまでは，データベース対応コントロール内の変更で項目のデータリンククラス内の値が変わりました。データ編集コントロール作成の最終段階は，新しい値でデータセットを更新することです。データベース対応コントロールの値を変更した人がコントロールの外をクリックするか（Tab）を押してコントロールを終了した後に，この変更が行われなければなりません。この処理は VCL と CLX で異なります。

VCL

VCL にはコントロール操作用のメッセージコントロール ID が定義されています。たとえばユーザーがコントロールを終了すると，そのコントロールに CM_EXIT メッセージが送られます。そのメッセージに応答するメッセージハンドラを記述できます。この場合，ユーザーがコントロールを終了すると，CM_EXIT 用のメッセージハンドラである CMExit メソッドはそれに応答して，データセットのレコードを項目のデータリンククラスの変更値で更新します。メッセージハンドラについての詳細は，第 51 章「メッセージとシステム通知の処理」を参照してください。

メッセージハンドラ内のデータセットを更新する手順は次のとおりです。

1. メッセージハンドラを TDBCalendar コンポーネントに追加します。

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    void __fastcall CMExit(TWMNoParams Message);
BEGIN_MESSAGE_MAP
  VCL_MESSAGE_HANDLER(CM_EXIT, TWMNoParams, CMExit)
END_MESSAGE_MAP
};
```

2. .CPP ファイルのコードを次のように記述します。

```
void __fastcall TDBCalendar::CMExit(TWMNoParams &Message)
{
```

```
  try
  {
    FDataLink.UpdateRecord();          // データベースを更新するようにデータリンクに通知する
  }
  catch(...)
  {
    SetFocus();                        // 失敗した場合は，フォーカスが移動しないようにする
    throw;
  }
}
```

CLX　CLX の場合，TWidgetControl に，入力フォーカスがコントロールから離れたときに呼び出されるプロテクトメソッド DoExit があります。このメソッドは，OnExit イベント用のイベントハンドラを呼び出します。このメソッドをオーバーライドすると，OnExit イベントハンドラを生成する前にデータセット内のレコードを更新できます。

ユーザーがコントロールを終了するときにデータセットを更新する手順は次のとおりです。

1. DoExit メソッドのオーバーライドを TDBCalendar コンポーネントに追加します。

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    DYNAMIC void __fastcall DoExit(void);
   ...
};
```

2. .CPP ファイルのコードを次のように記述します。

```
void __fastcall TDBCalendar::DoExit(void)
{
  try
  {
    FDataLink.UpdateRecord();          // データベースを更新するようにデータリンクに通知する
  }
  catch(...)
  {
    SetFocus();                        // 失敗した場合は，フォーカスが移動しないようにする
    throw;
  }
  TCustomGrid::DoExit(); // 継承メソッドに OnExit イベントを生成させる
}
```

第57章

ダイアログボックスのコンポーネント化

よく使うダイアログボックスは，コンポーネントにしてコンポーネントパレットに追加すると便利です。標準 コモンダイアログボックスを表すコンポーネントと同じように機能するダイアログボックスコンポーネントを作成できます。ダイアログボックスをコンポーネントにする目的は，プロジェクトに追加して設計時にプロパティを設定できるような，簡単なコンポーネントを作成することです。

ダイアログボックスをコンポーネントにする場合は，以下の作業が必要です。

1. コンポーネントのインターフェースの定義
2. コンポーネントの作成と登録
3. コンポーネントのインターフェースの作成
4. コンポーネントのテスト

ダイアログボックスに関連付けられた C++Builder の「ラッパー」コンポーネントは，実行時にダイアログボックスを作成し，ユーザーが指定したデータを渡します。ダイアログボックスコンポーネントは再利用でき，カスタマイズも可能です。

この章では C++Builder オブジェクトリポジトリに登録されている［バージョン情報］ダイアログを表示するラッパーコンポーネントと同じものを作成します。

メモ ABOUT.H，ABOUT.CPP，および ABOUT.DFM ファイルを作業用ディレクトリにコピーします。プロジェクトに ABOUT.CPP を追加して，ダイアログのラッパーコンポーネントのビルド時に ABOUT.OBJ ファイルが作成されるようにします。

コンポーネント化するダイアログボックスを設計する場合，制限はそれほどありません。また，ほとんどのフォームはダイアログボックスとして動作できます。

コンポーネントのインターフェースの定義

ダイアログボックス用のコンポーネントを作成する場合，まずアプリケーション開発者から見てどのようなコンポーネントを作成するのかという点を考察します。つまり，ダイアログボックスとそれを使用するアプリケーションとの間のインターフェースを考案します。

たとえば，コモンダイアログボックスコンポーネントにはプロパティがあります。アプリケーション開発者はそのプロパティをとおしてキャプションやコントロールを初期設定し，またダイアログボックスが閉じられた後は必要な情報を読み出します。この場合，情報を直接やり取りするのはラッパーコンポーネントのプロパティであって，ダイアログボックスのコントロールではありません。

インターフェースは情報を十分に受け渡す必要があります。つまり，アプリケーションが希望する形でダイアログボックスフォームを表示し，また，アプリケーションが必要とする情報を返す必要があります。コンポーネントのプロパティは，ダイアログボックスに表示されるすべてのデータを扱うものと考えてください。

［バージョン情報］ダイアログボックスの場合は情報を返す必要はありません。そのため，ラッパーのプロパティで受け渡す情報は，［バージョン情報］ダイアログボックスの表示を指定するための情報だけです。［バージョン情報］ダイアログボックスには，アプリケーションで指定できるフィールドが 4 つあるので，それに対応する 4 つの文字列型のプロパティを用意します。

コンポーネントの作成と登録

コンポーネントの作成は常に同じ方法で開始します。コンポーネントクラスを派生させ，コンポーネントの .CPP および .H ファイルを保存し，コンポーネントクラスを派生させて登録し，コンパイルしてコンポーネントパレットにインストールします。この処理についての概要は，45-8 ページの「新しいコンポーネントの作成」を参照してください。

このサンプルでは，一般的な手順でコンポーネントを作成します。コンポーネントに固有の点は以下のとおりです。

- TComponent から TAboutBoxDlg という新しいコンポーネント型を派生させる
- コンポーネントユニットのファイル名を ABOUTDLG.H と ABOUTDLG.CPP にする
- TAboutBoxDlg をコンポーネントパレットの［Samples］ページに登録する

次のような .H ファイルが作成されます。

```
#ifndef AboutDlgH
#define AboutDlgH
//---------------------------------------------------------------------------
#include <Sysutils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
class PACKAGE TAboutBoxDlg : public TComponent
{
private:
```

```
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

ユニットの .CPP ファイルは次のとおりです。

```
#include <vcl.h>
#pragma hdrstop
#include "AboutDlg.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
static inline void ValidCtrCheck(TAboutBoxDlg *)
{
  return new TAboutBoxDlg(NULL);
}
//---------------------------------------------------------------------------
namespace AboutDlg {
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TAboutBoxDlg)};
    RegisterComponents("Samples", classes, 0);
  }
}
```

メモ コンポーネントウィザードを使用してコンポーネントを作成した場合，TAboutDlg にはコンストラクタが追加されています。

CLX CLX アプリケーションでは，ヘッダーファイルの名前と場所が異なります。たとえば，<vcl \ controls.hpp> は CLX では <clx \ qcontrols.hpp> です。

この新しいコンポーネントの機能は TComponent に組み込まれている機能だけです。つまり，これは最も単純な非ビジュアルコンポーネントです。次のセクションでは，このコンポーネントとダイアログボックスとの間のインターフェースを作成します。

コンポーネントのインターフェースの作成

コンポーネントのインターフェースを作成する手順は以下のとおりです。

1. フォームユニットファイルのインクルード
2. インターフェースプロパティの追加
3. Execute メソッドの追加

フォームユニットファイルのインクルード

ラッパーコンポーネントにダイアログボックスを初期化して表示させるには，そのフォームのファイルをプロジェクトに記述します。

コンポーネントのヘッダーファイルに ABOUT.H および ABOUT.OBJ のリンクをインクルードします。

```
#include "About.h"
#pragma link "About.obj"
```

フォームヘッダーファイルでは，常にそのフォームクラスのインスタンスを宣言します。［バージョン情報］ダイアログボックスのフォームクラスは TAboutBox なので，ABOUT.H ファイルでは次の宣言が行われます。

```
extern TAboutBox *AboutBox;
```

インターフェースプロパティの追加

アプリケーション開発者がダイアログボックスをコンポーネントとして使用するにはいくつかのプロパティが必要です。そこで，必要なプロパティの宣言をコンポーネントのクラス宣言に記述します。

ラッパーコンポーネントのプロパティは，通常のコンポーネントのプロパティよりも単純です。ここで作成するのは，ラッパーコンポーネントがダイアログボックスとの間で受け渡すための永続的なデータです。このデータをプロパティのフォームで保持しておくと，アプリケーション開発者が設計時にデータを指定し，ラッパーコンポーネントが実行時にそのデータをダイアログボックスに渡せます。

インターフェースプロパティの宣言は，コンポーネントのクラス宣言に 2 つの部分を追加する必要があります。

- プライベートデータメンバーの宣言。ラッパーコンポーネントがプロパティの値を格納するための変数
- パブリッシュに設定したプロパティ自体の宣言。プロパティの名前を宣言し，値の格納場所として使用するデータメンバーを指定する

インターフェースプロパティには，アクセスメソッドは不要です。このプロパティは格納されているデータに直接アクセスします。プロパティ値を格納するデータメンバーの名前は通常，プロパティ名の先頭に F を加えた名前にします。データメンバーとプロパティは必ず同じ型にします。

たとえば，Year という整数型のインターフェースプロパティを宣言するコードを次に示します。

```
class PACKAGE TWrapper : public TComponent
{
private:
    int FYear;                          //  Year プロパティのデータを格納するデータメンバー
protected:
public:
__published:
    __property int Year = {read=FYear, write=FYear};        // 格納領域と一致するプロパティ
};
```

［バージョン情報］ダイアログボックスの場合には 4 つの文字列型のプロパティが必要です。各プロパティはそれぞれ，製品名，バージョン情報，著作権情報，コメントを表します。ABOUTDLG.H ファイルは次のようになります。

```
class PACKAGE TAboutBoxDlg : public TComponent
{
private:
    int FYear;
    String FProductName, FVersion, FCopyright, FComments;
```

```
protected:
public:
__published:
    __property int Year = {read=FYear, write=FYear};
    __property String ProductName = {read=FProductName, write=FProductName};
    __property String Version = {read=FVersion, write=FVersion};
    __property String Copyright = {read=FCopyright, write=FCopyright};
    __property String Comments = {read=FComments, write=FComments};
};
```

ここで，このコンポーネントをコンポーネントパレットにインストールしてフォームに配置すればプロパティの設定ができます。プロパティの値はフォームに保持され，ラッパーコンポーネントはダイアログボックスを実行するときにこの値を使用できます。

Execute メソッドの追加

最後に，コンポーネントのインターフェースには，ダイアログボックスを開く方法とダイアログボックスを閉じて戻り値を取得する方法を用意します。コモンダイアログボックスコンポーネントには Execute という論理型関数があり，ダイアログボックスで OK が選択されたときは **true** を返し，キャンセルされたときは **false** を返します。

Execute メソッドの宣言の例を次に示します。

```
class PACKAGE TMyWrapper : public TComponent
{
  ...
public:
    bool __fastcall Execute();
  ...
};
```

Execute には少なくとも次の機能が必要です。つまり，ダイアログボックスフォームを作成し，モード付きダイアログボックスとして表示し，ShowModal からの戻り値に応じて **true** か **false** を返します。

TMyDialogBox 型のダイアログボックスフォームの Execute メソッドの例を次に示します。この Execute メソッドには最小限の機能しかありません。

```
bool __fastcall TMyWrapper::Execute()
{
  DialogBox = new TMyDialogBox(Application);   // フォームを作成
    bool Result;
  try
  {
    Result = (DialogBox->ShowModal()  IDOK);  // 実行後，終了の方法に基づいて結果をセットする
  }
  catch(...)
  {
    Result = false;                           // 失敗したら Result に false を設定
  }
  DialogBox->Free();                          // フォームを破棄する
}
```

実際には，コードの多くを例外ハンドラ中に記述します。たとえば，ShowModal を呼び出す前に，ラッパーコンポーネントのインターフェースプロパティに基づいてダイアログボックスのプロパティ

を設定します。また，ShowModal を呼び出した後，ダイアログボックスの実行結果に基づいてインターフェースプロパティを設定することもあります。

この［バージョン情報］ダイアログボックスの場合には，ラッパーコンポーネントの 4 つのインターフェースプロパティを使用して，［バージョン情報］ダイアログボックスフォームのラベルの内容を設定します。［バージョン情報］ダイアログボックスはアプリケーションに情報を返さないので，ShowModal を呼び出した後は何もする必要はありません。［バージョン情報］ダイアログボックスのラッパーコンポーネントの Execute メソッドを次のように ABOUTDLG.CPP ファイルに記述します。

```
bool __fastcall TAboutBoxDlg::Execute()
{
  AboutBox = new TAboutBox(Application);      //  ［バージョン情報］ダイアログボックスを作成
  bool Result;
  try
  {
    if (ProductName == "")                    // 製品名が空白の場合は ...
      ProductName = Application->Title;       // ... アプリケーションのタイトルを代わりに使う
    AboutBox->ProductName->Caption = ProductName;   // 製品名をコピー
    AboutBox->Version->Caption = Version;           // バージョン情報をコピー
    AboutBox->Copyright->Caption = Copyright;       // 著作権情報をコピー
    AboutBox->Comments->Caption = Comments;         // 注釈をコピー
    AboutBox->Caption = "About " + ProductName;     //  ［バージョン情報］ダイアログの
                                                    // キャプションを設定
    Result = (AboutBox->ShowModal() == IDOK);       // 実行して結果を設定
  }
  catch(...)
  {
    Result = false;                         // 失敗したら Result に false を設定
    ...
  }
  AboutBox->Free();                         //  ［バージョン情報］ダイアログボックスを破棄する
  return Result == IDOK;                    // Result と IDOK を比較し，論理値を返す
}
```

ABOUTDLG.H ヘッダーファイルに，TAboutBoxDlg クラスの **public** 部に Execute メソッドの宣言を追加します。

```
class PACKAGE TAboutDlg : public TComponent
{
public:
     virtual bool __fastcall Execute();
};
```

コンポーネントのテスト

いったんダイアログボックスコンポーネントをインストールした後は，フォームにそれを配置して実行できます。テスト方法はコモンダイアログボックスの場合と同じです。［バージョン情報］ダイアログボックスを簡単にテストするには，フォームにコマンドボタンを追加し，そのボタンが選択されたときにダイアログボックスを実行するようにします。

たとえば，［バージョン情報］ダイアログボックスを作成してコンポーネントにし，コンポーネントパレットに追加したとします。このコンポーネントは次の手順で実行できます。

1. 新しいプロジェクトを作成します。
2. ［バージョン情報］ダイアログボックスコンポーネントをメインフォームに配置します。
3. フォームにコマンドボタンを配置します。
4. コマンドボタンをダブルクリックして，空のクリックイベントハンドラを作成します。
5. クリックイベントハンドラに次のコード行を記述します。

```
AboutBoxDlg1->Execute();
```

6. アプリケーションを実行します。

メインフォームが表示されたら，コマンドボタンをクリックします。［バージョン情報］ダイアログボックスが現れ，デフォルトのプロジェクトアイコンと Project1 という名前が表示されます。ダイアログボックスを閉じるには［OK］を選択します。

［バージョン情報］ダイアログボックスコンポーネントでは，プロパティを設定できます。プロパティを設定してアプリケーションを再度実行すれば，プロパティの設定を反映したダイアログボックスが表示されます。

第58章

IDE の拡張

Open Tools API を使用すると，独自のメニュー項目，ツールバーのボタン，動的フォーム作成ウィザードなどで IDE を拡張し，カスタマイズすることができます。Open Tools API は，多くの場合，単に Tools API と呼ばれます。Tools API は，IDE と対話し，IDE を制御する 100 個以上のインターフェース群（抽象クラス）であり，メインメニュー，ツールバー，メインアクションリストとメインイメージリスト，ソースエディタの内部バッファ，キーボードマクロとキーボードバインド，フォームエディタのフォームとそのコンポーネント，デバッガとデバッグ対象のプロセス，コード完了，メッセージ表示，To-Do リストなどで構成されます。

Tools API の使い方は，特定のインターフェースを実装するクラスを記述し，他のインターフェースが提供するサービスを呼び出すだけです。作成した Tools API コードをコンパイルし，設計時に設計時パッケージとして IDE にロードするか，DLL に入れておく必要があります。このように，Tools API 拡張の記述はプロパティエディタまたはコンポーネントエディタの記述に似ています。この章に入る前に，パッケージの使い方（第 15 章「パッケージとコンポーネントの操作」）とコンポーネントの登録（第 52 章「コンポーネントを設計時に利用できるようにする」）の基本を確実に理解しておいてください。第 13 章「VCL と CLX のための C++ 言語サポート」13-2 ページの「継承とインターフェース」の節をよく読み，Delphi スタイルのインターフェースについて調べるのもよいでしょう。

この章では，以下の事項について説明します。

- Tools API の概要
- ウィザードクラスの記述
- Tools API サービスの取得
- ファイルとエディタの処理
- フォームとプロジェクトの作成
- ウィザードに IDE イベントを通知する
- ウィザード DLL のインストール

Tools API の概要

Tools API の宣言は，すべて唯一のヘッダーファイル ToolsAPI.hpp に記述されます。名前空間は Toolsapi です。Tools API を使用するには，通常，designide パッケージを使います。つまり，設計時パッケージまたは実行時に使用する DLL として Tools API アドインを作成する必要があります。パッケージとライブラリについての詳細は，58-7 ページの「ウィザードパッケージのインストール」と 58-22 ページの「ウィザード DLL のインストール」を参照してください。

Tools API 拡張を記述するための主なインターフェースは IOTAWizard なので，IDE アドインのほとんどはウィザードと呼ばれます。C++Builder のウィザードと Delphi のウィザードはほとんどが相互運用可能です。Delphi で記述し，コンパイルしたウィザードを C++Builder で使用でき，C++Builder で記述したものを Delphi で使用できます。バージョン番号が同じ場合は最大の相互運用性が提供されますが，両製品の将来のバージョンで使用できるようなウィザードも記述できます。上位互換性についての詳細は，58-23 ページの「実行時パッケージなしで DLL を使う」を参照してください。

Tools API を使用するには，ToolsAPI ユニットで定義されたインターフェースを 1 つ以上実装するウィザードクラスを記述します。第 13 章 を参照し，C++Builder では Object Pascal インターフェースが抽象クラスとして表されることを思い出してください。このインターフェースを実装するには，抽象クラスのメンバー関数とその上位クラスをオーバーライドし，インターフェース GUID を認識する QueryInterface 関数を実装する必要があります。

ウィザードは Tools API が提供するサービスを利用します。それぞれのサービスは，関連の関数の集合で構成されるインターフェースです。インターフェースの実装は IDE の内部に隠蔽されます。Tools API が公開するのはそのインターフェース 1 つだけなので，開発者はインターフェースの実装を気にかけることなくウィザードを記述できます。さまざまなサービスを利用すると，ソースエディタ，フォームデザイナ，デバッガなどにアクセスできます。58-7 ページの「Tools API サービスの取得」で，このことについて詳しく説明します。

サービスとその他のインターフェースは，2 つの基本的なカテゴリに分類されます。これらのカテゴリは，名前の先頭部分にある文字列で区別されます。

- NTA （ネイティブ Tools API）が付くものは，IDE の TMainMenu オブジェクトなど，実際の IDE オブジェクトに直接アクセスします。このカテゴリのインターフェースを使う場合，ウィザードは必ず Borland パッケージを使用します。つまり，ウィザードは特定のバージョンの IDE に関連付けられます。ウィザードは，設計時パッケージまたは実行時に使用する DLL に配置されます。
- OTA（オープン Tools API）が付くものはパッケージを必要とせず，インターフェースのみを介して IDE にアクセスします。理論上は，開発者が Borland の **__fastcall** 呼び出し規約と AnsiString などの Object Pascal 型を処理できれば，COM スタイルのインターフェースをサポートする任意の言語でウィザードを記述できます。OTA インターフェースは IDE への全面的なアクセスを提供するわけではありませんが，OTA インターフェースを介して Tools API のほとんどすべての機能を利用できます。ウィザードで OTA インターフェースのみを使うと，特定のバージョンの IDE に依存しない DLL を記述できます。

Tools API のインターフェースには，プログラマが実装しなければならないものと IDE で実装されるものの 2 種類があります。ほとんどのインターフェースは後者に入ります。つまり，インターフェー

スは IDE の機能を定義しますが，実際の実装は隠されます。プログラマが実装するタイプのインターフェースは，さらにウィザード，ノーティファイア，クリエータの 3 つに分類されます。

- この節ですでに説明したように，ウィザードクラスは IOTAWizard インターフェースを実装します。派生インターフェースを実装することもあります。
- ノーティファイアは Tools API を構成するウィザード以外のインターフェースの 1 つです。IDE は，何か問題が発生したときに，ノーティファイアを使ってウィザードをコールバックします。ノーティファイアインターフェースを実装するクラスを記述して Tools API にノーティファイアを登録すると，IDE はユーザーがファイルを開くとき，ソースコードを編集するとき，フォームを変更するとき，デバッグセッションを開始するときなどにノーティファイアオブジェクトにコールバックします。ノーティファイアについては，58-18 ページの「ウィザードに IDE イベントを通知する」を参照してください。
- クリエータはプログラマが実装しなければならないもう 1 つのインターフェースです。Tools API では，クリエータを使って新しいユニット，プロジェクトなどのファイルを作成したり，既存のファイルを開いたりします。このことについての詳細は，58-14 ページの「フォームとプロジェクトの作成」を参照してください。

他に重要なインターフェースとして，モジュールとエディタがあります。モジュールインターフェースは，1 つ以上のファイルで構成されるオープンユニットです。エディタインターフェースはオープンファイルを表します。ソースファイルにはソースエディタ，フォームファイルにはフォームデザイナ，リソースファイルにはプロジェクトリソースといったように，さまざまなエディタインターフェースによって IDE のさまざまなファイルにアクセスします。これらについての詳細は，58-12 ページの「ファイルとエディタの処理」を参照してください。

以下の節では，ウィザードを記述する手順を説明します。各インターフェースについての詳細は，オンラインヘルプファイルを参照してください。

ウィザードクラスの記述

ウィザードクラスが実装するインターフェースによって，4 種類のウィザードがあります。表 58.1 で，この 4 種類のウィザードを説明します。

表 58.1　4 種類のウィザード

インターフェース型	説明
IOTAFormWizard	通常，新しいユニット，フォーム，その他のファイルを作成する
IOTAMenuWizard	ヘルプメニューに自動的に追加される
IOTAProjectWizard	通常，新しいアプリケーションまたはその他のプロジェクトを作成する
IOTAWizard	他のカテゴリに当てはまらないその他のウィザード

4 種類のウィザードの違いは，ユーザーがウィザードを呼び出す方法だけです。

- メニューウィザードは IDE のヘルプメニューに追加されます。ユーザーがこのメニュー項目を選択すると，IDE はウィザードの Execute() 関数を呼び出します。プレーンウィザードははるかに柔軟性が高いので，通常，メニューウィザードはプロトタイプやデバッグにのみ使用されます。

- フォームウィザードとプロジェクトウィザードは，オブジェクトリポジトリに登録されているのでリポジトリウィザードと呼ばれます。ユーザーはこれらのウィザードを［新規作成］ダイアログボックスから呼び出します。ユーザーは，［ツール｜リポジトリ］を選択すれば，これらのウィザードをオブジェクトリポジトリでも見ることもできます。ユーザーは［フォームの新規作成時に使用］チェックボックスをチェックするとフォームウィザードを利用できます。このボックスをチェックすると，ユーザーが［ファイル｜フォームの新規作成］を選択したときに，IDE はフォームウィザードを呼び出します。［メインフォーム］チェックボックスをチェックすることもできます。このボックスをチェックすると，IDE は新しいアプリケーションのデフォルトのフォームとしてフォームウィザードを使用します。プロジェクトウィザードを使う場合は［新規プロジェクト］チェックボックスをチェックします。ユーザーが［ファイル｜アプリケーションの新規作成］を選択すると，IDE は選択されたプロジェクトウィザードを呼び出します。
- 4 種類目のウィザードは他のカテゴリに当てはまらないウィザードです。プレーンウィザードは自動の処理を行いません。ウィザードを呼び出す方法はプログラマが定義しなければなりません。

Tools API にはウィザードに関する制約がありません。たとえば，プロジェクトの作成に必ずしもプロジェクトウィザードを使用する必要はありません。プロジェクトウィザードを記述して簡単にフォームを作成できますし，全く同様にフォームウィザードでもプロジェクトを作成できます（そのようなことが実際に必要であれば）。

ウィザードインターフェースの実装

すべてのウィザードクラスでは，最低限 IOTAWizard を実装する必要があります（その上位クラス，IOTANotifier と IInterface の実装も必要）。フォームウィザードとプロジェクトウィザードでは，そのすべての上位インターフェース，つまり IOTARepositoryWizard，IOTAWizard，IOTANotifier，IInterface を実装する必要があります。

IInterface の実装は，Object Pascal インターフェースの標準規則に従う必要があります。これは，COM インターフェースの規則と同じです。つまり，QueryInterface で型変換を行い，AddRef と Release で参照をカウントします。一般的な基本クラスを使うと，ウィザードクラスとノーティファイアクラスの記述を単純化できます。

たとえば，Delphi には TNotifierObject がありますが，これは IOTANotifier を関数本体が空の状態で実装したものです。次に示すように，同様のクラスを C++ で記述できます。

```
class PACKAGE NotifierObject : public IOTANotifier {
public:
  __fastcall NotifierObject() : ref_count(0) {}
  virtual __fastcall ~NotifierObject();
  void __fastcall AfterSave();
  void __fastcall BeforeSave();
  void __fastcall Destroyed();
  void __fastcall Modified();
protected:
  // インターフェース
  virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();
private:
  long ref_count;
};
```

IInterface インターフェースの実装は簡単です。

```
ULONG __stdcall NotifierObject::AddRef()
{
  return InterlockedIncrement(&ref_count);
}
ULONG __stdcall NotifierObject::Release()
{
  ULONG result = InterlockedDecrement(&ref_count);
  if (ref_count == 0)
    delete this;
  return result;
}
HRESULT __stdcall NotifierObject::QueryInterface(const GUID& iid, void** obj)
{
  if (iid == __uuidof(IInterface)) {
    *obj = static_cast<IInterface*>(this);
    static_cast<IInterface*>(*obj)->AddRef();
    return S_OK;
  }
  if (iid == __uuidof(IOTANotifier)) {
    *obj = static_cast<IOTANotifier*>(this);
    static_cast<IOTANotifier*>(*obj)->AddRef();
    return S_OK;
  }
  return E_NOINTERFACE;
}
```

ウィザードは IOTANotifier から継承するのでそのすべての関数を実装する必要がありますが，IDE は通常これらの関数を利用しないので，実装は空でもかまいません。

```
void __fastcall NotifierObject::AfterSave()  {}
void __fastcall NotifierObject::BeforeSave() {}
void __fastcall NotifierObject::Destroyed()  {}
void __fastcall NotifierObject::Modified()   {}
```

NotifierObject を基本クラスとして使うには，複数の継承を使用する必要があります。ウィザードクラスは，NotifierObject と 実装を要するウィザードインターフェース（IOTAWizard など）から継承する必要があります。IOTAWizard は IOTANotifier と IInterface から継承するので，派生クラスには曖昧な要素があります。たとえば，AddRef() などの関数は上位の継承関係のあらゆる下位クラスで宣言されます。, この問題を解決するために，首位の基本クラスとして 1 つの基本クラスを選択し，曖昧な関数をすべてこのクラスに割り当てます。クラスの宣言の例を次に示します。

```
class PACKAGE MyWizard : public NotifierObject, public IOTAMenuWizard {
  typedef NotifierObject inherited;
public:
  // IOTAWizard
  virtual AnsiString __fastcall GetIDString();
  virtual AnsiString __fastcall GetName();
  virtual TWizardState __fastcall GetState();
  virtual void __fastcall Execute();
// IOTAMenuWizard
  virtual AnsiString __fastcall GetMenuText();
  void __fastcall AfterSave();
  void __fastcall BeforeSave();
  void __fastcall Destroyed();
```

```
    void __fastcall Modified();
  protected:
    // IInterface
    virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
    virtual ULONG __stdcall AddRef();
    virtual ULONG __stdcall Release();
  };
```

クラスの実装に，次のコードを含めることもできます。

```
  ULONG __stdcall MyWizard::AddRef()  { return inherited::AddRef(); }
  ULONG __stdcall MyWizard::Release() { return inherited::Release(); }
  HRESULT __stdcall MyWizard::QueryInterface(const GUID& iid, void** obj)
  {
    if (iid == __uuidof(IOTAMenuWizard)) {
      *obj = static_cast<IOTAMenuWizard*>(this);
      static_cast<IOTAMenuWizard*>(*obj)->AddRef();
      return S_OK;
    }
    if (iid == __uuidof(IOTAWizard)) {
      *obj = static_cast<IOTAWizard*>(this);
      static_cast<IOTAWizard*>(*obj)->AddRef();
      return S_OK;
    }
    return inherited::QueryInterface(iid, obj);
  }
```

AfterSave，BeforeSave などは基本クラスの関数本体が空なので，派生クラスの関数本体も空の状態にすると，不必要な **inherited**::AfterSave() を回避できます。

インターフェースの簡単な実装

Object Pascal インターフェースは COM インターフェースに似ているので，COM ウィザードを使うとウィザードクラスを容易に記述できます。ただし，単純な Object Pascal インターフェースに比べて COM でははるかに大きなオーバーヘッドが必要です。基本クラスとわずかなコピー＆ペーストをうまく使えば，COM ウィザードを使わなくても容易に単純な実装を維持できることがわかるでしょう。たとえば，次のような単純なマクロを定義すると，QueryInterface を容易に実装できます。

```
  #define QUERY_INTERFACE(T, iid, obj)   \
    if ((iid) == __uuidof(T)) {          \
      *(obj) = static_cast<T*>(this);    \
      static_cast<T*>(*(obj))->AddRef(); \
      return S_OK;                       \
    }
```

このマクロは次のように使います。

```
  HRESULT __stdcall MyWizard::QueryInterface(const GUID& iid, void* obj)
  {
    QUERY_INTERFACE(IOTAMenuWizard, iid, obj);
    QUERY_INTERFACE(IOTAWizard, iid, obj);
    return inherited::QueryInterface(iid, obj);
  }
```

また，IOTAWizard と使用している派生クラスのすべてのメンバー関数をオーバーライドする必要があります。さまざまなウィザードインターフェースの関数は，そのほとんどにわかりやすい名前が付

いています。IDE はウィザードの関数を呼び出し，エンドユーザーに対してウィザードをどう表すか，そしてユーザーがウィザードを呼び出した場合にどう実行するかを決定します。

ウィザードクラスの記述が終わると，次の手順はウィザードをインストールすることです。

ウィザードパッケージのインストール

ウィザードパッケージには，他の設計時パッケージと同様に，1 つ以上の Register 関数が必要です（Register 関数についての詳細は第 52 章「コンポーネントを設計時に利用できるようにする」を参照してください）。Register 関数では，次に示すように，RegisterPackageWizard を呼び出して唯一の引数としてウィザードオブジェクトを渡すことによって，ウィザードをいくつでも登録できます。

```
namespace Example {
  void __fastcall PACKAGE Register()
  {
    RegisterPackageWizard(new MyWizard());
    RegisterPackageWizard(new MyOtherWizard());
  }
}
```

プロパティエディタやコンポーネントなども同じパッケージの一部として登録できます。

ただし，設計時パッケージはメインの C++Builder アプリケーションの一部なので，任意のフォーム名はアプリケーションと他の設計時パッケージのすべてを通じて一意でなければなりません。このことはパッケージを使用する上で大きな不都合です。他のプログラマかがフォームにどう名付けるかわかりません。

開発中は，他の設計時パッケージと同様に，パッケージマネージャで［インストール］ボタンをクリックしてウィザードパッケージをインストールします。IDE はこのパッケージをコンパイルし，リンクしてからロードします。パッケージを正常にロードできたかどうかを示すダイアログボックスが表示されます。

Tools API サービスの取得

有効な処理を行うには，ウィザードが IDE（エディタ，ウィンドウ，メニューなど）にアクセスする必要があります。これは，サービスインターフェースの役割です。Tools API には，ファイルアクションを実行するアクションサービス，ソースコードエディタにアクセスするエディタサービス，デバッガにアクセスするデバッガサービスなど，多くのサービスがあります。表 58.2 に，すべてのサービスインターフェースの概要を示します。

表 58.2　Tools API サービスインターフェース

インターフェース型	説明
INTAServices	IDE 特有のオブジェクト（メインメニュー，アクションリスト，イメージリスト，ツールバー）にアクセスする
IOTAActionServices	基本的なファイルアクション（ファイルを開く，閉じる，保存，再ロード）を実行する

表 58.2 Tools API サービスインターフェース（つづき）

インターフェース型	説明
IOTACodeCompletionServices	コード完了にアクセスし，ウィザードがカスタムコード完了マネージャをインストールできるようにする
IOTADebuggerServices	デバッガにアクセスする
IOTAEditorServices	ソースコードエディタとその内部バッファにアクセスする
IOTAKeyBindingServices	ウィザードがカスタムキーボードバインドを登録できるようにする
IOTAKeyboardServices	キーボードマクロおよびバインドにアクセスする
IOTAKeyboardDiagnostics	キー入力処理のデバッグを切り替える
IOTAMessageServices	メッセージ表示にアクセスする
IOTAModuleServices	オープンファイルにアクセスする
IOTAPackageServices	インストールされたすべてのパッケージとそのすべてのコンポーネントの名前を問い合わせる
IOTAServices	その他のサービス
IOTAToDoServices	To-Do リストにアクセスし，ウィザードがカスタム To-Do マネージャをインストールできるようにする
IOTAToolsFilter	ツールフィルタノーティファイアを登録する
IOTAWizardServices	ウィザードの登録と登録抹消を行う

特定のサービスインターフェースを使用する場合は，BorlandIDEServices 変数を該当するサービスに変換します。QueryInterface を呼び出すか，もう少し便利な Sysutils::Supports 関数を使用してください。たとえば次のようになります。

```
void set_keystroke_debugging(bool debugging)
{
  _di_IOTAKeyboardDiagnostics diag;
  if (Supports(BorlandIDEServices, IID_IOTAKeyboardDiagnostics, &diag))
    diag->KeyTracing = debugging;
}
```

ウィザードで特定のサービスをよく使う場合は，ウィザードクラスのデータメンバーとしてそのサービスへのポインタを保持します。DelphiInterface テンプレート（_di_IOTAModuleServices など）を使うと，Tools API がオブジェクトの持続時間を自動的に管理するので，開発者はウィザードのデストラクタで特別な作業をする必要がありません。

IDE 特有のオブジェクトの使い方

ウィザードは，IDE のメインメニュー，ツールバー，アクションリスト，イメージリストへの全面的なアクセスを提供します（ただし，IDE の多くのコンテキストメニューには Tools API からはアクセスできません）。この節では，ウィザードが IDE 特有のオブジェクトを使って IDE と対話する方法を単純な例で説明します。

INTAServices インターフェースの使い方

IDE 特有のオブジェクトの処理する場合の出発点は INTAServices インターフェースです。このインターフェースを使うと，イメージリストへのイメージの追加，アクションリストへのアクションの追加，メインメニューへのメニュー項目の追加，ツールバーへのボタンの追加が可能になります。メニュー項目とツールボタンにアクションを関連付けることができます。ウィザードが破棄されると，

そのウィザードで作成したオブジェクトもクリーンアップしなければなりませんが，ウィザードでイメージリストに追加したイメージを削除する必要はありません。イメージを削除すると，このウィザード以降に追加されたすべてのイメージのインデックスに混乱が生じます。

ウィザードは実際の IDE のオブジェクト，TMainMenu，TActionList，TImageList，TToolBar を使うので，他のアプリケーションと同様にコードを記述できます。一方，IDE のクラッシュや，重要な機能の無効化（［ファイル］メニューの削除など）を招く恐れもあります。

イメージをイメージリストに追加する

ウィザードを呼び出すメニュー項目を追加するとします。さらに，ウィザードを呼び出すツールバーのボタンをユーザーが追加できるようにします。最初の手順は IDE のイメージリストにイメージを追加することです。これで，イメージのインデックスをアクションに利用できます。メニュー項目とツールバーのボタンでもイメージを使用します。イメージエディタを使って，16 × 16 のビットマップリソースを含むリソースファイルを作成します。次のコードをウィザードのコンストラクタに追加します。

```
_di_INTAServices services;
Supports(BorlandIDEServices, IID_INTAServices, &services);
// イメージをイメージリストに追加する
Graphics::TBitmap* bitmap(new Graphics::TBitmap());
bitmap->LoadFromResourceName(reinterpret_cast<unsigned>(HInstance), "Bitmap1");
int image = services->AddMasked(bitmap, bitmap->TransparentColor,
                                 "Tempest Software.intro wizard image");
delete bitmap;
```

HInstance の型が適切ではないので，LoadFromResourceID が要求する型に変換する必要があります。さらに，リソースファイルに指定した名前または ID を使用して確実にそのリソースをロードします。#pragma resource を使ってパッケージにリソースファイルを追加します。イメージの背景の色を選択する必要があります。背景に色を付けたくない場合は，ビットマップにない色を選択します。

アクションリストにアクションを追加する

イメージインデックスを使って次のようにアクションを作成します。ウィザードでは OnExecute イベントと OnUpdate イベントを使います。一般に，ウィザードは OnUpdate イベントを使ってアクションを有効または無効にします。*OnUpdate* イベントは直ちに返されるようにしておきます。そうしないと，ユーザーはウィザードをロードした後，IDE が遅くなるように感じます。アクションの OnExecute イベントは，ウィザードの Execute メソッドに似ています。メニュー項目を使ってフォームウィザードまたはプロジェクトウィザードを呼び出す場合は，OnExecute が直接 Execute を呼び出すようにします。

```
action = new TAction(0);
action->ActionList = services->ActionList;
action->Caption    = GetMenuText();
action->Hint       = "Display a silly dialog box";
action->ImageIndex = image;
action->OnUpdate   = action_update;
action->OnExecute  = action_execute;
```

メニュー項目は，その Action プロパティを新しく作成されたアクションに設定します。メニュー項目の作成で注意を要する点は，挿入位置の把握です。次の例では［表示］メニューを検索し，新しいメニュー項目を［表示］メニューの最初の項目として挿入しています（一般に，絶対的な位置に頼る

のは得策ではありません。別のウィザードがいつメニューに挿入されるかわかりません。C++Builder の将来のバージョンでメニューの順序が変わる可能性もあります。メニュー内で特定の名前のメニュー項目を検索するのがよいでしょう。わかりやすいように，単純な例を次に示します)。

```
for (int i = 0; i < services->MainMenu->Items->Count; ++i)
{
  TMenuItem* item = services->MainMenu->Items->Items[i];
  if (CompareText(item->Name, "ViewsMenu") == 0)
  {
    menu_item = new TMenuItem(0);
    menu_item->Action = action;
    item->Insert(0, menu_item);
  }
}
```

IDE のアクションリストにアクションを追加すると，ユーザーがツールバーをカスタマイズするときにアクションが表示されます。ユーザーはアクションを選択し，任意のツールバーにボタンとして追加できます。ウィザードがアンロードされると，このことが原因で問題が発生します。つまり，存在しないアクションへの参照（ポインタ）と OnClick イベントハンドラによって，すべてのツールボタンが終了します。アクセス違反を回避するために，アクションを参照するすべてのツールボタンをウィザードで検出し，このようなボタンを削除します。

ツールバーボタンの削除

ツールバーからボタンを削除する適当な関数はないので，CM_CONTROLCHANGE メッセージを送信する必要があります。最初のパラメータは変更するコントロールです。2 番目のパラメータがゼロの場合はこのコントロールが削除され，ゼロ以外の場合はツールバーにコントロールが追加されます。デストラクタは，ツールバーのボタンを削除してからアクションとメニュー項目を削除します。これらの項目を削除すると，IDE の ActionList と MainMenu の項目が自動的に削除されます。

```
void __fastcall remove_action (TAction* action, TToolBar* toolbar)
{
  for (int i = toolbar->ButtonCount; --i >= 0; )
  {
    TToolButton* button = toolbar->Buttons[i];
    if (button->Action == action)
    {
      // toolbar から button を削除
      toolbar->Perform(CM_CONTROLCHANGE, WPARAM(button), 0);
      delete button;
    }
  }
}
__fastcall MyWizard::~MyWizard()
{
  _di_INTAServices services;
  Supports(BorlandIDEServices, INTAServices, &services);
  // すべてのツールバーをチェックしてこのアクションを使う
  // ボタンを削除
  remove_action(action, services->ToolBar[sCustomToolBar]);
  remove_action(action, services->ToolBar[sDesktopToolBar]);
  remove_action(action, services->ToolBar[sStandardToolBar]);
  remove_action(action, services->ToolBar[sDebugToolBar]);
  remove_action(action, services->ToolBar[sViewToolBar]);
```

```
    remove_action(action, services->ToolBar[sInternetToolBar]);
    delete menu_item;
    delete action;
  }
```

この単純な例からわかるように，ウィザードは IDE ときわめて柔軟に対話します。しかし，柔軟性に伴う問題もあります。ポインタの参照先が存在しないなどのアクセス違反が発生しやすくなります。次の節では，この種の問題を診断するためのヒントを示します。

ウィザードのデバッグ

特有の Tools API を使ったウィザードを記述すると，IDE のクラッシュの原因になるコードを書いてしまう恐れがあります。ウィザードを記述してインストールしても，思いどおりに機能しない場合もあります。設計時コードを処理する際の大きな問題の 1 つにデバッグがあります。しかし，この問題は容易に解決できます。ウィザードは C++Builder 自体にインストールされるので，［実行｜実行時引数］を選択し，パッケージのホストアプリケーションを C++Builder の実行可能ファイル（bcb.exe）に設定するだけです。

パッケージをデバッグする場合（またはデバッグが必要な場合）は，パッケージをインストールしないでください。かわりに，メニューバーから［実行｜実行］を選択します。これで，C++Builder の新しいインスタンスが起動します。新しいインスタンスで，［コンポーネント｜パッケージのインストール］を選択すると，コンパイル済みのパッケージがインストールされます。C++Builder の元のインスタンスに戻ると，ウィザードのソースコードにブレークポイントを設定できる場所を示す目印の青いドットが表示されます（表示されない場合は，コンパイラオプションをチェックしてデバッグが有効になっていることを確認し，適切なパッケージをロードしたことを確認し，さらにプロセスモジュールをチェックしてロードしたい .bpl ファイルをロードしたことを確認します）。

YVCL，CLX，RTL のコードはこの方法ではデバッグできませんが，ウィザード自体は完全にデバッグできるので，適切に機能しない部分を見つけることができます。

インターフェースのバージョン番号

IOTAMessageServices など一部のインターフェースの宣言に注目すると，名前が似ている別のインターフェースから継承したことがわかります。たとえば，IOTAMessageServices50 は IOTAMessageServices40 からの継承です。バージョン番号をこのように使うと，コードは C++Builder のリリース間の変更に対応できます。

Tools API は COM の基本原則に従うので，インターフェースとその GUID の変更はありません。新しいリリースでインターフェースに機能を追加すると，Tools API は古いインターフェースから継承した新しいインターフェースを宣言します。GUID は，古い変更のないインターフェースに結び付いたものと同じです。新しいインターフェースは新しい GUID の商標を取得します。古い GUID を使った古いウィザードは引き続き機能します。

また，Tools API はインターフェース名を変更してソースコードの互換性を維持しようとします。この動作を確認するには，Tools API の 2 種類のインターフェース（Borland の実装とユーザーの実装）の違いを明らかにすることが重要です。IDE がインターフェースを実装すると，名前は最新バージョ

ンのインターフェースのものになります。新しい機能は既存のコードに影響を与えません。古いインターフェースには古いバージョン番号が付きます。

しかし，ユーザーが実装したインターフェースでは，基本インターフェースの新しいメンバー関数がユーザーコードの新しい関数を要求します。したがって，古いインターフェースに名前をそのまま残し，新しいインターフェース名の末尾にバージョン番号を付ける傾向があります。

メッセージサービスの例について考察します。C++Builder 6 では新しい機能，メッセージグループが導入されました。したがって，基本のメッセージサービスインターフェースは新しいメンバー関数を要求します。これらの機能は新しいインターフェースクラスで宣言されており，IOTAMessageServices という名前を維持します。古いメッセージサービスインターフェースは，IOTAMessageServices50（バージョン 5 の場合）という名前に変わります。古い IOTAMessageServices の GUID は，メンバー関数が同じなので新しい IOTAMessageServices50 の GUID と同じです。

ユーザーが実装するインターフェースの例として，IOTAIDENotifier について考察します。C++Builder 5 では，新しいオーバーロード関数，AfterCompile と BeforeCompile が追加されています。IOTAIDENotifier を使った既存のコードを変更する必要はありませんが，新しい機能を必要とする新しいコードは IOTAIDENotifier50 から継承された新しい関数をオーバーライドするように変更しなければなりません。バージョン 6 では新しい関数が追加されていないので，最新バージョンは IOTAIDENotifier50 になります。

経験則として，新しいコードを記述する場合は派生の最下位のクラスを使います。単に C++Builder の新しいリリースで既存のウィザードを再コンパイルする場合は，ソースコードだけを残します。.

ファイルとエディタの処理

次の問題に入る前に，Tools API がファイルを処理する方法について知っておく必要があります。主なインターフェースは IOTAModule です。モジュールとは，論理的に関連するオープンファイルの集まりです。たとえば，1 つのモジュールは 1 つのユニットを表します。モジュールには 1 つ以上のエディタがあります。エディタはそれぞれ実装（.cpp）ファイル，インターフェース（.h）ファイル，フォーム（.dfm または .xfm）ファイルを表します。エディタインターフェースは IDE のエディタの内部の状態を反映するので，ユーザーが変更を保存しなくても，ウィザードは変更されたコードとユーザーが参照するフォームを確認できます。

モジュールインターフェースの使い方

モジュールインターフェースを使用するには，モジュールサービス（IOTAModuleServices）を起動します。モジュールサービスのすべてのオープンモジュールを問い合わせたり，ファイル名またはフォーム名でモジュールを検索したり，モジュールインターフェースを取得するファイルを開くことができます。

プロジェクト，リソース，タイプライブラリなど，さまざまなファイルによるさまざまなモジュールがあります。モジュールインターフェースを特定の種類のモジュールインターフェースに変換し，モ

ジュールがその種類かどうかを調べます。例として，現在のプロジェクトグループインターフェースを取得する 1 つの方法を次に示します。

```
// 現在のプロジェクトグループを返す。プロジェクトグループがない場合は 0 を返す
_di_IOTAProjectGroup __fastcall CurrentProjectGroup()
{
  _di_IOTAModuleServices svc;
  Supports(BorlandIDEServices, IID_IOTAModuleServices, &svc);
  for (int i = 0; i < svc->ModuleCount; ++i)
  {
    _di_IOTAModule module = svc->Modules[i];
    _di_IOTAProjectGroup group;
    if (Supports(module, IID_IOTAProjectGroup, &group))
      return group;
  }
  return 0;
}
```

エディタインターフェースの使い方

各モジュールには 1 つ以上のエディタインターフェースがあります。一部のモジュールには，実装（.cpp）ファイル，インターフェース（.h）ファイル，フォーム（.dfm）ファイルなど，複数のエディタがあります。すべてのエディタは IOTAEditor インターフェースを実装します。どんなエディタかを調べるには，エディタを特定の型に変換します。たとえば，フォームエディタインターフェースを取得する場合は，次のように記述します。

```
// モジュールのフォームエディタを返す。ユニットにフォームがない場合は 0 を返す
_di_IOTAFormEditor __fastcall GetFormEditor(_di_IOTAModule module)
{
  for (int i = 0; i < module->ModuleFileCount; ++i)
  {
    _di_IOTAEditor editor = module->ModuleFileEditors[i];
    _di_IOTAFormEditor formEditor;
    if (Supports(editor, IID_IOTAFormEditor, &formEditor))
      return formEditor;
  }
  return 0;
}
```

エディタインターフェースを使うと，エディタ内部の状態にアクセスできます。開発者は，ユーザーが編集中のソースコードまたはコンポーネントの調査，ソースコード，コンポーネント，プロパティの変更，ソースエディタとフォームエディタの選択の切り替えなど，エンドユーザーが実行できるエディタアクションのほとんどを実行できます。

フォームエディタを使うと，ウィザードはフォームのすべてのコンポーネントにアクセスできます。各コンポーネント（ルートフォームまたはデータモジュールを含む）には，IOTAComponent インターフェースが関連付けられています。ウィザードによって，コンポーネントのほとんどのプロパティを調べたり変更したりできます。コンポーネントの完全な制御が必要な場合は，IOTAComponent インターフェースを INTAComponent に変換します。特有のコンポーネントインターフェースを使用すると，ウィザードは TComponent ポインタに直接アクセスできます。TFont など，クラス型のプロパティの読み取りまたは変更が必要な場合はこのことが重要です。この操作は，NTA スタイルのインターフェースを介してのみ可能です。

フォームとプロジェクトの作成

C++Builder には，多くのフォームウィザードとプロジェクトウィザードがすでにインストールされていますが，独自のウィザードを記述することもできます。オブジェクトリポジトリによって，開発者はプロジェクトで利用できる静的テンプレートを作成できますが，ウィザードは動的なのではるかに高度な機能を提供します。ウィザードでは，ユーザーにプロンプトが表示され，ユーザーの応答によってさまざまなファイルを作成できます。この節では，フォームウィザードとプロジェクトウィザードを記述する方法を説明します。

モジュールの作成

フォームウィザードまたはプロジェクトウィザードは，通常，新しいファイルを 1 つ以上作成します。ただし，実際のファイルでなく，名称未設定，未保存のモジュールが作成されるにすぎません。ユーザーが保存するときに，IDE によってファイル名を入力するプロンプトが表示されます。ウィザードはクリエータオブジェクトを使ってこのようなモジュールを作成します。

クリエータクラスは，IOTACreator から継承したクリエータインターフェースを実装します。ウィザードはクリエータオブジェクトをモジュールサービスの CreateModule メソッドに渡し，IDE はモジュールの作成に必要なパラメータをクリエータオブジェクトにコールバックします。

たとえば，新しいフォームを作成するフォームウィザードは，通常，**false** を返す GetExisting() と **true** を返す GetUnnamed() を実装します。これで，名前のないモジュールが作成され（したがって，ファイルを保存する前にユーザーが名前を指定する必要があります），既存のファイルではバックアップされません（したがって，ユーザーは変更しなくてもファイルを保存する必要があります）。クリエータには，他にも作成するファイルの種類（プロジェクト，ユニット，フォームなど）を IDE に通知するメソッド，ファイルの内容を提供するメソッド，フォーム名，上位クラス名などの重要な情報を提供するメソッドがあります。追加のコールバックによって，ウィザードは新しく作成されたプロジェクトにモジュールを追加したり，新しく作成したフォームにコンポーネントを追加したりします。

新しいファイルを作成するには（多くの場合，フォームウィザードまたはプロジェクトウィザードで要求される），通常，新しいファイルの内容を提供する必要があります。そのためには，IOTAFile インターフェースを実装する新しいクラスを記述する必要があります。ウィザードがデフォルトのファイル内容で作成できる場合は，IOTAFile を返す関数から 0 ポインタを返すことができます。

たとえば，貴社組織では各ファイルの先頭に必ず標準のコメントブロックを記述するものとします。この作業はオブジェクトリポジトリの静的テンプレートを使っても実行できますが，作成者と作成日を反映するように手作業でコメントブロックを訂正する必要があります。一方，クリエータを使うとファイルが作成されるときに動的にコメントブロックを挿入できます。

最初の手順は新しいユニットとフォームを作成するウィザードを記述することです。クリエータの関数のほとんどは，ゼロ，空の文字列などのデフォルト値を返します。デフォルト値によって，Tools API にそのデフォルトの動作を指示します。GetCreatorType をオーバーライドして，作成するモジュールの種類（ユニットまたはフォーム）を Tools API に知らせます。ユニットを作成する場合は，マクロ sUnit を返します。フォームを作成する場合は，sForm を返します。コードを単純にするために，コンストラクタの引数としてクリエータ型の 1 つのクラスを使用します。このクリエータ型

をデータメンバーに保存して，GetCreatorType がその値を返せるようにします。NewImplSource と NewIntfSource をオーバーライドして，任意のファイルの内容を返します。

```
class PACKAGE Creator : public IOTAModuleCreator {
public:
  __fastcall Creator(const AnsiString creator_type)
    : ref_count(0), creator_type(creator_type) {}
  virtual __fastcall ~Creator();

// IOTAModuleCreator
  virtual AnsiString __fastcall GetAncestorName();
  virtual AnsiString __fastcall GetImplFileName();
  virtual AnsiString __fastcall GetIntfFileName();
  virtual AnsiString __fastcall GetFormName();
  virtual bool __fastcall GetMainForm();
  virtual bool __fastcall GetShowForm();
  virtual bool __fastcall GetShowSource();
  virtual _di_IOTAFile __fastcall NewFormFile(
    const AnsiString FormIdent, const AnsiString AncestorIdent);
  virtual _di_IOTAFile __fastcall NewImplSource(
    const AnsiString ModuleIdent, const AnsiString FormIdent,
    const AnsiString AncestorIdent);
  virtual _di_IOTAFile __fastcall NewIntfSource(
    const AnsiString ModuleIdent, const AnsiString FormIdent,
    const AnsiString AncestorIdent);
  virtual void __fastcall FormCreated(
    const _di_IOTAFormEditor FormEditor);

// IOTACreator
  virtual AnsiString __fastcall GetCreatorType();
  virtual bool __fastcall GetExisting();
  virtual AnsiString __fastcall GetFileSystem();
  virtual _di_IOTAModule __fastcall GetOwner();
  virtual bool __fastcall GetUnnamed();

protected:
  // IInterface
  virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();

private:
  long ref_count;
  const AnsiString creator_type;
};
```

クリエータのメンバーのほとんどは，ゼロまたは空の文字列を返します。論理メソッドは **true** を返します。ただし，GetExisting は **false** を返します。もっとも興味深いメソッドは GetOwner です。このメソッドは，現在のプロジェクトモジュールへのポインタを返します。プロジェクトがない場合は 0 を返します。現在のプロジェクトまたは現在のプロジェクトグループを簡単に検出する方法はありません。すべてのオープンモジュールについて，GetOwner を繰り返す必要があります。プロジェクトグループが見つかった場合は，これが唯一開いているプロジェクトグループなので，GetOwner はその現在のプロジェクトを返します。それ以外の場合，この関数は最初に見つけたプロジェクトモジュールを返します。開いているプロジェクトがない場合は 0 を返します。

```
_di_IOTAModule __fastcall Creator::GetOwner()
{
  // 現在のプロジェクトを返す
```

```
  _di_IOTAProject result = 0;
  _di_IOTAModuleServices svc = interface_cast<IOTAModuleServices>(BorlandIDEServices);
  for (int i = 0; i < svc->ModuleCount; ++i)
  begin
    _di_IOTAModule module = svc->Modules[i];
    _di_IOTAProject project;
    _di_IOTAProjectGroup group;
    if (Supports(module, IID_IOTAProject, &project)) {
      // 最初のプロジェクトモジュールを記憶
      if (result == 0)
        result = project;
    } else if (Supports(module, IID_IOTAProjectGroup, &group)) {
      // プロジェクトグループが検出されたのでアクティブなプロジェクトを返す
      result = group->ActiveProject;
      break;
    }
  }
  return result;
}
```

クリエータは，NewFormSource からの 0 を返して，デフォルトのフォームファイルを生成します。おもしろいメソッドとして，NewImplSource と NewIntfSource があります。これらは，ファイルの内容を返す IOTAFile のインスタンスを作成します。

ファイルクラスは，IOTAFile インターフェースを実装します。これは，ファイル年齢として -1（ファイルが存在しないことを表す）を返し，ファイルの内容を文字列として返します。単純なファイルクラスを維持するためには，クリエータで文字列を作成し，ファイルクラスはこれを渡すだけにします。

```
class File : public IOTAFile {
public:
  __fastcall File(const AnsiString source);
  virtual __fastcall ~File();
  AnsiString __fastcall GetSource();
  System::TDateTime __fastcall GetAge();
protected:
  // IInterface
  virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();
private:
  long ref_count;
  AnsiString source;
};
__fastcall File::File(const AnsiString source)
  : ref_count(0), source(source)
{}
AnsiString __fastcall File::GetSource()
{
  return source;
}
System::TDateTime __fastcall File::GetAge()
{
  return -1;
}
```

ファイル内容のテキストをリソースに保存すると変更しやすくなりますが，単純にするためにこの例ではソースコードをウィザードで変更しています。次の例では，フォームがあると仮定してソースコードを生成しています。プレーンユニットの単純なケースを容易に追加できます。FormIdent をテストして，空の場合はプレーンユニットを作成し，それ以外の場合はフォームユニットを作成します。コードの基本的な骨組みは，IDE のデフォルトと同じですが（もちろん先頭にコメントを追加），任意の変更が可能です。

```
_di_IOTAFile __fastcall Creator::NewImplSource(
  const AnsiString ModuleIdent,
  const AnsiString FormIdent,
  const AnsiString AncestorIdent)
{
  const AnsiString form_source =
"/*-----------------------------------------------------------------------\n"
" %m - description\n"
" Copyright © %y Your company, inc.\n"
" Created on %d\n"
" By %u\n"
"  -----------------------------------------------------------------------*/\n"
"\n"
"#include <vcl.h>\n"
"#pragma hdrstop\n"
"\n"
"#include \"%m.h\"\n"
"//-----------------------------------------------------------------------\n"
"#pragma package(smart_init)\n"
"#pragma resource \"*.dfm\"\n"
"T%f *%f;\n"
"//-----------------------------------------------------------------------\n"
"__fastcall T%m::T%m(TComponent* Owner)\n"
"        : T%a(Owner)\n"
"{\n"
"}\n"
"//-----------------------------------------------------------------------\n";
  return new File(expand(form_source, ModuleIdent, FormIdent,
                         AncestorIdent));
}
```

ソースコードに %m や %y の形の文字列があることに注目してください。これらは意識の上では printf または Format コントロールと同様ですが，ウィザードの展開機能によって展開されます。特に，%m はモジュールまたはユニットの識別子，%f はフォーム名に，%a は上位クラス名に展開されます。フォーム名に大文字の T を挿入してフォームの型名として使用する方法に注目してください。その他の書式指定子には，コメントブロックを生成しやすくするものもあります。たとえば，%d は日付，%u はユーザー，%y は年を表します（展開されたコードは Tools API とは無関係なので，読者の課題とします）。

NewIntfSource は NewImplSource に似ていますが，インターフェース（.h）ファイルを生成します。

最後の手順は，2 つのフォームウィザード（クリエータ型としてユニットを使用するものとフォームを使用するもの）を作成することです。INTAServices を使用すると，ユーザーにとっての新しい利点として［ファイル｜新規作成］メニューに各ウィザードを呼び出すメニュー項目を追加できます。メニュー項目の OnClick イベントハンドラは，ウィザードの Execute 関数を呼び出すことができます。

一部のウィザードでは，IDE における他の状況に応じて，メニュー項目を有効または無効にする必要があります。たとえば，ソースコード制御システムにプロジェクトをチェックインするウィザードは，IDE でファイルが開いていない場合は，［チェックイン］メニュー項目を無効にする必要があります。次の節で説明するノーティファイアを使うと，この機能をウィザードに追加できます。.

ウィザードに IDE イベントを通知する

適切に動作するウィザードを記述する上で重要な点は，ウィザードが IDE イベントに応答できるようにすることです。特に，モジュールインターフェースを追跡するウィザードは，ウィザードがインターフェースを解放できるように，ユーザーがモジュールをいつ閉じるかを検知する必要があります。このために，ウィザードにはノーティファイアが必要です。したがって，開発者はノーティファイアクラスを記述する必要があります。

すべてのノーティファイアクラスは 1 つ以上のノーティファイアインターフェースを実装します。ノーティファイアインターフェースはコールバックメソッドを定義し，ウィザードはノーティファイアオブジェクトを Tools API で登録し，IDE は重要な出来事が発生するとノーティファイアにコールバックします。

すべてのノーティファイアインターフェースは IOTANotifier から継承しますが，そのすべてのメソッドが特定のノーティファイアに使用されるわけではありません。表 58.3 に，すべてのノーティファイアインターフェースを示し，それぞれについて簡単に説明します。

表 58.3　ノーティファイアインターフェース

インターフェース型	説明
IOTANotifier	すべてのノーティファイアの抽象基本クラス
IOTABreakpointNotifier	デバッガのブレークポイントの発生または変更
IOTADebuggerNotifier	デバッガのプログラムの実行，あるいはブレークポイントの追加または削除
IOTAEditLineNotifier	ソースエディタ内の行の移動の追跡
IOTAEditorNotifier	ソースファイルの変更または保存，あるいはエディタにおけるファイルの切り替え
IOTAFormNotifier	フォームの保存，あるいはフォームまたはフォーム（またはデータモジュール）上の任意のコンポーネントの変更
IOTAIDENotifier	プロジェクトのロード，パッケージのインストールなどのグローバル IDE イベント
IOTAMessageNotifier	メッセージの表示におけるタブ（メッセージグループ）の追加と削除
IOTAModuleNotifier	モジュールの変更，保存，名前の変更
IOTAProcessModNotifier	デバッガにおけるプロセスモジュールのロード
IOTAProcessNotifier	デバッガにおけるスレッドとプロセスの作成または破棄
IOTAThreadNotifier	デバッガにおけるスレッドの状態の変更
IOTAToolsFilterNotifier	ツールフィルタの呼び出し

ノーティファイアの使い方を調べるために，前の例について考察します。この例では，各ソースファイルにコメントを追加するウィザードを，モジュールクリエータを使って作成します。コメントにはユニットの初期名が記載されていますが，ユーザーはほとんどの場合，別の名でファイルを保存しま

す。この場合，ウィザードが実際のファイル名に合わせてコメントを更新すれば，ユーザーには好都合です。

これにはモジュールノーティファイアが必要です。ウィザードは CreateModule が返すモジュールインターフェースを保存し，これを使ってモジュールノーティファイアを登録します。モジュールノーティファイアはユーザーがファイルを変更または保存するときに通知を受け取りますが，これらのイベントはこのウィザードにとって重要ではありません。したがって，AfterSave と関連の関数はすべて本体が空です。重要な関数は ModuleRenamed です。IDE は，ユーザーがファイルを新しい名前で保存した場合にこの関数を呼び出します。モジュールノーティファイアクラスの宣言は次のとおりです。

```
class ModuleNotifier : public NotifierObject, public IOTAModuleNotifier
{
  typedef NotifierObject inherited;
public:
  __fastcall ModuleNotifier(const _di_IOTAModule module);
  __fastcall ~ModuleNotifier();
// IOTAModuleNotifier
  virtual bool __fastcall CheckOverwrite();
  virtual void __fastcall ModuleRenamed(const AnsiString NewName);
// IOTANotifier
  void __fastcall AfterSave();
  void __fastcall BeforeSave();
  void __fastcall Destroyed();
  void __fastcall Modified();
protected:
  // IInterface
  virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();
private:
  _di_IOTAFormEditor formEditor;
  AnsiString name;        // モジュールの元の名前を記憶
  int index;              // ノーティファイアインデックス
};
```

ノーティファイアを記述する 1 つの方法は，コンストラクタでノーティファイアにそれ自体を自動的に登録させることです。デストラクタはノーティファイアの登録を抹消します。モジュールノーティファイアの場合は，ユーザーがファイルを閉じると IDE が Destroyed メソッドを呼び出します。この場合，はそれ自体の登録を抹消し，モジュールインターフェースへの参照を破棄します。IDE はノーティファイアへの参照を破棄します。これで参照カウントがゼロになり，オブジェクトが解放されます。したがって，デストラクタは慎重に記述する必要があります。ノーティファイアはすでに登録を抹消されている場合があります。

```
__fastcall ModuleNotifier::ModuleNotifier(const _di_IOTAModule module)
  : index(-1), module(module)
{
  // このノーティファイアを登録
  index = module->AddNotifier(this);
  // 元のモジュール名を記憶
  name = ChangeFileExt(ExtractFileName(module->FileName), "");
}
__fastcall ModuleNotifier::~ModuleNotifier()
{
  // ノーティファイアの登録抹消（まだの場合）
```

```
  if (index >= 0)
    module->RemoveNotifier(index);
}
void __fastcall ModuleNotifier::Destroyed()
{
  // モジュールインターフェースの破棄とノーティファイアのクリーンアップ
  if (index >= 0)
  {
    // ノーティファイアの登録抹消
    module->RemoveNotifier(index);
    index = -1;
  }
  module = 0;
}
```

ユーザーがファイル名を変更すると，IDE はノーティファイアの ModuleRenamed 関数にコールバックします。この関数では，新しい名前がパラメータになります。ウィザードはこの名前を使ってファイルのコメントを更新します。ソースバッファを編集するために，ウィザードは編集位置インターフェースを使います。ウィザードが該当する位置を検出し，該当するテキストを検出したことを再確認し，このテキストを新しい名前で置き換えます。

```
void __fastcall ModuleNotifier::ModuleRenamed(const AnsiString NewName)
{
  // 新しいファイル名からモジュール名を取得
  AnsiString ModuleName = ChangeFileExt(ExtractFileName(NewName), "");
  for (int i = 0; i < module->GetModuleFileCount(); ++i)
  {
    // すべてのソースエディタバッファを更新
    _di_IOTAEditor editor = module->GetModuleFileEditor(i);
    _di_IOTAEditBuffer buffer;
    if (Supports(editor, IID_IOTAEditBuffer, &buffer))
    {
      _di_IOTAEditPosition pos = buffer->GetEditPosition();
      // モジュール名の位置はコメントの 2 行目
      // 先頭の空白をスキップして元のモジュール名をコピー
      // 位置が正しいことを再確認
      pos->Move(2, 1);
      pos->MoveCursor(mmSkipWhite | mmSkipRight);
      AnsiString check = pos->RipText("", rfIncludeNumericChars | rfIncludeAlphaChars);
      if (check == name)
      {
        pos->Delete(check.Length());    // 元の名前を削除
        pos->InsertText(ModuleName);    // 新しい名前を挿入
        name = ModuleName;              // 新しい名前を記憶
      }
    }
  }
}
```

ユーザーがモジュール名の上にコメントを挿入する場合はどうでしょうか。この場合は，編集行ノーティファイアを使ってモジュール名が記載された行番号を追跡する必要があります。これには，IOTAEditLineNotifier インターフェースと IOTAEditLineTracker インターフェースを使います。これらのインターフェースについては，オンラインヘルプを参照してください。

ノーティファイアは慎重に記述する必要があります。ノーティファイアがそのウィザードより長く存続しないようにします。たとえば，ウィザードを使って新しいユニットを作成してからウィザードをアンロードしても，ユニットに関連付けられたノーティファイアは依然として存在します。結果は予測できませんが，多くの場合，IDE がクラッシュします。したがって，ウィザードはそのすべてのノーティファイアを追跡し，ウィザードが破棄される前に，ノーティファイアの登録をすべて抹消する必要があります。一方，ユーザーがまずファイルを閉じると，モジュールノーティファイアは Destroyed 通知を受け取ります。つまり，ノーティファイアはそれ自体の登録を抹消してモジュールのすべての参照を解放する必要があります。ノーティファイアはウィザードのマスターノーティファイアリストからもそれ自体を削除しなければなりません。

ウィザードの最新バージョンの Execute 関数を次に示します。この関数は新しいモジュールを作成し，モジュールインターフェースを使ってモジュールノーティファイアを作成してから，モジュールノーティファイアをインターフェースリスト（TInterfaceList）に保存します。

```
void __fastcall DocWizard::Execute()
{
  _di_IOTAModuleServices svc;
  Supports(BorlandIDEServices, IID_IOTAModuleServices, &svc);
  _di_IOTAModule module = svc->CreateModule(new Creator(creator_type));
  _di_IOTAModuleNotifier notifier = new ModuleNotifier(module);
  list->Add(notifier);
}
```

インターフェースリストについてウィザードのデストラクタを繰り返し実行し，リストにあるすべてのノーティファイアの登録を抹消します。IDE にも同じインターフェースがあるので，インターフェースリストを使ってインターフェースを解放するだけでは不十分です。ノーティファイアオブジェクトを解放するには，IDE にノーティファイアインターフェースを解放するように指示する必要があります。この場合，デストラクタはノーティファイアにそのモジュールが破棄されたと錯覚させます。さらに複雑な状況では，ノーティファイアクラス用に Unregister という別の関数を記述するのがよいかもしれません。

```
__fastcall DocWizard::~DocWizard()
{
  // リストにあるすべてのノーティファイアの登録を抹消
  for (int i = list->Count; --i >= 0; )
  {
    _di_IOTANotifier notifier;
    Supports(list->Items[i], IID_IOTANotifier, &notifier);
    // 関連のオブジェクトが破棄されたように装う
    // ノーティファイアはこれを認識してノーティファイア自体をクリーンアップ
    notifier->Destroyed();
    list->Delete(i);
  }
  delete list;
  delete item;
}
```

その他のウィザードは，ウィザードの登録，メニュー項目のインストールなどの一般的な細かい作業を行います。次の節では，ウィザードの登録の詳細な作業について，パッケージでなく DLL による方法を説明します。

ウィザード DLL のインストール

ウィザードは DLL にもインストールできます。ウィザードが動的 RTL と実行時パッケージを使用していることを確認します。プロジェクトオプションの実行時パッケージのリストに designide パッケージを追加します。DLL は，INITWIZARD0001 という初期化関数をエクスポートする必要があります。IDE は，DLL をロードするときに固有のエクスポート名を探してその関数を呼び出します。モジュール定義ファイルを使うと，関数をエクスポートできます。それ以外の場合は，**__declspec**(dllexport) 宣言を使ってエクスポートします。後者の場合は，名前の変形も回避しなければならないので，**extern** "C" で関数を宣言します。

関数は TWizardInitProc 型でなければなりません。この関数では，登録関数へのポインタが引数の 1 つになります。パッケージウィザードが RegisterPackageWizard を呼び出すのと同様に，登録関数を呼び出して各ウィザードオブジェクトを登録します。初期化関数は，正常に終了した場合に **true** を返し，障害が発生した場合は **false** を返します。次の例では，初期化関数を記述する方法を示します。

```
extern "C" bool __stdcall __declspec(dllexport) INITWIZARD0001(
  const _di_IBorlandIDEServices,
  TWizardRegisterProc RegisterProc,
  TWizardTerminateProc&)
{
  RegisterProc(new MyWizard());
  RegisterProc(new MyOtherWizard());
  return true;
}
```

実行時パッケージを使用する場合，初期化関数に渡す最初の引数は重要ではありません（最初のパラメータについては，58-23 ページの「実行時パッケージなしで DLL を使う」を参照）。最後の引数は終了手続きの関数ポインタへの参照です。この関数は，グローバルクリーンアップを実行するために使います。通常，最後のパラメータは無視できます。ウィザードがデストラクタでウィザード自体の後にクリーンアップするためです。

DLL をインストールするには，次に示すキーの下のレジストリにエントリを追加します。

```
HKEY_CURRENT_USER \ Software \ Borland \ C++Builder \ 6.0 \ Experts
```

エントリ名は，ウィザードの ID 文字列のようにユニークな名前でなければなりません。この値は DLL への絶対パスです。C++Builder を再起動すると，Experts キーの下のすべての DLL をロードします。DLL は，IDE が動作している間はロードされた状態で残ります。C++Builder が終了すると，DLL がアンロードされます。これでデバッグが遅くなります。開発中，設計時パッケージが望ましいのはこのためです。ウィザードをリリースできる状態になったら，設計時パッケージを DLL に変更します。

C++Builder でバージョンが適切でないというメッセージが表示された場合，このメッセージは INITWIZARD0001 関数が見つからないことを表しています。名前を正しく入力したかどうかを調べ，名前の変形がなく正しくエクスポートしたことを再確認します。

パッケージでなく DLL を使う主な利点は，名前の不一致の問題を回避できることです。DLL では，必要な任意のフォームを作成でき，他のウィザードとの名前の重複を心配することはありません。DLL のもう 1 つの利点は，特定のバージョンの C++Builder に依存しないように DLL を設計できるこ

とです。ただし，この場合は実行時パッケージを一切使用できません。この問題については，次の節で説明します。

実行時パッケージなしで DLL を使う

ウィザード DLL では，必ずしも実行時パッケージを使う必要はありません。実行時パッケージを使用する主な利点は，DLL が小規模なことです。ウィザードは IDE にロードされた場合にのみ有効なので，C++Builder パッケージを利用でき，小規模な DLL を利用すればよいことがわかります。一方，パッケージはバージョンに固有です。C++Builder と Delphi の複数のバージョンでウィザードを利用する場合は，実行時パッケージを利用できません。

BorlandIDEServices 変数は designide パッケージ内でのみ定義されるので，この変数を使用できません。かわりに，初期化関数に渡す最初の引数を使います。この引数はたまたま同じ値をとります。ウィザードはこの値を保存し，すべての Tools API サービスへのアクセスに使用します。例を示します。

```
extern "C" bool __stdcall __declspec(dllexport) INITWIZARD0001(
  const _di_IBorlandIDEServices svc,
  TWizardRegisterProc reg,
  TWizardTerminateProc&)
{
  LocalIDEServices = svc;
  reg(new DocWizard(sUnit));
  reg(new DocWizard(sForm));
  return true;
}
AnsiString __fastcall DocWizard::GetDesigner()
{
  _di_IOTAServices svc;
  Supports(LocalIDEServices, IID_IOTAServices, &svc);
  return svc->GetActiveDesignerType();
}
```

Tools API の設計では，新しいバージョンの C++Builder に対して DLL が引き続き機能することが保証されます。古いインターフェースはすべてその GUID を保持し，新しいインターフェースは新しい GUID を採用します。Tools API でのバージョン番号と GUID の使い方については，58-11 ページの「インターフェースのバージョン番号」を参照してください。

バージョンに依存しないためには，ウィザードは特有の（NTA）インターフェースを使用できません。これでウィザードの機能が限定されます。最大の制限は，ウィザードがメニュー バーなどの IDE オブジェクトにアクセスできないことです。ただし，Tools API インターフェースのほとんどは OTA インターフェースなので，ウィザードがどのようにインストールされても，便利でおもしろいウィザードを記述できます。

付録 A

ANSI 処理系独自の機能

ANSI C には，明示的には定義されていない部分も残されています。そうした部分は，C コンパイラの処理系の開発者が自由に定義してよいことになっています。この付録では，そうした処理系独自の機能の詳細をボーランドがどのように定義しているかを説明します。各トピックの番号は，1990 年 2 月の C ANSI/ISO 規格に対応しています。

C と C++ との間には相違点がありますが，ここで説明しているのは C についてだけです。C++ 準拠に関する情報は，Borland Comunity Web Site（http://community.borland.com/cpp/）を参照してください。

2.1.1.3 How to identify a diagnostic. 診断メッセージの表示指定方法

プログラム実行時に適切なオプションを組み合わせて使用したときに，コンパイラが生成する Fatal, Error，または Warning で始まるメッセージはすべて ANSI が指定した意味での診断メッセージです。この実装を有効にするために必要なオプションを次に示します。

表 A.1　ANSI 規格に準じるために必要なオプション

オプション	機能
-A	ANSI のキーワードだけを有効にする
-C-	コメントのネストはできない
-i32	識別子の有効文字数を先頭から 32 文字までとする
-p-	C の呼び出し規約を使用する
-w-	すべての警告をオフにする
-wbei	不適切な初期化子に関する警告をオンにする
-wbig	大きすぎる定数に関する警告をオンにする
-wcpt	移植性のないポインタの比較に関する警告をオンにする
-wdcl	型あるいは記憶クラスを伴わない宣言に関する警告をオンにする
-wdup	同等でないマクロ定義の重複に関する警告をオンにする
-wext	extern と static の両方で宣言された変数に関する警告をオンにする
-wfdt	typedef を使った関数定義に関する警告をオンにする
-wrpt	移植性のないポインタの変換に関する警告をオンにする
-wstu	未定義の構造体に関する警告をオンにする
-wsus	疑わしいポインタ変換に関する警告をオンにする

表 A.1　ANSI 規格に準じるために必要なオプション（つづき）

オプション	機能
-wucp	signed および **unsigned char** へのポインタの混在に関する警告をオンにする
-wvrt	値を返す **void** 関数に関する警告をオンにする

ここで言及していない他のオプションは，必要に応じて設定することができます。

2.1.2.2.1 The semantics of the arguments to main. main に対する引数の意味

プログラムが DOS 上で実行される場合には，argv[0] はプログラム名を指します。

残りの argv 文字列は，DOS コマンドライン引数の各構成要素を指します。引数の区切りであるホワイトスペースは削除され，隣りあう非ホワイトスペース文字の各並びは単一の引数として扱われます。引用符で囲まれた文字列は正しく（スペースを含む 1 つの文字列として）扱われます。

2.1.2.3 What constitutes an interactive device. 対話型装置が指すもの

コンソールのようなすべての装置を指します。

2.2.1 The collation sequence of the execution character set. 実行文字セットの大小順序

実行文字セットの照合順序では，ASCII の文字の値を使用します。

2.2.1 Members of the source and execution character sets. ソースファイルでの文字セットおよび実行文字セットの定義

ソースおよび実行文字セットは，ASCII セットとシフト JIS セットです。^Z（〔Ctrl〕＋〔Z〕）を除くすべての文字を文字列リテラル，文字定数，およびコメントに使用することができます。

2.2.1.2 Multibyte characters. 多バイト文字

C++Builder は，マルチバイト文字をサポートしています。

2.2.2 The direction of printing. 出力の方向

左から右への出力が，通常の方向です。

2.2.4.2 The number of bits in a character in the execution character set. 実行文字セットにおける文字のビット数

実行文字セット中の文字は 8 ビットです。

3.1.2 The number of significant initial characters in identifiers. 識別子の意味のある文字の長さ

先頭から 250 文字が有効文字ですが，コマンドラインオプション（**-i**）を使えば有効文字数を変更できます。内部識別子および外部識別子はともに，同一の有効文字数を使用します（C++ の識別子の有効文字数には制限はありません）。

3.1.2 Whether case distinctions are significant in external identifiers.
外部識別子の意味のある長さの文字における英小文字および英大文字の区別

通常コンパイラは，リンカに大文字と小文字の区別を要求します。コマンドラインオプション（**-lc-**）を使えば，大文字小文字を区別しないようにできます。IDE では，［プロジェクト｜オプション｜リンカ（詳細)］を選択して［大 / 小文字を無視してリンク］をチェックすることもできます。

3.1.2.5 The representations and sets of values of the various types of integers.
各整数型と数値範囲

表 A.2　C++ **における表示指定診断**

データ型	最小値	最大値
signed char	-128	127
unsigned char	0	255
signed short	-32,768	32,767
unsigned short	0	65,535
signed int	-2,147,483,648	-2,147,483,647
unsigned int	0	4,294,967,295
signed long	-2,147,483,648	2,147,483,647
unsigned long	0	4,294,967,295

すべての **char** 型は，記憶域として 8 ビットのバイトを 1 つ使用します。

すべての **short** 型は，2 バイトを使用します。

すべての **int** 型は，4 バイトを使用します。

すべての **long** 型は，4 バイトを使用します。

アラインメントが要求された場合（**-a**），**char** 以外のすべての整数型オブジェクトは偶数バイト境界に配置されます。要求されたアラインメントが **-a4** の場合，4 バイトのアラインメントになります。char はアラインメントされません。

3.1.2.5 The representations and sets of values of the various types of floating-point numbers. **浮動小数点型の表現と数値範囲**

C++Builder の浮動小数点型において，Intel 8087 で用いられている IEEE 浮動小数点形式が使用されます。**float** 型は 32 ビット IEEE 実数形式，**double** 型は 64 ビット IEEE 実数形式，**long double** 型は 80 ビット IEEE 拡張実数形式を，それぞれ使用します。

3.1.3.4 The mapping between source and execution character sets.
ソース文字セットおよび実行文字セット間の対応

文字列リテラルあるいは文字定数内のすべての文字は，実行されるプログラムの中でもそのままです。ソース文字セットと実行文字セットはまったく同じです。

3.1.3.4 The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant.
基本実行文字集合あるいはワイド文字定数のための拡張実行文字集合で表現されない文字や拡張表記を含む整数文字定数の値
ワイド文字はサポートされています。

3.1.3.4 The current locale used to convert multibyte characters into corresponding wide characters for a wide character constant.
多バイト文字を，対応する（ワイド文字定数の）ワイド文字に変換する際に使用される現在のロケール
ワイド文字定数は認識されます。

3.1.3.4 The value of an integer constant that contains more than one character, or a wide character constant that contains more than one multibyte character. 2 **文字以上の文字を含む単純文字定数，あるいは** 2 **文字以上の多バイト文字を含むワイド文字定数の値**
文字定数は，1 文字または 2 文字を含むことができます。32 ビットコンパイラでは，4 文字まで含むことができます。2 文字が含まれる場合には，最初の文字は下位バイトに，2 番目の文字は上位バイトに割り当てられます。

3.2.1.2 The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented.
整数をより少ないビット数の符号付き整数に変換，あるいは符号なし整数を同じビット数の符号付き整数に変換したとき，変換後の値を表現できない場合の結果
これらの変換は，単に上位ビットを切り捨てることによって行われます。符号付き整数は 2 の補数として記憶されるため，結果の数もそのような値として解釈されます。サイズの小さい整数の上位ビットがゼロでなければ，その値は負の値として解釈されます。ゼロの場合は正の値として解釈されます。

3.2.1.3 The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value.
汎整数型の値を浮動小数点型に変換するときに，元の値を正確に表現できない場合の丸めの方向
整数値は最近似値を表すように丸められます。したがって，たとえば **long** 値（2^{31} -1）は **float** 値 2^{31} に変換されます。中間値（丸められる桁が 5 の場合）は，IEEE の標準算術規則に従って丸められます。

3.2.1.4 The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number. **浮動小数点型をより精度の高い浮動小数点型に変換する場合の切り捨てあるいは丸めの方向**
値は，最近似値を表すように丸められます。中間値（丸められる桁が 5 の場合）は，IEEE の標準算術規則に従って丸められます。

3.3 The results of bitwise operations on signed integers.
ビット単位の演算子が符号付き整数型に用いられたときの結果

ビットごとの演算子は，符号付き整数に対して，対応する符号なし整数の場合と同じように適用されます。符号ビットは，通常のデータビットとして扱われます。その結果は，通常の 2 の補数の符号付き整数として解釈されます。

3.3.2.3 What happens when a member of a union object is accessed using a member of a different type. 異なる型のメンバーを用いて共用体オブジェクトのメンバーにアクセスしたときの動作

このアクセスは，単にそこに格納されているビットへのアクセスとして許されます。別のメンバーを用いて浮動小数点メンバーにどのようにアクセスすればよいかを理解するには，浮動小数点値のビットのコード化の詳細を把握しておく必要があります。格納されているメンバーのサイズが，アクセスに使われたメンバーよりも小さい場合，残りのビットは小さいメンバーが格納される前の値を持つことになります。

3.3.3.4 The type of integer required to hold the maximum size of an array.
配列の最大サイズの保持に必要な整数型

通常の配列ではその型は **unsigned int** で，大きい配列の場合には **signed long** になります。

3.3.4 The result of casting a pointer to an integer or vice versa.
ポインタと整数の間の型変換の結果

同一サイズの整数とポインタ間で変換が行われるときは，どのビットも変更されません。サイズの大きい型から小さい型に変換されるときは，上位ビットが切り捨てられます。小さい整数型から大きいポインタ型に変換されるときは，まず整数がそのポインタ型と同一サイズの整数型に拡張されます。

したがって，符号付き整数は新しいバイトを満たすために符号拡張されます。同様に，小さいポインタ型が大きい整数型に変換される場合には，まずそのポインタ型がその整数型と同じサイズのポインタ型に拡張されます。

3.3.5 The sign of the remainder on integer division.
整数の除算における剰余の符号

オペランドの 1 つだけが負の場合，剰余の符号は負になります。両方のオペランドがともに正またはともに負の場合には，剰余は正になります。

3.3.6 The type of integer required to hold the difference between two pointers to elements of the same array, ptrdiff_t.
同一配列の要素を指す 2 つのポインタの差の保持に必要な整数型 ptrdiff_t

signed int 型です。

3.3.7 The result of a right shift of a negative signed integral type.
符号付き汎整数型で負の値を持つ場合の右シフトの結果

負の符号付きの値は，右シフトされると符号拡張されます。

3.5.1 The extent to which objects can actually be placed in registers by using the *register* storage-class specifier. **記憶クラス指定子** register **を用いることによりレジスタ内に実際に配置できるオブジェクトの範囲**

1 バイト，2 バイト，または 4 バイトの整数型あるいはポインタ型で宣言されたオブジェクトは，レジスタ内に置くことがきます。最低 2 つ，最大 6 つのレジスタを使用することができます。実際に使われるレジスタの数は，関数の一時的な値にどのレジスタが使われるかによります。

3.5.2.1 Whether a plain int bit-field is treated as a signed int or as an unsigned int bit field. **単なる** int **型のビットフィールドの最上位ビット位置を，符号ビットとして扱うか**

int とだけ指定されたビットフィールドは，符号ビット **signed int** のビットフィールドとして取り扱われます。

3.5.2.1 The order of allocation of bit fields within an int. int **内のビットフィールド割り当ての順序**

ビットフィールドは，下位ビットから上位ビットの順で割り当てられます。

3.5.2.1 The padding and alignment of members of structures. **構造体メンバーのパディングと境界調整**

デフォルトでは，構造体にはパディングは使用されません。ワードアラインメントオプション（**-a**）を使用した場合，構造体は偶数アドレスにパディングされ，文字あるいは文字配列型を持たないすべてのメンバーは 2 の倍数のオフセットに境界付けられます。

3.5.2.1 Whether a bit-field can straddle a storage-unit boundary. **ビットフィールドは，記憶域単位の境界をまたぐことができるか**

アラインメントオプション（**-a**）が指定されていないときは，ビットフィールドがワード境界にまたがることがありますが，隣接した 5 つのバイトに渡って格納されることはありません。

3.5.2.2 The integer type chosen to represent the values of an enumeration type. **列挙型の値を表すために選択される整数型**

列挙子は，すべて **int** 型以上のサイズの整数型として格納されます。列挙子の値が **int** 型に収まらない場合は，**long** 型または **unsigned long** 型として格納されます。これは，コンパイラの **-b** オプションを指定した場合のデフォルトです。

-b- オプションを指定すると，列挙型は値を表すことができる最小の整数型に格納されます。これには，**signed char**，**unsigned char**，**signed short**，**unsigned short**，**signed int**，**unsigned int**，**signed long**，および **unsigned long** などすべての整数型が該当します。

C++ では，すべての列挙型を **int** 型以上の整数型として格納することは適切ではないため，必ず **-b-** オプションを指定しなければなりません。

3.5.3 What constitutes an access to an object that has volatile-qualified type. volatile **修飾型のオブジェクトへのアクセス**

volatile オブジェクトへの参照は，オブジェクトにアクセスするものとします。隣接するメモリ位置へのアクセス時もオブジェクトのアクセスになるかどうかは，そのメモリがハードウェア上でどのよ

うな構成になっているかによります。VRAM のような特殊装置のメモリでは，その装置自体の構成によって決まります。通常のメモリでは，**volatile** オブジェクトは非同期割り込みによりアクセスされる可能性のあるメモリにのみ使用されます。したがって，隣接オブジェクトへのアクセスは本オブジェクトには影響を与えません。

3.5.4 The maximum number of declarators that can modify an arithmetic, structure, or union type. **算術型，構造体型，共用体型を変更できる宣言の最大数**

宣言の数に関する特定の制限はありません。宣言の数はかなり多くまで許されますが，関数内でいくつものブロック中で何層にもネストされているときには，許される宣言の数は減少します。ファイルレベルで許される宣言の最大数は，少なくとも 50 はあります。

3.6.4.2 The maximum number of case values in a switch statement. switch **文における** case **名札の最大数**

switch 文における **case** 値の数に特定の制限はありません。**case** 情報を保持できる十分なメモリがある限り，コンパイラはすべての **case** 値を受け入れます。

3.8.1 Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. **条件取り込みを制御する定数式内の単一文字からなる文字定数の値が，実行文字集合内の同じ文字定数値に一致するか，またそのような文字定数が負の値を持ってよいか**

すべての文字定数は，条件指令中の定数も含めて，同じ（実行）文字セットを使用します。1 文字定数は，その文字が符号付き（デフォルト，**-K** オプションが指定されていない状態）であれば負になることもあります。

3.8.2 The method for locating includable source files. **インクルード可能なソースファイルの検索手順**

不等号カッコ内（）のインクルードファイル名は，コマンドラインでインクルードディレクトリが与えられている場合は，インクルードディレクトリのそれぞれでファイルが検索されます。インクルードディレクトリの検索は，次の順で行われます。

1. コマンドラインで指定されたディレクトリ
2. BCC32.CFG 内で指定されているディレクトリ
3. インクルードディレクトリが 1 つも指定されていない場合は，カレントディレクトリのみを検索

3.8.2 The support for quoted names for includable source files. 2 **つの " " 区切り記号で囲まれたソースファイルのファイル名のサポート**

引用符内で示されたファイル名の場合には，次の順でファイルが検索されます。

1. #include 文を含むファイルと同じディレクトリ
2. #include 文を含むファイルを，#include 文によってインクルードしているファイルと同じディレクトリ
3. カレントディレクトリ
4. **/I** コンパイラオプションによって指定されたパス
5. 環境変数 INCLUDE によって指定されたパス

3.8.2 The mapping of source file name character sequences. **ソースファイル名の文字の並びの対応付け**

インクルードファイル名中の円記号は，エスケープシーケンスではなく，独立した文字として処理されます。大文字と小文字の違いはいっさい無視されます。

3.8.8 The definitions for __DATE__ and __TIME__ when they are unavailable. __DATE__ と __TIME__ が使用できない場合の定義

__DATE__ と __TIME__ は常に使用でき，オペレーティングシステムの日付と時刻が使用されます。

4.1.1 The decimal point character. **小数点文字**

ピリオド（.）です。

4.1.5 The type of the sizeof operator, *size_t*. sizeof **演算子の型** size_t

size_t の型は **unsigned int** です。

4.1.5 The null pointer constant to which the macro NULL expands. **マクロ** NULL **が展開する空ポインタ定数**

NULL は，**int** の 0 あるいは **long** の 0 になります。どちらも 32 ビットの符号付きの値です。

4.2 The diagnostic printed by and the termination behavior of the assert function. assert **関数によって出力される診断メッセージと終了動作**

出力される診断メッセージは，「Assertion failed: *expression*，file *filename*，line *nn*」という形式で，*expression* は偽となった assert 中の式，*filename* はソースファイル名，*nn* は assert が呼び出された行番号を示しています。

このメッセージが出力された直後に abort が呼び出されます。

4.3 The implementation-defined aspects of character testing and case-mapping functions. **文字種類の判定および文字の変換の関数の処理系定義**

4.3.1 で述べられているものがすべてです。

4.3.1 The sets of characters tested for by the isalnum, isalpha, iscntrl, islower, isprint and isupper functions. isalnum，isalpha，iscntrl，islower，isprint，isupper **の各関数によりテストされる文字**

デフォルトの C ロケールの場合，最初の 128 個の ASCII 文字です。それ以外の場合には，256 個の文字すべてです。

4.5.1 The values returned by the mathematics functions on domain errors. **定義域エラーに対して数学関数が返す値**

IEEE 形式の NAN（not a number: 非数）です。

4.5.1 Whether the mathematics functions set the integer expression *errno* to the value of the macro ERANGE on underflow range errors.
アンダーフローを起こす値域エラーに対して，整数式 errno **が数学関数からマクロ** ERANGE **の値を得るか**
セットしません。セットされるのは，領域エラー，特異値エラー，オーバーフローエラー，精度の損失エラーの場合のみです。

4.5.6.4 Whether a domain error occurs or zero is returned when the fmod function has a second argument of zero. fmod **関数が第** 2 **引数がゼロのときに，定義域エラーが発生するか，あるいはゼロが返されるか**
定義域エラーは発生しません。fmod(x，0) は 0 を返します。

4.7.1.1 The set of signals for the signal function. signal **関数のシグナル**
SIGABRT，SIGFPE，SIGILL，SIGINT，SIGSEGV，および SIGTERM です。

4.7.1.1 The semantics for each signal recognized by the signal function. signal **関数が認識するシグナルの意味**
「signal」の解説を参照してください。

4.7.1.1 The default handling and the handling at program startup for each signal recognized by the signal function. signal **関数が認識する各シグナルに対する省略時の処理とプログラム開始時の処理**
「signal」の解説を参照してください。

4.7.1.1 If the equivalent of signal(sig, SIG_DFL); is not executed prior to the call of a signal handler, the blocking of the signal that is performed. signal(sig，SIG_DFL); **と同等のことをシグナル処理ルーチンの呼び出しより前に，実行するかまたはシグナルを遮断するか**
signal(sig, SIG_DFL);（あるいはこれと同等なもの）は，常に実行されます。

4.7.1.1 Whether the default handling is reset if the SIGILL signal is received by a handler specified to the signal function. signal **関数に指定された処理ルーチンが** SIGILL **シグナルを受け取った場合に，省略時の操作はリセットされるか**
リセットされません。

4.9.2 Whether the last line of a text stream requires a terminating newline character. **テキストストリームの最終行が，終了を示す改行文字を必要とするか**
何も必要とされません。

4.9.2 Whether space characters that are written out to a text stream immediately before a newline character appear when read in.
テキストストリームに，改行文字の直前に書き込まれた空白文字の並びが，読み込まれるときに現れるか
存在し，この空白文字も読み込まれます。

4.9.2 The number of null characters that may be appended to data written to a binary stream. **バイナリストリームに書き込まれたデータの最後に付加されたヌル文字**
ありません。

4.9.3 Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file. **追加モードでオープンされたストリームに結び付けられたファイル位置表示子は，最初にファイルの始めまたは終わりのどちらに位置付けされるか**
追加モードストリームのファイル位置標識は，最初はファイルの先頭に置かれます。これは，各書き込みの前にファイルの終わりにリセットされます。

4.9.3 Whether a write on a text stream causes the associated file to be truncated beyond that point. **テキストストリームへの書き込みが，結び付けられたファイルを最終書き込み点の直後で切り捨てるか**
0 バイトの書き込みは，バッファリングの方式によって，切り捨てられる場合と切り捨てられない場合があります。0 バイトの書き込みにおける実際の動作は定義されていないと考えるのが妥当です。

4.9.3 The characteristics of file buffering. **ファイルバッファリングの特性**
ファイルのバッファリングは，完全バッファリング，行バッファリング，あるいはバッファリングなし，のいずれかを指定できます。ファイルがバッファリングされる場合，ファイルを開くときに 512 バイトのデフォルトバッファが作成されます。

4.9.3 Whether a zero-length file actually exists.
長さ 0 **のファイルが実際に存在するか**
存在します。

4.9.3 Whether the same file can be open multiple times.
同一のファイルを引き続いて再オープンできるか
オープンできます。

4.9.4.1 The effect of the remove function on an open file.
ファイルがオープンされているときの remove **関数の動作**
すでにオープンされているファイルに対して特別なチェックは行われません。責任はプログラマに委ねられています。

4.9.4.2 The effect if a file with the new name exists prior to a call to rename.
rename **関数を呼び出す前に，新しいファイル名を持つファイルが存在していたときの** rename **関数の動作**
rename は -1 を返し，errno には EEXIST がセットされます。

4.9.6.1 The output for %p conversion in fprintf. fprintf **の** %p **変換に対する出力**
8 桁の 16 進数（XXXXXXXX）で，ゼロパディング，大文字で出力されます（%81X と同じ）。

4.9.6.2 The input for %p conversion in fscanf. fscanf **の** %p **変換に対する入力**
4.9.6.1 を参照してください。

4.9.6.2 The interpretation of a -(hyphen) character that is neither the first nor the last character in the scanlist for a %[conversion in fscanf.
fscanf **の** %[**変換における，スキャンリスト中の最初の文字でも最後の文字でもない** -**（ハイフン）の解釈**
scanf の解説を参照してください。

4.9.9.1 The value the macro errno is set to by the fgetpos or ftell function on failure. fgetpos **あるいは** ftell **関数が失敗したときにマクロ** errno **に設定される値**
EBADF ファイル番号が正しくない

4.9.10.4 The messages generated by perror. perror が生成するメッセージ

Win32 で生成されるメッセージ	
Arg list too big（引数リストが大きすぎる）	Math argument（数学引数）
Attempted to remove current directory （カレントディレクトリを削除しようとした）	Memory arena trashed （メモリブロックが破壊された）
Bad address（不正なアドレス）	Name too long（名前が長すぎる）
Bad file number（不正なファイル番号）	No child processes（子プロセスがない）
Block device required（ブロック装置が必要）	No more files（これ以上ファイルはない）
Broken pipe（パイプの破損）	No space left on device（装置に領域が足りない）
Cross-device link（装置間リンク）	No such device（そのような装置は存在しない）
Error 0（エラー 0）	No such device or address （そのような装置またはアドレスはない）
Exec format error（実行フォーマットエラー）	No such file or directory （そのようなファイルまたはディレクトリはない）
Executable file in use（実行形式ファイルを使用中）	No such process（そのようなプロセスはない）
File already exists（ファイルがすでに存在している）	Not a directory（ディレクトリではない）
File too large（ファイルが大きすぎる）	Not enough memory（メモリ不足）
Illegal seek（不正なシーク）	Not same device（装置が異なる）
Inappropriate I/O control operation （不適切な入出力制御処理）	Operation not permitted（処理が許されていない）
Input/output error（入出力エラー）	Path not found（パスが見つからない）
Interrupted function call（割り込み関数の呼び出し）	Permission denied（アクセスは拒否された）
Invalid access code（無効なアクセスコード）	Possible deadlock（処理不可能）
Invalid argument（無効な引数）	Read-only file system（読み込み専用ファイルシステム）
Invalid data（無効なデータ）	Resource busy（リソース使用中）
Invalid environment（無効な環境）	Resource temporarily unavailable （リソースは一時的に使用不可能）
Invalid format（無効な書式）	Result too large（結果が大きすぎる）
Invalid function number（無効なファンクション番号）	Too many links（リンクが多すぎる）
Invalid memory block address （無効なメモリブロックアクセス）	Too many open files （オープンファイルが多すぎる）
Is a directory（ディレクトリである）	

4.10.3 The behavior of calloc, malloc, or realloc if the size requested is zero.
要求されるサイズがゼロの場合の，calloc，malloc，realloc の動作

calloc と malloc は要求を無視して 0 を返します。realloc はブロックを解放します。

4.10.4.1 The behavior of the abort function with regard to open and temporary files.
オープンされているファイルとテンポラリファイルに関する abort 関数の動作

ファイルバッファはフラッシュされず，ファイルはクローズされません。

4.10.4.3 The status returned by exit if the value of the argument is other than zero, EXIT_SUCCESS, or EXIT_FAILURE.
引数の値が，ゼロ，EXIT_SUCCESS**，**EXIT_FAILURE **以外のときに，**exit **によって返されるステータス**
特にありません。ステータスは渡されたとおりに返されます。ステータスは **signed char** で表されます。

4.10.4.4 The set of environment names and the method for altering the environment list used by getenv.
環境リストを変更するために，getenv **により使用される方法と環境名**
環境文字列はオペレーティングシステムの中で SET コマンドによって定義されたものです。putenv を使用して，現在のプログラムの実行期間中は文字列を変更できますが，環境文字列を永続的に変更するためには，SET コマンドを使わなければなりません。

4.10.4.5 The contents and mode of execution of the string by the system function. system **関数による文字列の実行のモードと内容**
文字列はオペレーティングシステムコマンドとして解釈されます。COMSPEC か CMD.EXE が使用され，引数文字列は実行すべきコマンドとして渡されます。オペレーティングシステムの内部コマンドも，バッチファイルや実行可能プログラムと同様に実行されます。

4.11.6.2 The contents of the error message strings returned by strerror.
strerror **により返されるエラーメッセージ文字列の内容**
4.9.10.4 を参照してください。

4.12.1 The local time zone and Daylight Saving Time. **ローカル時間帯と夏時間**
マシンのローカルな時刻と日付の定義に準じます。

4.12.2.1 The era for clock. **クロックの年代**
クロックが示す時刻どおりです（開始時刻はプログラムの実行開始時刻）。

4.12.3.5 The formats for date and time. **日付と時刻の書式**
C++Builder では ANSI の書式が使用されます。

付録 B

WebSnap サーバー側スクリプトリファレンス

この付録では，WebSnap がどのようにスクリプトを使用して WebSnap Web サーバーアプリケーション内で動的 HTML ページを生成するかを説明します。対象読者は，Web ページモジュール内でアダプタページプロデューサではなくページプロデューサを使用する開発者です。アダプタページプロデューサはスクリプトの生成を自動的に行いますが，通常のページプロデューサではページテンプレート内のスクリプトを手動で追加しなければなりません。ここで示す情報は，ページテンプレートでのスクリプトの作成に役立ちます。

この付録は，アダプタページプロデューサを使用していて，その出力について詳しく知りたいユーザーにも有用です。ただし，スクリプト操作は高度なトピックです。基本的な WebSnap アプリケーションを作成する場合は，スクリプトの生成方法を理解する必要はありません。

この付録は 3 つの節で構成されています。1 番目の節では，スクリプト内でアクセスできるさまざまなオブジェクト型について説明します。2 番目の節では，WebSnap アプリケーション内で定義されるグローバルオブジェクトについて説明します。3 番目の節では，JScript の例を示し，HTML ページテンプレート内でスクリプトをどのように使用すれば Web サーバーアプリケーションから情報を取り出せるかを説明します。

この付録は，オブジェクトのスクリプトインターフェースについての API リファレンスとして書かれています。オブジェクトプロパティの説明には，プロパティ名，プロパティの型（text や Boolean など），プロパティが読み書き可能かどうかなどが含まれています。メソッドの説明は，メソッド名と呼び出すときの構文で始まります。

オブジェクト型

表 B.1 は，スクリプト可能な一般的なオブジェクト型を示しています。これらの型は，グローバルオブジェクトのようなオブジェクトのプロパティとして一般的に使われています。スクリプト可能なすべてのオブジェクト型がこの表に示されているわけではなく，インスタンスに異なる名前を付けることができるオブジェクト型だけが示されています。たとえば，アプリケーションオブジェクト型があります。アプリケーション内でインスタンス化されるアプリケーションオブジェクト型は 1 つだけな

ので，アプリケーション型はグローバルオブジェクトの節で Application オブジェクトとして説明されています。

表 B.1　WebSnap のオブジェクト型

オブジェクト型	説明
Adapter 型 （B-2 ページ）	アダプタのプロパティとメソッドを定義します。アダプタには，Module のプロパティを使って名前でアクセスできます。
AdapterAction 型 （B-4 ページ）	アダプタアクションのプロパティとメソッドを定義します。アクションには，アダプタのプロパティを使って名前で参照されます。
AdapterErrors 型 （B-6 ページ）	アダプタの Errors プロパティを定義します。Errors プロパティは，アクションの実行時またはページの生成時に発生したエラーをリストするために使用します。
AdapterField 型 （B-6 ページ）	アダプタフィールドのプロパティとメソッドを定義します。フィールドは，アダプタのプロパティを使って名前で参照されます。
AdapterFieldValues 型 （B-10 ページ）	アダプタフィールドの Values プロパティのプロパティとメソッドを定義します。
AdapterFieldValuesList 型 （B-10 ページ）	アダプタフィールドの ValuesList プロパティのプロパティとメソッドを定義します。
AdapterHiddenFields 型 （B-11 ページ）	アダプタの HiddenFields プロパティと HiddenRecordFields プロパティを定義します。
AdapterImage 型 （B-11 ページ）	アダプタフィールドとアダプタアクションの Image プロパティを定義します。
Module 型 （B-12 ページ）	モジュールのプロパティを定義します。モジュールには，Modules 変数のプロパティを使って名前でアクセスできます。
Page 型 （B-12 ページ）	ページのプロパティを定義します。ページには，Pages オブジェクトのプロパティを使って名前でアクセスできます。Page オブジェクトを使って，生成中のページにアクセスできます。

Adapter 型

アダプタのプロパティとメソッドを定義します。アダプタには，モジュールのプロパティを使って名前でアクセスできます（例，ModuleName.Adapter）。

アダプタは，データ項目を表すフィールドコンポーネントと，コマンドを表すアクションコンポーネントを持っています。サーバー側スクリプト文では，HTML フォームおよびテーブルを作成するために，アダプタフィールドの値とアダプタアクションのパラメータにアクセスします。

プロパティ

Actions: Enumerator

参照：Adapter 型の Fields プロパティ（後述），例 8（B-22 ページ）

アクションオブジェクトを列挙します。Actions プロパティを使って，アダプタのアクションをループ処理することができます。

CanModify: Boolean, read

参照：Adapter 型（後述）および AdapterField 型（B-6 ページ）の CanView プロパティ

エンドユーザーがこのアダプタのフィールドを変更する許可を得ているかどうかを示します。CanModify プロパティを使うと，エンドユーザーの権利に応じて異なる HTML を動的に生成することができます。たとえば，CanModify が True の場合には <input> 要素，CanModify が False の場合には <p> をページに含めることができます。

CanView: Boolean, read

参照：Adapter 型（前述）および AdapterField 型（B-6 ページ）の CanModify

エンドユーザーがこのアダプタのフィールドを見る許可を得ているかどうかを示します。CanModify プロパティを使うと，エンドユーザーの権利に応じて異なる HTML を動的に生成することができます。

ClassName_: text, read

参照：Name_（後述）

アダプタコンポーネントのクラス名を示します。

Errors: AdapterErrors, read

参照：AdapterErrors 型（B-6 ページ），例 7（B-22 ページ）

HTTP リクエストの処理中に検出されたエラーを列挙します。アダプタは，HTML ページの生成時またはアダプタアクションの実行時に発生したエラーを捕捉します。Errors プロパティを使うと，このエラーを列挙し，エラーメッセージを HTML ページに表示することができます。

Fields: Enumerator

参照：Actions

フィールドオブジェクトを列挙します。Fields プロパティを使って，アダプタのフィールドをループ処理することができます。

HiddenFields: AdapterHiddenFields

参照：HiddenRecordFields，AdapterHiddenFields 型（B-11 ページ），例 10（B-24 ページ），例 22（B-36 ページ）

アダプタのステート情報を渡す非表示入力フィールドを定義します。ステート情報には，たとえば TDataSetAdapter のモードがあります。このモードの値は，Edit または Insert のいずれかです。TDataSetAdapter を HTML フォームの生成に使用する場合，HiddenFields プロパティはこのモード用の非表示フィールドを定義します。HTML フォームが実行されるときに，HTTP リクエストにはこの非表示フィールドの値が含まれます。アクションの実行時に，モード値が HTTP リクエストから取り出されます。モードが Insert の場合，新しい行がデータセットに挿入されます。モードが Edit の場合，データセットの行が更新されます。

HiddenRecordFields: AdapterHiddenFields

参照：HiddenFields，AdapterHiddenFields 型（B-11 ページ），例 10（B-24 ページ），例 22（B-36 ページ）

HTML フォームの各行（レコード）に必要なステート情報を渡す非表示入力フィールドを定義します。たとえば，TDataSetAdapter を HTML フォームの生成に使用する場合，HiddenRecordFields プロパティで HTML テーブルの各行のキー値を識別する非表示フィールドを定義します。HTML

フォームが実行されるときに，HTTP リクエストにはこれらの非表示フィールドの値が含まれます。データセットの複数の行を更新するアクションの実行時に，TDataSetAdapter はこれらのキー値を使用して，更新する行を特定します。

Mode: text, read/write

参照：例 10（B-24 ページ）

アダプタのモードを設定または取得します。

一部のアダプタはモードをサポートしています。たとえば，TDataSetAdapter は，Edit，Insert，Browse，および Query モードをサポートします。モードによってアダプタの動作が異なります。TDataSetAdapter が Edit モードの場合，実行されるフォームはテーブルの行を更新します。TDataSetAdapter が Insert モードの場合，実行されるフォームはテーブルに行を挿入します。

Name_: text, read

アダプタの変数名を示します。

Records: Enumerator, read

参照： 例 9（B-23 ページ）

アダプタのレコードを列挙します。Records プロパティを使って，アダプタのレコードをループ処理して HTML テーブルを生成することができます。

AdapterAction 型

参照：Adapter 型（B-2 ページ），AdapterField 型（B-6 ページ）

AdapterAction 型は，アダプタアクションのプロパティとメソッドを定義します。

プロパティ

Array: Enumerator

参照：例 11（B-25 ページ）

アダプタアクションのコマンドを列挙します。Array プロパティを使って，コマンドをループ処理することができます。Array は，アクションが複数のコマンドをサポートしない場合は Null です。

複数のコマンドをサポートするアクションには，たとえば TAdapterGotoPageAction があります。このアクションは，親アダプタによって定義される各ページごとに 1 つのコマンドを持っています。Array プロパティは，エンドユーザーがハイパーリンクをクリックしてページにジャンプできる，ハイパーリンクのシリーズを生成するために使用されます。

AsFieldValue: text, read

参照：AsHREF，例 10 （B-24 ページ），例 21（B-35 ページ）

非表示フィールド内で実行できるテキスト値を提供します。

AsFieldValue は，アクションの名前とアクションのパラメータを識別します。この値は，__act という非表示フィールドに入ります。HTML フォームの実行時に，アダプタディスパッチャは HTTP リクエストから値を取り出し，その値を使ってアダプタアクションを見つけて呼び出します。

AsHREF: text, read

参照：AsFieldValue，例 11 （B-25 ページ）

<a> タグの href 属性値として使用できるテキスト値を提供します。

AsHREF は，アクションの名前とアクションのパラメータを識別します。この値を，リクエストを実行するアンカータグに入れて，このアクションを実行します。HTML フォーム上のアンカータグはフォームを実行しないことに注意してください。実行されるフォーム値をアクションで使用する場合は，非表示フォームフィールドと AsFieldValue を使ってアクションを識別します。

CanExecute: Boolean, read

エンドユーザーがこのアクションを実行する権利を持っているかどうかを示します。

DisplayLabel: text, read

参照：例 21（B-35 ページ）

このアダプタアクションの HTML 表示ラベルを提示します。

DisplayStyle: string, read

参照：例 21（B-35 ページ）

このアクションの HTML 表示スタイルを提示します。

サーバー側スクリプトは，HTML を生成する方法を，DisplayStyle を使って決定することができます。組み込みのアダプタは，以下の表示スタイルのいずれかを返します。

値	意味
''	未定義の表示スタイル
'Button'	<input type="submit"> として表示する
'Anchor'	<a> を使用する

Enabled: Boolean, read

参照：例 21（B-35 ページ）

このアクションを HTML ページ上で有効にする必要があるかどうかを示します。

Name: string, read

このアダプタアクションの変数名を示します。

Visible: Boolean, read

このアダプタフィールドを HTML ページ上で見えるようにする必要があるかどうかを示します。

メソッド

LinkToPage(PageSuccess, PageFail): AdapterAction, read

参照：例 10（B-24 ページ），例 11（B-25 ページ），例 21（B-35 ページ），Page オブジェクト，AdapterAction 型（B-4 ページ）

LinkToPage を使って，アクションの実行後に表示するページを指定します。最初のパラメータは，アクションの実行が成功した場合に表示するページの名前です。2 番目のパラメータは，アクションの実行中にエラーが発生した場合に表示するページの名前です。

AdapterErrors 型

参照：Adapter 型の Errors プロパティ（B-2 ページ）

AdapterErrors 型は，アダプタの Errors プロパティのプロパティを定義します。

プロパティ

Field: AdapterField, read

参照：AdapterField 型（B-6 ページ）

エラーの原因となったアダプタフィールドを識別します。

このプロパティは，エラーが特定のアダプタフィールドに関連付けられない場合は Null です。

ID: integer, read

エラーの数値識別子を提供します。

このプロパティは，ID が定義されていない場合はゼロです。

Message: text, read

参照：例 7（B-22 ページ）

エラーのテキスト記述を提供します。

AdapterField 型

参照：Adapter 型（B-2 ページ），AdapterAction 型（B-4 ページ）

AdapterField 型は，アダプタフィールドのプロパティとメソッドを定義します。

プロパティ

CanModify: Boolean, read

参照：AdapterField 型（後述）および Adapter 型（B-2 ページ）の CanView プロパティ

エンドユーザーがこのフィールドの値を変更する権利を持っているかどうかを示します。

CanView: Boolean, read

参照：AdapterField 型（前述）および Adapter 型（B-2 ページ）の CanModify プロパティ

エンドユーザーがこのフィールドの値を見る権利を持っているかどうかを示します。

DisplayLabel: text, read

このアダプタフィールドの HTML 表示ラベルを提示します。

DisplayStyle: text, read

参照：InputStyle（後述），ViewMode（後述），例 17（B-32 ページ）

DisplayStyle は，フィールドの値の読み出し専用表現を表示する方法を提示します。

サーバー側スクリプトは，HTML を生成する方法を，DisplayStyle を使って決定することができます。アダプタフィールドは，以下の表示スタイルのいずれかを返します。

値	HTML 表示スタイル
''	未定義
'Text'	<p> を使用する
'Image'	<img> を使用する。フィールドの Image プロパティは，src プロパティを定義する
'List'	<UL> を使用する。Values プロパティを列挙して，各 <LI> 項目を生成する

ViewMode プロパティは，HTML の生成に InputStyle または DisplayStyle を使用するかどうかを示します。

DisplayText: text, read

参照：EditText（後述），例 9（B-23 ページ）

アダプタフィールドの値を読み出し専用で表示するときに使用するテキストを提供します。DisplayText の値には，数値フォーマッティングを含めることができます。

DisplayWidth: integer, read

参照：MaxLength（後述）

アダプタフィールドの値を表示する幅（文字数）を提示します。

表示幅が未定義の場合は -1 が返されます。

EditText: text, read

参照：DisplayText（前述），例 10（B-24 ページ）

このアダプタフィールド用の HTML 入力を定義するときに使用するテキストを提供します。EditText の値は，通常はフォーマッティングされません。

Image: AdapterImage 型 , read

参照：AdapterImage 型（B-11 ページ），例 12（B-27 ページ）

このアダプタフィールドのイメージを定義するオブジェクトを提供します。

アダプタフィールドがイメージを提供しない場合は Null です。

InputStyle: text, read

参照：DisplayStyle（前述），ViewMode（後述），例 17（B-32 ページ）

このフィールドの HTML 入力スタイルを提示します。

サーバー側スクリプトは，HTML 要素を生成する方法を，InputStyle を使って決定することができます。アダプタフィールドは，以下の入力スタイルのいずれかを返します。

値	意味
''	未定義の入力スタイル
'TextInput'	<input type="text"> を使用する
'PasswordInput'	<input type="password"> を使用する
'Select'	<select> を使用する。ValuesList プロパティを列挙して，各 <option> 項目を生成する
'SelectMultiple'	<select multiple> を使用する。ValuesList プロパティを列挙して，各 <option> 項目を生成する
'Radio'	ValuesList プロパティを列挙して，1 つ以上の <input type="radio"> を生成する
'CheckBox'	ValuesList プロパティを列挙して，1 つ以上の <input type="checkbox"> を生成する
'TextArea'	<textarea> を使用する
'File'	<input type="file"> を使用する

ViewMode プロパティは，HTML の生成に InputStyle または DisplayStyle を使用するかどうかを示します。

InputName: text, read

参照：例 10（B-24 ページ）

このアダプタフィールドを編集する HTML 入力要素の名前を提供します。

HTML の <input>，<select>，または <textarea> 要素を生成するときに InputName を使用すると，アダプタコンポーネントは，HTTP リクエスト内の名前 / 値ペアをアダプタフィールドに関連付けることができます。

MaxLength: integer, read

参照：DisplayWidth（前述）

このフィールドに入力できる最大長（文字数）を示します。

最大長が未定義の場合は MaxLength は -1 になります。

Name: text, read

アダプタフィールドの変数名を返します。

Required: Boolean, read

フォームが実行されるときに，このアダプタフィールドに値が必要かどうかを示します。

Value: variant, read

参照：Values（後述），DisplayText（前述），EditText（前述）

計算で使用できる値を返します。たとえば，2 つのアダプタフィールド値を加算するときに Value を使います。

Values: AdapterFieldValues, read

参照：ValuesList（後述），AdapterFieldValues 型（B-9 ページ），Value（前述），例 13（B-27 ページ）

フィールドの値のリストを返します。Values プロパティは，このアダプタフィールドが複数の値をサポートしていない場合は Null です。複数値フィールドは，たとえばエンドユーザーが選択リストで複数の値を選択できるようにするために使用されます。

ValuesList: AdapterFieldValuesList, read

参照：Values（前述），AdapterFieldValuesList 型（B-10 ページ），例 13（B-27 ページ）

このアダプタフィールドの選択肢のリストを提供します。ValuesList は，HTML の選択リスト，チェックボックスグループ，またはラジオボタングループを生成するときに使用します。ValuesList 内の各項目は，それぞれ値を持ち，名前を持つこともできます。

Visible: Boolean, read

このアダプタフィールドを HTML ページ上で見えるようにする必要があるかどうかを示します。

ViewMode: text, read

参照：DisplayStyle（前述），InputStyle（前述），例 17（B-32 ページ）

このアダプタフィールド値を HTML ページ上に表示する方法を提示します。

アダプタフィールドは，以下の表示モードのいずれかを返します。

値	表示モード
''	未定義
'Input'	<input>，<textarea>，または <select> を使用して，編集可能な HTML フォーム要素を生成する
'Display'	<p>， ，または <img> を使用して，読み出し専用の HTML を生成する

ViewMode プロパティは，HTML の生成に InputStyle または DisplayStyle を使用するかどうかを示します。

メソッド

IsEqual(Value): Boolean

参照：例 16（B-31 ページ）

この関数を呼び出すと，変数をアダプタフィールドの値と比較できます。

AdapterFieldValues 型

参照：AdapterField 型の Values プロパティ（B-6 ページ）

フィールドの値のリストを提供します。複数値アダプタフィールドは，このプロパティをサポートします。複数値フィールドは，たとえばエンドユーザーが選択リストで複数の値を選択できるようにするために使用されます。

プロパティ

Records: Enumerator, read

参照：例 15（B-30 ページ）

値のリスト内のレコードを列挙します。

Value: variant, read

参照：ValueField（後述）

現在の列挙項目の値を返します。

ValueField: AdapterField, read

参照：AdapterField 型（B-6 ページ），例 15（B-30 ページ）

現在の列挙項目のアダプタフィールドを返します。ValueField は，たとえば現在の列挙項目の DisplayText を取得するために使用します。

メソッド

HasValue(Value): Boolean

参照：例 14（B-28 ページ）

指定された値が，フィールド値のリストに含まれるかどうか示します。このメソッドは，HTML 選択リストで項目を選択するか，チェックボックスのグループで項目をチェックするかを決めるために使用します。

AdapterFieldValuesList 型

参照：Adapter 型（B-2 ページ）

このアダプタフィールドに設定可能な値のリストを提供します。

ValuesList は，HTML の選択リスト，チェックボックスグループ，またはラジオボタングループを生成するときに使用します。ValuesList 内の各項目は，それぞれ値を持ち，名前を持つこともできます。

プロパティ

Image: AdapterImage, read

現在の列挙項目のイメージを返します。項目がイメージを持っていない場合は Null を返します。

Records: Enumerator, read

値のリスト内のレコードを列挙します。

Value: variant, read

現在の列挙項目の値を返します。

ValueField: AdapterField, read

参照：AdapterField 型（B-6 ページ），例 15（B-30 ページ）

現在の列挙項目のアダプタフィールドを返します。ValueField は，たとえば現在の列挙項目の DisplayText を取得するために使用します。

ValueName: text, read

現在の項目のテキスト名を返します。値が名前を持っていない場合は，ValueName は空白になります。

メソッド

ImageOfValue(Value): AdapterImage

この値に関連付けられているイメージを探します。イメージがない場合は Null を返します。

NameOfValue(Value): text

この値に関連付けられている名前を探します。値が見つからない場合，または値が名前を持っていない場合は，空白の文字列を返します。

AdapterHiddenFields 型

参照：Adapter 型の HiddenFields および HiddenRecordFields プロパティ（B-2 ページ）

アダプタが変更を実行するために使用する HTML フォーム上で必要とする非表示フィールドの名前と値へのアクセスを提供します。

プロパティ

Name: text, read

列挙されている非表示フィールドの名前を返します。

Value: text, read

列挙されている非表示フィールドの文字列値を返します。

メソッド

WriteFields(Response)

参照：例 10（B-24 ページ），例 22（B-36 ページ）

非表示フィールドの名前と値を，<input type="hidden"> を使って書き込みます。

このメソッドを呼び出して，すべての HTML 非表示フィールドを HTML フォームに書き込みます。

AdapterImage 型

参照：AdapterField 型（B-11 ページ），AdapterAction 型（B-4 ページ）

アクションまたはフィールドに関連付けられているイメージを表します。

プロパティ

AsHREF: text, read

参照：例 11（B-25 ページ），例 12（B-27 ページ）

HTML <img> 要素の定義に使用できる URL を提供します。

Module 型

参照：Modules オブジェクト（B-16 ページ）

アダプタコンポーネントは，モジュールのプロパティを使って名前で参照できます。また，モジュールを使って，モジュールのスクリプティング可能なオブジェクト（通常はアダプタ）を列挙することもできます。

プロパティ

Name_: text, read

参照：例 20（B-34 ページ）

モジュールの変数名を識別します。これは，Modules 変数のプロパティとしてモジュールにアクセスするときに使用される名前です。

ClassName_: text, read

参照：例 20（B-34 ページ）

モジュールのクラス名を識別します。

Objects: Enumerator

参照：例 20（B-34 ページ）

Objects を使って，モジュール内のスクリプティング可能なオブジェクト（通常はアダプタ）を列挙できます。

Page 型

参照：Page オブジェクト（B-16 ページ），例 20（B-34 ページ）

ページのプロパティとメソッドを定義します。

プロパティ

CanView: Boolean, read

エンドユーザーがこのページを見る権利を持っているかどうかを示します。

ページは，アクセス権を登録します。CanView は，ページによって登録されている権利を，エンドユーザーに与えられている権利と比較します。

DefaultAction: AdapterAction 型 , read

参照：例 6（B-21 ページ）

このページに関連付けられているデフォルトのアダプタアクションを識別します。

デフォルトアクションは，通常はパラメータをページに渡さなければならないときに使用されます。DefaultAction は，Null であってもかまいません。

HREF: text, read

参照：例 5（B-21 ページ）

このページへのハイパーリンクを <a> タグを使って生成するために使用できる URL を提供します。

LoginRequired: Boolean, read

エンドユーザーがこのページにアクセスする前にログインする必要があるかどうかを示します。

ページは，LoginRequired フラグを登録します。True の場合，エンドユーザーはログインしなければこのページへのアクセスを許可されません。

Name: text, read

参照：例 5（B-21 ページ）

登録されたページの名前を提供します。

ページが公開される場合，ページの名前が HTTP リクエストのパス情報のサフィックスであれば，PageDispatcher はページを生成します。

Published: Boolean, read

参照：例 5（B-21 ページ）

エンドユーザーが，ページ名を URL のサフィックスとして指定することによってこのページにアクセスできるかどうかを示します。

ページは，Published フラグを登録します。ページディスパッチャは，公開されたページを自動的にディスパッチします。通常 Published プロパティは，ページへのハイパーリンクを持つメニューの生成時に使用されます。Published が False に設定されているページは，このメニューに含まれません。

Title: text, read

参照：例 5（B-21 ページ），例 18（B-33 ページ）

ページのタイトルを提供します。

タイトルは，通常はエンドユーザーに表示されます。

グローバルオブジェクト

グローバルオブジェクトは，サーバー側スクリプトで参照することができます。ソースコード内のオブジェクト参照に似たスクリプト内のグローバルオブジェクト参照を作成できます。次に例を示します。

```
<%= Application.Title %>
```

これは Web ページ内でのアプリケーションのタイトルを表示します。

グローバルスクリプトオブジェクトを以下の表に示します。

表 B.2　WebSnap グローバルオブジェクト

オブジェクト	説明
Application オブジェクト（B-14 ページ）	アプリケーションアダプタのフィールドとアクション（Title フィールドなど）にアクセスします。
EndUser オブジェクト（B-15 ページ）	エンドユーザーアダプタのフィールドとアクション（エンドユーザーのための DisplayName，Login アクション，Logout アクションなど）にアクセスします。
Modules オブジェクト（B-16 ページ）	データモジュールまたはページモジュールを名前で参照します。Modules 変数を使って，アプリケーションのモジュールを列挙することもできます。
Page オブジェクト（B-16 ページ）	生成されるページのプロパティ（ページの Title など）にアクセスします。
Pages オブジェクト（B-16 ページ）	登録されているページを名前で参照します。Pages 変数を使って，アプリケーションの登録ページを列挙することもできます。
Producer オブジェクト（B-16 ページ）	透過タグを含めることができる HTML コンテンツを記述します。
Request オブジェクト（B-17 ページ）	HTTP リクエストのプロパティとメソッドにアクセスします。
Response オブジェクト（B-17 ページ）	HTTP レスポンスに HTML コンテンツを書き込みます。
Session オブジェクト（B-18 ページ）	エンドユーザーのセッションのプロパティにアクセスします。

Application オブジェクト

参照：Adapter 型（B-2 ページ）

Application オブジェクトは，アプリケーションに関する情報へのアクセスを提供します。

Application オブジェクトを使って，アプリケーションアダプタのフィールドとアクション（Title フィールドなど）にアクセスします。Application オブジェクトは Adapter なので，フィールドやアクションを追加してカスタマイズすることができます。アプリケーションアダプタに追加されたフィールドとアクションには，Application オブジェクトのプロパティの名前でアクセスできます。

プロパティ

Designing: Boolean, read

参照：例 1（B-19 ページ）

この Web アプリケーションが IDE 内で設計中であるかどうかを示します。

Designing プロパティを使うと，設計モードのときと Web アプリケーションの実行中とで異なる HTML を条件によって生成することができます。

ModulePath: text, read

参照：QualifyFileName メソッド

Web アプリケーションの実行形式の格納場所を示します。

ModulePath を使って，実行形式と同じディレクトリにあるファイルの名前を作成できます。

ModuleFileName: text, read

参照：QualifyFileName メソッド

実行形式の完全に限定されたファイル名を示します。

Title: text, read

参照：例 18（B-33 ページ）

アプリケーションのタイトルを示します。

Title プロパティは，TApplicationAdapter の Title プロパティの値を持ちます。通常，この値は HTML ページの上部に表示されます。

メソッド

QualifyFileName(FileName): text

参照：例 1（B-19 ページ）

相対のファイル名またはディレクトリ参照を絶対参照に変換します。

QualifyFileName は，Web アプリケーション実行形式のディレクトリ位置を使って，完全に限定されていないファイル名を限定します。このメソッドは，完全に限定されたファイル名を返します。FileName パラメータが完全に限定されている場合は，そのファイル名がそのまま返されます。設計モードでは，FileName パラメータはプロジェクトファイルのディレクトリ位置で限定されます。

EndUser オブジェクト

参照：Adapter 型（B-2 ページ）

現在のエンドユーザーに関する情報へのアクセスを提供します。

EndUser を使用すると，エンドユーザーアダプタのフィールドとアクション（エンドユーザーの DisplayName など）にアクセスできます。エンドユーザーアダプタに追加されたフィールドとアクションには，EndUser オブジェクトのプロパティの名前でアクセスできます。

プロパティ

DisplayName: text, read

参照：例 19（B-33 ページ）

エンドユーザーの名前を示します。

LoggedIn: Boolean, read

参照：例 19（B-33 ページ）

そのエンドユーザーがログインしているかどうかを示します。

LoginFormAction: AdapterAction 型 , read

参照：例 19（B-33 ページ），AdapterAction 型（B-4 ページ）

ユーザーのログインに使用するアダプタアクションを提供します。

LogoutAction: AdapterAction 型 , read

参照：例 19（B-33 ページ），AdapterAction 型（B-4 ページ）

ユーザーのログアウトに使用するアダプタアクションを提供します。

Modules オブジェクト

参照：例 2（B-20 ページ），例 20（B-34 ページ）

Modules オブジェクトは，現在の HTTP リクエストのサービスを行うためにインスタンス化されてアクティブ化されているすべてのモジュールへアクセスを提供します。

特定のモジュールを参照するには，モジュールの名前を Modules 変数のプロパティとして使用します。アプリケーション内のすべてのモジュールを列挙するには，Modules オブジェクトを使って列挙子を作成します。

Page オブジェクト

参照：例 5（B-21 ページ），Page 型（B-12 ページ）

Page オブジェクトは，生成中のページのプロパティへのアクセスを提供します。

Page オブジェクトのプロパティとメソッドについての詳細は，Page 型を参照してください。

Pages オブジェクト

参照：例 5（B-21 ページ）

Pages オブジェクトは，アプリケーションによって登録されているすべてのページへのアクセスを提供します。

特定のページを参照するには，ページの名前を Pages 変数のプロパティとして使用します。アプリケーション内のすべてのページを列挙するには，Pages オブジェクトを使って列挙子を作成します。

Producer オブジェクト

参照：Response オブジェクト（B-17 ページ）

Producer オブジェクトを使って，透過タグを含むテキストを記述します。タグは，ページプロデューサによって変換され，HTTP レスポンスに書き込まれます。テキストに透過タグが含まれていない場合は，Response オブジェクトを使った方がパフォーマンスが高くなります。

プロパティ

Content: text, read/write

HTTP レスポンスのコンテンツ部分へのアクセスを提供します。

Content を使うと，HTTP レスポンスのコンテンツ部分全体の読み出しまたは書き込みができます。Content を設定すると，透過タグが翻訳されます。透過タグを使用しない場合は，Response.Content を使った方がパフォーマンスが高くなります。

メソッド

Write(Value)

HTTP リクエストのコンテンツ部分に，透過タグのサポートを追加します。

Write メソッドを使うと，HTTP リクエストのコンテンツ部分にコンテンツを追加できます。Write は，次のような透過タグを変換します

```
Write('Translate this: <#MyTag>')
```

透過タグを使用しない場合は，Response.Write を使った方がパフォーマンスが高くなります。

Request オブジェクト

HTTP リクエストへのアクセスを提供します。

Request オブジェクトのプロパティを使って，HTTP リクエストに関する情報にアクセスします。

プロパティ

Host: text, read

HTTP リクエストの Host ヘッダーの値を返します。

Host は，TWebRequest の Host プロパティと同じです。

PathInfo: text, read

URL の PathInfo 部分を返します。

PathInfo は，TWebRequest の InternalPathInfo プロパティと同じです。

ScriptName: text, read

URL のスクリプト名部分を返します。スクリプト名部分は，Web サーバーアプリケーションの名前を指定します。

ScriptName は，TWebRequest の InternalScriptName プロパティと同じです。

Response オブジェクト

参照：Producer オブジェクト (B-16 ページ)

HTTP レスポンスへのアクセスを提供します。Response オブジェクトを使って，HTTP レスポンスのコンテンツ部分に書き込みを行います。透過タグを使用する場合は，Response オブジェクトではなく Producer オブジェクトを使用してください。

プロパティ

Content: text, read/write

HTTP レスポンスのコンテンツ部分へのアクセスを提供します。

Content を使うと，HTTP レスポンスのコンテンツ部分全体の読み出しまたは書き込みができます。

メソッド

Write(Value)

参照：例 5（B-21 ページ）

HTTP リクエストのコンテンツ部分に Value を追加します。

Write メソッドを使うと，HTTP リクエストのコンテンツに Value を追加できます。Write は，透過タグを変換しません。

1 つまたは複数の透過タグを含む文字列を書き込むには，Producer オブジェクトの Write メソッドを使用します。

Session オブジェクト

Session オブジェクトは，セッションの ID と値へのアクセスを提供します。

セッションは，短時間のあいだ必要なエンドユーザーについての情報をトラッキングするために使用されます。

プロパティ

SessionID.Value: text, read/write

現在のエンドユーザーのセッションの ID へのアクセスを提供します。

Values(Name): variant, read

現在のエンドユーザーのセッションに格納されている値へのアクセスを提供します。

JScript の例

以下の JScript の例は，サーバー側スクリプティングのプロパティとメソッドの使い方を示しています。

表 B.3　サーバー側スクリプティングの JScript の例

例	説明
例 1（B-19 ページ）	Application オブジェクトの QualifyFilename メソッドを使用して，イメージへの相対パス参照を生成します。
例 2（B-20 ページ）	モジュールを参照する変数を宣言します。
例 3（B-20 ページ）	Web アプリケーション内のモジュールを列挙し，それらの名前を HTML テーブルに表示します。
例 4（B-20 ページ）	登録されたページを参照する変数を宣言します。

表 B.3　サーバー側スクリプティングの JScript の例（つづき）

例	説明
例 5（B-21 ページ）	登録されたページを列挙して，公開されているページへのハイパーリンクのメニューを生成します。
例 6（B-21 ページ）	登録されたページを列挙して，公開されているページのデフォルトアクションへのハイパーリンクのメニューを生成します。
例 7（B-22 ページ）	アダプタによって検出されたエラーのリストを書き出します。
例 8（B-22 ページ）	アダプタのすべてのアクションオブジェクトを列挙して，アクションオブジェクトのプロパティ値を HTML テーブルに表示します。
例 9（B-23 ページ）	アダプタのレコードを列挙して，アダプタフィールドの値を HTML テーブルに表示します。
例 10（B-24 ページ）	HTML フォームを生成し，アダプタフィールドを編集して，アダプタアクションを実行します。
例 11（B-25 ページ）	アダプタアクションを表示してページングをサポートします。
例 12（B-27 ページ）	アダプタフィールドのイメージを <img> タグを使って表示します。
例 13（B-27 ページ）	アダプタフィールドを，<select> タグと <option> タグを使って表示します。
例 14（B-28 ページ）	アダプタフィールドをチェックボックスのグループとして表示します。
例 15（B-30 ページ）	アダプタフィールドの値を， タグと タグを使って表示します。
例 16（B-31 ページ）	アダプタフィールドをラジオボタンのグループとして表示します。
例 17（B-32 ページ）	アダプタフィールドの DisplayStyle，InputStyle，および ViewMode プロパティを使って HTML を生成します。
例 18（B-33 ページ）	Application オブジェクトと Page オブジェクトのプロパティを使ってページヘッダーを生成します。
例 19（B-33 ページ）	EndUser オブジェクトのプロパティを使って，エンドユーザー名，ログインコマンド，およびログアウトコマンドを表示します。
例 20（B-34 ページ）	モジュール内のスクリプティング可能なオブジェクトをリストします。
例 21（B-35 ページ）	アダプタアクションの DisplayStyle プロパティを使って HTML を生成します。
例 22（B-36 ページ）	HTML テーブルを生成して，複数の詳細レコードを更新します。

例 1

参照：Application オブジェクトの Designing および QualifyFileName プロパティ（B-14 ページ），Request オブジェクトの PathInfo プロパティ（B-17 ページ）

この例は，イメージへの相対パス参照を生成しています。スクリプトが設計モードの場合は実際のディレクトリを参照し，そうでない場合は仮想ディレクトリを参照します。

```
<%
     function PathInfoToRelativePath(S)
     {
       var R = '';
       var L = S.length
       I = 0
       while (I < L)
       {
         if (S.charAt(I) == '/')
           R = R + '../'
         I++
       }
```

```
        return R
      }

      function QualifyImage(S)
      {
        if (Application.Designing)
          return Application.QualifyFileName("..\\images\\" + S);   // 相対ディレクトリ
        else
          return PathInfoToRelativePath(Request.PathInfo) + '../images/' + S;
                                                                    // 仮想ディレクトリ
      }
    %>
```

例 2

参照：Modules オブジェクト（B-16 ページ）

この例は，WebModule1 を参照する変数を宣言しています。

```
<% var M  = Modules.WebModule1 %>
```

例 3

参照：Modules オブジェクト（B-16 ページ）

例 3 は，インスタンス化されたモジュールを列挙して，その変数名とクラス名をテーブルに表示します。

```
<table border=1>
<tr><th>Name</th><th>ClassName</th></tr>
<%
   var e = new Enumerator(Modules)
   for (; !e.atEnd(); e.moveNext())
   {
%>
  <tr><td><%=e.item().Name_%></td><td><%=e.item()ClassName._%></td></tr>
<%
   }
%>
</table>
```

例 4

参照：Pages オブジェクト（B-16 ページ），Page オブジェクトの Title プロパティ（B-16 ページ）

この例は，Home というページを参照する変数を宣言しています。また，Home のタイトルを表示します。

```
<% var P = Pages.Home %>
<p><%= P.Title %></p>
```

例 5

参照：Pages オブジェクト（B-16 ページ），Page オブジェクトの Published および HREF プロパティ（B-16 ページ），Response オブジェクトの Write メソッド（B-17 ページ）

この例は，登録されたページを列挙して，公開されているすべてのページへのハイパーリンクのメニューを作成しています。

```
<table>
<td>
<%   e = new Enumerator(Pages)
     s = ''
     c = 0
     for (; !e.atEnd(); e.moveNext())
     {
       if (e.item().Published)
       {
         if (c>0) s += ' | '
         if (Page.Name != e.item().Name)
           s += '<a href="' + e.item().HREF + '">' + e.item().Title + '</a>'
         else
           s += e.item().Title
         c++
       }
     }
     if (c>1) Response.Write(s)
%>
</td>
</table>
```

例 6

参照：Page 型の DefaultAction プロパティ（B-12 ページ）

この例は，登録されたページを列挙して，デフォルトアクションへのハイパーリンクを表示するメニューを作成しています。

```
<table>
<td>
<%  e = new Enumerator(Pages)
     s = ''
     c = 0
     for (; !e.atEnd(); e.moveNext())
     {
       if (e.item().Published)
       {
         if (c>0) s += ' | '
         if (Page.Name != e.item().Name)
           if (e.item().DefaultAction != null)
             s += '<a href="'+e.item().DefaultAction.AsHREF+'"> +e.item().Title+'</a>'
           else
             s += '<a href="' + e.item().HREF + '">' + e.item().Title + '</a>'
         else
           s += e.item().Title
         c++
```

```
        }
      }
      if (c>1) Response.Write(s)
%>
</td>
</table>
```

例 7

参照：Adapter 型の Errors プロパティ（B-2 ページ），AdapterErrors 型（B-6 ページ），Modules オブジェクト（B-16 ページ），Response オブジェクトの Write メソッド（B-17 ページ）

この例は，アダプタによって検出されたエラーのリストを書き出しています。

```
<% {
   var e = new Enumerator(Modules.CountryTable.Adapter.Errors)
   for (; !e.atEnd(); e.moveNext())
   {
     Response.Write("<li>" + e.item().Message)
   }
   e.moveFirst()
   } %>
```

例 8

参照：Adapter 型の Actions プロパティ（B-2 ページ），AdapterAction 型（B-4 ページ）

この例は，アダプタのすべてのアクションを列挙して，アクションのプロパティ値をテーブルに表示しています。

```
<% // 1 つのアクションのいくつかのプロパティをテーブルに表示する
   function DumpAction(A)
   {
%>
      <table border="1">
         <tr><th COLSPAN=2><%=A.Name%></th>
         <tr><th>AsFieldValue:</th><td><%= A.AsFieldValue %></td>
         <tr><th>AsHREF:</th><td><%= A.AsHREF %></span>
         <tr><th>DisplayLabel:</th><td><%= A.DisplayLabel %></td>
         <tr><th>Enabled:</th><td><%= A.Enabled %></td>
         <tr><th>CanExecute:</th><td><span class="value"><%= A.CanExecute %></td>
    </table>
<%
  }
%>

<% // アダプタ内のあらゆるアクションに対して DumpAction 関数を呼び出す
   function DumpActions(A)
   {
     var e = new Enumerator(A)
     for (; !e.atEnd(); e.moveNext())
     {
       DumpAction(e.item())
     }
   }
```

```
%>


<%
// Adapter1 という名前のアダプタ内のアクションのプロパティを表示する
DumpActions(Adapter1.Actions) %>
```

例 9

参照：Adapter 型の Records プロパティ（B-2 ページ），AdapterField 型の DisplayText プロパティ（B-6 ページ）

この例は，アダプタのレコードを列挙して，HTML テーブルを生成しています。

```
<%
// アダプタとフィールドの変数を定義する

vAdapter=Modules.CountryTable.Adapter
vAdapter_Name=vAdapter.Name
vAdapter_Capital=vAdapter.Capital
vAdapter_Continent=vAdapter.Continent
%>

<%
// すべてのセルが境界を持つように列テキストを記述する関数
function WriteColText(t)
{
   Response.Write((t!="")?t:" ")
}
%>

<table border="1">
  <tr>
  <th>Name</th>
  <th>Capital</th>
  <th>Continent</th>
<%
   // アダプタ内のすべてのレコードを列挙し，フィールド値を記述する

   var e = new Enumerator(vAdapter.Records)
   for (; !e.atEnd(); e.moveNext())
   { %>
      <tr>
      <td><div><% WriteColText(vAdapter_Name.DisplayText) %></div></td>
      <td><div><% WriteColText(vAdapter_Capital.DisplayText) %></div></td>
      <td><div><% WriteColText(vAdapter_Continent.DisplayText) %></div></td>
      </tr>
<%
   }
%>
</table>
```

例 10

参照：AdapterAction 型の LinkToPage および AsFieldValue プロパティ（B-4 ページ），AdapterField 型の InputName および DisplayText プロパティ（B-6 ページ），Adapter 型の HiddenFields および HiddenRecordFields プロパティ（B-2 ページ）

この例は，HTML フォームを生成し，アダプタフィールドを編集して，アダプタアクションを実行しています。

```
<%
// アダプタ，フィールド，およびアクションの変数を定義する

vAdapter=Modules.CountryTable.Adapter
vAdapter_Name=vAdapter.Name
vAdapter_Capital=vAdapter.Capital
vAdapter_Continent=vAdapter.Continent
vAdapter_Apply=vAdapter.Apply
vAdapter_RefreshRow=vAdapter.RefreshRow

// アダプタのモードが設定されていなければ Edit モードにする。モードが設定されている場合は
// アダプタアクションによってモードが設定されたと思われる。たとえば，行を挿入するアクションならば，
// アダプタを Insert モードに設定する

if (vAdapter.Mode=="")
  vAdapter.Mode="Edit"
%>
<form name="AdapterForm1" method="post">

  <!-- この非表示フィールドは，フォームが送信されるときに実行されるアクションを定義する -->

  <input type="hidden" name="__act">


<%
   // アダプタによって定義される非表示フィールドを記述する

   if (vAdapter.HiddenFields != null)
   {
     vAdapter.HiddenFields.WriteFields(Response)
   } %>
<% if (vAdapter.HiddenRecordFields != null)
   {
     vAdapter.HiddenRecordFields.WriteFields(Response)
   } %>
  <table>
    <tr>
      <td>
        <table>
          <tr>
          <!-- アダプタのフィールドを編集する入力フィールドを記述する -->

          <td>Name</td>
          <td ><input type="text" size="24" name="<%=vAdapter_Name.InputName%>" value="
             <%= vAdapter_Name.EditText %>" ></td>
          </tr>
```

```
          <tr>
          <td>Capital</td>
          <td ><input type="text" size="24" name="<%=vAdapter_Capital.InputName%>"
             value="<%= vAdapter_Capital.EditText %>" ></td>
          </tr>
          <tr>
          <td>Continent</td>
          <td ><input type="text" size="24" name="<%=vAdapter_Continent.InputName%>"
             value="<%= vAdapter_Continent.EditText %>" ></td>
          </tr>
        </table>
      </td>
    </tr>
    <tr>
      <td>
        <table>
          <!-- アクションを実行する送信ボタンを記述する。アクションの実行後にこのページが
               再生成されるように LinkToPage を使用する -->

          <tr>
          <td><input type="submit" value="Apply"
            onclick = "AdapterForm1.__act.value='
              <%=vAdapter_Apply.LinkToPage(Page.Name).AsFieldValue%>'"></td>
          <td><input type="submit" value="Refresh"
            onclick = "AdapterForm1.__act.value='
              <%=vAdapter_RefreshRow.LinkToPage(Page.Name).AsFieldValue%>'"> </td>
          </tr>
        </table>
      </td>
    </tr>
  </table>
</form>
```

例 11

参照：AdapterAction 型の Array および AsHREF プロパティ（B-4 ページ）

この例は，アダプタアクションを表示してページングをサポートしています。PrevPage，GotoPage，および NextPage アクションがハイパーリンクとして表示されます。GotoPage アクションには，コマンドの配列があります。そのコマンドを列挙して，各ページにジャンプするハイパーリンクを生成します。

```
<%
   // アダプタとアクションの変数を定義する

   vAdapter = Modules.WebDataModule1.QueryAdapter
   vPrevPage = vAdapter.PrevPage
   vGotoPage = vAdapter.GotoPage
   vNextPage = vAdapter.NextPage
%>

<!-- ページ間のハイパーリンクを表示するテーブルを生成する -->

<table cellpadding="5">
<tr>
```

```
<td>
<%
   // Prevpage は "<<" を表示する。このコマンドが有効な場合にのみアンカータグを使用する

   if (vPrevPage.Enabled)
   { %>
     <a href="<%=vPrevPage.LinkToPage(Page.Name).AsHREF%>"><<<</a>
<%
   }
   else
   {%>
     <a><<<</a>
 <%} %>
<%
   // GotoPage はコマンドのリストを持つ。このリストをループ処理する。
   // このコマンドが有効な場合にのみアンカータグを使用する

   if (vGotoPage.Array != null)
   {
     var e = new Enumerator(vGotoPage.Array)
     for (; !e.atEnd(); e.moveNext())
     {
%>
       <td>
<%     if (vGotoPage.Enabled)
       { %>
          <a href="<%=vGotoPage.LinkToPage(Page.Name).AsHREF%>">
            <%=vGotoPage.DisplayLabel%></a>
<%     }
       else
       { %>
          <a><%=vGotoPage.DisplayLabel%></a>
<%     }
%>
       </td>
<%
     }
   }
%>
<td>
<%
   // NextPage は ">>" を表示する。このコマンドが有効な場合にのみアンカータグを使用する

   if (vNextPage.Enabled)
   { %>
     <a href="<%=vNextPage.LinkToPage(Page.Name).AsHREF%>">>>></a>
<%
   }
   else
   {%>
     <a>>>></a>
 <%} %>
</td>
</table>
```

例 12

参照：AdapterField 型の Image プロパティ（B-6 ページ）

例 12 は，アダプタフィールドのイメージを表示しています。

```
<%
// アダプタとフィールドの変数を宣言する

vAdapter=Modules.WebDataModule3.DataSetAdapter1
vGraphic=vAdapter.Graphic
%>

<!-- アダプタフィールドをイメージとして表示する -->
<img src="<%=(vGraphic.Image!=null) ? vGraphic.Image.AsHREF : ''%>"
alt="<%=vGraphic.DisplayText%>">
```

例 13

参照：AdapterField 型の Values および ValuesList プロパティ（B-6 ページ）

この例は，アダプタフィールドを，HTML の <select> 要素と <option> 要素を使って表示しています。

```
<%
// アダプタフィールドの HTML 選択オプションを定義するオブジェクトを返す
// 返されたオブジェクトには以下の要素がある
//
// text - <option> 要素を含む文字列
// count - <option> 要素の数
// multiple - 'multiple' か '' のどちらかを含む文字列
//            この値は <select> 要素の属性として使用する
//
// 次のように使用する
//   obj=SelOptions(f)
//   Response.Write('<select size="' + obj.count + '" name="' + f.InputName + '" ' +
//     obj.multiple + '>' + obj.text + '</select>')

function SelOptions(f)
{
   var s=''
   var v=''
   var n=''
   var c=0
   if (f.ValuesList != null)
   {
     var e = new Enumerator(f.ValuesList.Records)
     for (; !e.atEnd(); e.moveNext())
     {
       s+= '<option'
       v = f.ValuesList.Value;
       var selected
       if (f.Values == null)
          selected = f.IsEqual(v)
       else
          selected = f.Values.HasValue(v)
       if (selected)
```

```
            s += ' selected'
          n = f.ValuesList.ValueName;
          if (n=='')
          {
            n = v
            v = ''
          }
          if (v!='') s += ' value="' + v + '"'
          s += '>' + n + '</option>\r\n'
          c++
        }
        e.moveFirst()
      }
      r = new Object;
      r.text = s
      r.count = c
      r.multiple = (f.Values == null) ?'' : 'multiple'
      return r;
   }
%>

<%
   // アダプタフィールドの HTML 選択オプションを生成する
   function WriteSelectOptions(f)
   {
      obj=SelOptions(f)
%>
      <select size="<%=obj.count%>" name="<%=f.InputName%>" <%=obj.multiple%> >
        <%=obj.text%>
      </select>
<%
   }
%>
```

例 14

参照：AdapterField 型の Values および ValuesList プロパティ（B-6 ページ）

この例は，アダプタフィールドを <input type="checkbox"> 要素のグループとして表示しています。

```
<%
// アダプタフィールドの HTML チェックボックスを定義するオブジェクトを返す
// 返されたオブジェクトには以下の要素がある
//
// text - <input type=checkbox> 要素を含む文字列
// count - <option> 要素の数
//
// 3 つの列を持ち，追加の属性がないチェックボックスグループを定義するには，
// 次のように使用する
//   obj=CheckBoxGroup(f, 3, '')
//   Response.Write(obj.text)
//
function CheckBoxGroup(f,cols,attr)
{
   var s=''
   var v=''
```

```
  var n=''
  var c=0;
  var nm=f.InputName
  if (f.ValuesList == null)
  {
    s+= '<input type="checkbox"'
    if (f.IsEqual(true)) s+= ' checked'
    s += ' value="true"' + ' name="' + nm + '"'
    if (attr!='') s+= ' ' + attr
    s += '></input>\r\n'
    c = 1
  }
  else
  {
    s += '<table><tr>'
    var e = new Enumerator(f.ValuesList.Records)
    for (; !e.atEnd(); e.moveNext())
    {
      if (c % cols == 0 && c != 0) s += '</tr><tr>'
      s+= '<td><input type="checkbox"'
      v = f.ValuesList.Value;
      var checked
      if (f.Values == null)
        checked = (f.IsEqual(v))
      else
        checked = f.Values.HasValue(v)
      if (checked)
        s+= ' checked'
      n = f.ValuesList.ValueName;
      if (n=='')
        n = v
      s += ' value="' + v + '"' + ' name="' + nm + '"'
      if (attr!='') s+= ' ' + attr
      s += '>' + n + '</input></td>\r\n'
      c++
    }
    e.moveFirst()
    s += '</tr></table>'
  }
  r = new Object;
  r.text = s
  r.count = c
  return r;
 }
%>

<%
  // アダプタフィールドをチェックボックスのグループとして表示する
  function WriteCheckBoxGroup(f, cols, attr)
  {
     obj=CheckBoxGroup(f, cols, attr)
     Response.Write(obj.text);
  }
%>
```

例 15

参照：AdapterField 型の Values および ValuesList プロパティ（B-6 ページ），AdapterFieldValues 型（B-9 ページ）

この例は，アダプタフィールドの値を， 要素と <li> 要素を使って読み出し専用値のリストとして表示しています。

```
<%
// アダプタフィールドの HTML リスト値を定義するオブジェクトを返す
// 返されたオブジェクトには以下の要素がある
//
// text - <li> 要素を含む文字列
// count - 要素の数
//
// アダプタフィールドが複数の値をサポートしない場合は
// text が空白，count はゼロになる
//
// このアダプタフィールド値の読み出し専用リストの表示を定義するには
// 次のように使用する
//   obj=ListValues(f)
//   Response.Write('<ul>' + obj.text + '</ul>')
//
function ListValues(f)
{
   var s=''
   var v=''
   var n=''
   var c=0;
   r = new Object;
   if (f.Values != null)
   {
     var e = new Enumerator(f.Values.Records)
     for (; !e.atEnd(); e.moveNext())
     {
       s+= '<li>'
       s += f.Values.ValueField.DisplayText;
       s += '</li>'
       c++
     }
     e.moveFirst()
   }
   r.text = s
   r.count = c
   return r;
}
%>

<%
// アダプタフィールドを読み出し専用の値のリストとして記述する
function WriteListValues(f)
{
   obj=ListValues(f)
%>
   <ul><%=obj.text%></ul>
<%
```

```
}
%>
```

例 16

参照：AdapterFieldValuesList 型（B-10 ページ），AdapterField 型の Values および ValuesList プロパティ（B-6 ページ）

この例は，アダプタフィールドを <input type="radio"> 要素のグループとして表示しています。

```
<%
// アダプタフィールドの HTML ラジオボタンを定義するオブジェクトを返す
// 返されたオブジェクトには以下の要素がある
//
// text - <input type=radio> 要素を含む文字列
// count - 要素の数
//
// 3 つの列を持ち，追加の属性がないラジオボタングループを定義するには，
// 次のように使用する
//   obj=RadioGroup(f, 3, '')
//   Response.Write(obj.text)
//

function RadioGroup(f,cols,attr)
{
   var s=''
   var v=''
   var n=''
   var c=0;
   var nm=f.InputName
   if (f.ValuesList == null)
   {
     s+= '<input type="radio"'
     if (f.IsEqual(true)) s+= ' checked'
     s += ' value="true"' + ' name="' + nm + '"'
     if (attr!='') s+= ' ' + attr
     s += '></input>\r\n'
     c = 1
   }
   else
   {
     s += '<table><tr>'
     var e = new Enumerator(f.ValuesList.Records)
     for (; !e.atEnd(); e.moveNext())
     {
       if (c % cols == 0 && c != 0) s += '</tr><tr>'
       s+= '<td><input type="radio"'
       v = f.ValuesList.Value;
       var checked
       if (f.Values == null)
         checked = (f.IsEqual(v))
       else
         checked = f.Values.HasValue(v)
       if (checked)
         s+= ' checked'
       n = f.ValuesList.ValueName;
```

```
        if (n=='')
        {
        n = v
        }
        s += ' value="' + v + '"' + ' name="' + nm + '"'
        if (attr!='') s+= ' ' + attr
        s += '>' + n + '</input></td>\r\n'
        c++
      }
      e.moveFirst()
      s += '</tr></table>'
    }
    r = new Object;
    r.text = s
    r.count = c
    return r;
  }
%>

<%
   // アダプタフィールドをラジオボタンのグループとして表示する
   function WriteRadioGroup(f, cols, attr)
   {
      obj=RadioGroup(f, cols, attr)
      Response.Write(obj.text);
   }
%>
```

例 17

参照：AdapterField 型の DisplayStyle，ViewMode，および InputStyle プロパティ（B-6 ページ）

この例は，アダプタフィールドの InputStyle，DisplayStyle，および ViewMode プロパティの値に従って，アダプタフィールドの HTML を生成しています。

```
<%
   function WriteField(f)
   {
     Mode = f.ViewMode
     if (Mode == 'Input')
     {
       Style = f.InputStyle
       if (Style == 'SelectMultiple' || Style == 'Select')
          WriteSelectOptions(f)
       else if (Style == 'CheckBox')
         WriteCheckBoxGroup(f, 2, '')
       else if (Style == 'Radio')
         WriteRadioGroup(f, 2, '')
       else if (Style == 'TextArea')
       {
%>
         <textarea wrap=OFF name="<%=f.InputName%>"><%= f.EditText %></textarea>
<%
       }
       else if (Style == 'PasswordInput')
       {
```

```
%>
          <input type="password" name="<%=f.InputName%>"/>
<%
        }
        else if (Style == 'File')
        {
%>
           <input type="file" name="<%=f.InputName%>"/>
<%
        }
        else
        {
%>
          <input type="input" name="<%=f.InputName%>" value="<%= f.EditText %>"/>
<%
        }
      }
      else
      {
        Style = f.DisplayStyle
        if (Style == 'List')
          WriteListValues(f)
        else if (Style == 'Image')
        {
%>
          <img src="<%=(f.Image!=null) ? f.Image.AsHREF : ''%>" alt="<%=f.DisplayText%>">
<%      }
        else
          Response.Write('<p>' + f.DisplayText + '</p>')
      }
    }
%>
```

例 18

参照：Page オブジェクト（B-16 ページ），Application オブジェクトの Title プロパティ（B-14 ページ）

この例は，Application オブジェクトと Page オブジェクトのプロパティを使ってページヘッダーを生成しています。

```
<html>
<head>
<title>
<%= Page.Title %>
</title>
</head>
<body>
<h1><%= Application.Title %></h1>

<h2><%= Page.Title %></h2>
```

例 19

参照：EndUser オブジェクト（B-15 ページ）

この例は，EndUser オブジェクトのプロパティを使って，エンドユーザー名，ログインコマンド，およびログアウトコマンドを表示しています。

```
<% if (EndUser.Logout != null)
   {
     if (EndUser.DisplayName != '')
     {
%>
       <h1>Welcome <%=EndUser.DisplayName %></h1>
<%   }
     if (EndUser.Logout.Enabled) {
%>
       <a href="<%=EndUser.Logout.AsHREF%>">Logout</a>
<%   }
     if (EndUser.LoginForm.Enabled) {
%>
       <a href="<%=EndUser.LoginForm.AsHREF%>">Login</a>
<%   }
   }
%>
```

例 20

参照：Module 型（B-12 ページ）

この例は，モジュール内のスクリプティング可能なオブジェクトをリストしています。

```
<%
   // モジュール内のすべてのスクリプティング可能なオブジェクトの名前とクラス名を
   // リストする HTML テーブルを記述する
   function ListModuleObjects(m)
   {
%>
     <p></p>
     <table border="1">
     <tr>
     <th colspan="2"><%=m.Name_ + ': ' + m.ClassName_%></th>
     </tr>
<%
     var e = new Enumerator(m.Objects)
     for (; !e.atEnd(); e.moveNext())
     {
%>
     <tr>
     <td>
        <%= e.item().Name_ + ': ' + e.item().ClassName_ %>
     </td>
     </tr>
<%
     }
%>
     </table>
<%
   }
%>
```

例 21

参照：AdapterAction 型の DisplayStyle および Enabled プロパティ（B-4 ページ）

この例は，アダプタアクションの DisplayStyle プロパティに基づいて，アダプタアクションの HTML を生成しています。

```
<%
   // DisplayStyle プロパティを使用してアダプタアクションの HTML を記述する
   //
   // a - アクション
   // cap - キャプション。空白の場合は，そのアクションの表示ラベルが使用される
   // fm - HTML フォームの名前
   // p - アクションの実行後に表示するページの名前。空白の場合は，現在のページが使用される
   //
   // この関数はアクションの Array プロパティを使用していない。このアクションは
   // コマンドが 1 つであるとみなされている
   //
   function WriteAction(a, cap, fm, p)
   {
       if (cap == '')
         cap = a.DisplayLabel
       if (p == '')
         p = Page.Name
       Style = a.DisplayStyle
       if (Style == 'Anchor')
       {

          if (a.Enabled)
          {
            // href プロパティは使用しないこと。そのかわりに，HTML フォームフィールドが
            // HTTP リクエストの一部になるようにフォームを実行する
%>
            <a href=""onclick="<%=fm%>.__act.value='
               <%=a.LinkToPage(p).AsFieldValue%>';<%=fm%>.submit();return false;">
               <%=cap%></a>
<%
          }
          else
          {
%>
            <a><%=cap%></a>
<%
          }
       }
       else
       {
%>
          <input type="submit" value="<%= cap%>"
onclick="<%=fm%>.__act.value='<%=a.LinkToPage(p).AsFieldValue%>'">
<%
       }
   }
%>
```

例 22

参照：Adapter 型の HiddenFields，HiddenRecordFields，および Mode プロパティ（B-2 ページ）

この例は，HTML テーブルを生成して，複数の詳細レコードを更新しています。

```
<%
vItemsAdapter=Modules.DM.ItemsAdapter
vOrdersAdapter=Modules.DM.OrdersAdapter
vOrderNo=vOrdersAdapter.OrderNo
vCustNo=vOrdersAdapter.CustNo
vPrevRow=vOrdersAdapter.PrevRow
vNextRow=vOrdersAdapter.NextRow
vRefreshRow=vOrdersAdapter.RefreshRow
vApply=vOrdersAdapter.Apply
vItemNo=vItemsAdapter.ItemNo
vPartNo=vItemsAdapter.PartNo
vDiscount=vItemsAdapter.Discount
vQty=vItemsAdapter.Qty
%>

<!-- 2 つのアダプタを使用して複数の詳細レコードを更新する
     orders アダプタはマスターデータセットに関連付けられる
     items アダプタは詳細データセットに関連付けられる
     グリッド内の各行は items アダプタからの値を表示する
     1 つの列は Qty を編集するための <input> 要素を表示する
     適用ボタンは各詳細レコード内の Qty 値を更新する -->

<!-- 注文番号と顧客番号の値を表示する -->
<h2>OrderNo: <%= vOrderNo.DisplayText %></h2>
<h2>CustNo: <% vCustNo.DisplayText %></h2>

<%
   // このフォームは Qty フィールドを更新するので
   // items アダプタを Edit モードにする
   vItemsAdapter.Mode = 'Edit'
%>

<form name="AdapterForm1" method="post">

  <!-- 実行されたアクション名とパラメータ用の非表示フィールドを定義する -->
  <input type="hidden" name="__act">

<%
   // orders アダプタと items アダプタのステート情報
   // が入る非表示フィールドを記述する
   if (vOrdersAdapter.HiddenFields != null)
     vOrdersAdapter.HiddenFields.WriteFields(Response)
   if (vItemsAdapter.HiddenFields != null)
     vItemsAdapter.HiddenFields.WriteFields(Response)

   // orders アダプタの現在のレコードのステート情報
   // が入る非表示フィールドを記述する
   if (vOrdersAdapter.HiddenRecordFields != null)
     vOrdersAdapter.HiddenRecordFields.WriteFields(Response)%>

  <table border="1">
```

```
    <tr>
    <th>ItemNo</th>
    <th>PartNo</th>
    <th>Discount</th>
    <th>Qty</th>
  </tr>
<%
   var e = new Enumerator(vItemsAdapter.Records)
   for (; !e.atEnd(); e.moveNext())
   { %>
     <tr>
     <td><%=vItemNo.DisplayText%></td>
     <td><%=vPartNo.DisplayText%></td>
     <td><%=vDiscount.DisplayText%></td>
     <td><input type="text" name="<%=vQty.InputName%>" value="<%= vQty.EditText %>" ></td>
     </tr>
<%
     // items アダプタの各レコードのステート情報
     // が入る非表示フィールドを記述する。Qty フィールドを更新するときに
     // items アダプタがこれを必要とする

     if (vItemsAdapter.HiddenRecordFields != null)
       vItemsAdapter.HiddenRecordFields.WriteFields(Response)
   }
%>
  </table>
  <p></p>
  <table>
    <td><input type="submit" value="Prev Order"

onclick="AdapterForm1.__act.value='<%=vPrevRow.LinkToPage(Page.Name).AsFieldValue%>'"></td>
    <td><input type="submit" value="Next Order"

onclick="AdapterForm1.__act.value='<%=vNextRow.LinkToPage(Page.Name).AsFieldValue%>'"></td>
    <td><input type="submit" value="Refresh"

onclick="AdapterForm1.__act.value='<%=vRefreshRow.LinkToPage(Page.Name).AsFieldValue%>'"></
td>
    <td><input type="submit" value="Apply"

onclick="AdapterForm1.__act.value='<%=vApply.LinkToPage(Page.Name).AsFieldValue%>'"></td>
  </table>
</form>
```

索引

記号

数字

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

あ

い

う

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

と

ひ

ふ

へ

ほ

ま

み

め

も

ゆ

よ

ら

り

る

れ

ろ

わ