今から始めるGo言語

2014-09-29

林部 祐太

概要

- なぜGo言語か
- Hello World
- Goの文法の概要
- go言語の良い所
- まとめ
- Appendix
    - Tour guide of go
    - 参考文献

謝辞

本スライドは，各種参考文献，とりわけ WEB+DB PRESS Vol.82のGo特集 を参考にさせていただきました． ありがとうございました．

なぜGo言語か

コンパイル言語への不満点

例:C=1972年～，C++=1983年～

- 標準ライブラリが貧弱
  - 外部ライブラリの利用も面倒
  - 書き始めるのに気合が必要（気が滅入る文字列・ネットワーク・並列処理）
- 盛り沢山で複雑な機能
- 互換性の維持のために残された文法
- 面倒なメモリ管理
- コーディングスタイルが人によってバラバラ
- コンパイル・クロスコンパイルが面倒

スクリプト言語への不満

例:python2=2000年～

- （コンパイル言語と比べると）遅い
- 動的型付けは，むしろ不便
  - 引数の型を宣言できない
  - 予期しない型キャスト
- 一般公開が面倒
  - 利用者側のランタイムやライブラリのバージョンに気を使わないといけない
  - インストールさせるのも面倒

Go言語

- 最近の言語(2009年～)
  - これまでの言語の問題点をよく研究して作られている
  - 文法はシンプルに，ライブラリやツールをリッチに
- 色んな言語の良い所取り
  - 並列処理が簡単（関数呼び出しの前にgoと書くだけ）
  - 静的型付けなので，コンパイル時にバグに気づく
  - スクリプト言語のようにサクっと動かすことも可能
  - コーディングスタイルを自動的に統一できる(go fmt)
  - 全て静的リンク(依存ライブラリが全くない)ので配布が簡単
  - コンパイルが早い，クロスコンパイルも簡単
  - GC, stringがUTF-8，型推論，複数戻り値，匿名関数，豊かな標準ライブラリ …

Go言語の概要

- 開発者: Google
- 設計者
    - Robert Griesemer（Google ChromeのV8エンジンの開発者）
    - Rob Pike（Plan9，UTF-8の開発者）
    - Ken Thompson（UNIX，C言語，UTF8の開発者）
- 最新リリース: 1.3.2 (2014-09-26)
- 強い型付け

Hello World

練習1: Hello World

http://play.golang.orgですぐに試せる

```
1 package main
2
3 import ” fmt”
4
5 func main() {
6         fmt.Println(” Hello, 世界” )
7 }
```

Go言語のインストール

マニュアルに従えば簡単にインストールできる

- Linux
  1. Go言語のダウンロードページに行く
  2. go$VERSION.$OS-$ARCH.tar.gzをダウンロードする
  3. 適当な場所に展開する（例:/usr/local/go/）
  4. 環境変数を設定する（例:.zshrcに追記する）

```
export GOROOT=/usr/local/go
export GOPATH=~/.go
export PATH=$GOROOT/bin:$GOPATH/bin:$PATH
```

  5. go versionを実行してみる
- Mac
  1. インストーラーを使ってインストールする
  2. Linuxと同様に環境変数を設定する

.vimrcの設定

1. 補助ツールをインストール
   - gocode: コード補完
   - godef: 定義元へのジャンプのために godefをインストールしておく

```
go get -u github.com/nsf/gocode
go get -u code.google.com/p/rog-go/exp/cmd/godef
```

2. 以下のvimプラグインの導入・設定を行う
   - NeoBundle: デファクトスタンダードのプラグイン管理ツール
   - syntastic: 文法チェッカ
   - neocomplete (neocomplcache): コード補完
   - vim-ft-go, vim-go-extra: goの基礎設定
3. .vimrcの設定を行う

.emacsの設定

1. gocode, godefをインストール
2. go-mode, go-autocomplete, go-eldocを導入する
3. 以下の設定を行う

```
(eval-after-load " go-mode"
  ' (progn
 (require ' go-autocomplete)
 (add-hook ' go-mode-hook ' go-eldoc-setup)

 ;; key bindings
 (define-key go-mode-map (kbd " M-." ) ' godef-jump)
 (define-key go-mode-map (kbd " M-," ) ' pop-tag-mark)))
```

- 参考

練習2: Hello world++

1. ライブラリのインストール
    - `go get github.com/jessevdk/go-flags` (デファクトスタンダードの引数解析ライブラリ)
2. 入力を少し加工して返すプログラムをコピペする
    - `hello-io.go`をUTF-8で保存
    - 参考
        - mainパッケージ内のmain関数が、まず最初(初期化処理後)に実行
        - メソッド名は全て大文字で始まる
        - 型推論:=がある
3. 実行してみる
    - `go run hello-io.go`
    - `go run hello-io.go -i file.txt`

練習2: Hello world++ (Cont'd)

4. コンパイルしてみる

```
go build hello-io.go
```

5. クロスコンパイルしてみる (詳細はこちら)

```
# windows 32bit用
GOOS=windows GOARCH=386 go build hello-io.go
# macintosh 64bit用
GOOS=darwin  GOARCH=amd64 go build -o hello.mac hello-io.go
```

- (注)クロスコンパイルには事前に設定が必要

```
cd /usr/local/go/src
sudo GOOS=windows GOARCH=386 CGO_ENABLED=0 ./make.bash
sudo GOOS=windows GOARCH=amd64 CGO_ENABLED=0 ./make.bash
sudo GOOS=darwin GOARCH=386 CGO_ENABLED=0 ./make.bash
sudo GOOS=darwin GOARCH=amd64 CGO_ENABLED=0 ./make.bash
```

よく使うコマンドラインツール

コマンド	用途
go build	プログラムのビルド
go run	プログラムの実行
go get	ファイブパッケージの取得
go test	テストの実行
go env	環境変数の確認
go version	バージョンの確認
go fmt	ファイルの整形（エディタから自動で呼ぶ設定を推奨）

Goの文法の概要

予約語は25個

```
break        default      func         interface    select
case         defer        go           map          struct
chan         else         goto         package      switch
const        fallthrough  if           range        type
continue     for          import       return       var
```

特徴のある予約語: chan defer go select

事前宣言済み識別子

- 型:

  ```
  bool byte complex64 complex128 error float32 float64
  int int8 int16 int32 int64 rune string
  uint uint8 uint16 uint32 uint64 uintptr
  ```

- 定数:

  ```
  true false iota
  ```

- ゼロ値:

  ```
  nil
  ```

- 組み込み関数:
  - append delete make newをよく使う

    ```
    append cap close complex copy delete imag len
    make new panic print println real recover
    ```

（注） panic, recoverは基本的に使わない

go言語の良い所

言語仕様が小さく簡潔だが柔軟

- フォーマッタが付属しており簡単に整形できる
- 多様な書き方を認めず，バグを生みやすい表現は排除し，言語の仕様を小さく保っている
  - if文の波括弧の省略は不可

```
1 if n ==10
2 n = 14
```

  - 三項演算子はない（見にくい）
  - while文はない(for文で表現可能なので)
- switch文が柔軟で使いやすい
  - 1つのcaseが終わると(fallthroughが無い限り)swichを抜ける
  - caseに複数の値を指定できる
  - case文に式も書ける

型

- 型推論: 型が自明なら型を明示しなくてもよい

```
1 number := 3
2 val, err := SomeOperation()
```

- type: 新しい型を定義できる（aliasではない）

```
1 type ProductID int
2 var id0 ProductID = 7
3 var num int = 3
4 id0 = ProductID(num) //ok
5 id0 = num //error
```

```
1 x := 3
2 y := 4.5
3 z := y / float64(x) //OK → 陽にキャストするので型変換に関するバグが減る
4 z := y / x  //error
```

文字列/map処理が簡単

スクリプト言語のように簡単に文字列/map処理ができる

```
1  package main
2
3  import ” fmt”
4  import ” strings”
5
6  func main() {
7      mymap := make(map[string]string)
8      mymap[” 晴れ” ] = ” 良い天気”
9
10     a := ” 明日は 晴れ です”
11     items := strings.Split(a, ”  ” )
12     val, ok := mymap[items[1]]
13     items[1] = val
14     if ok && strings.HasSuffix(a, ” です” ) {
15         //「明日は-良い天気-ですかね?」が表示される
16         fmt.Printf(” %sかね?\n” , strings.Join(items, ” -” ))
17     }
18 }
```

例外機構は無い

- 複数の戻り値を返せるので，エラーも戻り値として返す
  - エラーは最後の戻り値として返しerrという変数に入れる慣習がある
- エラー処理を忘れにくい
  - 未使用変数があるとコンパイルできない
  - 複数戻り値がある場合，全て何らかの変数に代入しないとエラーになる
  - _に代入することで無視はできるが，無視したことが陽に分かるコードになる
  - (注) エラー変数を使いまわすと気づかないことがある

```
1 file1,err := os.Open(” /tmp/no_such_file” ) // ここで起きたエラーは処理されてない
2 file2,err := os.Open(” /tmp/hoge” )
```

ポインタの扱いが楽

- メモリリークの原因となるポインタ演算は不可
- メモリ管理はガベージコレクション(GC)に任せればよい
    - ガーベージコレクションの技術進化
    - 冗長なメモリ管理のコーディングをしなくてよい
    - 自動GCでマルチスレッドプログラミングが簡単になる

ライブラリ周りがしっかりしている

- 標準ライブラリが充実している（今時の言語なら当たり前）
  - JSON
  - ファイル
  - ネットワーク
  - 文字列テンプレート
- 他人のライブラリも簡単に導入できる
  - `go get github.com/xxx/yyy`するだけ
  - `go get -u all`でライブラリのアップデートも簡単にできる

interfaceがエレガント

- typeやfuncがinterface Xで定義してある関数を全て実装すると…
  - 全て自動的にXを実装していることになる(英語FAQではsatisfyと表現)
  - javaのimplementsのようにインターフェースを実装していることを明示する必要はない
  - 他のオブジェクトとの関係を気にせず，どういうメソッドを持っていれば良いかだけを考えればよい
- インターフェース型の値は，それらのメソッドを実装する任意の値をもつことができる

interfaceの例

```
1  package main
2
3  import ” fmt”
4
5  type Fooer interface {
6      Foo() string
7      ImplementsFooer()
8  }
9
10 type Bar struct {}
11 func (b *Bar) Foo() string { return ” bar”  }
12 func (b *Bar) ImplementsFooer() { fmt.Println(” implements bar” ) }
13
14 //インタフェースFooerにあるメソッドを全て実装していれば，なんでも引数として与えることが出来る
15 func foo(arg Fooer) {
16     arg.Foo()
17 }
```

並列処理が簡単

- 並列プログラムに必要な機能をサポート
  - ゴルーチン(goroutine): 軽量スレッド
  - チャンネル(channel): データをやり取りする仕組み
- 関数呼び出しの前にgoと書くだけで別のgoroutineで実行される
- ゴルーチンの生成コストは低いので気軽にgoroutineを作って良い

練習3: ステータスコードチェッカ

- urlsという文字列スライスにあるURLのウェブページにアクセスし，帰ってきたステータスコード表示するプログラム
- 色々なバリエーション
  - 安直な実装: URLに順番にアクセスして，結果を表示する
  - 並列化: 一斉にアクセスして，帰ってきたものから順番に表示する
  - タイムアウト: 1秒以上応答がない場合は無視する
  - バッファ付き並列化: mainの処理が遅くてもゴルーチンが終えられるようにする
  - 並列数の上限: ゴルーチンの同時起動数を制限する

安直な実装: go-net0.go

```
1  package main
2
3  import (
4          ” fmt”
5          ” log”
6          ” net/http”
7  )
8
9  func main() {
10         urls := []string{
11                 ” http://example.com” ,
12                 ” http://example.net” ,
13                 ” http://example.co.jp” ,
14                 ” http://example.org” ,
15         }
16
17         for _, url := range urls {
18                 res, err := http.Get(url)
19                 if err != nil {
20                         log.Fatal(err)
21                 }
22                 defer res.Body.Close()
23                 fmt.Printf(” %s\t%s\n” , url, res.Status)
24         }
25 }
```

並列化: go-net1.go

```
1  func getStatus(urls []string) ←chan string {
2      statusChan := make(chan string)
3      for _, url := range urls {
4          go func(url string) { //匿名関数
5              res, err := http.Get(url)
6              if err != nil {
7                  statusChan ← err.Error()
8                  return
9              }
10             defer res.Body.Close()
11             statusChan ← fmt.Sprintf(" %s\t%s" , url, res.Status)
12         }(url)
13
14     }
15     return statusChan
16 }
```

mainの処理を変更する

```
1  //内部で呼び出したゴルーチンの終了は待たずにメインスレッドは続行される
2  statusChan := getStatus(urls)
3  for i := 0; i < len(urls); i++ {
4          fmt.Printf(" %s\n" , ←statusChan)
5  }
```

タイムアウト: go-net2.go

```
1  func getStatusWithTimeout(urls []string) {
2      statusChan := getStatus(urls)
3      timeout := time.After(time.Second)
4      for {
5          select {
6          case status := ←statusChan:
7              fmt.Printf(" %s\n" , status)
8          case ←timeout:
9              return
10         }
11     }
12 }
```

mainの処理を変更する

```
1  getStatusWithTimeout(urls)
```

バッファ付き並列化: go-net3.go

- makeに第2引数が無いとバッファ無し
  - main()の処理が遅いと，ゴルーチンは最後のチャンネルに書き込む処理ができず，待た無くてはいけず，メモリに負荷がかかる
- URLの数だけバッファを確保し
  - ゴルーチンはチャンネルに書き込んで終了できる

```
1 func getStatus(urls []string) ←chan string {
2 statusChan := make(chan string, len(urls))
```

  - main()の処理に時間がかかっても，ゴルーチンは終了できる

```
1 statusChan := getStatus(urls)
2 for i := 0; i < len(urls); i++ {
3     fmt.Printf(" %s\n" , ←statusChan)
4     time.Sleep(time.Second) //時間のかかる処理
5 }
```

同時起動数制限: go-net4.go

- URLの数が多い場合，大量のゴルーチンが起動し，メモリに負荷がかかる
- バッファ付きのチャンネルで同時起動数制限

```
1  func getStatus(urls []string) ←chan string {
2  statusChan := make(chan string, len(urls)) //URLの数だけバッファを確保
3  var empty struct{}
4  limit := make(chan struct{}, 2) //同時ゴルーチン起動数=2
5  for _, url := range urls {
6      select {
7      case limit ← empty:
8          go func(url string) { //匿名関数
9              res, err := http.Get(url)
10             if err != nil {
11                 statusChan ← err.Error()
12                 return
13             }
14             defer res.Body.Close()
15             statusChan ← fmt.Sprintf(" %s\t%s" , url, res.Status)
16             ←limit //終わったら1つ読みだして空きを作る
17         }(url)
18     }
19 }
20 return statusChan
```

まとめ

まとめ

- 言語仕様はシンプルさを保ち，周辺ツールが充実
- goはこれからますます発展しそうな言語
- 触れなかった話題
    - テスト
        - `go test ./...`
    - 開発するフォルダ
        - `$GOPATH/src/bitbucket.org/username/project_name/`

Appendix

Tour guide of go

A tour of go

- goの機能を順番に知ることが出来るチュートリアル
- 演習問題は難しく，人によってはツマラナイので必ずしもしなくてよいと思う
- 色々な人がやっているのでブログ等が参考になる
    - 忙しい人のためのA Tour of Go
- 以下では，各自tourで知るべき箇所をメモする

基礎的な事柄 (tour 1-23)

- package
- import
  - import時に名前を指定できる
- 最初の文字が大文字ならば公開
- 関数の形式
  - 引数，戻り値（複数，名前付き宣言）
- 変数宣言の方法
- const
- for
- if-then-else
- switch

structとポインタ (tour 25-29)

- ポインタは，値渡しではなく参照渡しの時したい時に使う
  - ポインタ演算はできない
  - ある種の型（文字列，インターフェイス，チャネル，マップ，スライス）の値はそもそもポインタのようなものなので，それらのポインタはあまり意味がないので注意
  - (参考) Goでxxxのポインタを取っているプログラムはだいたい全部間違っている
- newするとゼロ初期化された構造体のポインタが帰ってくる

```
var t *T = new(T)
```

スライス (tour 30-34)

- 配列（固定長）はシビアにメモリ管理したいとき以外はあまり使わず，スライス（可変長）をよく使う
- makeで作る

```
a := make([]int, 5)
a2 := make([]int, 2, 3) //leng=2, capa=3
c := []int{1,2,3}
```

```
b := make([]int, 0, 5) // len(b)=0, cap(b)=5
b = b[:cap(b)] // len(b)=5, cap(b)=5
b = b[1:]      // len(b)=4, cap(b)=4
```

- sliceの初期値はnil
- nilのsliceは長さ0で容量も0

makeの補足

少しややこしいので慣れが必要

呼び出し	型T	結果
make(T, n)	slice	長さn、キャパシティnであるT型のスライス
make(T, n, m)	slice	長さn、キャパシティmであるT型のスライス
make(T)	map	T型のマップ
make(T, n)	map	要素の初期容量がnであるT型のマップ
make(T)	channel	T型の同期チャネル
make(T, n)	channel	バッファサイズがnであるT型の非同期チャネル

range (tour 34-35)

- forのrangeでsliceやmapをひとつずつ反復処理
- rangeで得た値に対して処理しても元の値は変更されないので注意

```
1 for idx, v := range items{
2 if idx == 2{
3     v[idx] = 30 //items[2]は30になる
4     v = 40 //items[2]は40にはならない!
5 }
6 }
```

- 不要な変数は_に代入する

```
1 for _, value := range pow {
2 fmt.Printf(" %d\n" , value)
3 }
```

map (tour 37-40)

- mapの宣言方法と初期化

```
var m map[string]Vertex=  make(map[string]Vertex)
```

- mapの値の削除方法

```
delete(m, key)
```

- mapの値の存在チェック

```
elem, ok = m[key]
```

関数 (tour 42-43)

- 関数はfuncで宣言する
- 関数は変数に代入可能
- クロージャー

switch (tour 45-47)

- fallthrough
- 条件は複数とれる
- 条件のないswitchはswitch trueと書いたことと同じ

メソッド (tour 50-52)

- method receiverをfuncキーワードとメソッド名の間に書いてメソッドを定義する

```
1 func (self *MyStrcture) doSomething()int{
```

- method receiverが値型(MyStrcture)だとコピーされる
- method receiverがポインタ型(*MyStrcture)だと参照渡し

interface (tour 53-54)

- C++の純粋抽象基底クラスや，javaのinterfaceのようなもの
- interface Xで定義してある関数を全て実装するtypeやfuncは全て自動的にXを実装していることになる
  - 「ダックタイピング」
  - javaのimplementsのようにインターフェースを実装していることを明示する必要はない
  - 他のオブジェクトとの関係を気にせず，どういうメソッドを持っていれば良いかだけを考えればよい
- インターフェース型の値は，それらのメソッドを実装する任意の値をもつことができる

Error (tour 55)

- stringを返すError()というメソッドを実装すれば，新しいerrorを定義できる
- errorというinterfaceがgoで予め定義されている

goroutine (tour 63)

- goの後ろに関数を書けば新しいgoroutine上で実行される

```
1 go f(x, y, z)
```

- Goのランタイムに管理される軽量なスレッド
- 引数x,y,zは現在のgoroutineで評価
- fは新しいgoroutineで実行
- goroutineは同じアドレス空間で実行される
- 使用するコア数をデフォルトの1からMAXに変更するにはruntime.GOMAXPROCS(runtime.NumCPU())

channel (tour 64-66)

- goroutine間でデータを受け渡しするのに使う

```
1 ch1 := make(chan int) // int型の値を送受信できるチャンネルをつくる
2 ch1 ← v // vをチャネルchへ送る
3 v := ←ch1 // ch から受信して代入
```

```
1 var ch2 chan← float64  // float64の送信のみできるチャンネル
2 var ch3 ←chan int      // intの受信のみできるチャンネル
```

- チャンネルはバッファでき，バッファがいっぱいになったときにまとめて送信する

```
1 ch := make(chan int, 100)
```

- チャンネルはcloseできき受信時に第二戻り値がfalseになる（通常closeの必要はない）

```
1 v, ok := ←ch
```

select (tour 67-68)

```
1 select {
2     case c ← x:
3         x, y = y, x+y
4     case ←quit:
5         fmt.Println(” quit” )
6 }
```

- goroutineを複数の通信操作で待たせ，いずれかのcaseを実行できるまでブロック
- 条件が一致したcaseを実行する
- 複数が一致した場合caseはランダムに選ばれる
- どのcaseにも一致しないのであればdefaultのcaseが実行される
- for{}でくくることが多い
    - 参考: For文の中でSelectを使う時は関数に

参考文献

色々な情報

- まずはFAQや公式ドキュメントを読んでみよう
- WEB+DB PRESS Vol.82のGo特集も参考になる
- Goのコンセプトを知ろう
  - グーグル,C/C++に代わる新言語「Go」をOSSで公開
  - コンパイルが速くて、スクリプト言語的に書ける言語が欲しかった
  - Go言語の気に入ったところ/気に入らなかったところ
  - Golangでエレガントだと思うこと
- tipsを知ろう
  - effective-goではない何か
  - Goプログラマであるかを見分ける10の質問
  - Golang のオフィシャルが提供するインタフェースまとめ

ライブラリ関連情報

- Awesome Go
- Goで使ってみたライブラリとツールの感想をそこはかとなく書くよ
- GoでJSONのシリアライズ・デシリアライズ
- Go言語でコマンドラインオプション使うなら。便利パッケージgo-flags
- jessevdk/go-flagsを使ってみる
- godepを利用して依存ライブラリの管理を行う