Kotlin コルーチンを 理解しよう 2019

2019/08/24

八木 俊広

About Me

八木 俊広

Toshihiro Yagi

@sys1yagi

Software engineer at

Ubie

https://techbooster.booth.pm/items/1485567

TechBooster 【C96新刊】
Androidプログラミング短編集：王女とカルテットの宝探し

Ubieについて

AI問診 Ubie

AIで、医師の業務効率化を支援

患者が使うタブレット問診票と、
医師が使うエディタ等を開発

初診問診の事務を**1/3に効率化**。
医師が、余裕をもって患者と向き合えるように

今日話すこと

- コルーチンとはなにか、なにがうれしいのか
- Kotlinにおけるコルーチンの仕組み
- Kotlinコルーチンのきほん
- コルーチンスコープと構造化された並行性
- コルーチンと設計
- コルーチンのテスト

コルーチンとはなにか、
なにがうれしいのか

コルーチンとはなにか

メルヴィン・コンウェイの
1963年の論文が初出

※実装は機械語で1958年にあったらしい

Design of a Separable Transition-Diagram Compiler*

MELVIN E. CONWAY
Directorate of Computers, USAF
L. G. Hanscom Field, Bedford, Mass.

A COBOL compiler design is presented which is compact enough to permit rapid, one-pass compilation of a large subset of COBOL on a moderately large computer. Versions of the same compiler for smaller machines require only two working tapes plus a compiler tape. The methods given are largely applicable to the construction of ALGOL compilers.

Introduction

This paper is written in rebuttal of three propositions widely held among compiler writers, to wit: (1) syntax-directed compilers [1] suffer practical disadvantages over other types of compilers, chiefly in speed; (2) compilers should be written with compilers; (3) COBOL [2] compilers must be complicated. The form of the rebuttal is to describe a high-speed, one-pass, syntax-directed COBOL compiler which can be built by two people with an assembler in less than a year.

The compiler design presented here has the following properties.

1. It processes full elective COBOL except for automatic segmentation and its byproducts, such as those properties of the ALTER verb which are affected by segmentation. The verbs DEFINE, ENTER, USE and INCLUDE are accessible to the design but were not included in the prototype coded at the Case Computing Center.

2. It can be implemented as a true one-pass compiler (with load-time fixup of forward references to procedure names) on a machine with 10,000 to 16,000 words of high-speed storage. In this configuration it processes a source deck as fast as current one-pass algebraic compilers.

3. It can be segmented into many possible configurations, depending on the source computer's storage size, such that (a) once a segment leaves high-speed storage it will not be recalled; (b) only two working tapes are required, and no tape sorting is needed. One such configuration requires five segments for a machine with 8000 six-bit characters of core storage.

Of course any compiler can be made *one-pass* if the high-to make this design (in which all tables are accessed while stored in memory) practical on contemporary computers. None of these techniques is limited in application to COBOL compilers. The following specific techniques are discussed: the coroutine method of separating programs, transition diagrams in syntactical analysis, data name qualification analysis, and instruction generation for conditional statements.

The algorithms described were verified on the 5000-word Burroughs 220 at the Case Institute of Technology Computing Center. A two-pass configuration was planned for that machine, and first-pass code was checked out through the syntactical analysis. At the time the project was discontinued a complete COBOL syntax checker was operating at 140 fully-punched source cards per minute. (The Case 220 had a typical single-address instruction time of 100 microseconds.) Remarks presented later suggest that a complete one-pass version of the compiler, which would be feasible on a 10,000-word machine, would run at well over 100 source cards per minute.

Coroutines and Separable Programs

That property of the design which makes it amenable to many segment configurations is its *separability*. A program organization is separable if it is broken up into processing modules which communicate with each other according to the following restrictions: (1) the only communication between modules is in the form of discrete items of information; (2) the flow of each of these items is along fixed, one-way paths; (3) the entire program can be laid out so that the input is at the left extreme, the output is at the right extreme, and everywhere in between all information items flowing between modules have a component of motion to the right.

Under these conditions each module may be made into a *coroutine*; that is, it may be coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines. Thus, coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program.[1] There is no bound placed by this definition on the number of inputs and outputs a coroutine may have.

The coroutine notion can greatly simplify the conception of a program when its modules do not communicate with each other synchronously. Consider, as an example, a program which reads cards and writes the string of characters it finds, column 1 of card 1 to column 80 of card 1, then column 1 of card 2, and so on, with the following wrinkle: every time there are adjacent asterisks they will be paired off from the left and each "**" will be replaced

http://melconway.com/Home/pdf/compiler.pdf

コルーチンとはなにか

メルヴィン・コンウェイの1963年の論文が初出

※実装は機械語で1958年にあったらしい

Design of a Separable Transition-Diagram Compiler*

MELVIN E. CONWAY
Directorate of Computers, USAF
L. G. Hanscom Field, Bedford, Mass.

A COBOL compiler design is presented which is compact enough to permit rapid, one-pass compilation of a large subset of COBOL on a moderately large computer. Versions of the same compiler for smaller machines require only two working tapes plus a compiler tape. The methods given are largely applicable to the construction of ALGOL compilers.

Introduction

This paper is written in rebuttal of three propositions widely held among compiler writers, to wit: (1) syntax-directed compilers [1] suffer practical disadvantages over other types of compilers, chiefly in speed; (2) compilers should be written with compilers; (3) COBOL [2] compilers must be complicated. The form of the rebuttal is to describe a high-speed, one-pass, syntax-directed COBOL compiler which can be built by two people with an assembler in less than a year.

The compiler design presented here has the following properties.

1. It processes full elective COBOL except for automatic segmentation and its byproducts, such as those properties of the ALTER verb which are affected by segmentation. The verbs DEFINE, ENTER, USE and INCLUDE are accessible to the design but were not included in the prototype coded at the Case Computing Center.

2. It can be implemented as a true one-pass compiler (with load-time fixup of forward references to procedure names) on a machine with 10,000 to 16,000 words of high-speed storage. In this configuration it processes a source deck as fast as current one-pass algebraic compilers.

3. It can be segmented into many possible configurations, depending on the source computer's storage size, such that (a) once a segment leaves high-speed storage it will not be recalled; (b) only two working tapes are required, and no tape sorting is needed. One such configuration requires five segments for a machine with 8000 six-bit characters of core storage.

Of course any compiler can be made *one-pass* if the high-

to make this design (in which all tables are accessed while stored in memory) practical on contemporary computers. None of these techniques is limited in application to COBOL compilers. The following specific techniques are discussed: the coroutine method of separating programs, transition diagrams in syntactical analysis, data name qualification analysis, and instruction generation for conditional statements.

The algorithms described were verified on the 5000-word Burroughs 220 at the Case Institute of Technology Computing Center. A two-pass configuration was planned for that machine, and first-pass code was checked out through the syntactical analysis. At the time the project was discontinued a complete COBOL syntax checker was operating at 140 fully-punched source cards per minute. (The Case 220 had a typical single-address instruction time of 100 microseconds.) Remarks presented later suggest that a complete one-pass version of the compiler, which would be feasible on a 10,000-word machine, would run at well over 100 source cards per minute.

Coroutines and Separable Programs

That property of the design which makes it amenable to many segment configurations is its *separability*. A program organization is separable if it is broken up into processing modules which communicate with each other according to the following restrictions: (1) the only communication between modules is in the form of discrete items of information; (2) the flow of each of these items is along fixed, one-way paths; (3) the entire program can be laid out so that the input is at the left extreme, the output is at the right extreme, and everywhere in between all information items flowing between modules have a component of motion to the right.

Under these conditions each module may be made into a *coroutine*; that is, it may be coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines. Thus, coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program.[1] There is no bound placed by this definition on the number of inputs and outputs a coroutine may have.

The coroutine notion can greatly simplify the conception of a program when its modules do not communicate with each other synchronously. Consider, as an example, a program which reads cards and writes the string of characters it finds, column 1 of card 1 to column 80 of card 1, then column 1 of card 2, and so on, with the following wrinkle: every time there are adjacent asterisks they will be paired off from the left and each "**" will be replaced

http://melconway.com/Home/pdf/compiler.pdf

1967年にSimulaがプログラミング言語としては初めてコルーチンを実装したらしい

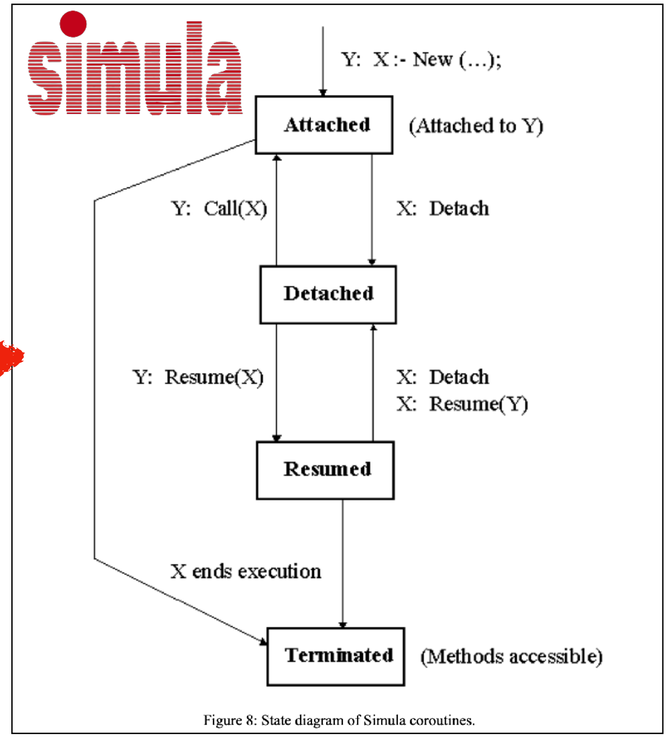


Figure 8: State diagram of Simula coroutines.

http://staff.um.edu.mt/jskl1/talk.html

コルーチンとはなにか

メルヴィン・コンウェイの
1963年の論文が初出

※実装は機械語で1958年にあったらしい

Design of a Separable Transition-Diagram Compiler*

MELVIN E. CONWAY
Directorate of Computers, USAF
L. G. Hanscom Field, Bedford, Mass.

A COBOL compiler design is presented which is compact enough to permit rapid, one-pass compilation of a large subset of COBOL on a moderately large computer. Versions of the same compiler for smaller machines require only two working tapes plus a compiler tape. The methods given are largely applicable to the construction of ALGOL compilers.

Introduction

This paper is written in rebuttal of three propositions widely held among compiler writers, to wit: (1) syntax-directed compilers [1] suffer practical disadvantages over other types of compilers, chiefly in speed; (2) compilers should be written with compilers; (3) COBOL [2] compilers must be complicated. The form of the rebuttal is to describe a high-speed, one-pass, syntax-directed COBOL compiler which can be built by two people with an assembler in less than a year.

The compiler design presented here has the following properties.

1. It processes full elective COBOL except for automatic segmentation and its byproducts, such as those properties of the ALTER verb which are affected by segmentation. The verbs DEFINE, ENTER, USE and INCLUDE are accessible to the design but were not included in the prototype coded at the Case Computing Center.

2. It can be implemented as a true one-pass compiler (with load-time fixup of forward references to procedure names) on a machine with 10,000 to 16,000 words of high-speed storage. In this configuration it processes a source deck as fast as current one-pass algebraic compilers.

3. It can be segmented into many possible configurations, depending on the source computer's storage size, such that (a) once a segment leaves high-speed storage it will not be recalled; (b) only two working tapes are required, and no tape sorting is needed. One such configuration requires five segments for a machine with 8000 six-bit characters of core storage.

Of course any compiler can be made *one-pass* if the high-to make this design (in which all tables are accessed while stored in memory) practical on contemporary computers. None of these techniques is limited in application to COBOL compilers. The following specific techniques are discussed: the coroutine method of separating programs, transition diagrams in syntactical analysis, data name qualification analysis, and instruction generation for conditional statements.

The algorithms described were verified on the 5000-word Burroughs 220 at the Case Institute of Technology Computing Center. A two-pass configuration was planned for that machine, and first-pass code was checked out through the syntactical analysis. At the time the project was discontinued a complete COBOL syntax checker was operating at 140 fully-punched source cards per minute. (The Case 220 had a typical single-address instruction time of 100 microseconds.) Remarks presented later suggest that a complete one-pass version of the compiler, which would be feasible on a 10,000-word machine, would run at well over 100 source cards per minute.

Coroutines and Separable Programs

That property of the design which makes it amenable to many segment configurations is its *separability*. A program organization is separable if it is broken up into processing modules which communicate with each other according to the following restrictions: (1) the only communication between modules is in the form of discrete items of information; (2) the flow of each of these items is along fixed, one-way paths; (3) the entire program can be laid out so that the input is at the left extreme, the output is at the right extreme, and everywhere in between all information items flowing between modules have a component of motion to the right.

Under these conditions each module may be made into a *coroutine*; that is, it may be coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines. Thus, coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program.[1] There is no bound placed by this definition on the number of inputs and outputs a coroutine may have.

The coroutine notion can greatly simplify the conception of a program when its modules do not communicate with each other synchronously. Consider, as an example, a program which reads cards and writes the string of characters it finds, column 1 of card 1 to column 80 of card 1, then column 1 of card 2, and so on, with the following wrinkle: every time there are adjacent asterisks they will be paired off from the left and each "**" will be replaced

http://melconway.com/Home/pdf/compiler.pdf

1967年にSimulaがプログラミング言語としては初めてコルーチンを実装したらしい

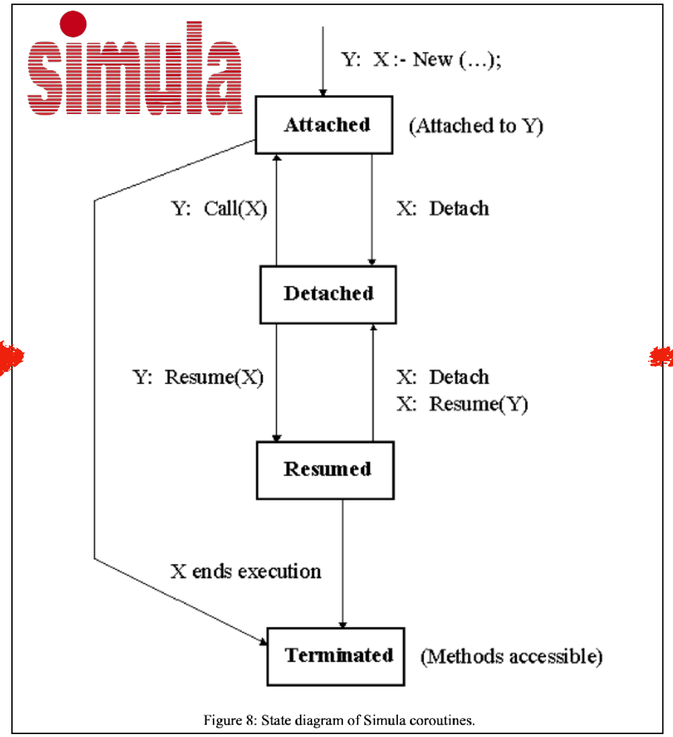


Figure 8: State diagram of Simula coroutines.

http://staff.um.edu.mt/jskl1/talk.html

様々な言語で色々な実装
generator, async/await...

Modula-2

Lua

python™

Ruby
A Programmer's Best Friend

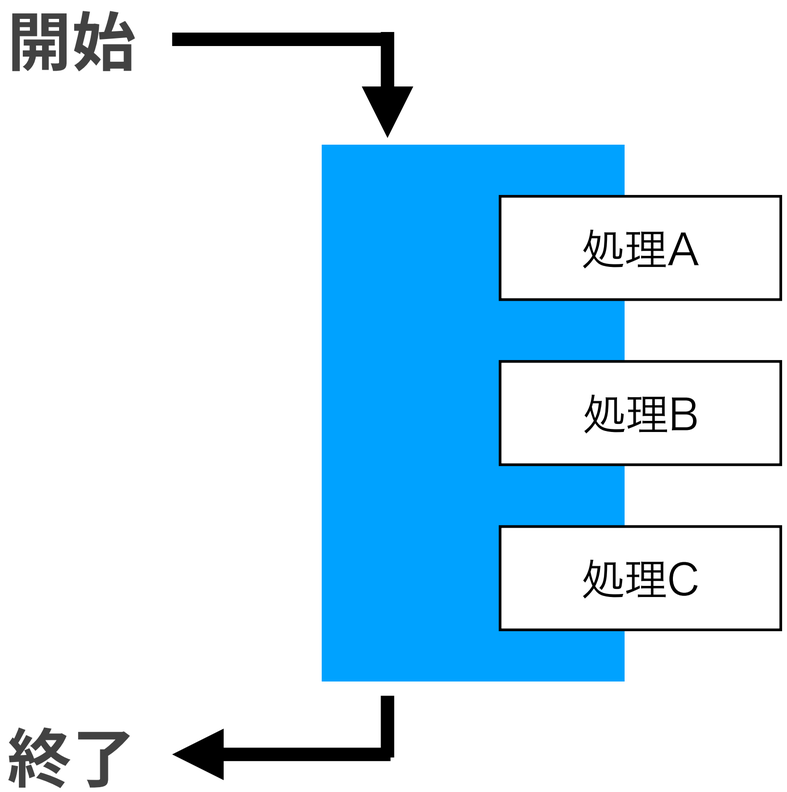
C#

GO

コルーチンは中断と再開を通常の関数に導入する

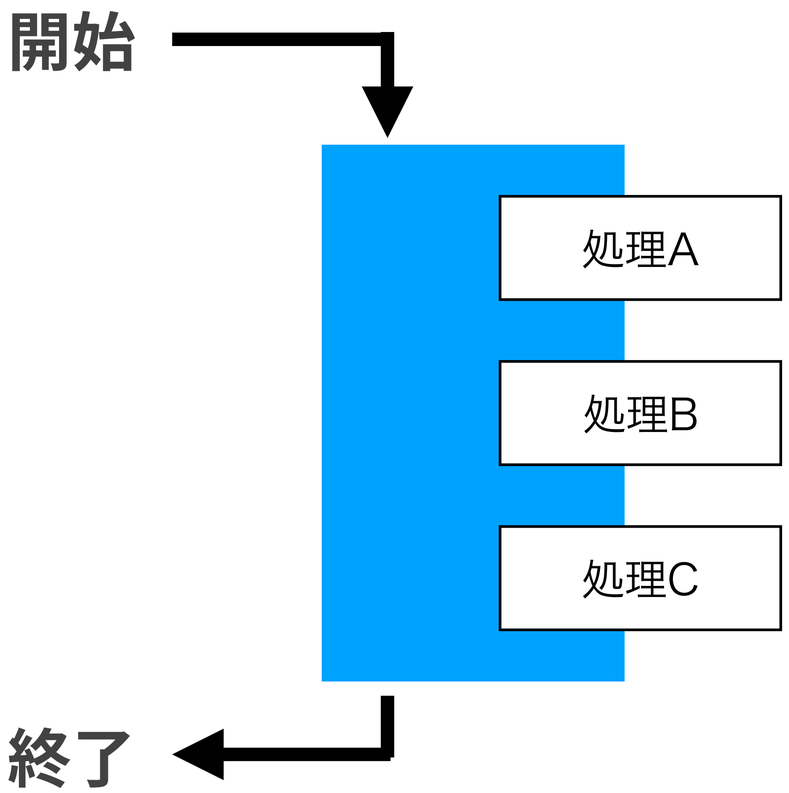
コルーチンは中断と再開を通常の関数に導入する

通常の関数

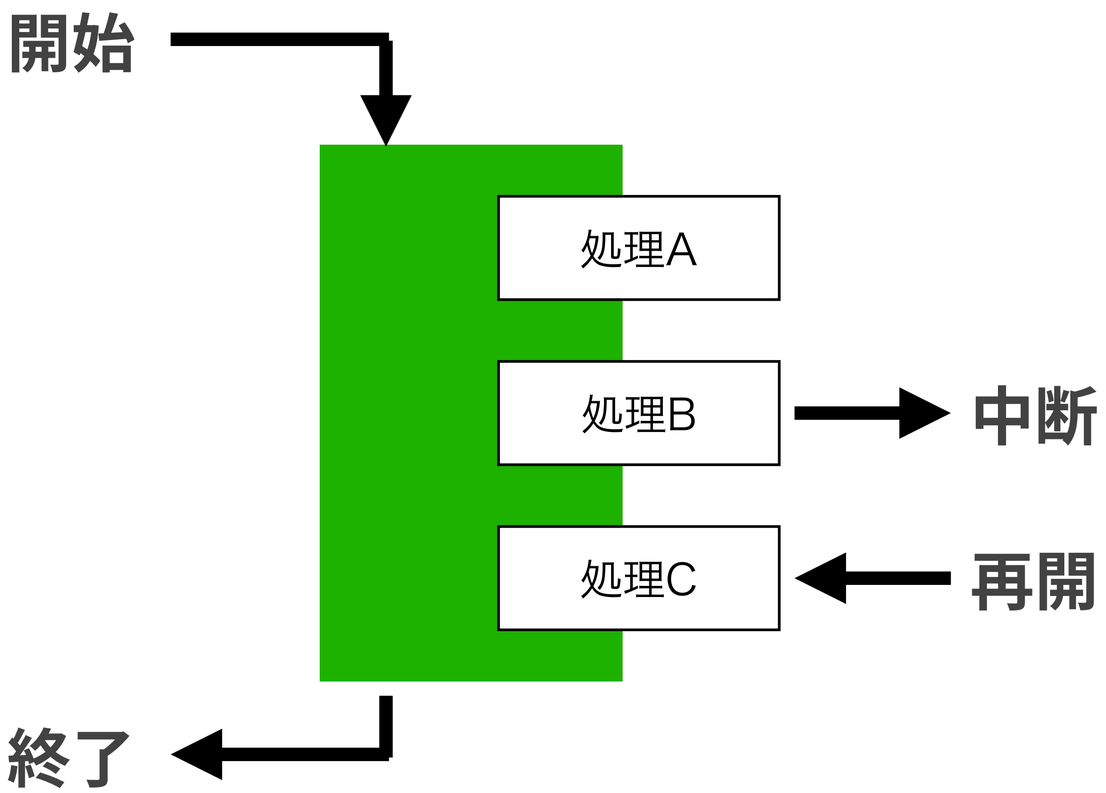


コルーチンは中断と再開を通常の関数に導入する

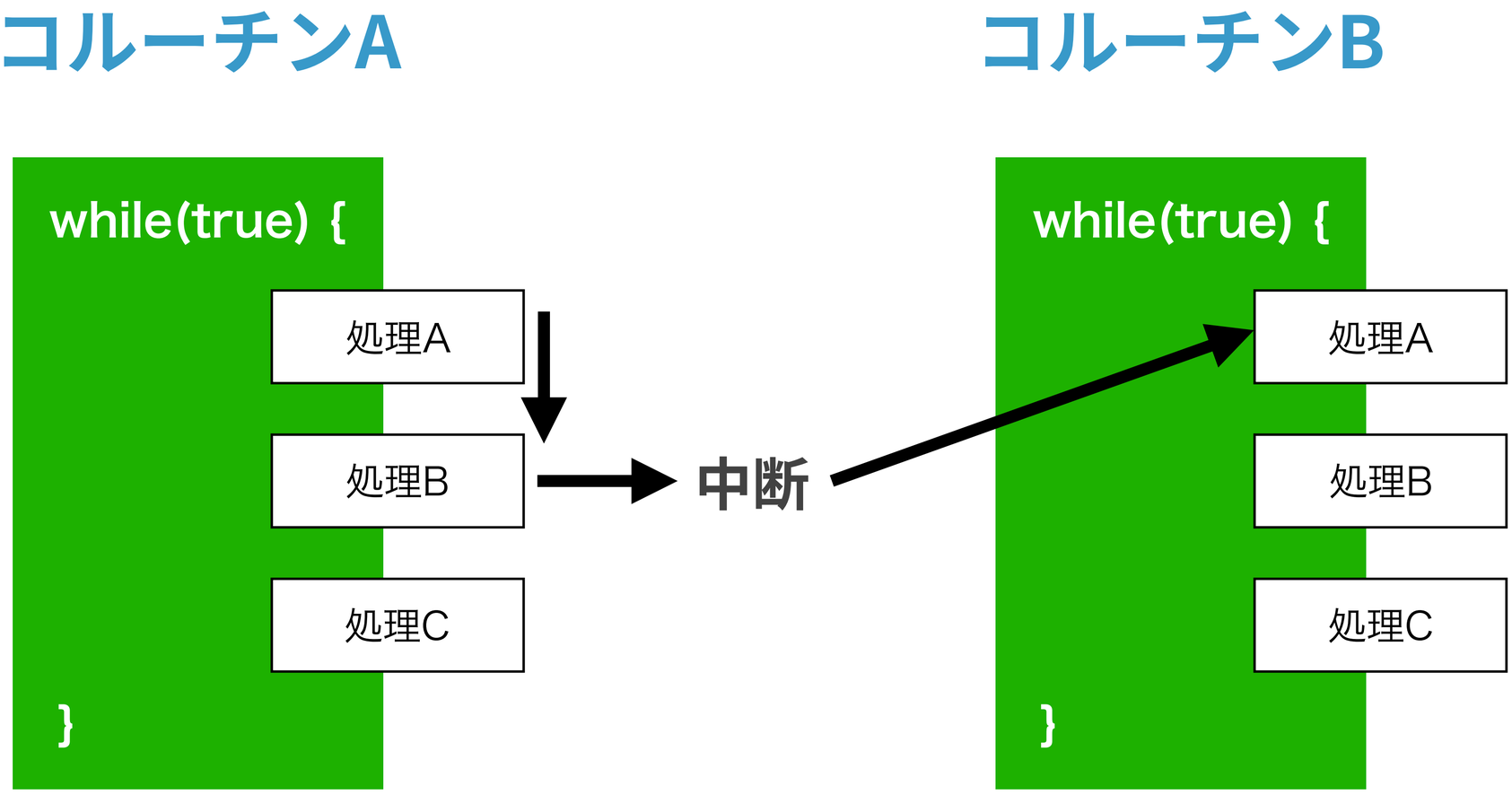
通常の関数



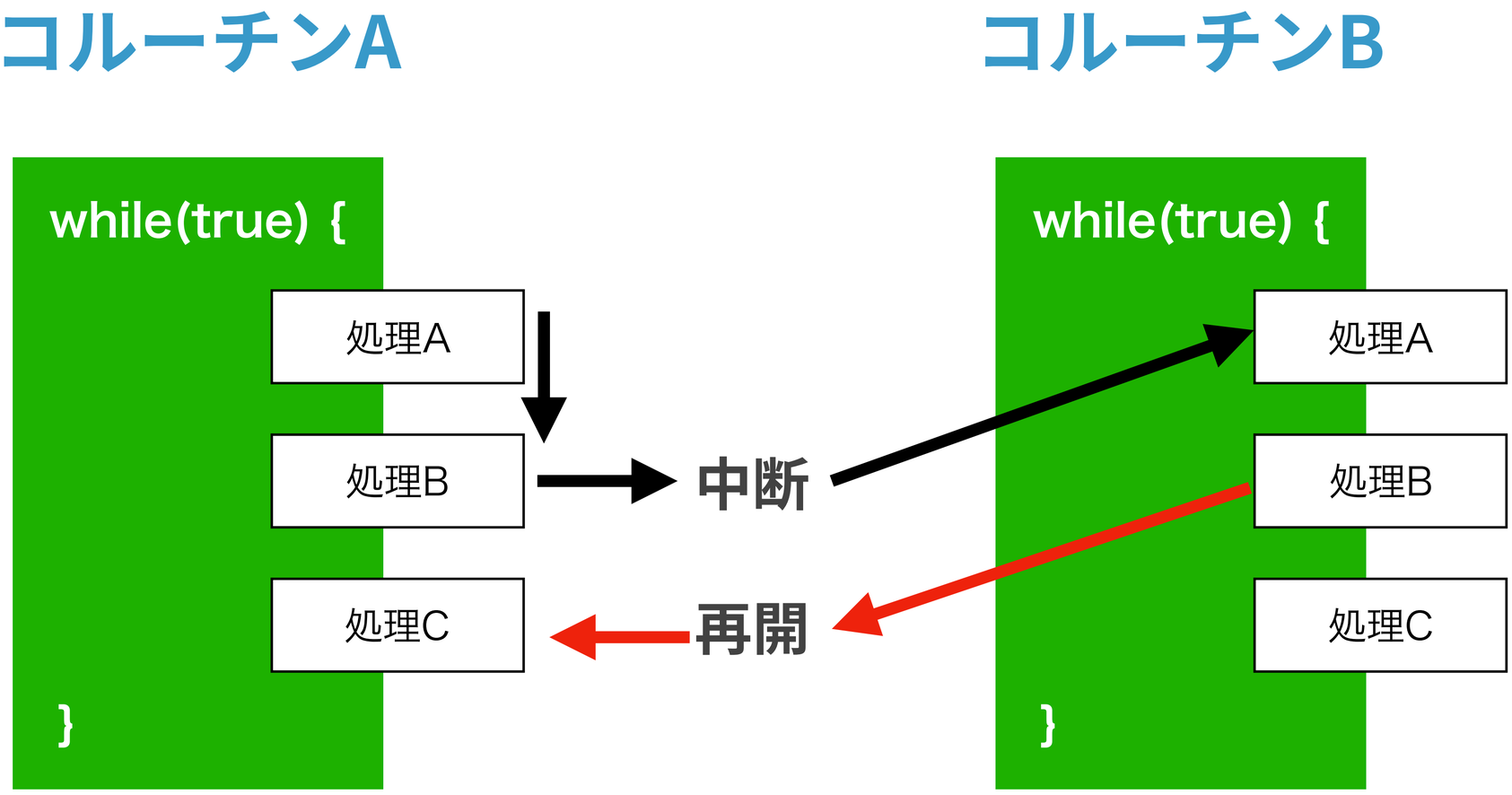
コルーチン



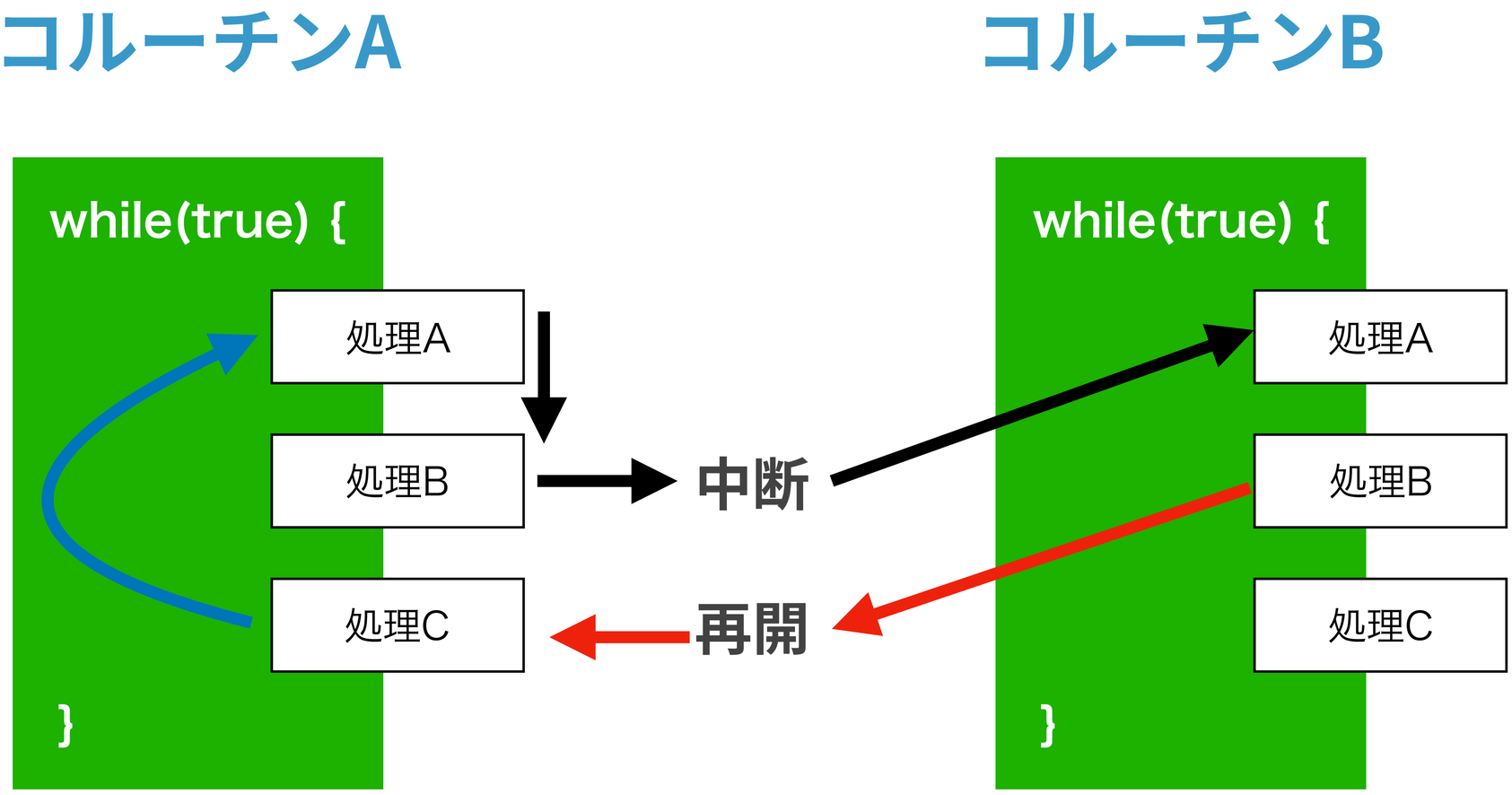
コルーチン同士で協調的に動作する（対称）



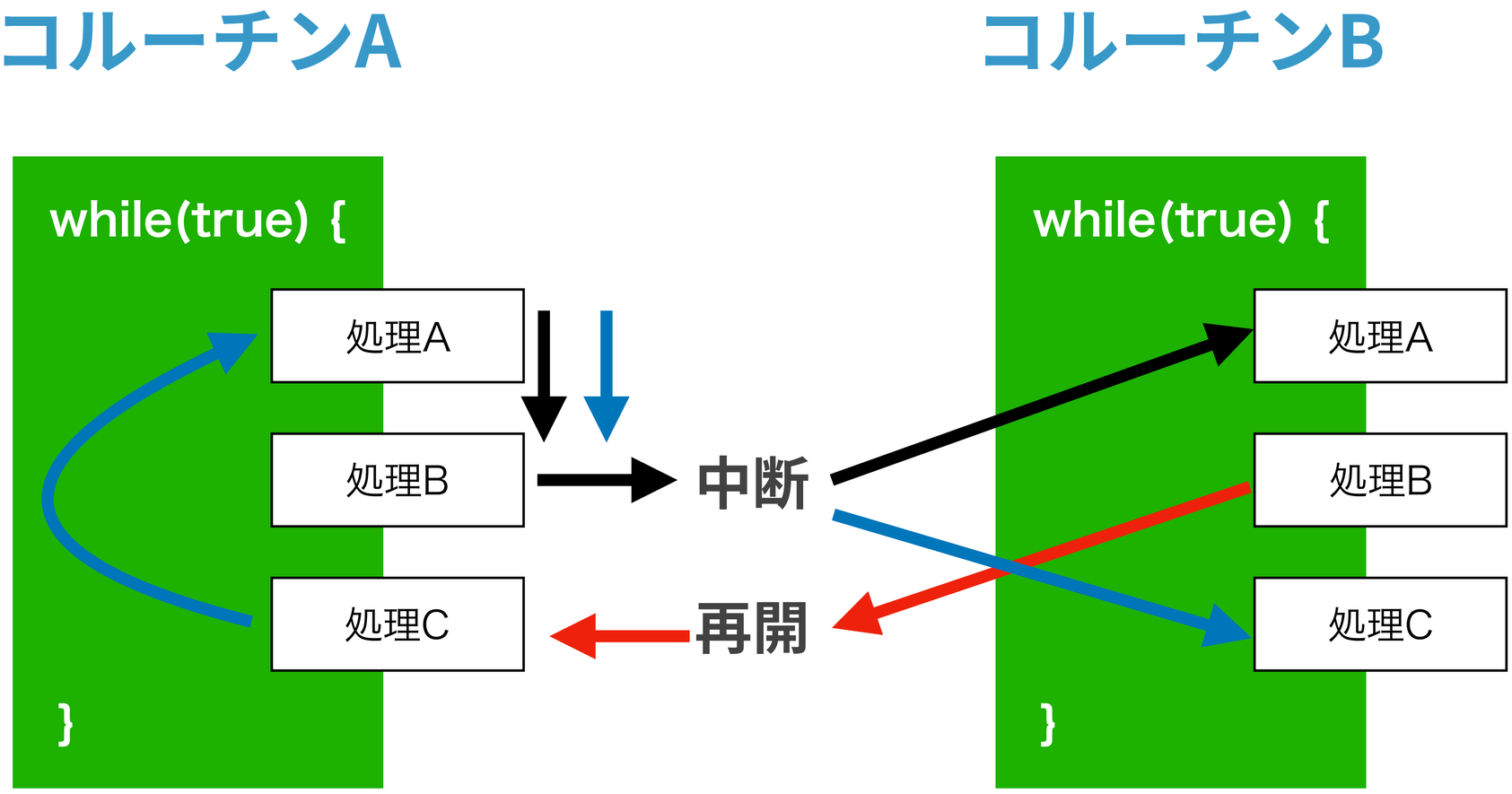
コルーチン同士で協調的に動作する（対称）



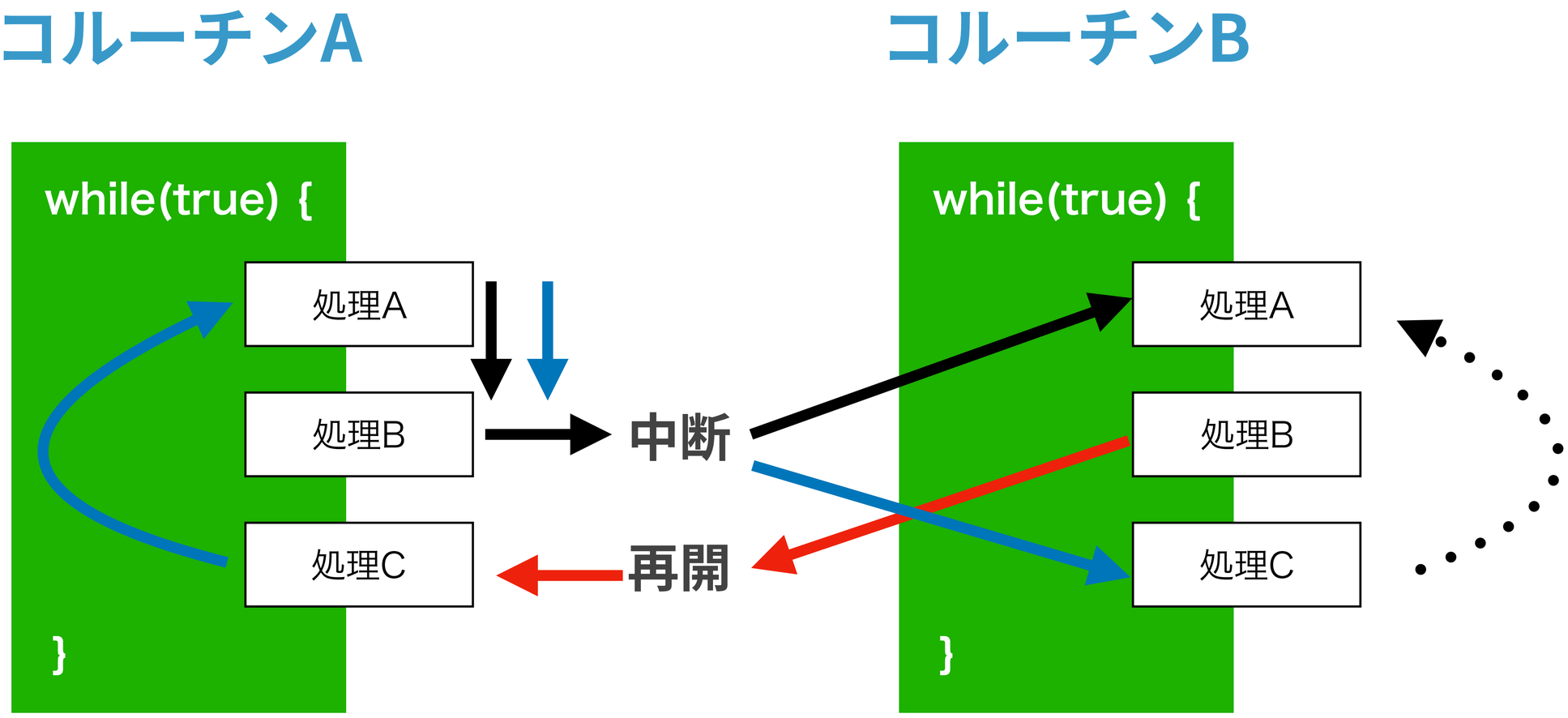
コルーチン同士で協調的に動作する（対称）



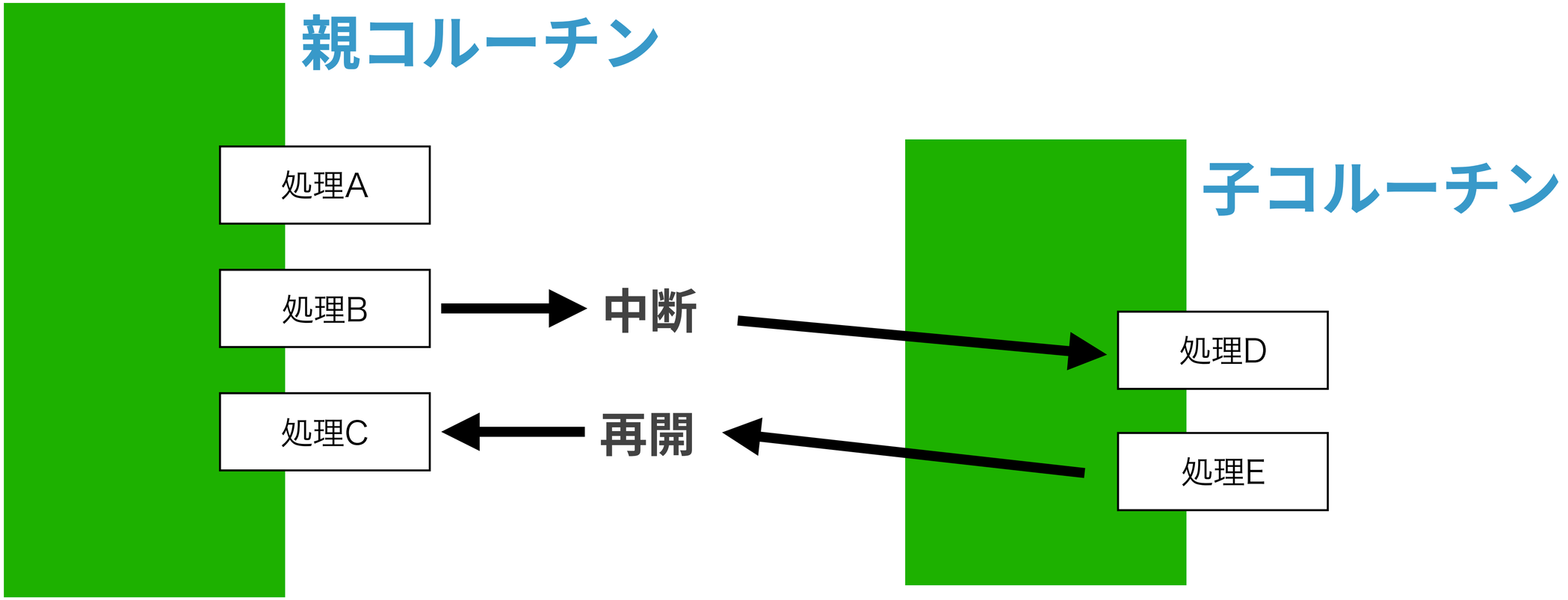
コルーチン同士で協調的に動作する（対称）



コルーチン同士で協調的に動作する（対称）

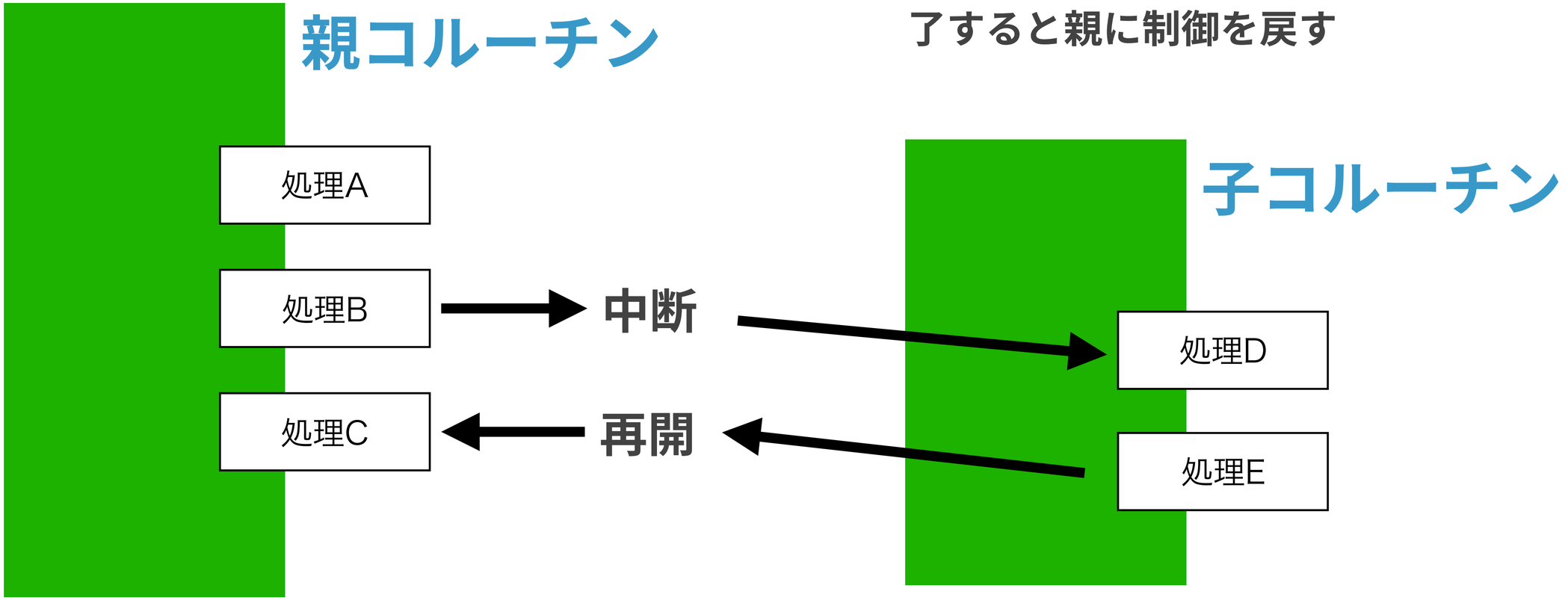


コルーチン同士で協調的に動作する（非対称）

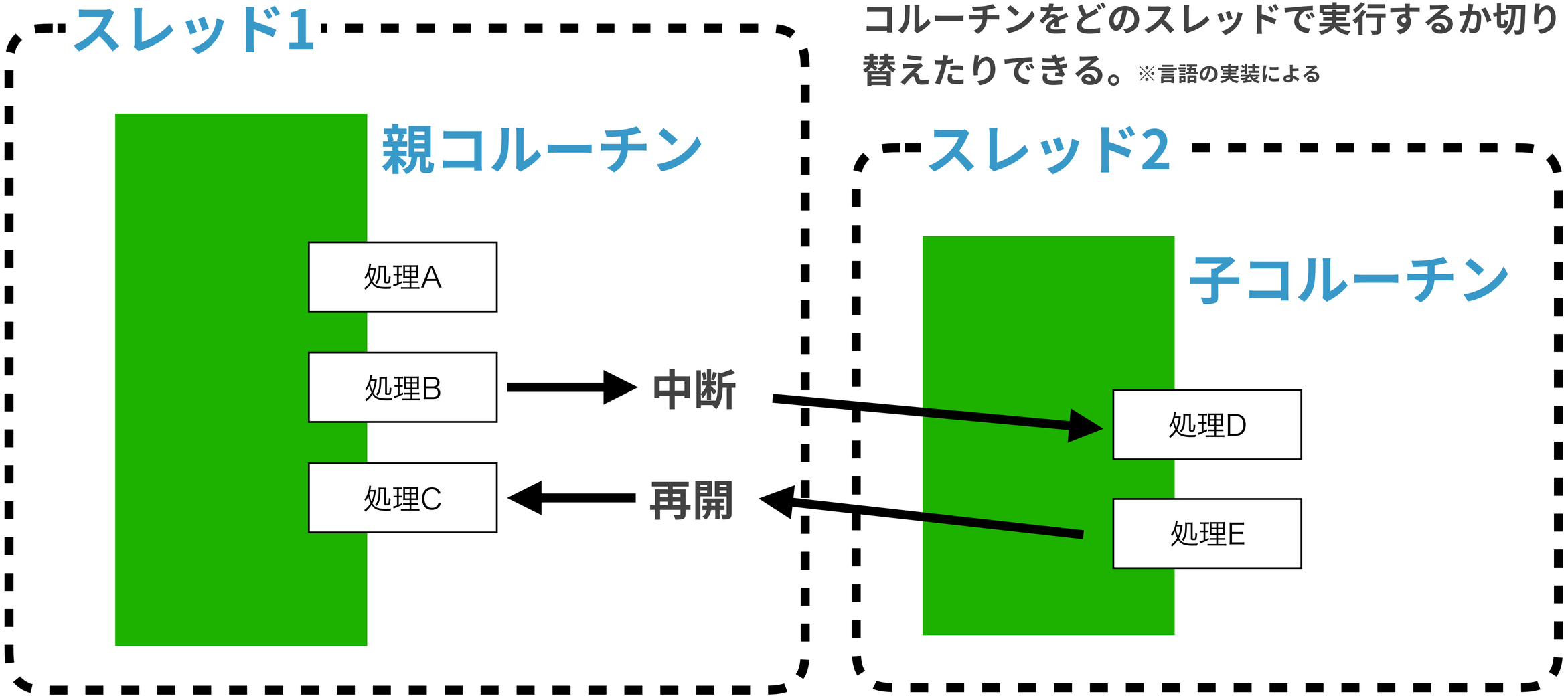


コルーチン同士で協調的に動作する（非対称）

子コルーチンが中断したり終了すると親に制御を戻す



コルーチン同士で協調的に動作する（非対称）



コルーチンをどのスレッドで実行するか切り替えたりできる。※言語の実装による

コルーチンはなにがうれしいのか

コルーチンはなにがうれしいのか

- 継続状況を表現しやすい
- スレッドより軽量である（Kotlin）

継続状況を表現しやすい

継続状況とはなにか

```
fun getProfile(id: Int, f: (Profile) -> Unit)

fun loadProfile(id: Int) {
  getProfile(token) { profile ->
    showProfile(profile)
  }
}
```

継続状況とはなにか

```
fun getProfile(id: Int, f: (Profile) -> Unit)

fun                                          {
    getProfile(token) { profile ->
        showProfile(profile)
    }
}
```

APIにリクエストして結果を返す関数

継続状況とはなにか

```
fun getProfile(id: Int, f: (Profile) -> Unit)

fun loadProfile(id:
  getProfile(token) { profile ->
    showProfile(profile)
  }
}
```

結果を受け取るコールバック

継続状況とはなにか

```
fun getProfile(id: Int, f: (Profile) -> Unit)

fun loadProfile(id: Int) {
  getProfile(token) { profile ->
    showProfile(profile)
  }
}
```

継続状況とはなにか

```
fun getProfile(id: Int, f: (Profile) -> Unit)

fun loadProfile(id: Int) {
  getProfile(token) { profile ->
    showProfile(profile)
  }
}
```

コールバックに続きの処理を書く

継続状況とはなにか

```
fun getProfile(id: Int, f: (Profile) -> Unit)

fun loadProfile(id: Int) {
  getProfile(token) { profile ->
    showProfile(profile)
  }
}
```

コールバックに続きの処理を書く

継続状況

継続状況を表現しやすい

```
suspend fun getProfile(id: Int): Profile


suspend fun loadProfile(id: Int) {
    val profile = getProfile(token)
    showProfile(profile)
}
```

継続状況を表現しやすい

```
suspend fun getProfile(id: Int): Profile


suspend fun loadProfile(id: Int) {
    val profile = getProfile(token)
    showProfile(profile)
}
```

ここで中断する

継続状況を表現しやすい

```
suspend fun getProfile(id: Int): Profile


suspend fun loadProfile(id: Int) {
    val profile = getProfile(token)
    showProfile(profile)
}
```

再開して続きを処理する

継続状況を表現しやすい

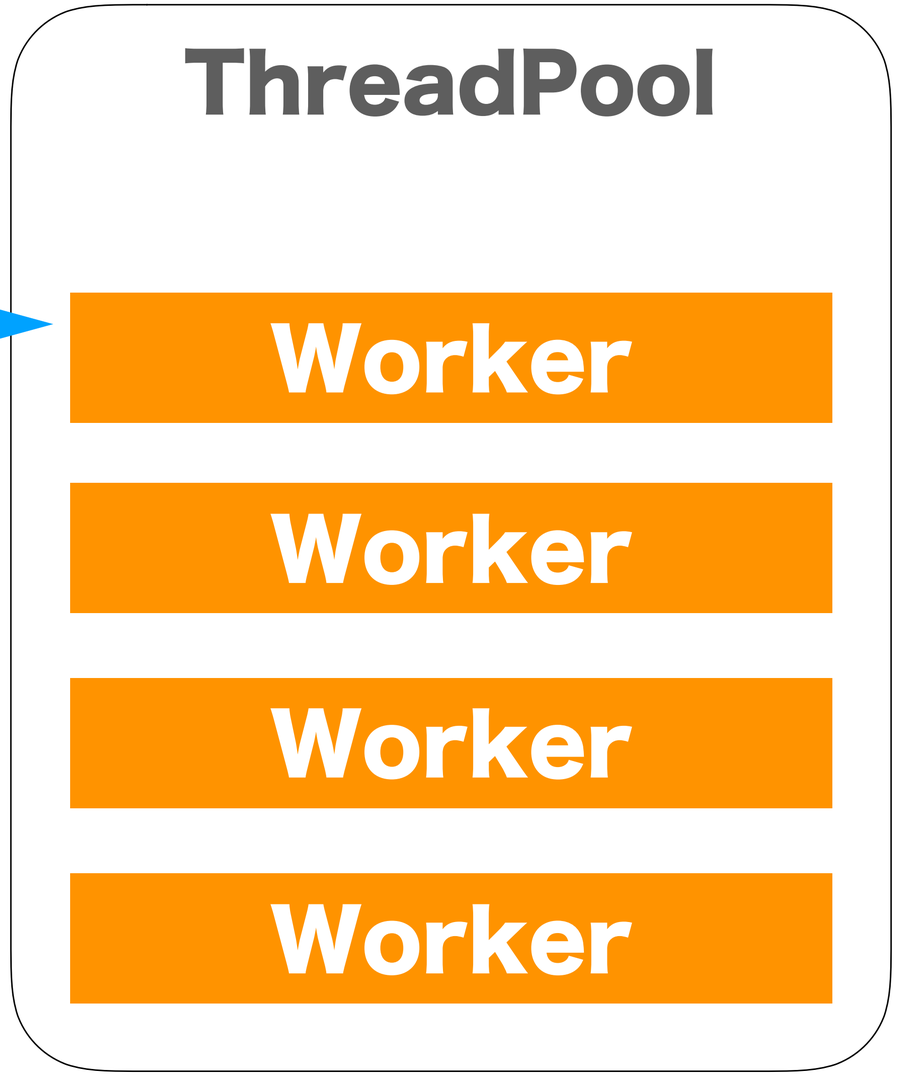
```
suspend fun getProfile(id: Int): Profile


suspend fun loadProfile(id: Int) {
    val profile = getProfile(token)
    showProfile(profile)
}
```

Kotlinでは実行スレッドを任意に変更できる

スレッドより軽量である（Kotlin）

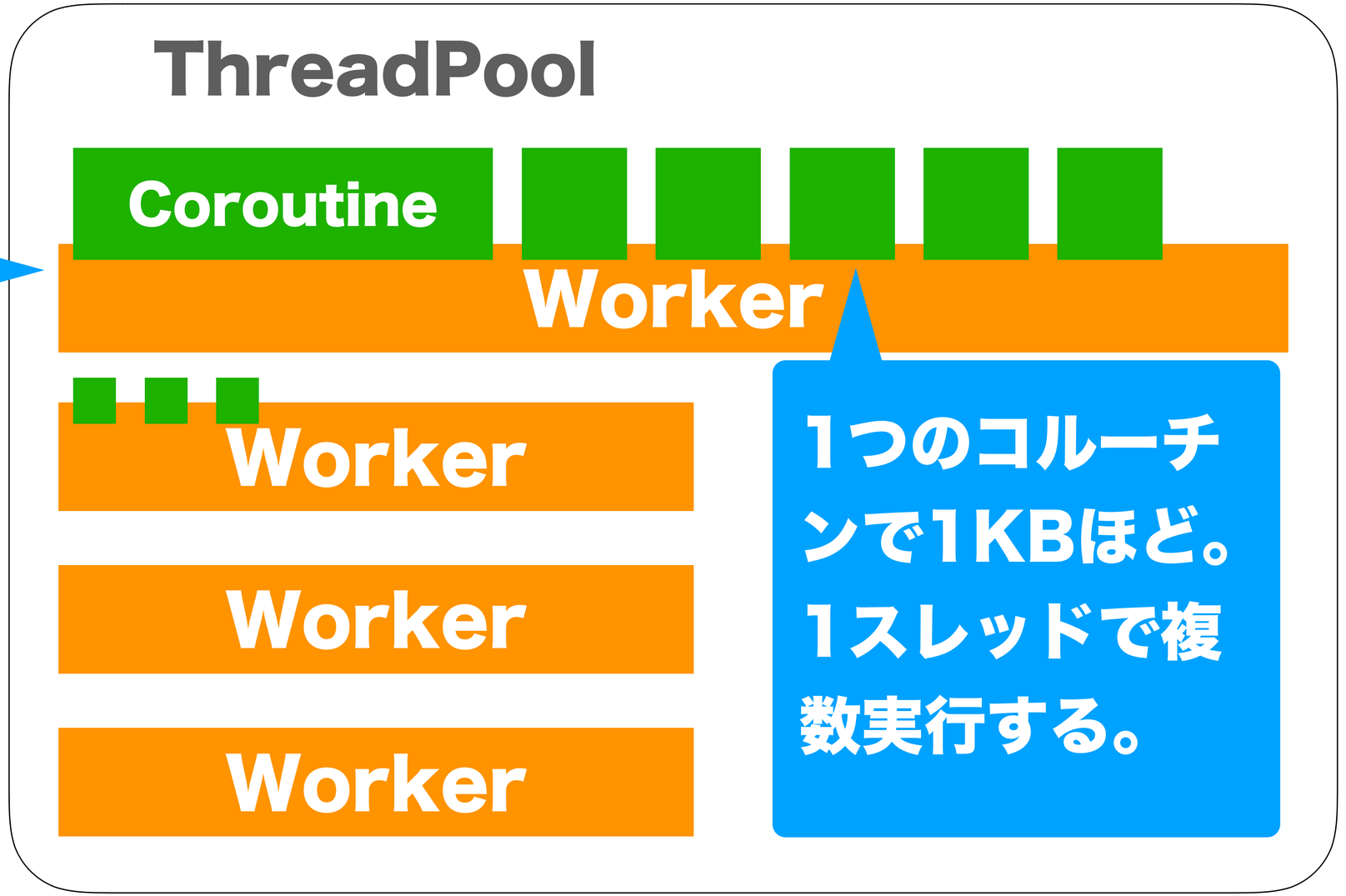
スレッドより軽量である（Kotlin）

1スレッド
1MB~2MBほど
メモリを使う



スレッドより軽量である（Kotlin）

1スレッド
1MB~2MBほど
メモリを使う



コルーチンとはなにか、なにがうれしいのか

- 中断と再開を通常の関数に導入する
- 継続状況を表現しやすい
- スレッドより軽量である

Kotlinにおけるコルーチンの仕組み

suspendキーワードをKoltinに導入

```
suspend fun getProfile(id: Int): Profile
```

suspendキーワードをKoltinに導入

```
suspend fun getProfile(id: Int): Profile
```

関数に付与することで中断を表す

suspendキーワードをKoltinに導入

```
suspend fun getProfile(id: Int): Profile


fun loadProfile(id: Int) {
  val profile = getProfile(id)
  showProfile(profile)
}
```

suspendキーワードをKoltinに導入

```
suspend fun getProfile(id: Int): Profile


fun loadProfile(id: Int) {
  val profile = getProfile(id) // Compile Error
  showProfile(profile)
}
```

suspendキーワードをKoltinに導入

```
suspend fun getProfile(id: Int): Profile


fun loadProfile(id: Int) {
    val profile = getProfile(id) // Compile Error
    showProfile(profile)
}
```

通常の関数からは呼び出せない

suspendキーワードをKoltinに導入

```
suspend fun getProfile(id: Int): Profile

suspend fun loadProfile(id: Int) {
    val profile = getProfile(id) // OK
    showProfile(profile)
}
```

suspendキーワードをKoltinに導入

```
suspend fun getProfile(id: Int): Profile

suspend fun loadProfile(id: Int) {···· 開始
  val profile = getProfile(id)·········· 中断
  showProfile(profile) ···················· 再開
} ····································· 終了
```

suspendラムダとコルーチン

```
suspend fun getProfile(id: Int): Profile


suspend fun loadProfile(id: Int) {
    val profile = getProfile(id)
    showProfile(profile)
}
```

通常の関数からsuspend関数が呼び出せないとすると、そもそもどうやって呼び出せばいいのか?

suspendラムダとコルーチン

```
fun launch(
    block: suspend () -> Unit
) { … }
```

suspendラムダとコルーチン

```
fun launch(
    block: suspend () -> Unit
) { … }
```

関数型にsuspendキーワードをつけたものを、suspendラムダと呼ぶ

suspendラムダとコルーチン

```
fun launch(
    block: suspend () -> Unit
) { … }
```

関数型にsuspendキーワードをつけたものを、suspendラムダと呼ぶ

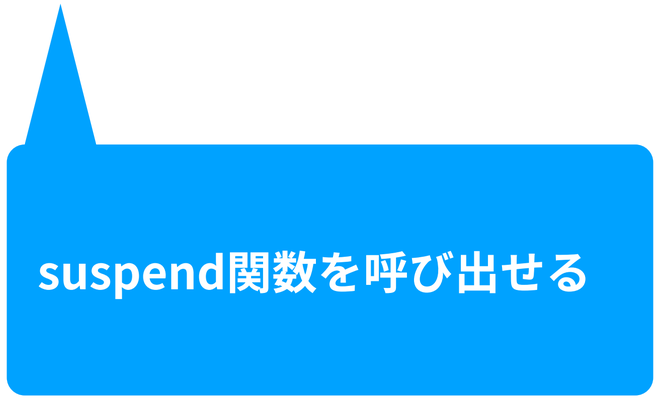
suspendラムダがコルーチン本体になる

suspendラムダとコルーチン

```
launch {
    val dataSource = ProfileDataSource()
    showLoading()
    dataSource.loadProfile(10)
    hideLoading()
}
```

suspendラムダとコルーチン

```
launch {
    val dataSource = ProfileDataSource()
    showLoading()
    dataSource.loadProfile(10)
    hideLoading()
}
```

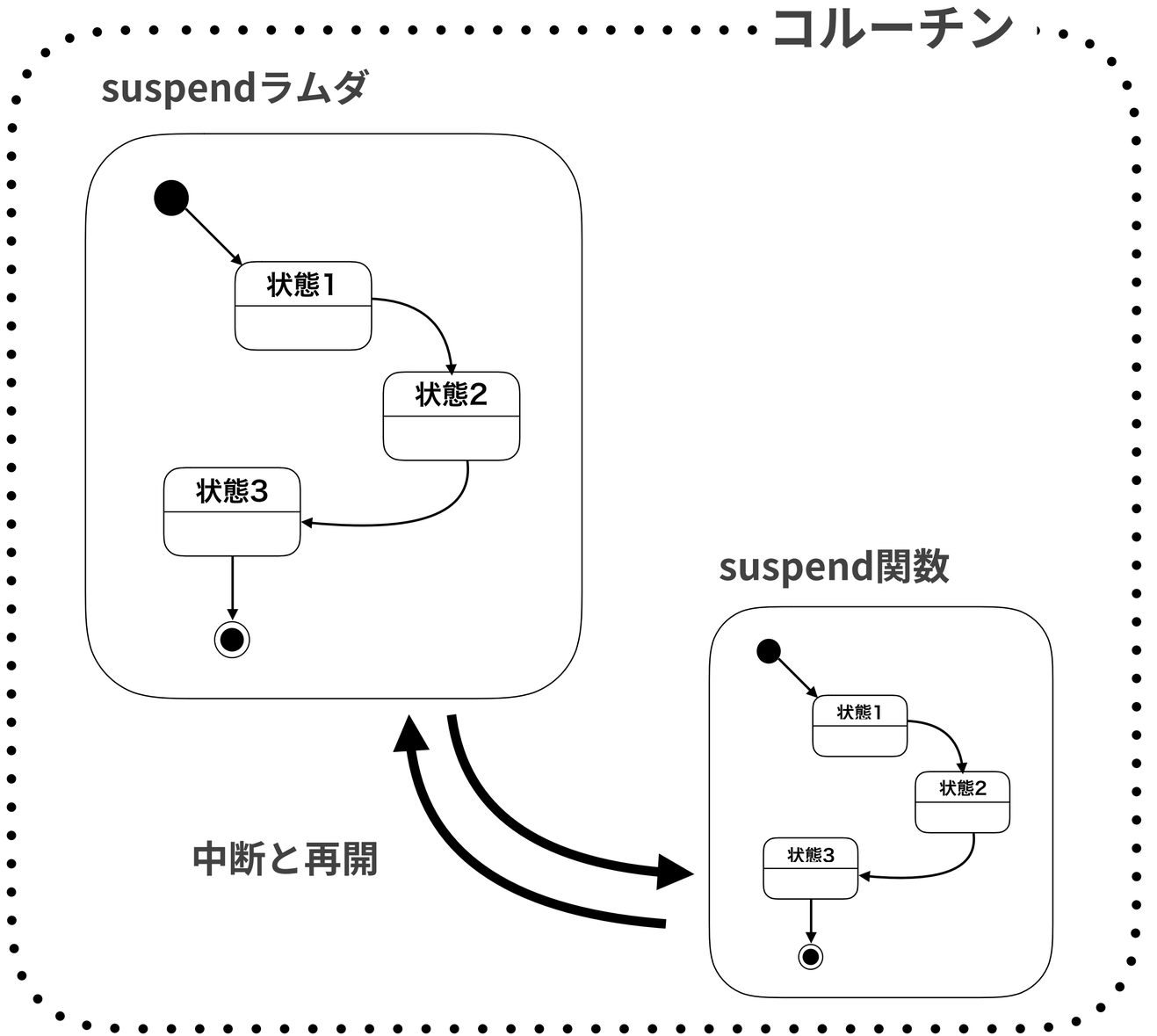


Kotlinにおけるコルーチンの仕組み

Kotlinプログラム

```
fun main() {
    GlobalScope.launch { this: CoroutineScope
        val dataSource = ProfileDataSource()
        showLoading()
        dataSource.loadProfile( id: 10)
        hideLoading()
    }
}
```

コンパイル



Kotlinは

Kotlinプログラム

```
fun main() {
    GlobalScope.launch { this: Coro
        val dataSource = Profile
        showLoading()
        dataSource.loadProfile(
        hideLoading()
    }
}
```

コンパイル

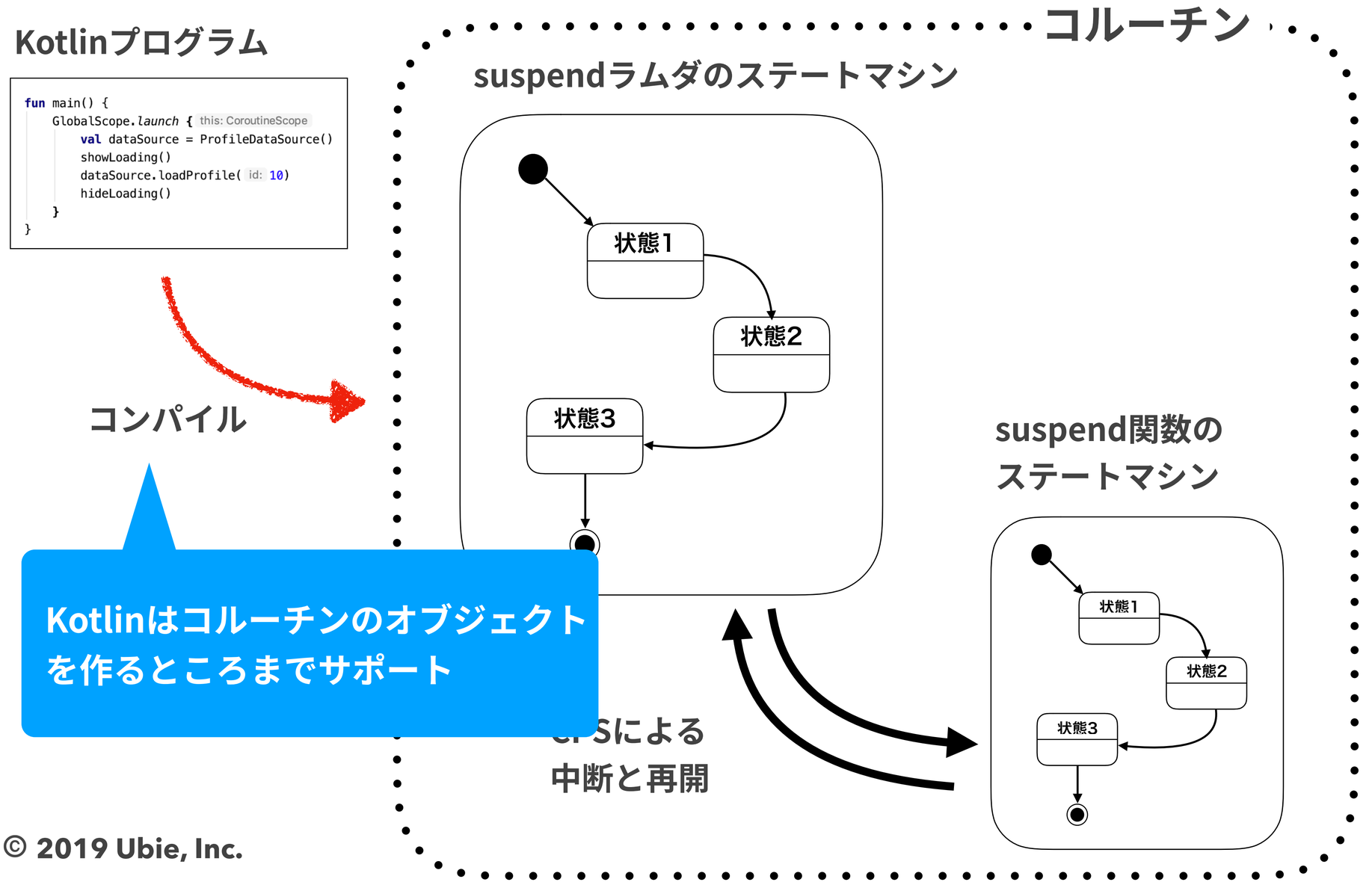
どのようにコルーチンに変換しているかの詳細は前回の資料を参照してください

Kotlinにおけるコルーチンの仕組み

Kotlinプログラム

```
fun main() {
    GlobalScope.launch { this: CoroutineScope
        val dataSource = ProfileDataSource()
        showLoading()
        dataSource.loadProfile( id: 10)
        hideLoading()
    }
}
```

コンパイル



Kotlinはコルーチンのオブジェクトを作るところまでサポート

Kotlinにおけるコルーチンの仕組み

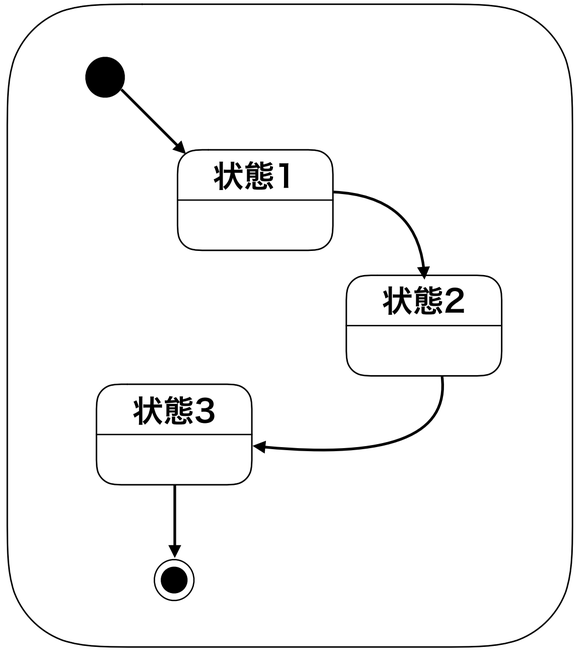
Kotlinプログラム

```
fun main() {
    GlobalScope.launch { this: CoroutineScope
        val dataSource = ProfileDataSource()
        showLoading()
        dataSource.loadProfile( id: 10)
        hideLoading()
    }
}
```

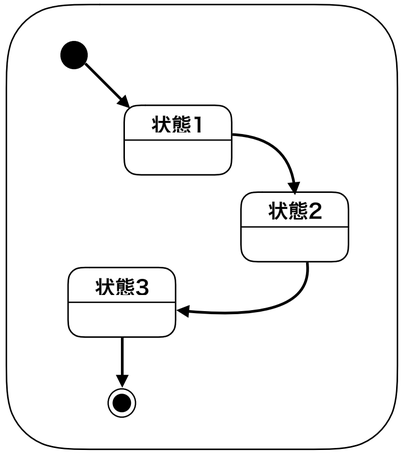
コンパイル

コルーチン

suspendラムダのステートマシン



suspend関数の
ステートマシン



CPSによる
中断と再開

コルーチンライブラリ

CoroutineDispatcher
launch, async
Job, Deferred
CoroutineScope
CoroutineStart…

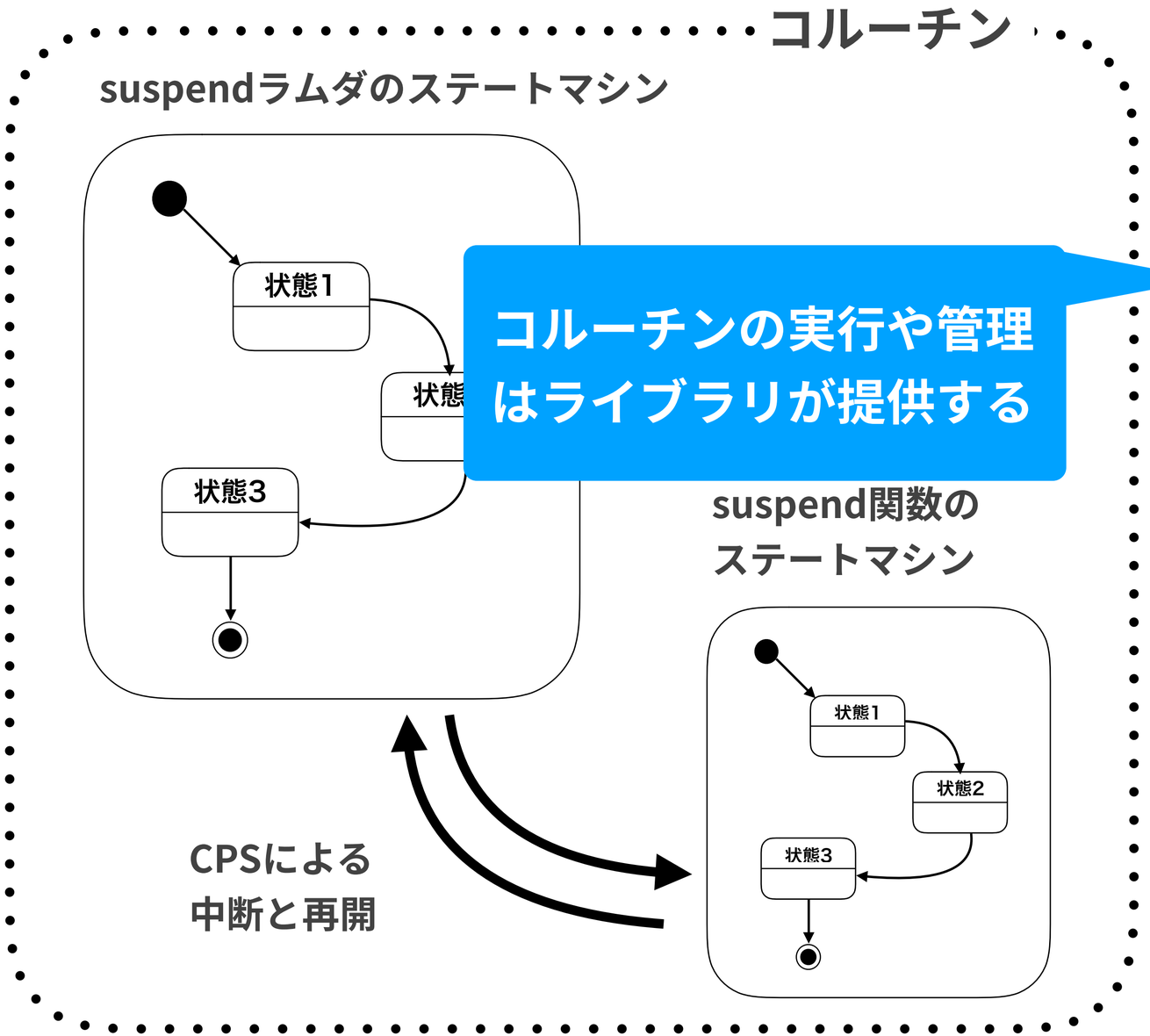
実行
キャンセル
スレッドプール
割り込み
例外ハンドラ
スコープ etc…

Kotlinにおけるコルーチンの仕組み

Kotlinプログラム

```
fun main() {
    GlobalScope.launch { this: CoroutineScope
        val dataSource = ProfileDataSource()
        showLoading()
        dataSource.loadProfile( id: 10)
        hideLoading()
    }
}
```

コンパイル



コルーチンライブラリ

CoroutineDispatcher
launch, async
Job, Deferred
CoroutineScope
CoroutineStart...

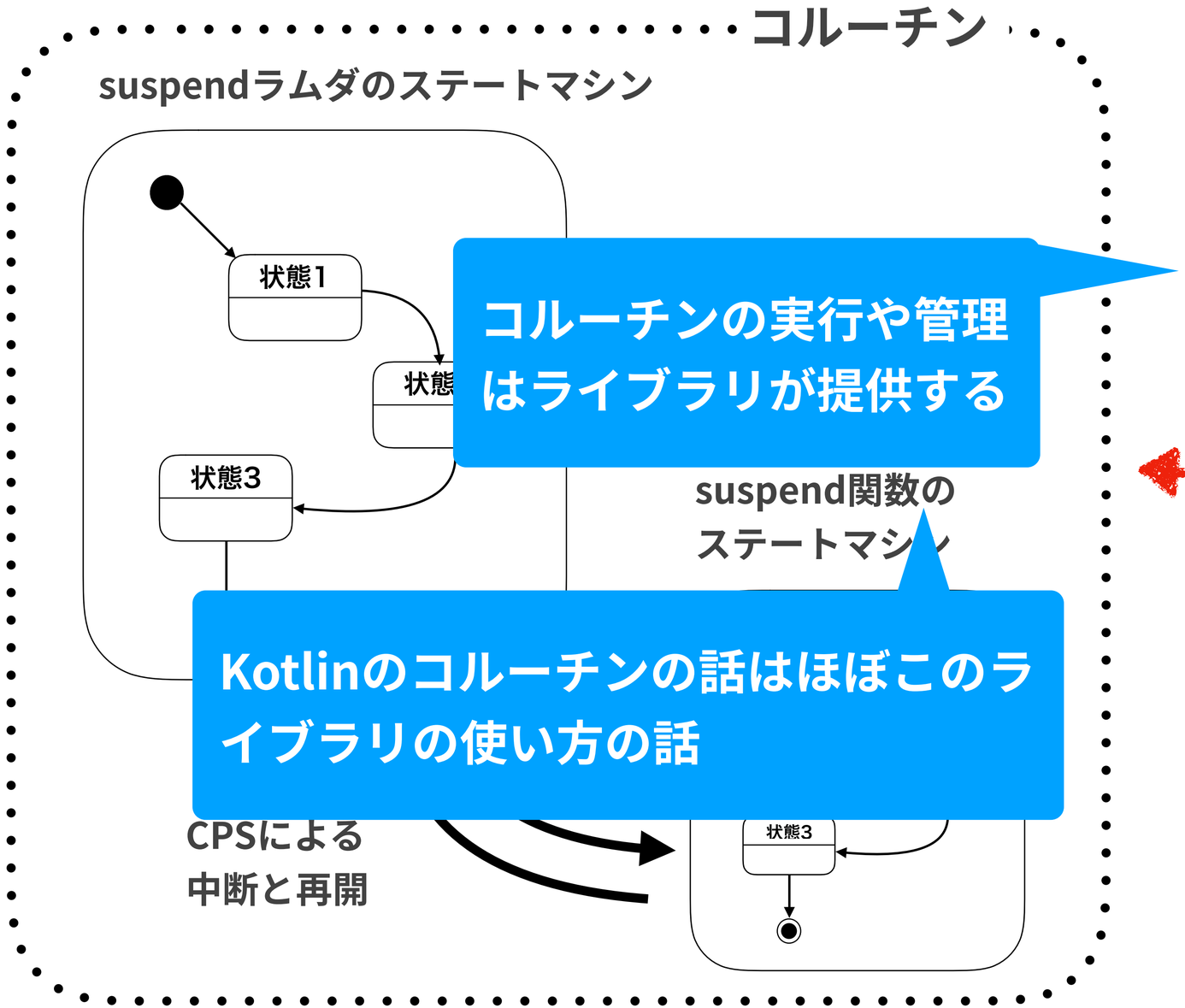
実行
キャンセル
スレッドプール
割り込み
例外ハンドラ
スコープ etc...

Kotlinにおけるコルーチンの仕組み

Kotlinプログラム

```
fun main() {
    GlobalScope.launch { this: CoroutineScope
        val dataSource = ProfileDataSource()
        showLoading()
        dataSource.loadProfile( id: 10)
        hideLoading()
    }
}
```

コンパイル



コルーチンの実行や管理はライブラリが提供する

コルーチンライブラリ

CoroutineDispatcher
launch, async
Job, Deferred
CoroutineScope
CoroutineStart...

実行
キャンセル
スレッドプール
割り込み
例外ハンドラ
スコープ etc...

Kotlinのコルーチンの話はほぼこのライブラリの使い方の話

Kotlinにおけるコルーチンの仕組み

- suspendキーワードを導入し、中断を表現するようにした
- suspendラムダがコルーチンの本体になる。suspendラムダは通常の関数の引数になる
- Kotlin自体はコルーチンのオブジェクトを作るところまで
- コルーチンライブラリが実行などを管理する仕組みを提供する

Kotlinコルーチンのきほん

コルーチンライブラリを導入する

build.gradle

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.0"
```

https://github.com/Kotlin/kotlinx.coroutines

コルーチンライブラリを導入する

build.gradle

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.0"
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.0"
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-play-services:1.3.0"
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-rx2:1.3.0"
```

https://github.com/Kotlin/kotlinx.coroutines

最初のコルーチン

```
fun main() {
  GlobalScope.launch {
    delay(100)
    println("World!")
  }
  println("Hello,")
  Thread.sleep(200)
}
```

最初のコルーチン

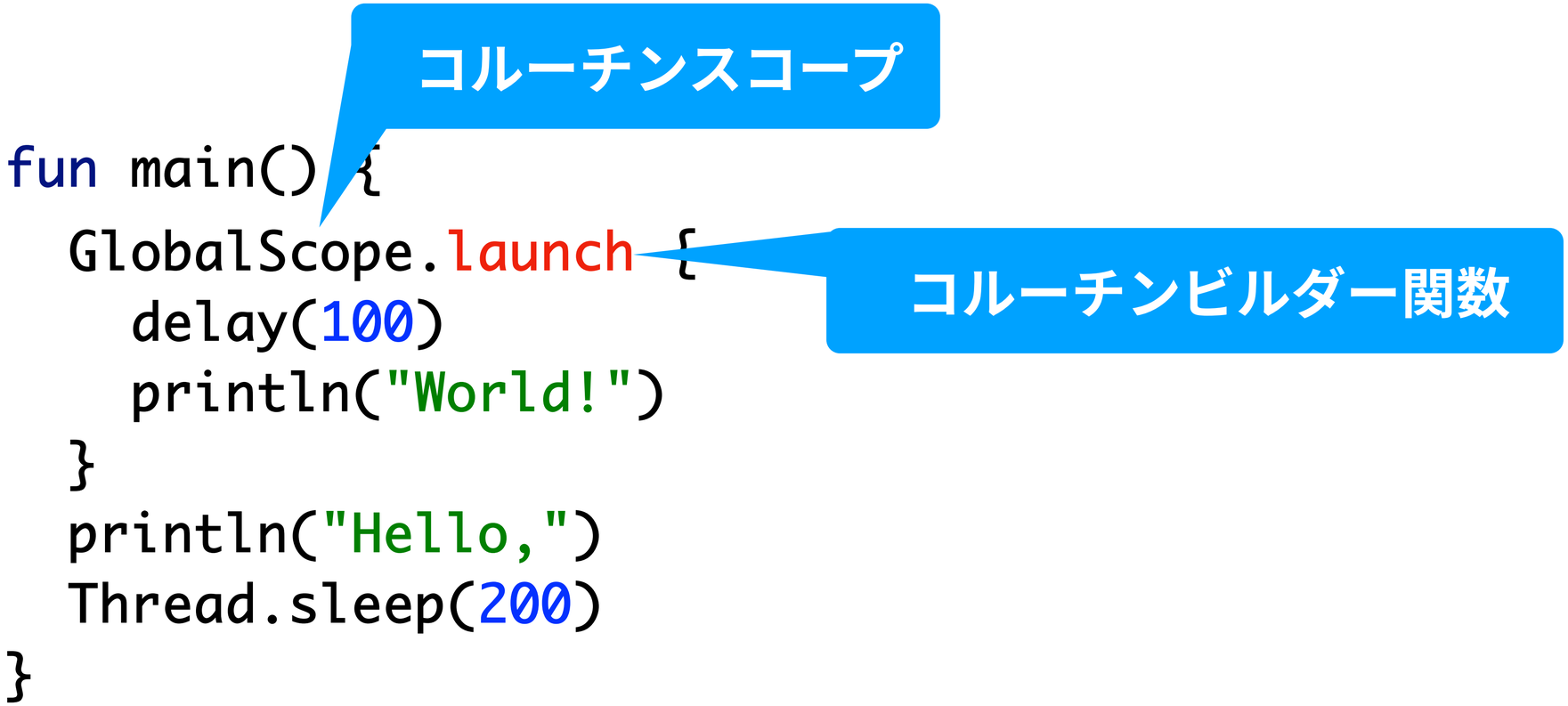
```
fun main() {
  GlobalScope.launch {
    delay(100)
    println("World!")
  }
  println("Hello,")
  Thread.sleep(200)
}
```

最初のコルーチン

コルーチンスコープ

```
fun main() {
  GlobalScope.launch {
    delay(100)
    println("World!")
  }
  println("Hello,")
  Thread.sleep(200)
}
```

最初のコルーチン




```
fun main() {
  GlobalScope.launch {
    delay(100)
    println("World!")
  }
  println("Hello,")
  Thread.sleep(200)
}
```

最初のコルーチン

```
fun main() {
    GlobalScope.launch {
        delay(100)
        println("World!")
    }
    println("Hello,")
    Thread.sleep(200)
}
```

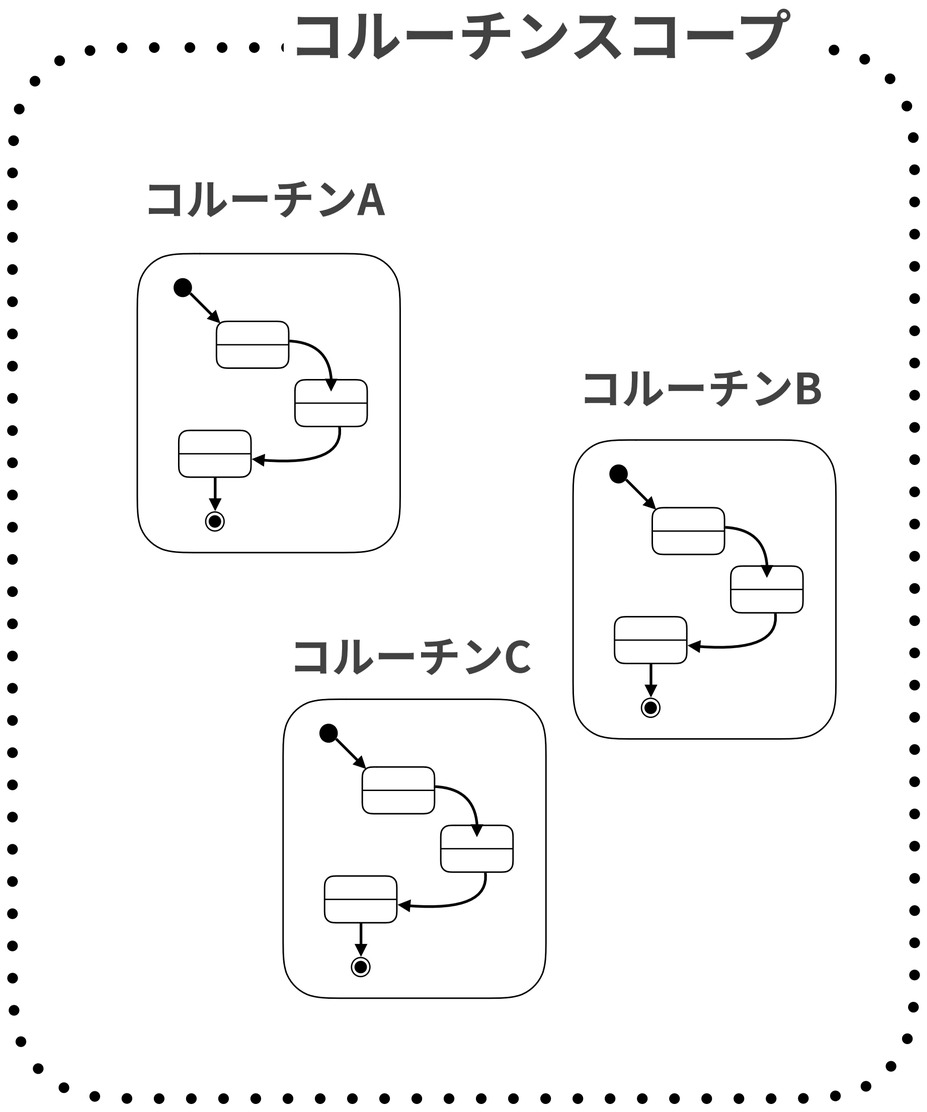
コルーチンスコープ

コルーチンビルダー関数

ライブラリが提供するのsuspend関数

コルーチンスコープ

- コルーチンが動作する範囲を表す
- コルーチンスコープの中でのみコルーチンを起動できる
- 複数のコルーチンを持てる
- ライフサイクルがあり、スコープを閉じると、中のコルーチンも閉じられる



コルーチンビルダー関数

- コルーチンを起動する関数
- 実行コンテキスト、開始方法、suspendラムダを渡してコルーチンを起動できる
- ノンブロッキングで動作する
- 結果を返さないlaunch関数と、結果を返すasync関数がある。

```
public fun CoroutineScope.launch(
    context: CoroutineContext
    start: CoroutineStart
    block: suspend CoroutineScope.() -> Unit
): Job

public fun <T> CoroutineScope.async(
    context: CoroutineContext
    start: CoroutineStart
    block: suspend CoroutineScope.() -> T
): Deferred<T>
```

suspend関数

- 中断を表す関数
- suspend関数はsuspend関数からしか呼び出せない
- 関数にsuspendキーワードを付与することで宣言できる

```
public suspend fun delay(timeMillis: Long)
```

最初のコルーチン

```
fun main() {
  GlobalScope.launch {
    delay(100)
    println("World!")
  }
  println("Hello,")
  Thread.sleep(200)
}
```

最初のコルーチン

```
fun main() {
  GlobalScope.launch { ……… ①
    delay(100)
    println("World!")
  }
  println("Hello,")
  Thread.sleep(200)
}
```

最初のコルーチン

```
fun main() {
  GlobalScope.launch { ……… ①
    delay(100)
    println("World!")
  }
  println("Hello,")   ……… ②
  Thread.sleep(200)
}
```

最初のコルーチン

```
fun main() {
  GlobalScope.launch { ……… ①
    delay(100)          ……… ③
    println("World!")
  }
  println("Hello,")   ……… ②
  Thread.sleep(200)
}
```

最初のコルーチン

```
fun main() {
  GlobalScope.launch { ……… ①
    delay(100)          ……… ③
    println("World!")  ……… ④
  }
  println("Hello,")   ……… ②
  Thread.sleep(200)
}
```

最初のコルーチン

```
fun main() {
  GlobalScope.launch { ……… ①
    delay(100)          ……… ③
    println("World!")  ……… ④
  }
  println("Hello,")   ……… ②
  Thread.sleep(200)   ……… ⑤
}
```

コルーチンとブロッキング

```
fun main() {
  GlobalScope.launch {
    delay(100)
    println("World!")
  }
  println("Hello,")
  Thread.sleep(200)
}
```

他の方法はないか…?

コルーチンとブロッキング

```
fun main() = runBlocking {
  GlobalScope.launch {
    delay(100)
    println("World!")
  }
  println("Hello,")
  delay(200)
}
```

コルーチンとブロッキング

```
fun main() = runBlocking {
  GlobalScope.launch {
    delay(100)
    println("World!")
  }
  println("Hello,")
  delay(200)
}
```

ブロッキングでコルーチンを実行できる

コルーチンとブロッキング

```
fun main() = runBlocking {
  GlobalScope.launch {
    delay(100)
    println("World!")
  }
  println("Hello,")
  delay(200)
}
```

コルーチンとブロッキング

```
fun main() = runBlocking {
  GlobalScope.launch {
    delay(100)
    println("World!")
  }
  println("Hello,")
  delay(200)
}
```

どちらにせよ待つ必要がある

コルーチンとブロッキング

```
fun main() = runBlocking {
  GlobalScope.launch {
    delay(100)
    println("World!")
  }
  println("Hello,")
  delay(200)
}
```

どちらにせよ待つ必要がある

launch関数の完了を待つ方法は他にないか？

コルーチンスコープと子コルーチン

```
fun main() = runBlocking { // this: CoroutineScope
  GlobalScope.launch {
    delay(100)
    println("World!")
  }
  println("Hello,")
  delay(200)
}
```

コルーチンスコープと子コルーチン

```
fun main() = runBlocking { // this: CoroutineScope
  GlobalScope.launch {
    delay(100)
    println("World!")
  }
  println("Hello,")
  delay(200)
}
```

block内はCoroutineScopeがレシーバ

コルーチンスコープと子コルーチン

```
fun main() = runBlocking { // this: CoroutineScope
  GlobalScope.launch {
    delay(100)
    println("W     d!")
  }
  println("Hell
  delay(200)
}
```

トップレベルのスコープ。アプリケーション（プロセス）のライフサイクルで動作する

コルーチンスコープと子コルーチン

```
fun main() = runBlocking { // this: CoroutineScope
  GlobalScope.launch {
    delay(100)
    println("World!")
  }
  println("Hello,")
  delay(200)
}
```

このコルーチンはrunBlockingのスコープに属さないため、明示的に終了を待つ必要がある

コルーチンスコープと子コルーチン

```
fun main() = runBlocking { // this: CoroutineScope
  this.launch {
    delay(100)
    println("World!")
  }
  println("Hello,")
  delay(200)
}
```

コルーチンスコープと子コルーチン

```
fun main() = runBlocking { // this: CoroutineScope
  this.launch {
    delay(100)
    println("World!")
  }
  println("Hello,")
  delay(200)
}
```

runBlockingの子コルーチンとして起動する

コルーチンスコープと子コルーチン

```
fun main() = runBlocking { // this: CoroutineScope
  launch {
    delay(100)
    println("World!")
  }
  println("Hello,")
}
```

コルーチンスコープと子コルーチン

```
fun main() = runBlocking { // this: CoroutineScope
  launch {
    delay(100)
    println("World!")
  }
  println("Hello,")
}
```

スコープは子コルーチンがすべて完了するまで待ってくれる

コルーチンのキャンセル

```
fun main() = runBlocking { // this: CoroutineScope
  launch {
    delay(100)
    println("World!")
  }
  println("Hello,")
}
```

コルーチンのキャンセル

```
fun main() = runBlocking { // this: CoroutineScope
  launch {
    while(true) {
      delay(100)
      println("World!")
    }
  }
  println("Hello,")
}
```

コルーチンのキャンセル

```
fun main() = runBlocking { // this: CoroutineScope
  launch {
    while(true) {
      delay(100)
      println("World!")
    }
  }
  println("Hello,")
}
```

時間経過では終了しないコルーチン

コルーチンのキャンセル

```
fun main() = runBlocking { // this: CoroutineScope
  launch {
    while(true) {
      delay(100)
      println("World!")
    }
  }
  println("Hello,")
}
```

ずっと待ってしまう

コルーチンのキャンセル

```
fun main() = runBlocking { // this: CoroutineScope
  val job = launch {
    while(true) {
      delay(100)
      println("World!")
    }
  }
  println("Hello,")
}
```

```
public fun CoroutineScope.launch(
    context: CoroutineContext
    start: CoroutineStart
    block: suspend CoroutineScope.() -> Unit
): Job
```

コルーチンのキャンセル

```
fun main() = runBlocking { // this: CoroutineScope
  val job = launch {
    while(true) {
      delay(100)
      println("World!")
    }
  }
  println("Hello,")
  delay(500)
  job.cancel()
}
```

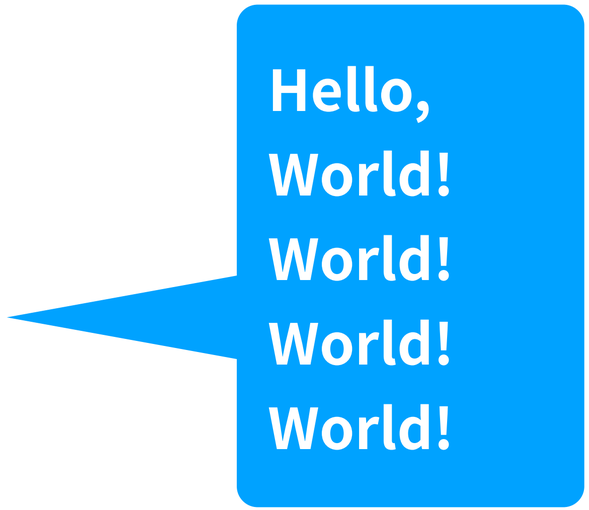
コルーチンのキャンセル

```
fun main() = runBlocking { // this: CoroutineScope
  val job = launch {
    while(true) {
      delay(100)
      println("World!")
    }
  }
  println("Hello,")
  delay(500)
  job.cancel()
}
```

jobを使ってキャンセルできる

コルーチンのキャンセル

```
fun main() = runBlocking { // this: CoroutineScope
  val job = launch {
    while(true) {
      delay(100)
      println("World!")
    }
  }
  println("Hello,")
  delay(500)
  job.cancel()
}
```



Kotlinコルーチンのきほん①

- コルーチンスコープはコルーチンが動作する範囲で、複数のコルーチンを起動できる
- コルーチンを起動するためのコルーチンビルダー関数（launch, async）があり、実行コンテキスト、開始方法とsuspendラムダを渡して使う
- コルーチンをブロッキングで実行するためのrunBlocking関数がある
- コルーチンスコープは子コルーチンの完了を待つ
- 時間経過で完了するコルーチンと、完了しないコルーチンがある。
- Job#cancel関数を使って明示的にコルーチンをキャンセルできる。

suspend関数と実行コンテキスト

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile {
        return apiClient.getProfile(id)
    }
}
```

suspend関数と実行コンテキスト

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile {
        return apiClient.getProfile(id)
    }
}
```

APIにリクエストするクライアント

suspend関数と実行コンテキスト

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile {
        return apiClient.getProfile(id)
    }
}
```

ブロッキングでリクエストする関数

suspend関数と実行コンテキスト

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile {
        return apiClient.getProfile(id)
    }
}
```

この関数はどのスレッドで動作するか？

suspend関数と実行コンテキスト

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile {
        return apiClient.getProfile(id)
    }
}
```

この関数はどのスレッドで動作するか？

関数を呼び出したスレッドで動作する

suspend関数と実行コンテキスト

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile = withContext(Dispatchers.IO) {
        apiClient.getProfile(id)
    }
}
```

suspend関数と実行コンテキスト

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile = withContext(Dispatchers.IO) {
        apiClient.getProfile(id)
    }
}
```

実行コンテキストを切り替える関数

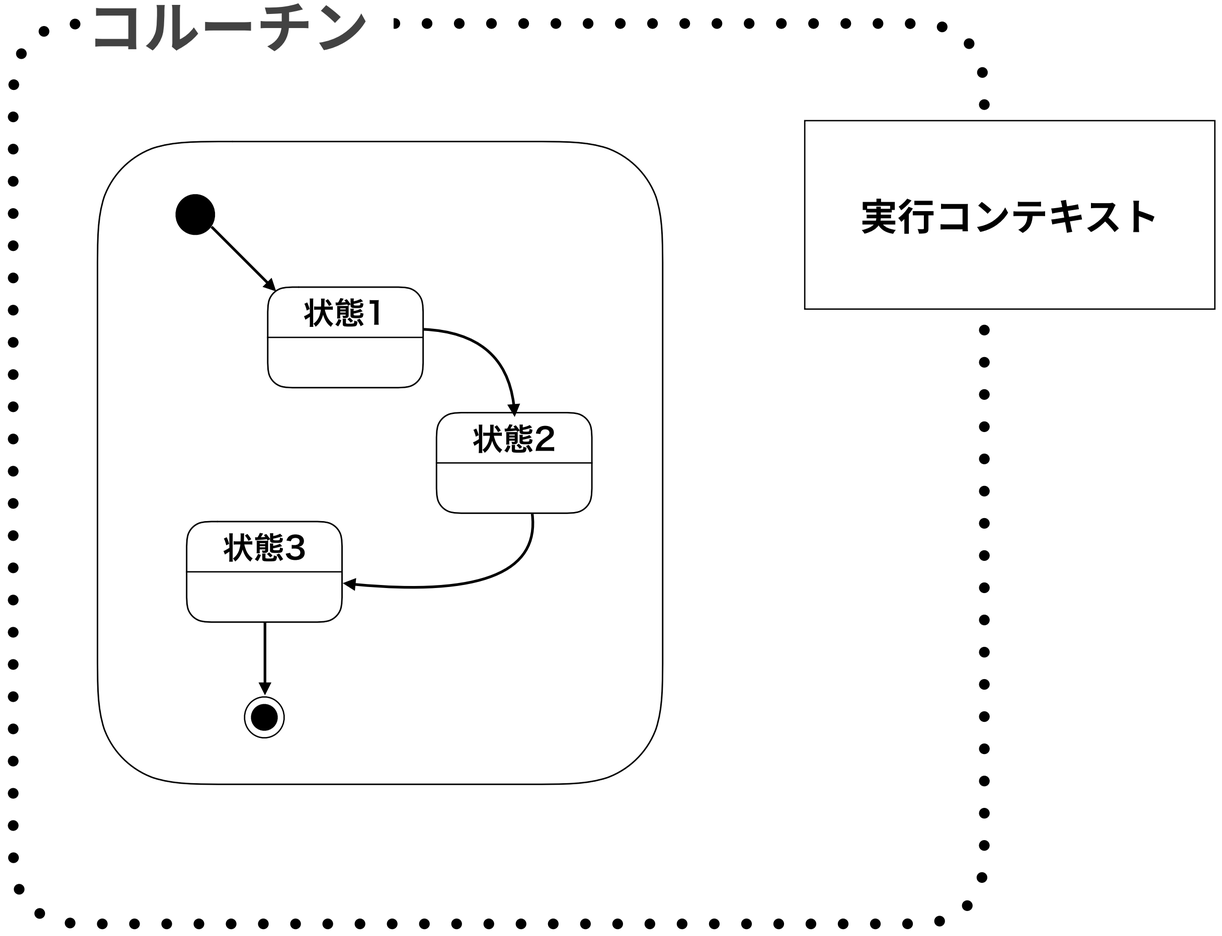
suspend関数と実行コンテキスト

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile = withContext(Dispatchers.IO) {
        apiClient.getProfile(id)
    }
}
```

コルーチンディスパッチャーの定数

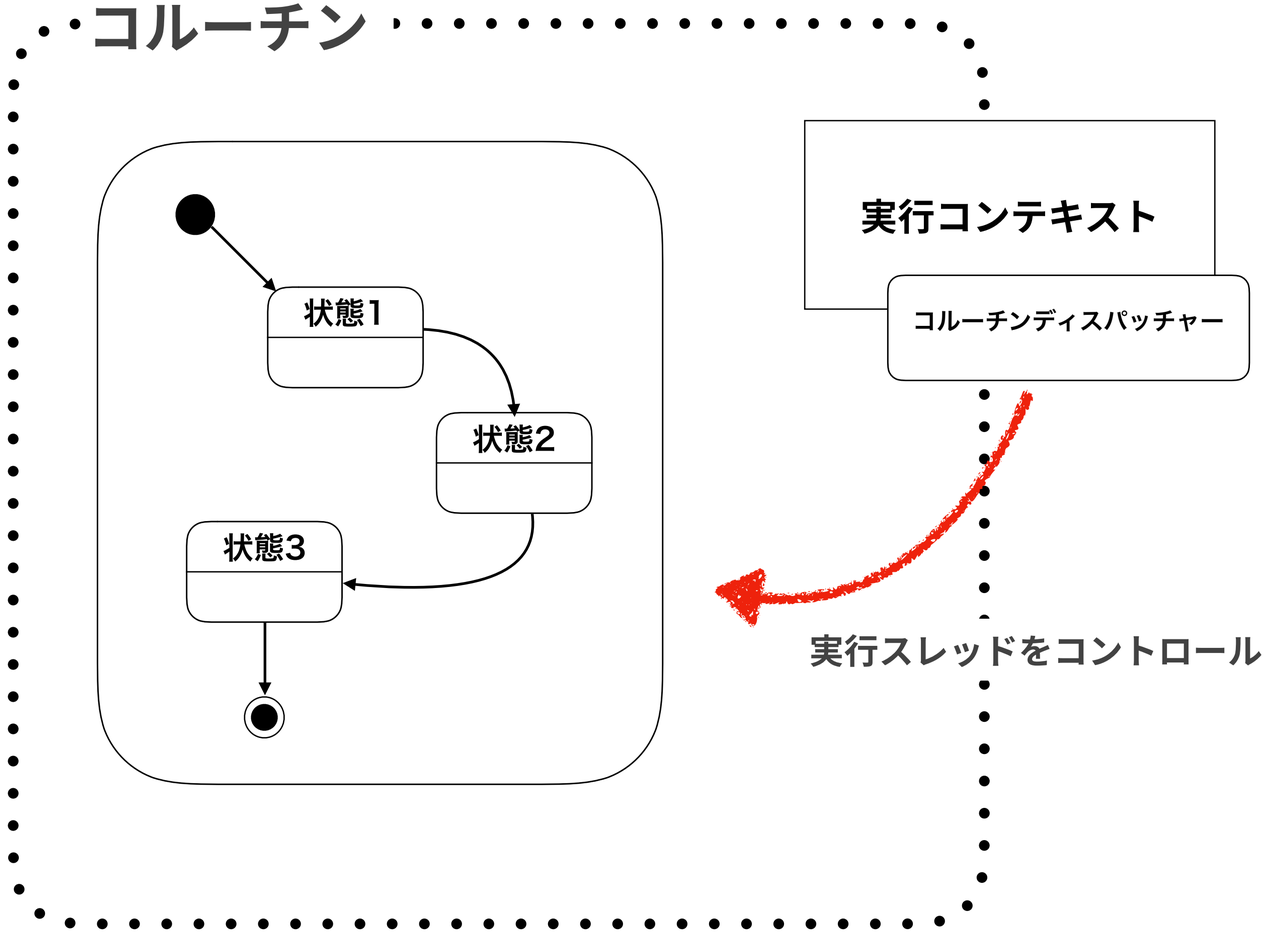
実行コンテキスト

- コルーチンの中で共有する要素のセット
- Job、コルーチンインターセプター、例外ハンドラなどコルーチンの動作に必要な要素を含んでいる



コルーチンディスパッチャー

- コルーチンインターセプターの実装
- コルーチンをどのスレッドで実行するかを選択できる
- 定数として4種類提供している
  - Default: CPUバウンド
  - IO: I/Oバウンド
  - Main: メインスレッド
  - UnConfined: 制限なし（通常は使わない）



suspend関数と実行コンテキスト

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile = withContext(Dispatchers.IO) {
        apiClient.getProfile(id)
    }
}
```

suspend関数と実行コンテキスト

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile = withContext(Dispatchers.IO) {
        apiClient.getProfile(id)
    }
}
```

IOバウンドのスレッドプールで動作するように切り替える

suspend関数の呼び出し

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile = withContext(Dispatchers.IO) {
        println("loadProfile: ${Thread.currentThread().id}")
        apiClient.getProfile(id)
    }
}
```

suspend関数の呼び出し

```
fun main() = runBlocking {
    println("start: ${Thread.currentThread().id}")
    val dataSource = ProfileDataSource(ApiClient())
    val profile = dataSource.loadProfile(10)
    println("end: ${Thread.currentThread().id}")
}
```

suspend関数の呼び出し

```
fun main() = runBlocking {
    println("start: ${Thread.currentThread().id}")
    val dataSource = ProfileDataSource(ApiClient())
    val profile = dataSource.loadProfile(10)
    println("end: ${Thread.currentThread().id}")
}
```

start: 1
loadProfile: 8
end: 1

例外のハンドリング

```
fun main() = runBlocking {
    println("start: ${Thread.currentThread().id}")
    val dataSource = ProfileDataSource(ApiClient())
    val profile = dataSource.loadProfile(10)
    println("end: ${Thread.currentThread().id}")
}
```

例外のハンドリング

```
fun main() = runBlocking {
    println("start: ${Thread.currentThread().id}")
    val dataSource = ProfileDataSource(ApiClient())
    try {
      val profile = dataSource.loadProfile(10)
      println("end: ${Thread.currentThread().id}")
    } catch (e: Exception) {
      println("error")
    }
}
```

例外のハンドリング

```
fun main() = runBlocking {
    println("start: ${Thread.currentThread().id}")
    val dataSource = ProfileDataSource(ApiClient())
    try {
      val profile = dataSource.loadProfile(10)
      println("end: ${Thread.currentThread().id}")
    } catch (e: Exception) {
      println("error")
    }
}
```

基本的にsuspend関数をtry-catchで囲む形で良い

実際に近い例（Android）

```
class ProfileViewModel(private val dataSource: ProfileDataSource) : ViewModel() {
    sealed class ViewState { /* 略 */ }

    private val _viewState = MutableLiveData<ViewState>()
    val viewState: LiveData<ViewState> = _viewState

    fun loadProfile(id: Long) {
        viewModelScope.launch {
            try {
                _viewState.value = ViewState.Progress
                dataSource.loadProfile(id)
                _viewState.value = ViewState.Completed
            } catch (e: Exception) {
                _viewState.value = ViewState.Error(e)
            }
        }
    }
}
```

実際に近い例（Android）

```
class ProfileViewModel(private val dataSource: ProfileDataSource) : ViewModel() {
    sealed class ViewState { /* 略 */ }

    private val _viewState = MutableLiveData<
    val viewState: LiveData<ViewState> = _vie

    fun loadProfile(id: Long) {
        viewModelScope.launch {
            try {
                _viewState.value = ViewState.Progress
                dataSource.loadProfile(id)
                _viewState.value = ViewState.Completed
            } catch (e: Exception) {
                _viewState.value = ViewState.Error(e)
            }
        }
    }
}
```

JetpackのViewModel

実際に近い例（Android）

```
class ProfileViewModel(private val dataSource: ProfileDataSource) : ViewModel() {
    sealed class ViewState { /* 略 */ }
    private val _viewState = MutableLiv
    val viewState: LiveData<ViewState>

    fun loadProfile(id: Long) {
        viewModelScope.launch {
            try {
                _viewState.value = ViewState.Progress
                dataSource.loadProfile(id)
                _viewState.value = ViewState.Completed
            } catch (e: Exception) {
                _viewState.value = ViewState.Error(e)
            }
        }
    }
}
```

ActivityやFragmentから呼び出す

実際に近い例（Android）

```
class ProfileViewModel(private val dataSource: ProfileDataSource) : ViewModel() {
    sealed class ViewState { /* 略 */ }

    private val _viewState = MutableLiveData<ViewState>()
    val viewState: LiveData<ViewState> =

    fun loadProfile(id: Long) {
        viewModelScope.launch {
            try {
                _viewState.value = ViewState.Progress
                dataSource.loadProfile(id)
                _viewState.value = ViewState.Completed
            } catch (e: Exception) {
                _viewState.value = ViewState.Error(e)
            }
        }
    }
}
```

lifecycle-viewmodel-ktx:2.2.0-alpha3
が提供するViewModelのコルーチンスコープ

実際に近い例（Android）

```
class ProfileViewModel(private val dataSource: ProfileDataSource) : ViewModel() {
    sealed class ViewState { /* 略 */ }

    private val _viewState = MutableLiveData<ViewState>()
    val viewState: LiveData<ViewState> = _viewState

    fun loadProfile(id: Long) {
        viewModelScope.launch {
            try {
                _viewState.value = ViewState.Progress
                dataSource.loadProfile(id)
                _viewState.value = ViewState.Completed
            } catch (e: Exception) {
                _viewState.value = ViewState.Error(e)
            }
        }
    }
}
```

メインスレッドで動作する

実際に近い例（Android）

```
class ProfileViewModel(private val dataSource: ProfileDataSource) : ViewModel() {
    sealed class ViewState { /* 略 */ }

    private val _viewState = MutableLiveData<ViewState>()
    val viewState: LiveData<ViewState> = _viewState

    fun loadProfile(id: Long) {
        viewModelScope.launch {
            try {
                _viewState.value = ViewState.Progress
                dataSource.loadProfile(id)
                _viewState.value = ViewState.Completed
            } catch (e: Exception) {
                _viewState.value
            }
        }
    }
}
```

この関数はDispatchers.IOで動作する

実際に近い例（Android）

```
class ProfileViewModel(private val dataSource: ProfileDataSource) : ViewModel() {
    sealed class ViewState { /* 略 */ }

    private val _viewState = MutableLiveData<ViewState>()
    val viewState: LiveData<ViewState> = _viewState

    fun loadProfile(id: Long) {
        viewModelScope.launch {
            try {
                _viewState.value = ViewState.Progress
                dataSource.loadProfile(id)
                _viewState.value = ViewState.Completed
            } catch (e: Exception) {
                _viewState.value = ViewState.Error(e)
            }
        }
    }
}
```

Kotlinコルーチンのきほん②

- コルーチンが動作するための要素セットとして実行コンテキストがある
- 用途ごとにスレッドプールなどをコルーチンディスパッチャーとして提供している
- withContext関数で実行コンテキストを切り替えられる
- 例外ハンドリングは基本的にtry-catchで行う
- 非同期処理のほとんどはこれらの組み合わせで対応できる

コルーチンスコープと構造化された並行性

直列な動作

```
fun loadContent() {
  viewModelScope.launch {
    showProgress()
    val id = 10L
    val profile = dataSource.loadProfile(id)
    val articles = dataSource.loadArticles(id)
    showContent(profile, articles)
    hideProgress()
  }
}
```

直列な動作

```
fun loadContent() {
  viewModelScope.launch {
    showProgress()
    val id = 10L
    val profile = dataSource.loadProfile(id)  --- 1
    val articles = dataSource.loadArticles(id)  --- 2
    showContent(profile, articles)
    hideProgress()
  }
}
```

順次、IOバウンドのスレッドプールで実行する

直列な動作

```
fun loadContent() {
  viewModelScope.launch {
    showProgress()
    val id = 10L
    val profile = dataSource.loadProfile(id)  --- 1
    val articles = dataSource.loadArticles(id)  --- 2
    showContent(profile, articles)
    hideProgress()
  }
}
```

順次、IOバウンドのスレッドプールで実行する

同時に実行するには？

並行な動作

```
fun loadContent() {
  viewModelScope.launch {
    showProgress()
    val id = 10L
    val profile: Deferred<Profile> = async { dataSource.loadProfile(id) }
    val articles: Deferred<List<Article>> = async { dataSource.loadArticles(id) }
    showContent(profile.await(), articles.await())
    hideProgress()
  }
}
```

並行な動作

```
fun loadContent() {
  viewModelScope.launch {
    showProgress()
    val id = 10L
    val profile: Deferred<Profile> = async { dataSource.loadProfile(id) }
    val articles: Deferred<List<Article>> = async { dataSource.loadArticles(id) }
    showContent(profile.await(), articles.await())
    hideProgress()
  }
}
```

Deferredを返し、コルーチンを実行する

並行な動作

```
fun loadContent() {
  viewModelScope.launch {
    showProgress()
    val id = 10L
    val profile: Deferred<Profile> = async { dataSource.loadProfile(id) }
    val articles: Deferred<List<Article>> = async { dataSource.loadArticles(id) }
    showContent(profile.await(), articles.await())
    hideProgress()
  }
}
```

Deferredから結果を取り出す

並行な動作

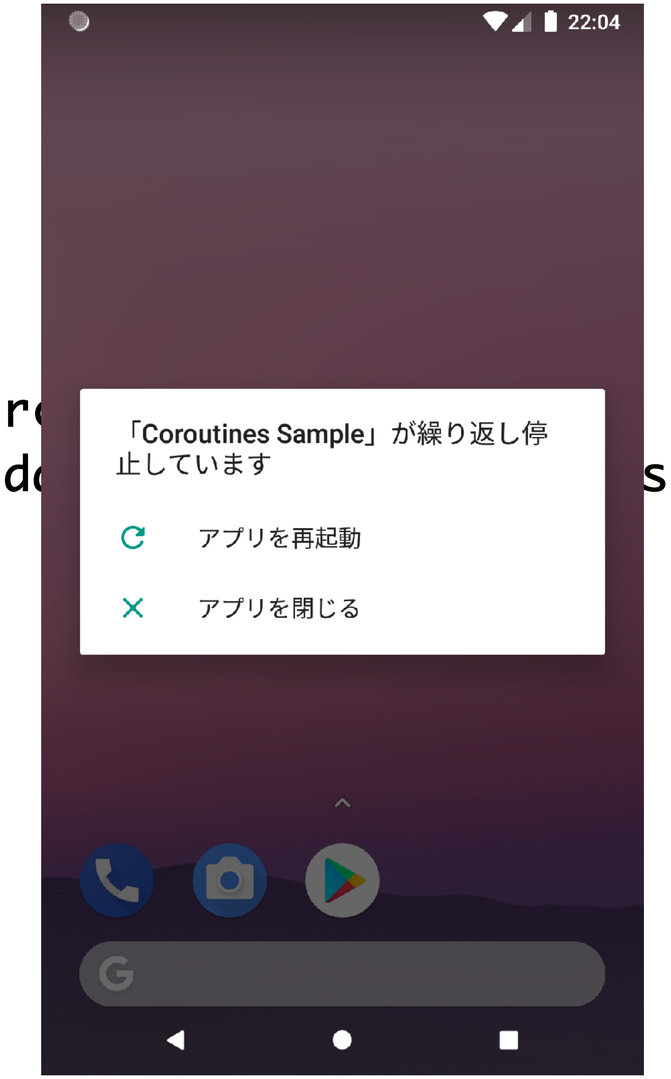
```
fun loadContent() {
  viewModelScope.launch {
    showProgress()
    val id = 10L
    val profile: Deferred<Profile> = async { dataSource.loadProfile(id) }
    val articles: Deferred<List<Article>> = async { dataSource.loadArticles(id) }
    showContent(profile.await(), articles.await())
    hideProgress()
  }
}
```

並行な動作

```
fun loadContent() {
  viewModelScope.launch {
    try {
      showProgress()
      val id = 10L
      val profile: Deferred<Profile> = async { dataSource.loadProfile(id) }
      val articles: Deferred<List<Article>> = async { dataSource.loadArticles(id) }
      showContent(profile.await(), articles.await())
    } catch (e: IOException) {
      showError(e)
    } finally {
      hideProgress()
    }
  }
}
```

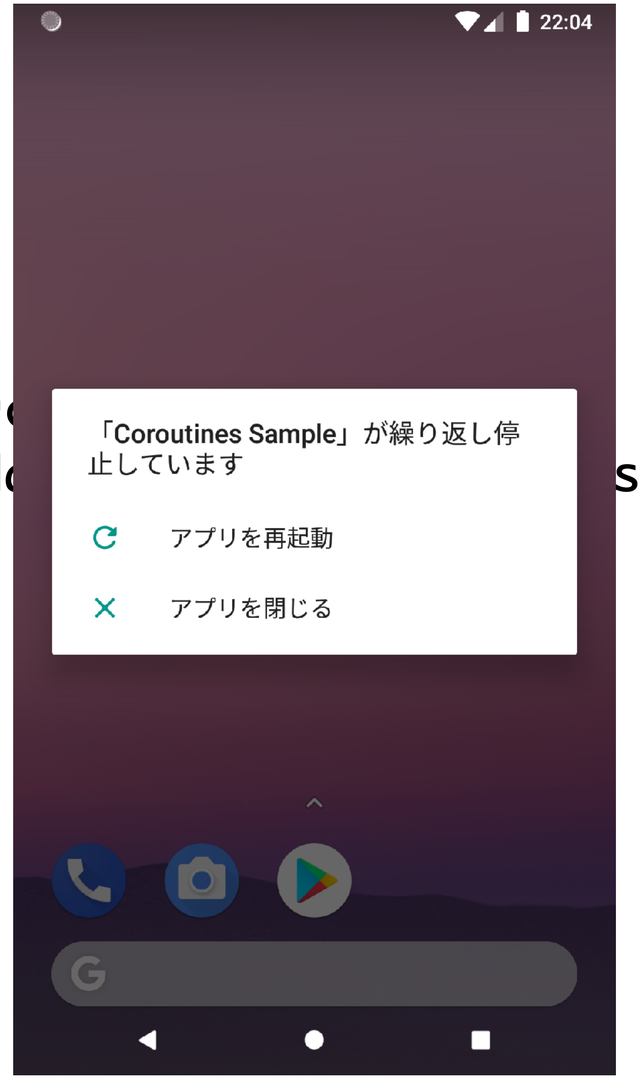
並行な動作

```
fun loadContent() {
  viewModelScope.launch {
    try {
      showProgress()
      val id = 10L
      val profile: Deferred<Profile> = async { dataSour
      val articles: Deferred<List<Article>> = async { d                    s(id) }
      showContent(profile.await(), articles.await())
    } catch (e: IOException) {
      showError(e)
    } finally {
      hideProgress()
    }
  }
}
```



並行な動作

```
fun loadContent() {
  viewModelScope.launch {
    try {
      showProgress()
      val id = 10L
      val profile: Deferred<Profile> = async { dataSour
      val articles: Deferred<List<Article>> = async { d                    s(id) }
      showContent(profile.await(), articles.await())
    } catch (e: IOException) {
      showError(e)
    } finally {
      hideProgress()
    }
  }
}
```



構造化された並行性

https://medium.com/@elizarov/structured-concurrency-722d765aa952

Structured concurrency

Roman Elizarov Follow

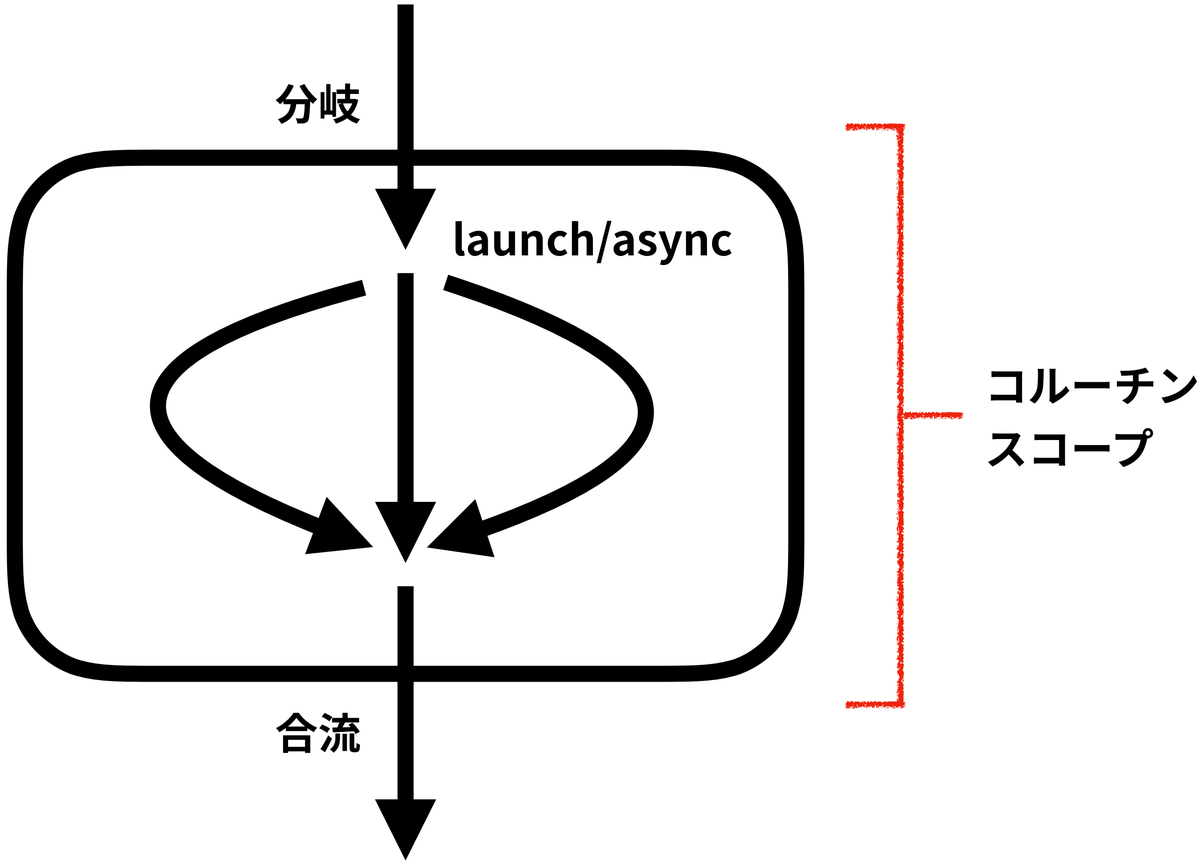
Sep 12 · 5 min read

Today marks the release of a version 0.26.0 of `kotlinx.coroutines` library and an introduction of *structured concurrency* to Kotlin coroutines. It is more than just a feature—it marks an ideology shift so big that I'm writing this post to explain it.

Since the initial rollout of Kotlin coroutines as an experimental feature in Kotlin 1.1 in the beginning of 2017 we've been working hard to explain the concept of coroutines to the programmers who used to think of concurrency in terms of threads, so our key analogy and motto was "**coroutines are light-weight threads**". Moreover, our key APIs were designed to be similar to thread APIs to ease the learning curve. This analogy works well in small-scale examples, but it does not help to explain the shift in programming style with coroutines.

構造化された並行性

- 並行処理の分岐を構造化するという考え方
- 分岐と合流の範囲を明示的に宣言することでエラーの範囲が決まる
- コルーチンスコープがひとつの分岐と合流を表す
- コルーチンスコープの親子関係で並行性を構造化する



並行性を構造化する

```
fun loadContent() {
  viewModelScope.launch {
    try {
      showProgress()
      val id = 10L
      val profile: Deferred<Profile> = async { dataSource.loadProfile(id) }
      val articles: Deferred<List<Article>> = async { dataSource.loadArticles(id) }
      showContent(profile.await(), articles.await())
    } catch (e: IOException) {
      showError(e)
    } finally {
      hideProgress()
    }
  }
}
```

並行性を構造化する

```
fun loadContent() {
  viewModelScope.launch {
    try {
      coroutineScope {
        showProgress()
        val id = 10L
        val profile: Deferred<Profile> = async { dataSource.loadProfile(id) }
        val articles: Deferred<List<Article>> = async { dataSource.loadArticles(id) }
        showContent(profile.await(), articles.await())
      }
    } catch (e: IOException) {
      showError(e)
    } finally {
      hideProgress()
    }
  }
}
```

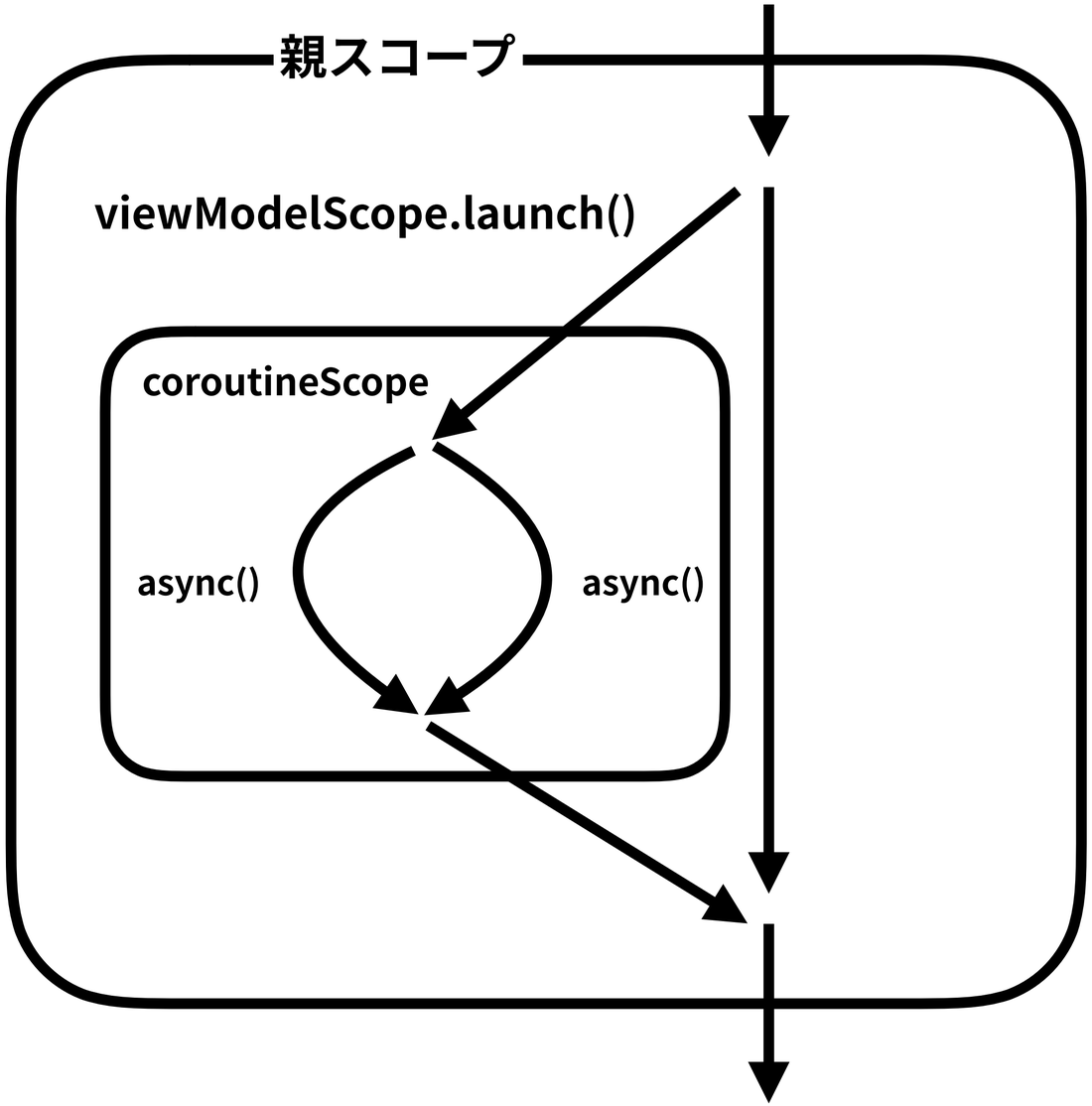
並行性を構造化する

```
fun loadContent() {
  viewModelScope.launch {
    try {
      coroutineScope {
        showProgress()
        val id = 10L
        val profile: Deferred<Profile> = async { dataSource.loadProfile(id) }
        val articles: Deferred<List<Article>> = async { dataSource.loadArticles(id) }
        showContent(profile.await(), articles.await())
      }
    } catch (e: IOException) {
      showError(e)
    } finally {
      hideProgress()
    }
  }
}
```

子スコープを作る関数

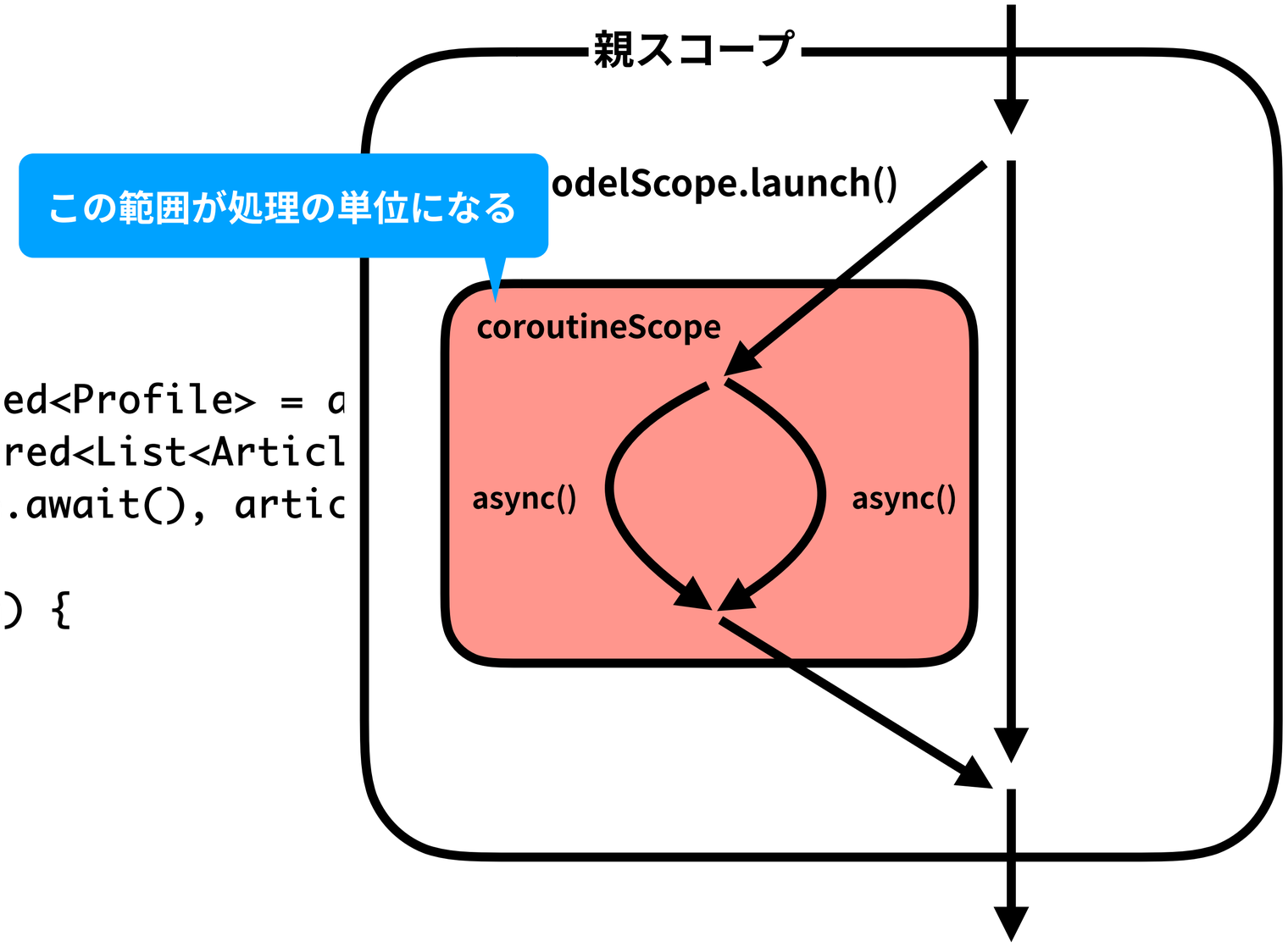
並行性を構造化する

```
fun loadContent() {
  viewModelScope.launch {
    try {
      coroutineScope {
        showProgress()
        val id = 10L
        val profile: Deferred<Profile> = a
        val articles: Deferred<List<Articl
        showContent(profile.await(), artic
      }
    } catch (e: IOException) {
      showError(e)
    } finally {
      hideProgress()
    }
  }
}
```



並行性を構造化する

```
fun loadContent() {
  viewModelScope.launch {
    try {
      coroutineScope {
        showProgress()
        val id = 10L
        val profile: Deferred<Profile> = a
        val articles: Deferred<List<Articl
        showContent(profile.await(), artic
      }
    } catch (e: IOException) {
      showError(e)
    } finally {
      hideProgress()
    }
  }
}
```

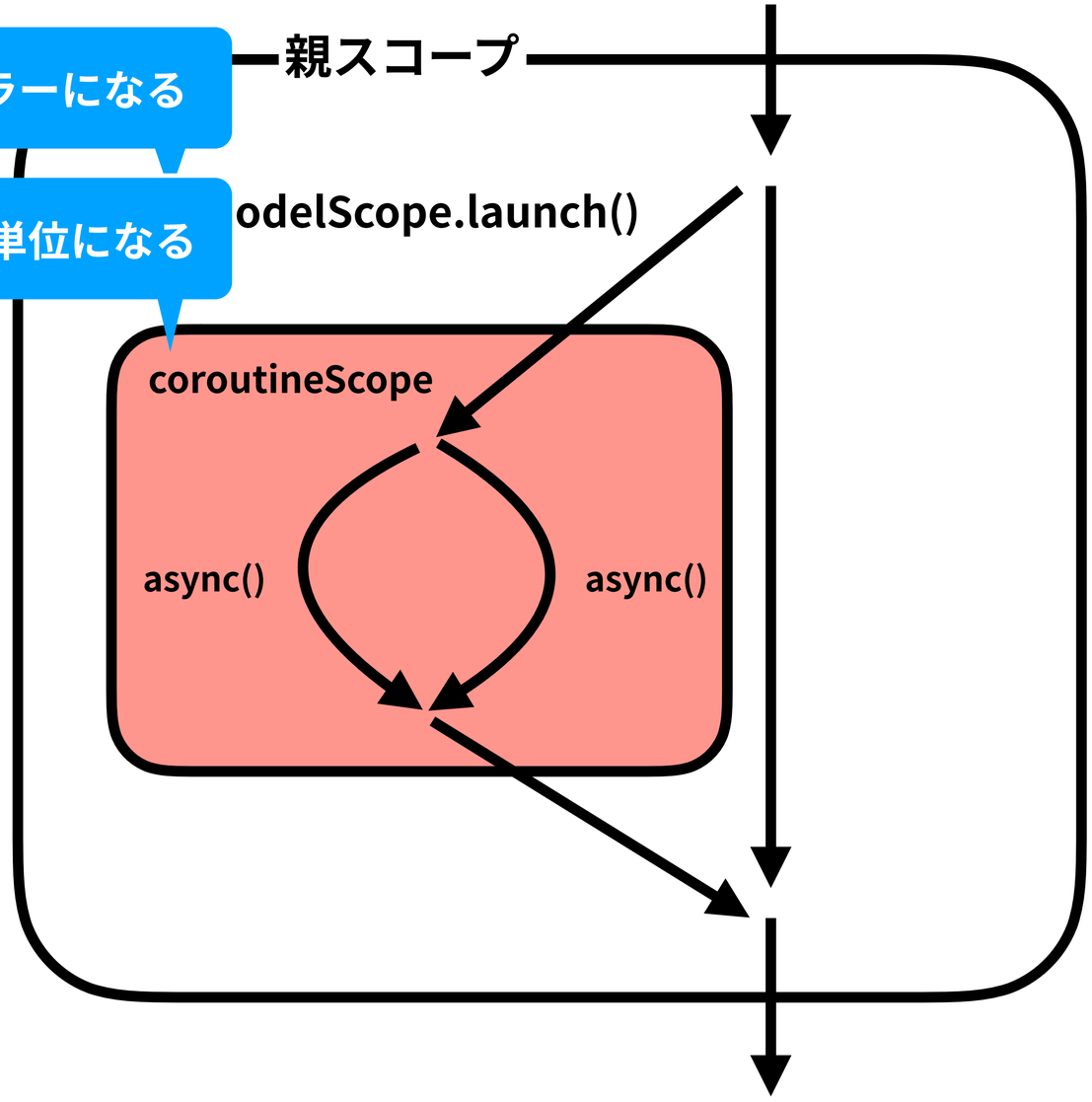


並行性を構造化する

```
fun loadContent() {
  viewModelScope.launch {
    try {
      coroutineScope {
        showProgress()
        val id = 10L
        val profile: Deferred<Profile> = a
        val articles: Deferred<List<Articl
        showContent(profile.await(), artic
      }
    } catch (e: IOException) {
      showError(e)
    } finally {
      hideProgress()
    }
  }
}
```

どれかが失敗すると全体がエラーになる

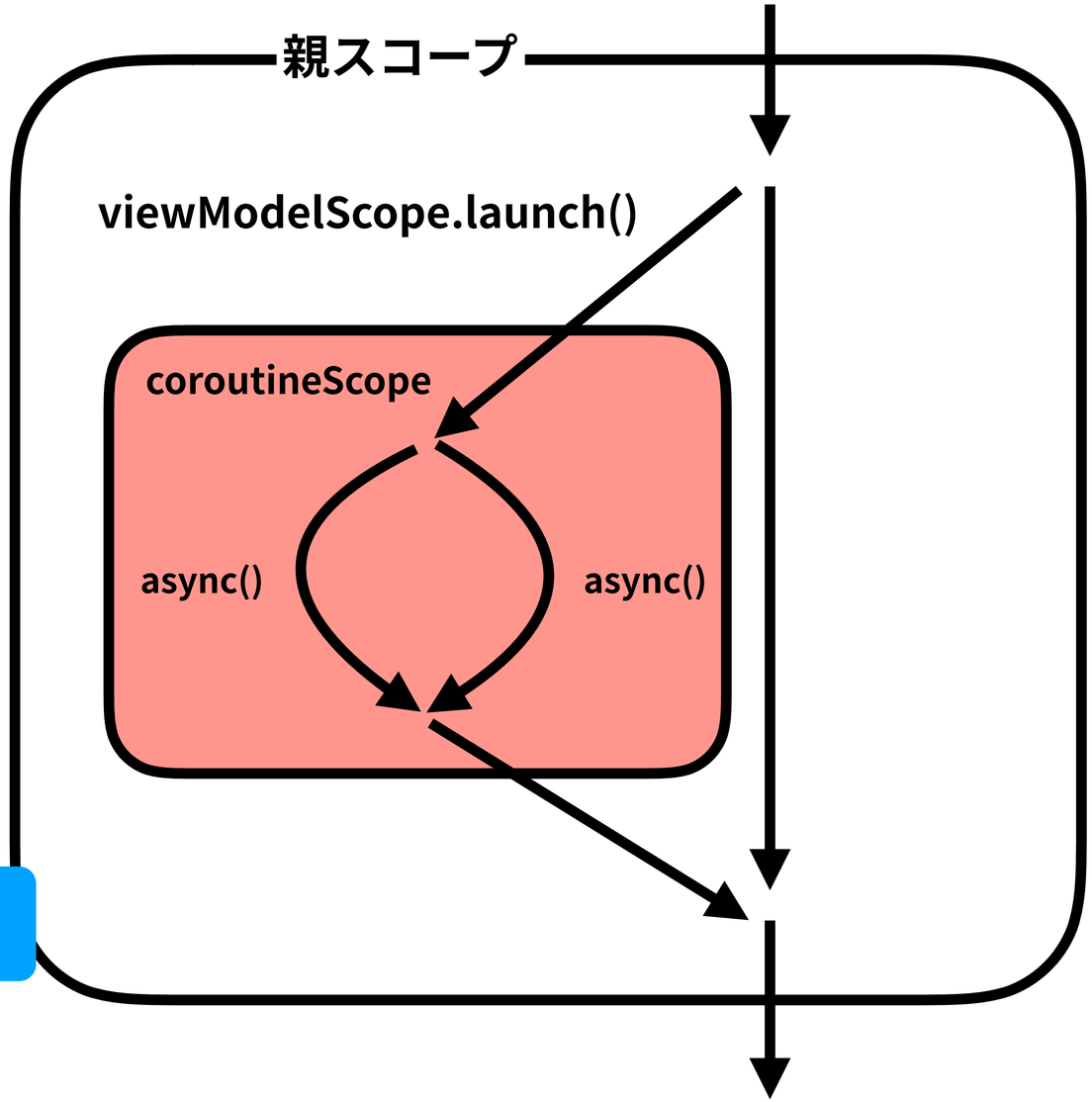
この範囲が処理の単位になる



並行性を構造化する

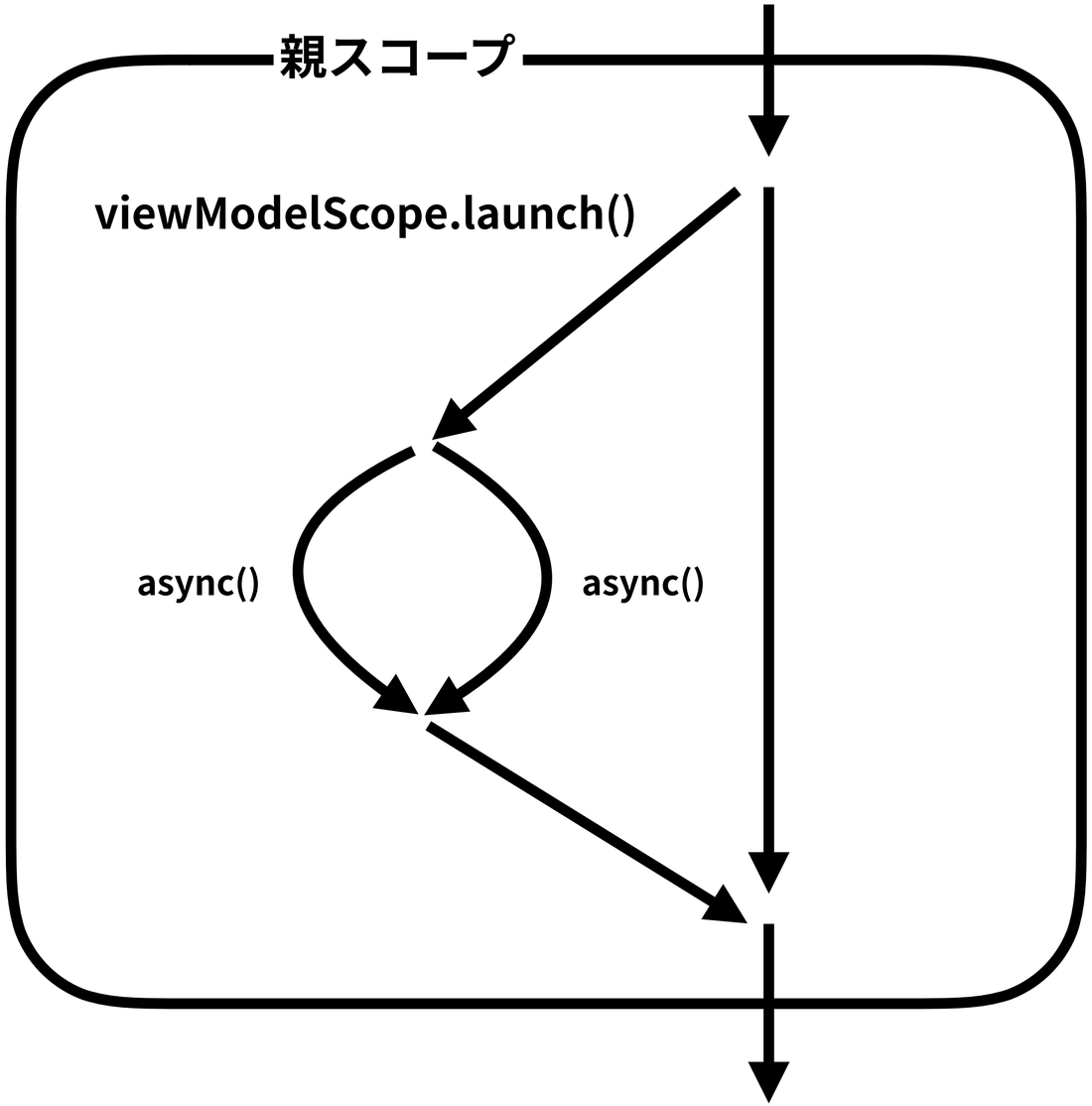
```
fun loadContent() {
  viewModelScope.launch {
    try {
      coroutineScope {
        showProgress()
        val id = 10L
        val profile: Deferred<Profile> = a
        val articles: Deferred<List<Articl
        showContent(profile.await(), artic
      }
    } catch (e: IOException) {
      showError(e)
    } finally {

    }
  }
}
```

スコープのエラーを透過的にハンドリングできる



並行性を構造化していなかったら

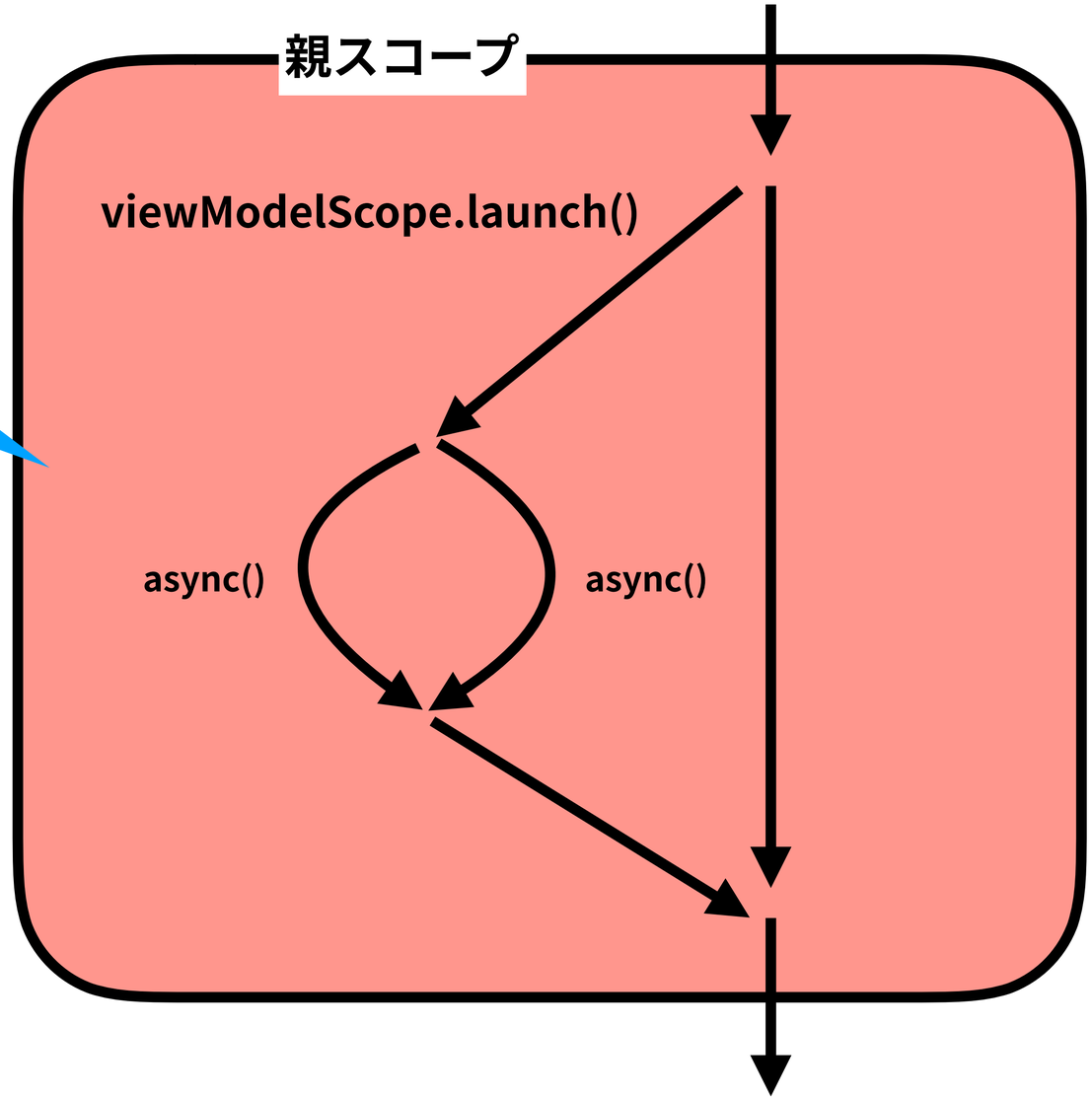
```
fun loadContent() {
  viewModelScope.launch {
    try {
      showProgress()
      val id = 10L
      val profile: Deferred<Profile> = asy
      val articles: Deferred<List<Article>
      showContent(profile.await(), article
    } catch (e: IOException) {
      showError(e)
    } finally {
      hideProgress()
    }
  }
}
```



並行性を構造化していなかったら

```
fun loadContent() {
  viewModelScope.launch {
    try {
      showPr
      val id
      val pr
      val articles: Deferred<List<Article>
      showContent(profile.await(), article
    } catch (e: IOException) {
      showError(e)
    } finally {
      hideProgress()
    }
  }
}
```

この範囲が処理の単位になるので、
エラーが発生すると、親スコープが終了する



コルーチンスコープと構造化された並行性

- コルーチンを直列で実行する場合は実行コンテキスト切り替え以外は基本的に気にしなくてよい
- コルーチンを分岐させる場合は並行性を構造化する必要がある
- 並行性の構造化とは分岐と合流の範囲を明示すること
- launch/asyncが分岐を表し、コルーチンスコープが範囲を表す
- coroutineScope関数を使って子スコープ（分岐と合流の範囲）を作ることができる
- 分岐と合流の範囲を構造化することで、処理とエラーの範囲を決められる

コルーチンと設計

コルーチンと設計

- コルーチンスコープとアプリケーションライフサイクルを合わせる
- suspend関数はメインセーフティで実装する
- コルーチンはメインセーフティで起動する

コルーチンと設計

- コルーチンスコープとアプリケーションライフサイクルを合わせる
- suspend関数はメインセーフティで実装する
- コルーチンはメインセーフティで起動する

コルーチンスコープとアプリケーションライフサイクルを合わせる

```
val scope = CoroutineScope(Dispatchers.Main + Job())

scope.launch { ... }

scope.coroutineContext.cancelChildren()
```

コルーチンスコープとアプリケーションライフサイクルを合わせる

```
val scope = CoroutineScope(Dispatchers.Main + Job())

scope.launch { ... }

scope.coroutineContext.cancelChildren()
```

実行コンテキストの設定とともに
コルーチンスコープを作れる

コルーチンスコープとアプリケーションライフサイクルを合わせる

```
val scope = CoroutineScope(Dispatchers.Main + Job())

scope.launch { ... }

scope.coroutineContext.cancelChildren()
```

スコープをつかってコルーチンを起動する

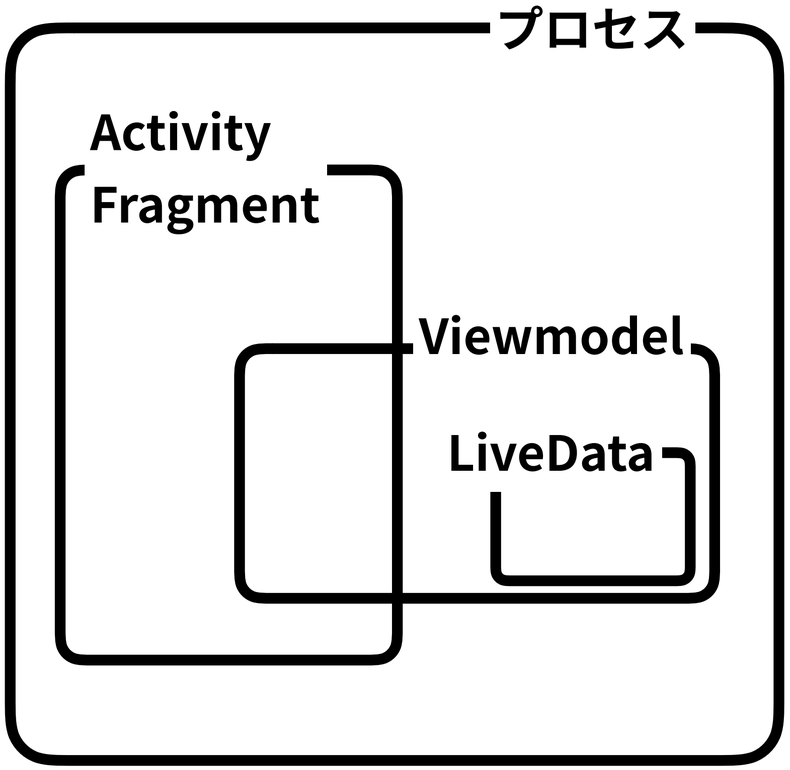
コルーチンスコープとアプリケーションライフサイクルを合わせる

```
val scope = CoroutineScope(Dispatchers.Main + Job())

scope.launch { ... }

scope.coroutineContext.cancelChildren()
```

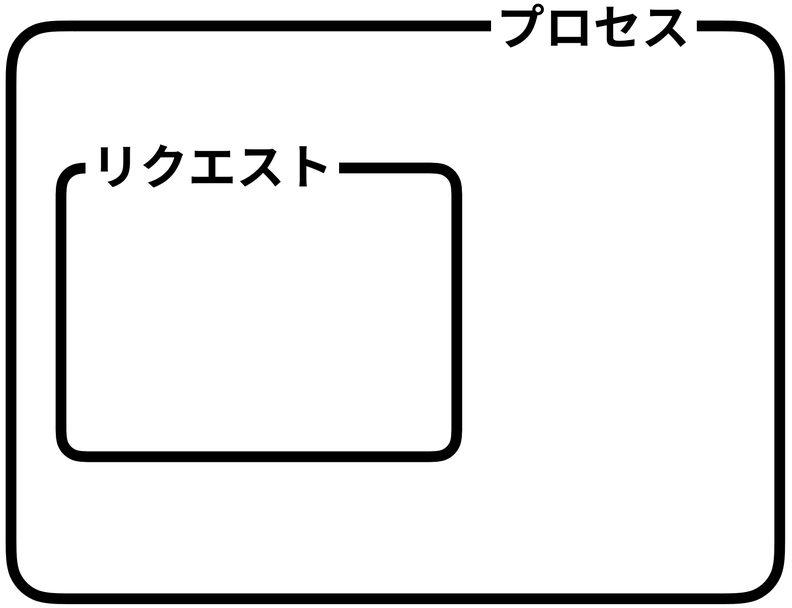
スコープを破棄するときにキャンセルしてリークを防ぐ

コルーチンスコープとアプリケーションライフサイクルを合わせる

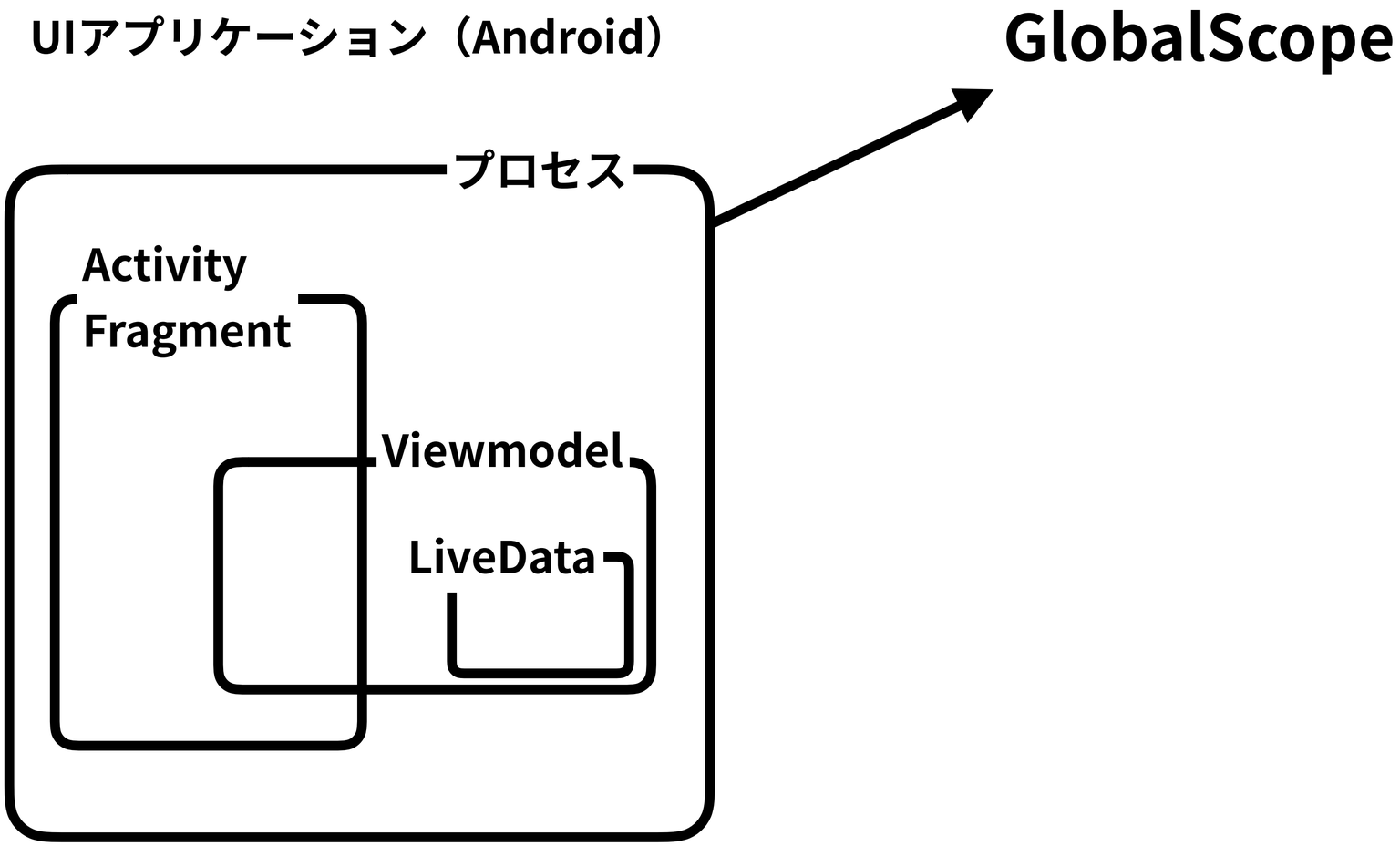
UIアプリケーション（Android）



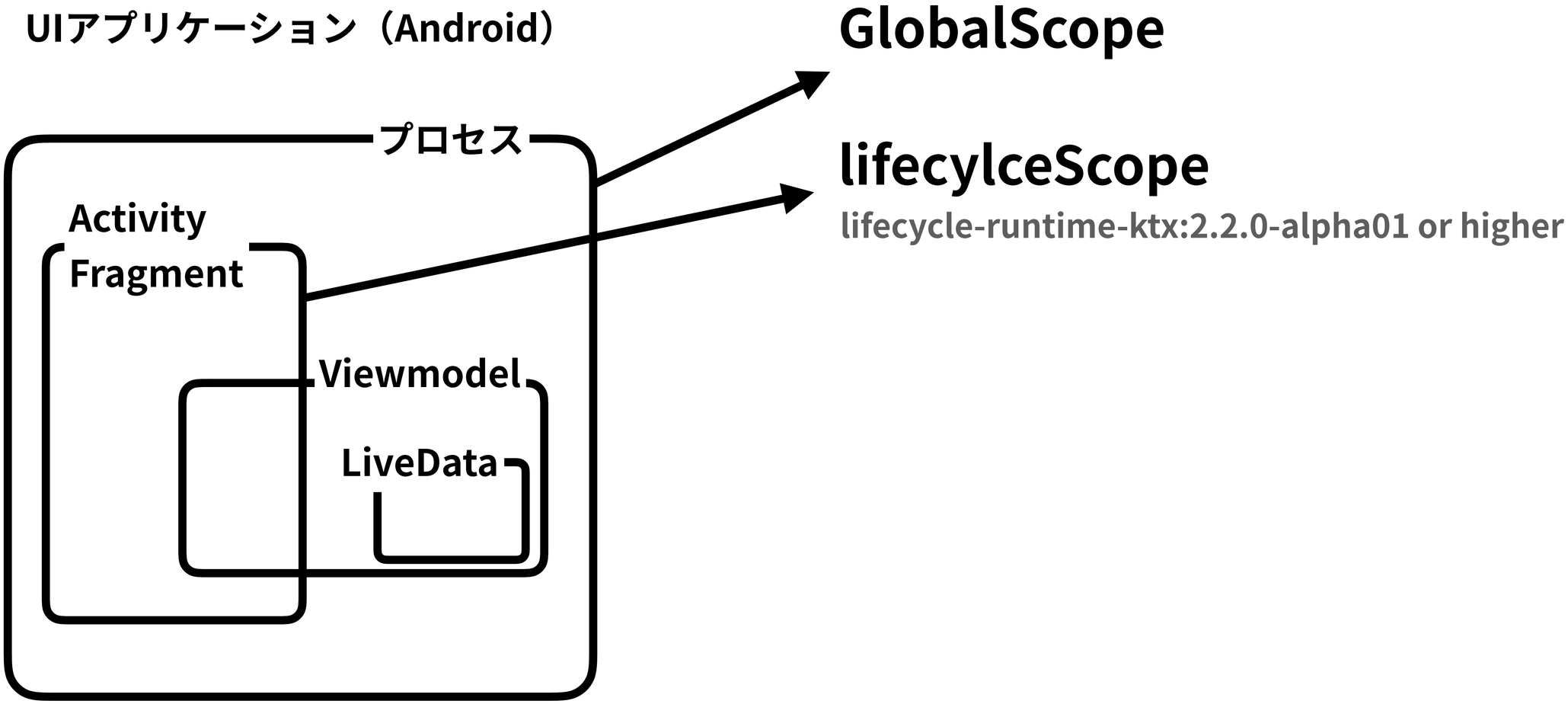
サーバーサイドアプリケーション（ktor）



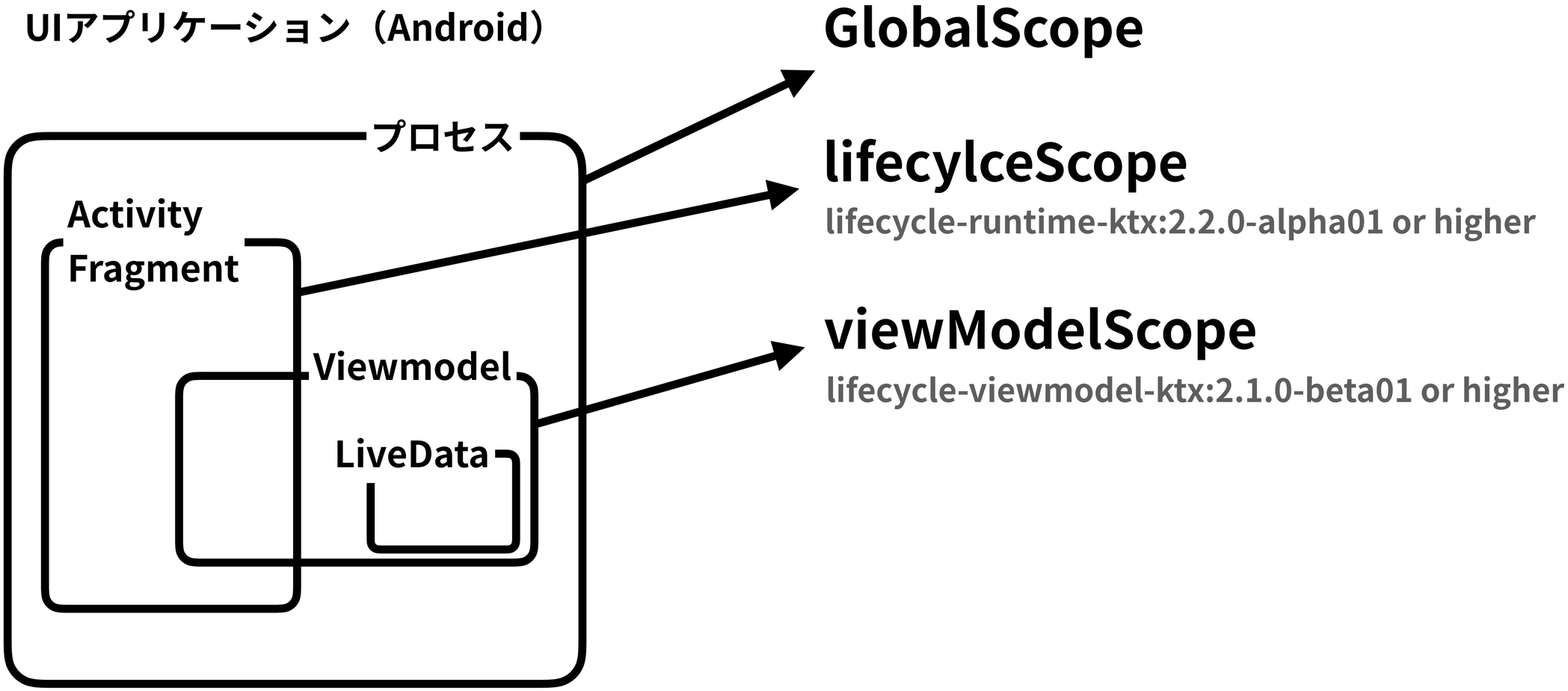
コルーチンスコープとアプリケーションライフサイクルを合わせる



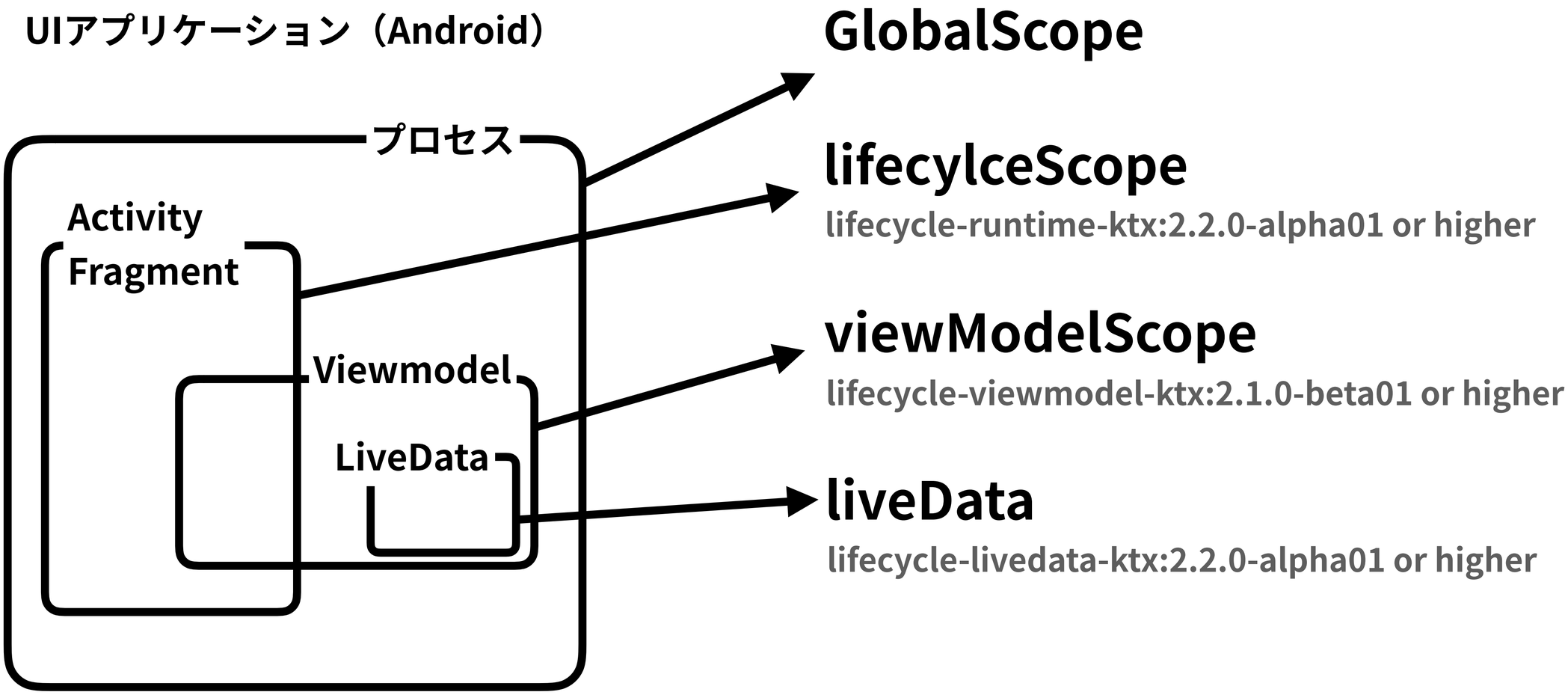
コルーチンスコープとアプリケーションライフサイクルを合わせる



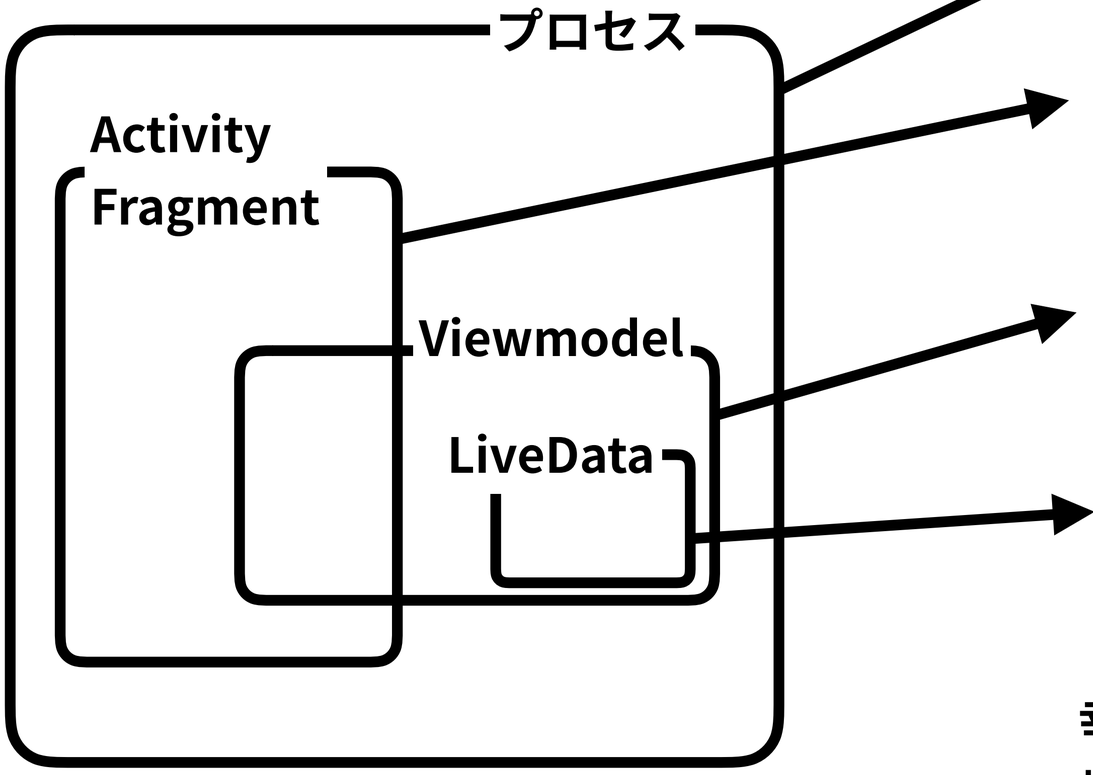
コルーチンスコープとアプリケーションライフサイクルを合わせる



コルーチンスコープとアプリケーションライフサイクルを合わせる



コルーチンスコープとアプリケーションライフサイクルを合わせる



GlobalScope

lifecylceScope

lifecycle-runtime-ktx:2.2.0-alpha01 or higher

viewModelScope

lifecycle-viewmodel-ktx:2.1.0-beta01 or higher

liveData

lifecycle-livedata-ktx:2.2.0-alpha01 or higher

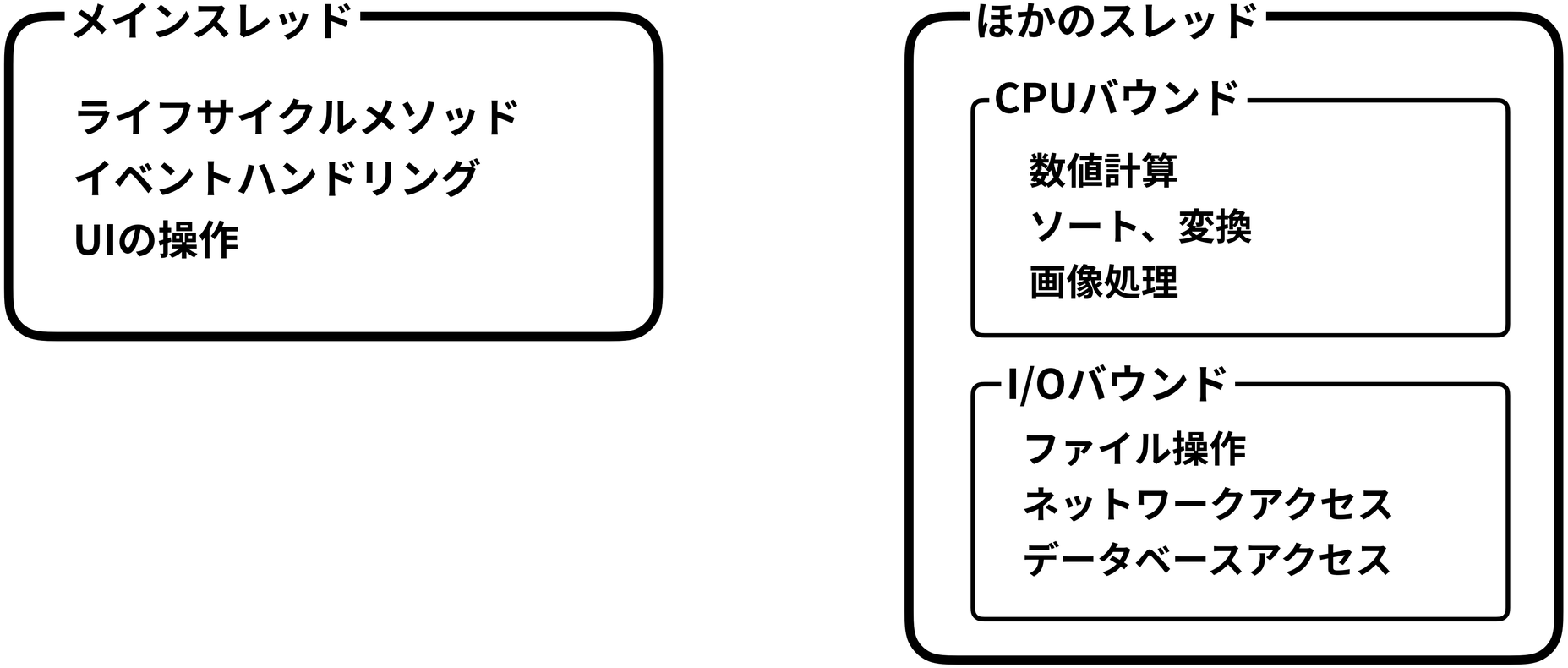
幸いなことにKotlinコルーチンのサポートが
様々なプラットフォームで進んでおり、自前で
用意する必要はほとんどなくなった

コルーチンと設計

- コルーチンスコープとアプリケーションライフサイクルを合わせる
- suspend関数はメインセーフティで実装する
- コルーチンはメインセーフティで起動する

suspend関数はメインセーフティで実装する

処理の分類



suspend関数はメインセーフティで実装する

処理の分類

メインスレッド

ライフサイクルメソッド
イベントハンドリング
UIの操作

メインスレッド以外で処理する場合、suspend関数をメインスレッドから安全に呼び出せるように実装する

ほかのスレッド

CPUバウンド

数値計算
ソート、変換
画像処理

I/Oバウンド

ファイル操作
ネットワークアクセス
データベースアクセス

suspend関数はメインセーフティで実装する

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile = withContext(Dispatchers.IO) {
        apiClient.getProfile(id)
    }
}
```

suspend関数はメインセーフティで実装する

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile = withContext(Dispatchers.IO) {
        apiClient.getProfile(id)
    }
}
```

ブロッキングする関数を使ってるので...

suspend関数はメインセーフティで実装する

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile = withContext(Dispatchers.IO) {
        apiClient.getProfile(id)
    }
}
```

メインスレッドから安全に呼び出せるようにする

suspend関数はメインセーフティで実装する

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile = withContext(Dispatchers.IO) {
        apiClient.getProfile(id)
    }
}
```

メインスレッドから安全に呼び出せるようにする

suspend関数をメインスレッドから呼び出す時に、実行スレッドを意識させない

コルーチンと設計

- コルーチンスコープとアプリケーションライフサイクルを合わせる
- suspend関数はメインセーフティで実装する
- コルーチンはメインセーフティで起動する

コルーチンはメインセーフティで起動する

```
class MainViewModel(val dataSource: ProfileDataSource) : ViewModel() {
  val profile: LiveData<Profile>
  val viewState: LiveData<ViewState>
  fun loadContent() {
    viewModelScope.launch {
      try {
        viewState.value = ViewState.Progress
        profile.value = dataSource.loadProfile(10L)
      } catch (e: IOException) {
        viewState.value = ViewState.Error(e)
      } finally {
        viewState.value = ViewState.Loaded
      }
    }
}
```

Good

コルーチンはメインセーフティで起動する

```
class MainViewModel(val dataSource: ProfileDataSource) : ViewModel() {
  val profile: LiveData<Profile>
  val viewState: LiveData<ViewState>
  fun loadContent() {
    viewModelScope.launch {
      try {
        viewState.value = ViewState.Progress
        profile.value = dataSource.loadProfile(10L)
      } catch (e: IOException) {
        viewState.value = ViewState.Error(e)
      } finally {
        viewState.value = ViewState.Loaded
      }
    }
}
```

Android JetpackのViewModel

Good

コルーチンはメインセーフティで起動する

```
class MainViewModel(val dataSource: ProfileDataSource) : ViewModel() {
  val profile: LiveData<Profile>
  val viewState: LiveData<ViewState>
  fun loadContent() {
    viewModelScope.launch {
      try {
        viewState
        profile.v                                    )
      } catch (e
        viewState.value = ViewState.Error(e)
      } finally {
        viewState.value = ViewState.Loaded
      }
    }
}
```

ViewModelのコルーチンスコープ
Mainディスパッチャーで動作する

Good

コルーチンはメインセーフティで起動する

```
class MainViewModel(val dataSource: ProfileDataSource) : ViewModel() {
  val profile: LiveData<Profile>
  val viewState: LiveData<ViewSt
  fun loadContent() {
    viewModelScope.launch {
      try {
        viewState.value = ViewState.Progress
        profile.value = dataSource.loadProfile(10L)
      } catch (e: IOException) {
        viewState.value = ViewState.Error(e)
      } finally {
        viewState.value = ViewState.Loaded
      }
    }
}
```

メインスレッドで動作する

Good

コルーチンはメインセーフティで起動する

```
class MainViewModel(val dataSource: ProfileDataSource) : ViewModel() {
  val profile: LiveData<Profile>
  val viewState: LiveData<ViewState>
  fun loadContent() {
    viewModelScope.launch(Dispatchers.IO) {
      try {
        viewState.postValue(ViewState.Progress)
        profile.postValue(dataSource.loadProfile(10L))
      } catch (e: IOException) {
        viewState.postValue(ViewState.Error(e))
      } finally {
        viewState.postValue(ViewState.Loaded)
      }
    }
}
```

Not Good

コルーチンはメインセーフティで起動する

```
class MainViewModel(val dataSource: ProfileDataSource) : ViewModel() {
  val profile: LiveData<Profile>
  val viewState: LiveData<ViewState>
  fun loadContent() {
    viewModelScope.launch(Dispatchers.IO) {
      try {
        viewState.postValue(ViewState.Progress)
        profile.postValue(dataSource.loadProfile(10L))
      } catch (e: IOException) {
        viewState.postValue(ViewState.Error(e))
      } finally {
        viewState.postValue(ViewState.Loaded)
      }
    }
}
```

異なるスレッドで動かす

Not Good

コルーチンはメインセーフティで起動する

```
class MainViewModel(val dataSource: ProfileDataSource) : ViewModel() {
  val profile: LiveData<Profile>
  val viewState: LiveData<ViewState>
  fun loadContent() {
    viewModelScope.launch(Dispatchers.IO) {
      try {
        viewState.postValue(ViewState.Progress)
        profile.postValue(dataSource.loadProfile(10L))
      } catch (e: IOException) {
        viewState.postValue(ViewState.Error(e))
      } finally {
        viewState.postValue(ViewState.Loaded)
      }
    }
}
```

一応LiveDataの場合postValueを使えばメインセーフティな操作ができる

Not Good

コルーチンはメインセーフティで起動する

```
class MainViewModel(val dataSource: ProfileDataSource) : ViewModel() {
  val profile: LiveData<Profile>
  val viewState: LiveData<ViewState>
  fun loadContent() {
    viewModelScope.launch(Dispatchers.IO) {
      try {
        viewState.postValue(ViewState.Progress)
        profile.postValue(dataSource.loadProfile(10L))
      } catch (e: IOException) {
        viewState.postValue(ViewState.Erro
      } finally {
        viewState.postValue(ViewState.Loade
      }
    }
}
```

ブロッキングする関数だとしても
問題なく動作できるが...

Not Good

コルーチンはメインセーフティで起動する

```
class MainViewModel(val dataSource: ProfileDataSource) : ViewModel() {
  val profile: LiveData<Profile>
  val viewState: LiveData<ViewState>
  fun loadContent() {
    viewModelScope.launch(Dispatchers.IO) {
      try {
        viewState.postValue(ViewState.Progress)
        profile.postValue(dataSource.loadProfile(10L))
      } catch (e: IOException) {
        viewState.postValue(ViewState.Erro
      } finally {
        viewState.postValue(ViewState.Loade
      }
    }
}
```

ブロッキングする関数だとしても問題なく動作できるが...

メインセーフティでない関数を呼び出すかもしれないことを意識させるべきではない

Not Good

コルーチンはメインセーフティで起動する

```
class MainViewModel(val dataSource: ProfileDataSource) : ViewModel() {
  val profile: LiveData<Profile>
  val viewState: LiveData<ViewState>
  fun loadContent() {
    viewModelScope.launch {
      try {
        viewState.value = ViewState.Progress
        profile.value = dataSource.loadProfile(10L)
      } catch (e: IOException) {
        viewState.value = ViewState.Error(e)
      } finally {
        viewState.value = ViewState.Loaded
      }
    }
}
```

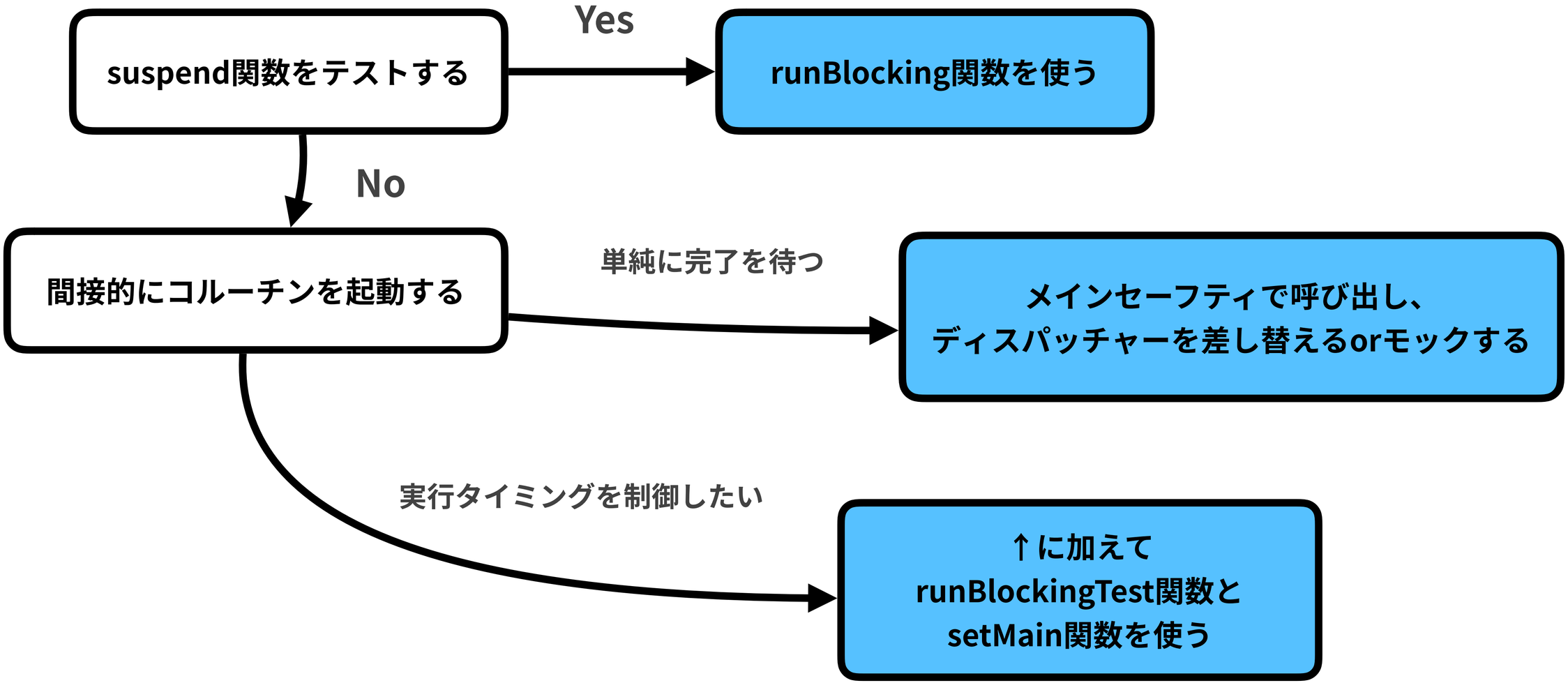
Good

コルーチンと設計

- コルーチンスコープとアプリケーションライフサイクルを合わせる。最近ではプラットフォームで公式のサポートをしてくれている
- suspend関数はメインセーフティで実装する
- コルーチンはメインセーフティで起動する。もし問題があればsuspend関数がメインセーフティでないかもしれない

コルーチンのテスト

コルーチンのテスト



コルーチンのテストライブラリ

- 実行タイミングを制御するテストディスパッチャーとrunBlockingTest関数
- Mainディスパッチャーを書き換えるsetMain関数

https://github.com/Kotlin/kotlinx.coroutines/tree/master/kotlinx-coroutines-test

Module kotlinx-coroutines-test

Test utilities for `kotlinx.coroutines`.

This package provides testing utilities for effectively testing coroutines.

Using in your project

Add `kotlinx-coroutines-test` to your project test dependencies:

```
dependencies {
    testImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.3.0-RC2'
}
```

コルーチンのテストライブラリ

build.gradle (Android)

```
testImplementation "org.jetbrains.kotlinx:kotlinx-coroutines-test:1.3.0"

testImplementation "androidx.test.ext:truth:1.2.0"
testImplementation "io.mockk:mockk:1.8.13.kotlin13"
testImplementation "io.mockk:mockk-android:1.8.13.kotlin13"
```

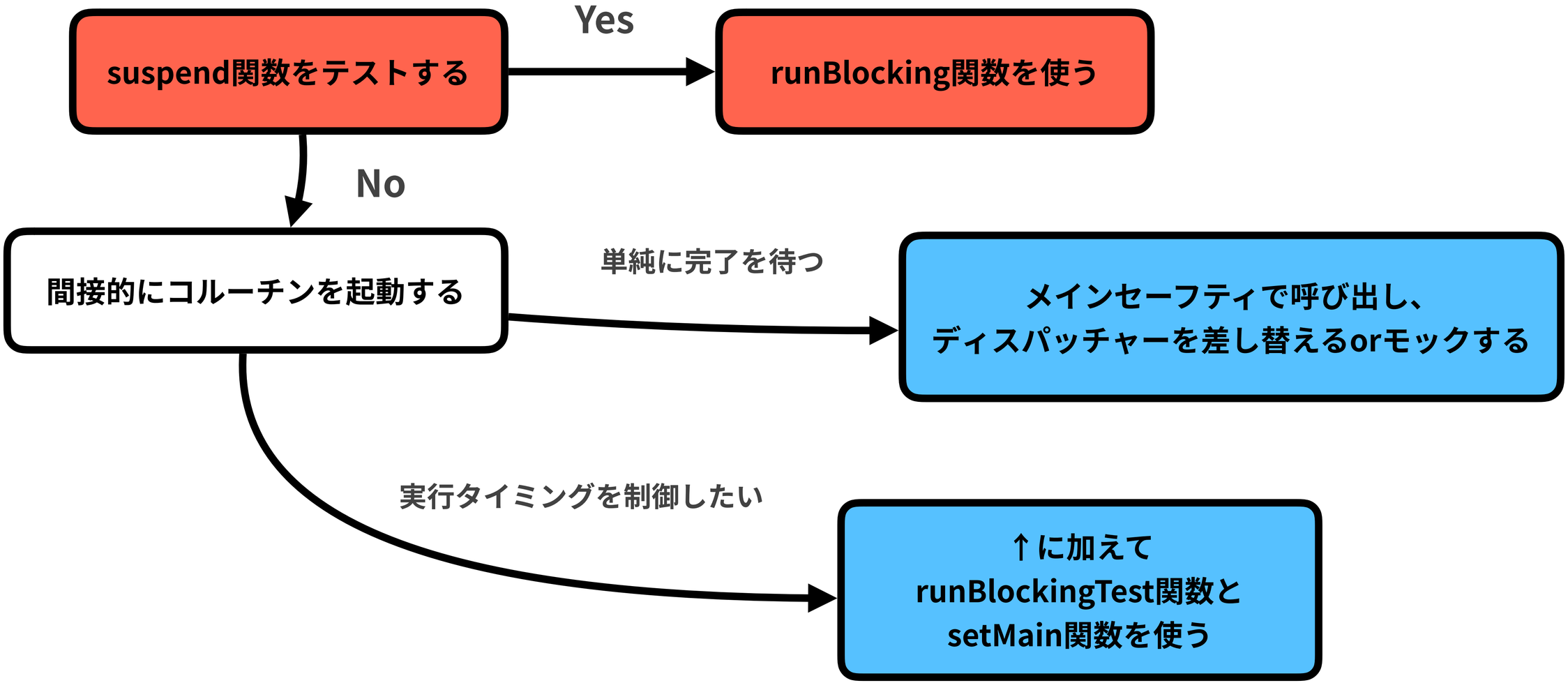
コルーチンのテストライブラリ

build.gradle (Android)

```
testImplementation "org.jetbrains.kotlinx:kotlinx-coroutines-test:1.3.0-RC2"

testImplementation "androidx.test.ext:truth:1.2.0"
testImplementation "io.mockk:mockk:1.8.13.kotlin13"
testImplementation "io.mockk:mockk-android:1.8.13.kotlin13"
```

アサーションとモックライブラリ

コルーチンのテスト



suspend関数のテスト

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile = withContext(Dispatchers.IO) {
        apiClient.getProfile(id)
    }
}
```

suspend関数のテスト

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile = withContext(Dispatchers.IO) {
        apiClient.getProfile(id)
    }
}
```

```
@Test
fun loadProfileSuccess(): Unit = runBlocking {
    val dataSource = ProfileDataSource(mockk {
        every { getProfile(any()) } returns Profile()
    })
    val profile = dataSource.loadProfile(10)
    assertThat(profile).isNotNull()
}
```

suspend関数のテスト

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile = withContext(Dispatchers.IO) {
        apiClient.getProfile(id)
    }
}
```

ブロッキングでコルーチンを実行

```
@Test
fun loadProfileSuccess(): Unit = runBlocking {
    val dataSource = ProfileDataSource(mockk {
        every { getProfile(any()) } returns Profile()
    })
    val profile = dataSource.loadProfile(10)
    assertThat(profile).isNotNull()
}
```

suspend関数のテスト

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile = withContext(Dispatchers.IO) {
        apiClient.getProfile(id)
    }
}

@Test
fun loadProfileSuccess(): Unit = runBlocking {
    val dataSource = ProfileDataSource(mockk {
        every { getProfile(any()) } returns Profile()
    })
    val profile = dataSource.loadProfile(10)
    assertThat(profile).isNotNull()
}
```

ApiClientをモック

suspend関数のテスト

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile = withContext(Dispatchers.IO) {
        apiClient.getProfile(id)
    }
}

@Test
fun loadProfileSuccess(): Unit = runBlocking {
    val dataSource = ProfileDataSource(mockk {
        every { getProfile(any()) } returns Profile()
    })
    val profile = dataSource.loadProfile(10)
    assertThat(profile).isNotNull()
}
```

suspend関数を呼び出す

suspend関数のテスト

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile = withContext(Dispatchers.IO) {
        apiClient.getProfile(id)
    }
}

@Test
fun loadProfileSuccess(): Unit = runBlocking {
    val dataSource = ProfileDataSource(mockk {
        every { getProfile(any()) } returns Profile()
    })
    val profile = dataSource.loadProfile(10)
    assertThat(profile).isNotNull()
}
```

アサーション

suspend関数のテスト

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile = withContext(Dispatchers.IO) {
        apiClient.getProfile(id)
    }
}
```

```
@Test
fun loadProfileSuccess(): Unit = runBlocking {
    val dataSource = ProfileDataSource(mockk {
        every { getProfile(any()) } returns Profile()
    })
    val profile = dataSource.loadProfile(10)
    assertThat(profile).isNotNull()
}
```

suspend関数のテスト

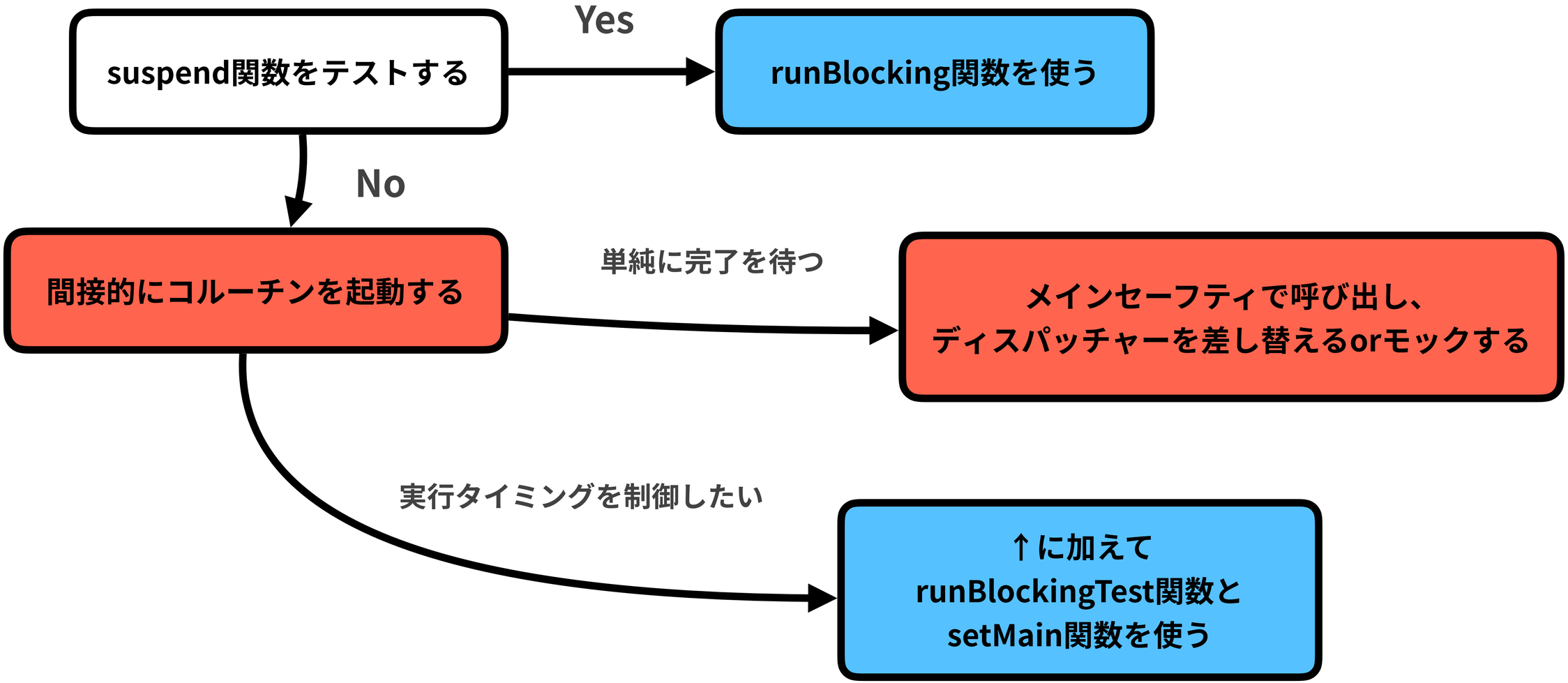
```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile = withContext(Dispatchers.IO) {
      coroutineScope {
        val profile = async { apiClient.getProfile(id) }
        val articles = async { apiClient.getArticles(id) }
        Profile(profile.await(), articles.await())
      }
    }
}
```

suspend関数のテスト

```
class ProfileDataSource(private val apiClient: ApiClient) {
    suspend fun loadProfile(id: Long): Profile = withContext(Dispatchers.IO) {
      coroutineScope {
        val profile = async { apiClient.getProfile(id) }
        val articles = async { apiClient.getArticles(id) }
        Profile(profile.await(), articles.await())
      }
    }
}
```

呼び出し元と異なるスコープでコルーチンを起動しない限りはこういう形などでも問題ない

コルーチンのテスト



間接的にコルーチンを起動する関数

```
class ReadingHistoryViewModel(private val database: Database) : ViewModel() {
    private val _viewState = MutableLiveData<ViewState>()
    val viewState: LiveData<ViewState> = _viewState

    fun addReadingHistory(history: RoomReadingHistory) {
        viewModelScope.launch {
            try {
                _viewState.value = ViewState.Progress
                database.readingHistoryDao().insert(history)
                _viewState.value = ViewState.Completed
            } catch (e: Exception) {
                _viewState.value = ViewState.Error(e)
            }
        }
    }
}
```

間接的にコルーチンを起動する関数

```
class ReadingHistoryViewModel(private val database: Database) : ViewModel() {
    private val _viewState = MutableLiveData<ViewState>()
    val viewState: LiveData<ViewSta                    viewState

    fun addReadingHistory(histo                        ory) {
        viewModelScope.launch {
            try {
                _viewState.value = ViewState.Progress
                database.readingHistoryDao().insert(history)
                _viewState.value = ViewState.Completed
            } catch (e: Exception) {
                _viewState.value = ViewState.Error(e)
            }
        }
    }
}
```

JetpackのViewModel

間接的にコルーチンを起動する関数

```
class ReadingHistoryViewModel(private val database: Database) : ViewModel() {
    private val _viewState = MutableLiveData<ViewState>()
    val viewState: LiveData<ViewState> = _viewState

    fun addReadingHist
        viewModelScope.launch {
            try {
                _viewState.value = ViewState.Progress
                database.readingHistoryDao().insert(history)
                _viewState.value = ViewState.Completed
            } catch (e: Exception) {
                _viewState.value = ViewState.Error(e)
            }
        }
    }
}
```

JetpackのRoomで、DAOでsuspend関数を宣言している

間接的にコルーチンを起動する関数

```
class ReadingHistoryViewModel(private val database: Database) : ViewModel() {
    private val _viewState = MutableLiveData<ViewState>()
    val viewState: LiveData<ViewState> = _viewState

    fun addReadingHistory(history: RoomReadingHistory) {
        viewModelScope.launch {
            try {
                                                    gress
                                                    ert(history)
                                                    pleted
            } catch (e: Exception) {
                _viewState.value = ViewState.Error(e)
            }
        }
    }
}
```

通常の関数呼び出しからコルーチンを起動する

間接的にコルーチンを起動する関数

```
class ReadingHistoryViewModel(private val database: Database) : ViewModel() {
    private val _viewState = MutableLiveData<ViewState>()
    val viewState: LiveData<ViewState> = _viewState

    fun addReadingHistory(history: RoomReadingHistory) {
        viewModelScope.launch {
            try {
                _viewState.value = ViewState.Progress
                database.readingHistoryDao().insert(history)
                _viewState.value = ViewState.Completed
            } catch (e: Exception) {
                _viewState.value =
            }
        }
    }
}
```

suspend関数を呼び出す

間接的にコルーチンを起動する関数

```
@RunWith(AndroidJUnit4::class)
class ReadingHistoryViewModelTest {
  val app by lazy { ApplicationProvider.getApplicationContext<Application>() }
  val database by lazy {
        Room.databaseBuilder(app, Database::class.java, "database")
            .allowMainThreadQueries()
            .build()
  }
  @Test
  fun insertTest() = runBlocking {
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(database.readingHistoryDao().historyCount()).isEqualTo(0)
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(database.readingHistoryDao().historyCount()).isEqualTo(1)
  }
}
```

間接的にコルーチンを起動する関数

```
@RunWith(AndroidJUnit4::class)
class ReadingHistoryViewModelTest {
  val app by lazy { <Application>() }
  val database by l
        Room.databa                                    base")
            .allowMainThreadQueries()
            .build()
  }
  @Test
  fun insertTest() = runBlocking {
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(database.readingHistoryDao().historyCount()).isEqualTo(0)
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(database.readingHistoryDao().historyCount()).isEqualTo(1)
  }
}
```

Roomを動かすためにAndroidJUnit4で動かす

間接的にコルーチンを起動する関数

```
@RunWith(AndroidJUnit4::class)
class ReadingHistoryViewModelTest {
  val app by lazy { ApplicationProvider.getApplicationContext<Application>() }
  val database by lazy {
        Room.databaseBuilder(app, Database::class.java, "database")
            .allowMainThreadQueries()
            .build()
  }
  @Test
  fun insertTest() =
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(database.readingHistoryDao().historyCount()).isEqualTo(0)
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(database.readingHistoryDao().historyCount()).isEqualTo(1)
  }
}
```

RoomDatabaseを初期化する

間接的にコルーチンを起動する関数

```
@RunWith(AndroidJUnit4::class)
class ReadingHistoryViewModelTest {
  val app by lazy { ApplicationProvider.getApplicationContext<Application>() }
  val database by lazy {
        Room.databaseBuilder(app, Database::class.java, "database")
            .allowMainThreadQueries()
            .build()
  }
  @Test
  fun insertTest() = runBlocking {
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(database.readingHistoryDao().historyCount()).isEqualTo(0)
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(database.readingHistoryDao().historyCount()).isEqualTo(1)
  }
}
```

アサーションのためにDAOを操作するのでrunBlockingで動かす

間接的にコルーチンを起動する関数

```
@RunWith(AndroidJUnit4::class)
class ReadingHistoryViewModelTest {
  val app by lazy { ApplicationProvider.getApplicationContext<Application>() }
  val database by lazy {
        Room.databaseBuilder(app, Database::class.java, "database")
            .allowMainThreadQueries()
            .build()
  }
  @Test
  fun insertTest() = runBlocking {
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(database.readingHistoryDao().historyCount()).isEqualTo(0)
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(database.readingHistoryDao().historyCount()).isEqualTo(1)
  }
}
```

Caused by: expected: 1
but was : 0

間接的にコルーチンを起動する関数

```
class ReadingHistoryViewModel(private val database: Database) : ViewModel() {
    private val _viewState = MutableLiveData<ViewState>()
    val viewState: LiveData<ViewState> = _viewState

    fun addReadingHistory(history: RoomReadingHistory) {
        viewModelScope.launch {
            try {
                _viewState.value = ViewState.Progress
                database.readingHistoryDao().insert(history)
                _viewState.value = ViewState.Completed
            } catch (e: Exception) {
                _viewState.value = ViewState.Error(e)
            }
        }
    }
}
```

間接的にコルーチンを起動する関数

```
class ReadingHistoryViewModel(private val database: Database) : ViewModel() {
    private val _viewState = MutableLiveData<ViewState>()
    val viewState: LiveData<ViewState> = _viewState

    fun addReadingHistory(history: RoomReadingHistory) {
        viewModelScope.launch {
            try {
                _viewState.value = ViewState.Progress
                database.readingHistoryDao().insert(history)
                _viewState.value = ViewState.Completed
            } catch (e: Exception) {
                _viewState.value
            }
        }
    }
}
```

テストではここまで到達して中断し、制御が戻る（アサーションに進んでしまう）

間接的にコルーチンを起動する関数

```
class ReadingHistoryViewModel(private val database: Database) : ViewModel() {
    private val _viewState = MutableLiveData<ViewState>()
    val viewState: LiveData<ViewState> = _viewState

    fun addReadingHistory(history: RoomReadingHistory) {
        viewModelScope.launch(Dispatchers.IO) {
            try {
                _viewStat  value = ViewState.Progress
                da                o().insert(history)
                _v                ate.Completed
            } catc
                _viewState.value = ViewState.Error(e)
            }
        }
    }
}
```

もしメインセーフティな実行じゃない場合はここで止まる

Not Good

間接的にコルーチンを起動する関数

```
class ReadingHistoryViewModel(private val database: Database) : ViewModel() {
    private val _viewState = MutableLiveData<ViewState>()
    val viewState: LiveData<ViewState> = _viewState

    fun addReadingHistory(history: RoomReadingHistory) {
        viewModelScope.launch {
            try {
                _viewState.value = ViewState.Progress
                database.readingHistoryDao().insert(history)
                _viewState.value = ViewState.Completed
            } catch (e: Exception) {
                _viewState.value = V
            }
        }
    }
}
```

この実行をテストスレッドで行いたい

間接的にコルーチンを起動する関数

```
@RunWith(AndroidJUnit4::class)
class ReadingHistoryViewModelTest {
  val app by lazy { ApplicationProvider.getApplicationContext<Application>() }
  val database by lazy {
        Room.databaseBuilder(app, Database::class.java, "database")
            .allowMainThreadQueries()
            .build()
  }
  @Test
  fun insertTest() = runBlocking {
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(database.readingHistoryDao().historyCount()).isEqualTo(0)
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(database.readingHistoryDao().historyCount()).isEqualTo(1)
  }
}
```

間接的にコルーチンを起動する関数

```
@RunWith(AndroidJUnit4::class)
class ReadingHistoryViewModelTest {
  val app by lazy { ApplicationProvider.getApplicationContext<Application>() }
  val database by lazy {
        Room.databaseBuilder(app, Database::class.java, "database")
            .allowMainThreadQueries()
            .setQueryExecutor { it.run() }
            .build()
  }
  @Test
  fun insertTest() = runBlocking {
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(database.readingHistoryDao().historyCount()).isEqualTo(0)
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(database.readingHistoryDao().historyCount()).isEqualTo(1)
  }
}
```

間接的にコルーチンを起動する関数

```
@RunWith(AndroidJUnit4::class)
class ReadingHistoryViewModelTest {
  val app by lazy { ApplicationProvider.getApplicationContext<Application>() }
  val database by lazy {
        Room.databaseBuilder(app, Database::class.java, "database")
            .allowMainThreadQueries()
            .setQueryExecutor { it.run() }
            .build()
  }
  @Test
  fun insertTest
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(database.readingHistoryDao().historyCount()).isEqualTo(0)
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(database.readingHistoryDao().historyCount()).isEqualTo(1)
  }
}
```

テストスレッドで実行するようにする

間接的にコルーチンを起動する関数

```
@RunWith(AndroidJUnit4::class)
class ReadingHistoryViewModelTest {
  val app by lazy { ApplicationProvider.getApplicationContext<Application>() }
  val database by lazy {
        Room.databaseBuilder(app, Database::class.java, "database")
            .allowMainThreadQueries()
            .setQueryExecutor { it.run() }
            .build()
  }
  @Test
  fun insertTest
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(
    viewModel.ad
    assertThat(
  }
}
```

テストスレッドで実行するようにする

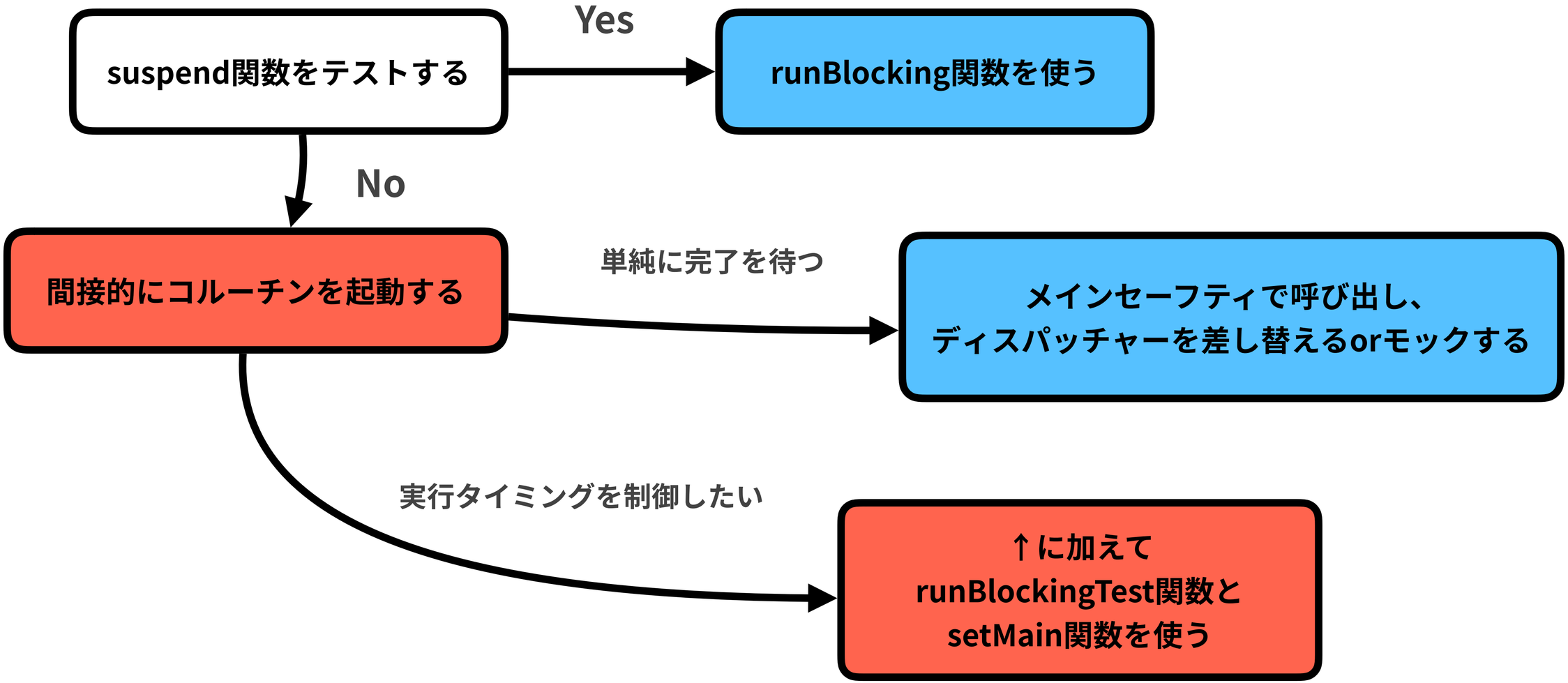
Main以外のディスパッチャを使うsuspend関数は、ディスパッチャを書き換えられるようにしておくのが望ましい

間接的にコルーチンを起動する関数

```
@RunWith(AndroidJUnit4::class)
class ReadingHistoryViewModelTest {
  val app by lazy { ApplicationProvider.getApplicationContext<Application>() }
  val database by lazy {
        Room.databaseBuilder(app, Database::class.java, "database")
            .allowMainThreadQueries()
            .setQueryExecutor { it.run() }
            .build()
  }
  @Test
  fun insertTest() = runBlocking {
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(database.readingHistoryDao().historyCount()).isEqualTo(0)
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(database.readingHistoryDao().historyCount()).isEqualTo(1)
  }
}
```

OK

コルーチンのテスト



コルーチンの実行タイミングを制御する

```
@Test
fun viewStateTest() = runBlocking {
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(viewModel.viewState.value).isNull()
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Progress)
    // 処理が完了後
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Completed)
}
```

コルーチンの実行タイミングを制御する

```
@Test
fun viewStateTest() = runBlocking {
    val viewModel = ReadingHistoryViewModel(database

    assertThat(viewModel.viewState.value).isNull()
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Progress)
    // 処理が完了後
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Completed)
}
```

ViewStateの変化をテストする

コルーチンの実行タイミングを制御する

```
@Test
fun viewStateTest() = runBlocking {
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(viewModel.viewState.value).isNull()
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Progress)
    // 処理が完了後
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Completed)
}
```

expected: Progress
but was : Completed

コルーチンの実行タイミングを制御する

```
class ReadingHistoryViewModel(private val database: Database) : ViewModel() {
    private val _viewState = MutableLiveData<ViewState>()
    val viewState: LiveData<ViewState> = _viewState

    fun addReadingHistory(history: RoomReadingHistory) {
        viewModelScope.launch {
            try {
                _viewState.value = ViewState.Progress
                database.readingHistoryDao().insert(history)
                _viewState.value = ViewState.Completed
            } catch (e: Exception) {
                _viewState.value = ViewState.Error(e)
            }
        }
    }
}
```

コルーチンの実行タイミングを制御する

```
class ReadingHistoryViewModel(private val database: Database) : ViewModel() {
    private val _viewState = MutableLiveData<ViewState>()
    val viewState: LiveData<ViewState> = _viewState

    fun addReadingHistory(history: RoomReadingHistory) {
        viewModelScope.launch {
            try {
                _viewState.value = ViewState.Progress
                database.readingHistoryDao().insert(history)
                _viewState.value = ViewState.Completed
            } catch (e: Exception) {
                _viewState.value = ViewState.Error(e)
            }
        }
    }
}
```

関数を呼び出したら最後までブロッキングで実行する

コルーチンの実行タイミングを制御する

```
class ReadingHistoryViewModel(private val database: Database) : ViewModel() {
    private val _viewState = MutableLiveData<ViewState>()
    val viewState: LiveData<ViewState> = _viewState

    fun addReadingHistory(history: Roo
        viewModelScope.launch {
            try {
                _viewState.value = ViewState.P ogress
                database.readingHistoryDao().insert(history)
                _viewState.value = ViewState.Completed
            } catch (e: Exception) {
                _viewState.value = ViewState.Error(e)
            }
        }
    }
}
```

一旦ここまで進めてアサーションして続きを進めたい

コルーチンの実行タイミングを制御する

```
class CoroutinesTestRule(
    val testDispatcher: TestCoroutineDispatcher = TestCoroutineDispatcher()
) : TestWatcher() {

    override fun starting(description: Description?) {
        super.starting(description)
        Dispatchers.setMain(testDispatcher)
    }

    override fun finished(description: Description?) {
        super.finished(description)
        Dispatchers.resetMain()
        testDispatcher.cleanupTestCoroutines()
    }
}
```

コルーチンの実行タイミングを制御する

```
class CoroutinesTestRule(
    val testDispatcher: TestCoroutineDispatcher = TestCoroutineDispatcher()
) : TestWatcher() {

    override fun start                                    ) {
        super.starting
        Dispatchers.se
    }

    override fun finished(description: Description?) {
        super.finished(description)
        Dispatchers.resetMain()
        testDispatcher.cleanupTestCoroutines()
    }
}
```

経過時間を制御できるディスパッチャ

コルーチンの実行タイミングを制御する

```
class CoroutinesTestRule(
    val testDispatcher: TestCoroutineDispatcher = TestCoroutineDispatcher()
) : TestWatcher() {

    override fun starting(description: Description?) {
        super.starting(description)
        Dispatchers.setMain(testDispatcher)
    }

    override fun finished(description: Description?) {
        super.finished(description)
        Dispatchers.resetMain()
        testDispatcher.cleanupTestCoroutines()
    }
}
```

テスト開始時にMainディスパッチャーを書き換える

コルーチンの実行タイミングを制御する

```
class CoroutinesTestRule(
    val testDispatcher: TestCoroutineDispatcher = TestCoroutineDispatcher()
) : TestWatcher() {

    override fun starting(description: Description?) {
        super.starting(description)
        Dispatchers.setMain(testDispatcher)
    }

    override fun finished(description: Description?) {
        super.finished(description)
        Dispatchers.resetMain()
        testDispatcher.cleanupTestCoroutines()
    }
}
```

テスト完了時の後処理

コルーチンの実行タイミングを制御する

```
@Test
fun viewStateTest() = runBlocking {
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(viewModel.viewState.value).isNull()
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Progress)
    // 処理が完了後
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Completed)
}
```

コルーチンの実行タイミングを制御する

```
@get:Rule
val rule = CoroutinesTestRule()

@Test
fun viewStateTest() = runBlocking {
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(viewModel.viewState.value).isNull()
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Progress)
    // 処理が完了後
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Completed)
}
```

コルーチンの実行タイミングを制御する

```
@get:Rule
val rule = CoroutinesTestRule()

@Test
fun viewStateTest() = rule.testDispatcher.runBlockingTest {
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(viewModel.viewState.value).isNull()
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Progress)
    // 処理が完了後
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Completed)
}
```

コルーチンの実行タイミングを制御する

```
@get:Rule
val rule = CoroutinesTestRule()

@Test
fun viewStateTest() = rule.testDispatcher.runBlockingTest {
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(viewModel.viewState.value).isNull()
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Progress)
    // 処理が完了後
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Completed)
}
```

コルーチンの実行タイミングを制御する

```
@get:Rule
val rule = CoroutinesTestRule()

@Test
fun viewStateTest() = rule.testDispatcher.runBlockingTest {
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(viewModel.viewState.value).isNull()
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Progress)
    // 処理が完了後
    assertThat(viewModel.viewState.value).isEqualTo(Read                    ompleted)
}
```

expected: Progress
but was : Completed

コルーチンの実行タイミングを制御する

```
@get:Rule
val rule = CoroutinesTestRule()

@Test
fun viewStateTest() = rule.testDispatcher.runBlockingTest {
    val readingHistoryDao: ReadingHistoryDao = mockk {
      coEvery { insert(any()) } coAnswers {
        delay(100)
        Unit
      }
    }
    val database: Database = mockk {
      every { readingHistoryDao() } returns readingHistoryDao
    }
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(viewModel.viewState.value).isNull()
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Progress)
    // 処理が完了後
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Completed)
}
```

コルーチンの実行タイミングを制御する

```
@get:Rule
val rule = CoroutinesTestRule()

@Test
fun viewStateTest() = rule.testDispatcher.runBlockingTest {
    val readingHistoryDao: ReadingHistoryDao = mockk {
      coEvery { insert(any()) } coAnswers {
        delay(100)
        Unit
      }
    }
    val database: Database = mockk {
      every { readingHistoryDao() } returns readingHistoryDao
    }
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(viewModel.viewState.value).isNull()
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Progress)
    // 処理が完了後
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Completed)
}
```

Roomデータベースをモックする

コルーチンの実行タイミングを制御する

```
@get:Rule
val rule = CoroutinesTestRule()

@Test
fun viewStateTest() = rule.testDispatcher.runBlockingTest {
    val readingHistoryDao: ReadingHistoryDao = mockk {
      coEvery { insert(any()) } coAnswers {
        delay(100)
        Unit
      }
    }
    val database: Database = mockk {
      every { readingHistoryDao() } returns readingHistoryDao
    }
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(viewModel.viewState.value).isNull()
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Progress)
    // 処理が完了後
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Completed)
}
```

テスト内で使われるDAOや関数をモックする

コルーチンの実行タイミングを制御する

```
@get:Rule
val rule = CoroutinesTestRule()

@Test
fun viewStateTest() = rule.testDispatcher.runBlockingTest {
    val readingHistoryDao: ReadingHistoryDao = mockk {
      coEvery { insert(any()) } coAnswers {
        delay(100)
        Unit
      }
    }
    val database: Database = mockk {
      every { readingHistoryDao() } returns readingHistoryDao
    }
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(viewModel.viewState.value).isNull()
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Progress)
    // 処理が完了後
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Completed)
}
```

任意の時間待たせる

コルーチンの実行タイミングを制御する

```
@get:Rule
val rule = CoroutinesTestRule()

@Test
fun viewStateTest() = rule.testDispatcher.runBlockingTest {
    val readingHistoryDao: ReadingHistoryDao = mockk {
      coEvery { insert(any()) } coAnswers {
        delay(100)
        Unit
      }
    }
    val database: Database = mockk {
      every { readingHistoryDao() } returns readingHistoryDao
    }
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(viewModel.viewState.value).isNull()
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Progress)
    // 処理が完了後
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Completed)
}
```

この時点ではReadingHistoryDao.insertを実行中

コルーチンの実行タイミングを制御する

```
@get:Rule
val rule = CoroutinesTestRule()

@Test
fun viewStateTest() = rule.testDispatcher.runBlockingTest {
    val readingHistoryDao: ReadingHistoryDao = mockk {
      coEvery { insert(any()) } coAnswers {
        delay(100)
        Unit
      }
    }
    val database: Database = mockk {
      every { readingHistoryDao() } returns readingHistoryDao
    }
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(viewModel.viewState.value).isNull()
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Progress)
    // 処理が完了後
    advanceTimeBy(200)
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Completed)
}
```

コルーチンの実行タイミングを制御する

```
@get:Rule
val rule = CoroutinesTestRule()

@Test
fun viewStateTest() = rule.testDispatcher.runBlockingTest {
    val readingHistoryDao: ReadingHistoryDao = mockk {
      coEvery { insert(any()) } coAnswers {
        delay(100)
        Unit
      }
    }
    val database: Database = mockk {
      every { readingHistoryDao() } returns readingHistoryDao
    }
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(viewModel.viewState.value).isNull()
    viewModel.addReadingHis
    assertThat(viewModel.vi                                    te.Progress)
    // 処理が完了後
    advanceTimeBy(200)
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Completed)
}
```

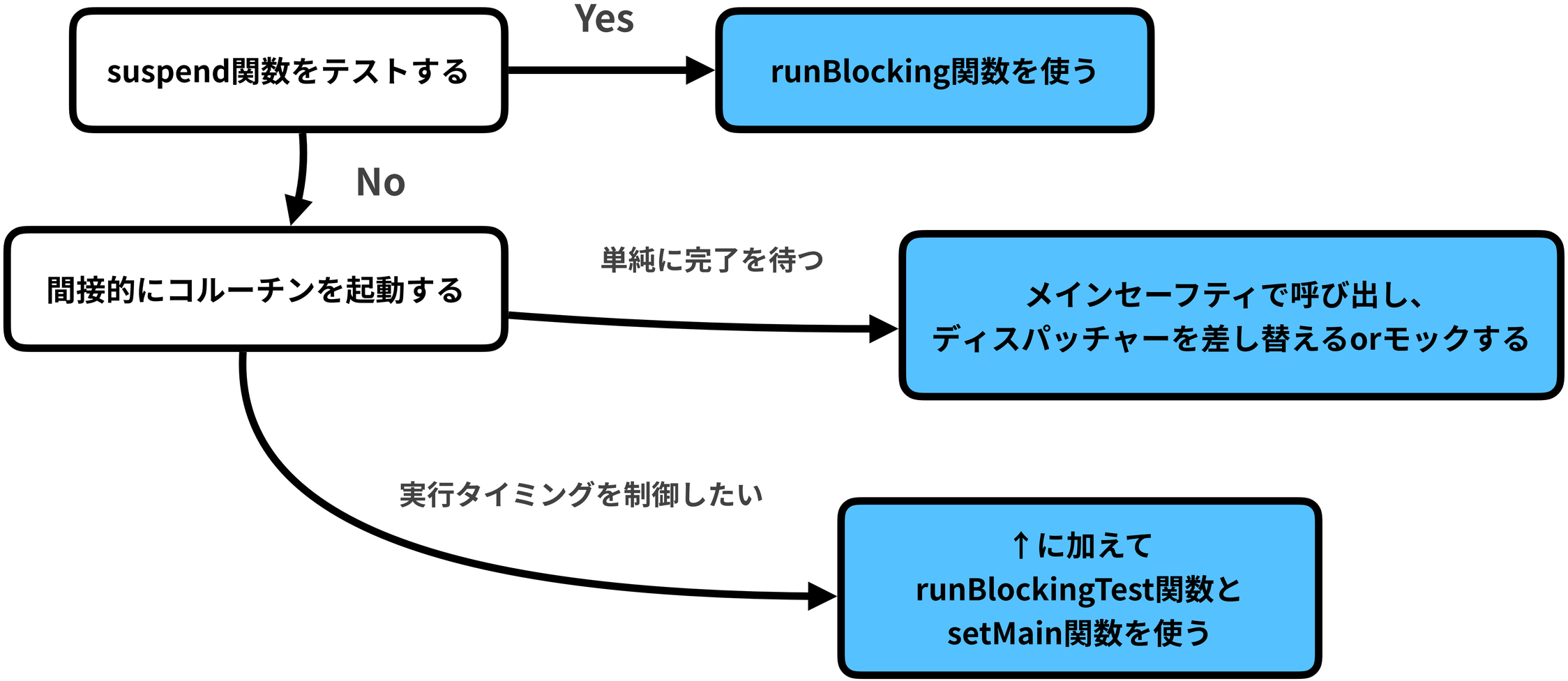
テストディスパッチャの時間を200ms進める関数

コルーチンの実行タイミングを制御する

```
@get:Rule
val rule = CoroutinesTestRule()

@Test
fun viewStateTest() = rule.testDispatcher.runBlockingTest {
    val readingHistoryDao: ReadingHistoryDao = mockk {
      coEvery { insert(any()) } coAnswers {
        delay(100)
        Unit
      }
    }
    val database: Database = mockk {
      every { readingHistoryDao() } returns readingHistoryDao
    }
    val viewModel = ReadingHistoryViewModel(database)

    assertThat(viewModel.viewState.value).isNull()
    viewModel.addReadingHistory(createDummyReadingHistory(1))
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Progress)
    // 処理が完了後
    advanceTimeBy(200)
    assertThat(viewModel.viewState.value).isEqualTo(ReadingHistoryViewModel.ViewState.Completed)
}
```

OK

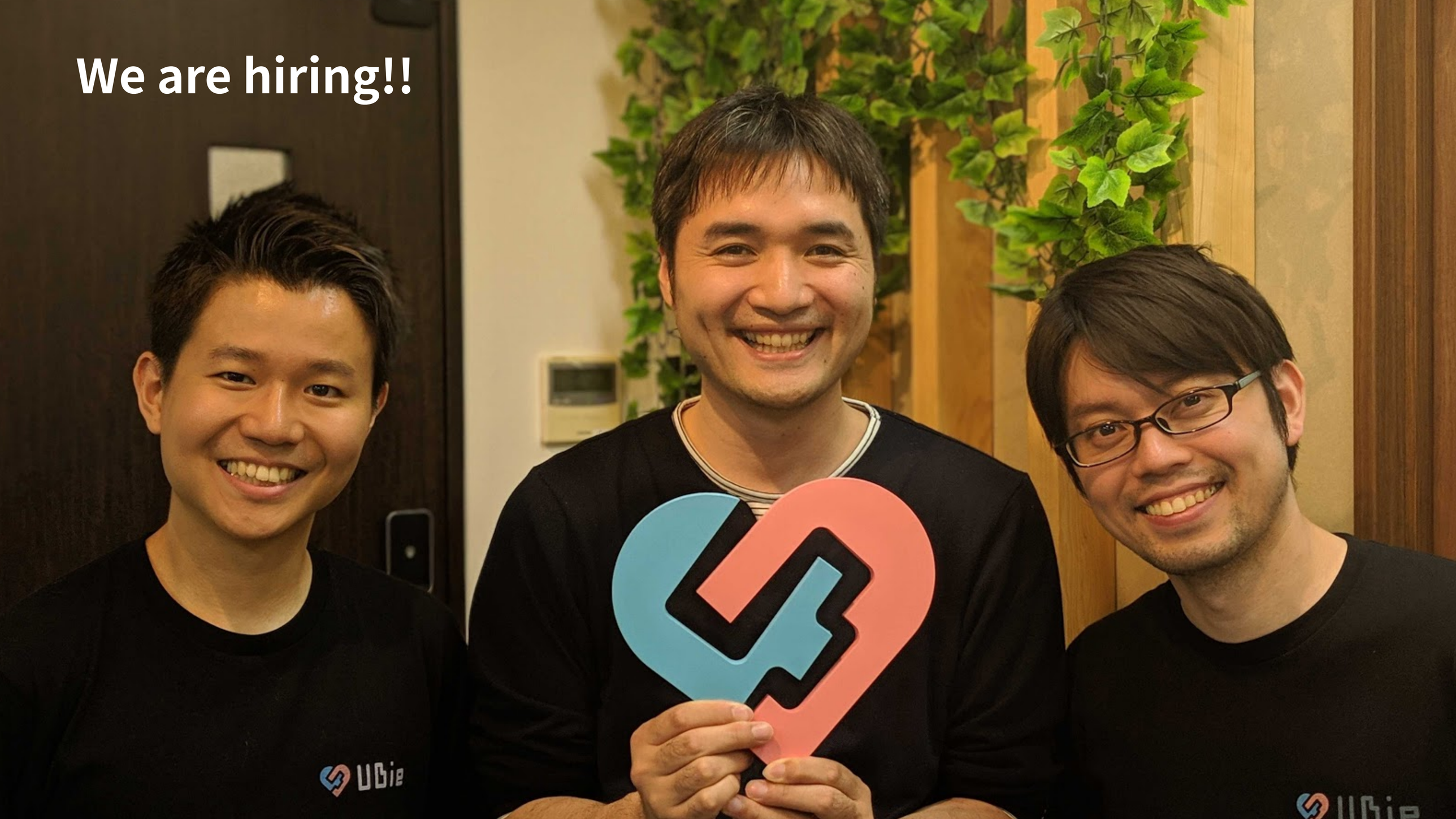
コルーチンのテストのまとめ



今日話したこと

- コルーチンとはなにか、なにがうれしいのか
- Kotlinにおけるコルーチンの仕組み
- Kotlinコルーチンのきほん
- コルーチンスコープと構造化された並行性
- コルーチンと設計
- コルーチンのテスト

We are hiring!!



ありがとうございました