Kotlin コルーチンを理解しよう

2018/08/25 Kotlin Fest

Toshihiro Yagi

Lang-8,Inc

About Me

- 八木 俊広
- @sys1yagi
- CTO of Lang-8,Inc
- Android Engineer

Jetpack Handbook 【C94新刊】

「第6章 WorkManagerとバックグラウンドタスク」

書きました

We are hiring!

https://hinative.com

Lang-8,Inc

今日話すこと

- コルーチンとはなにか
- Kotlinはどのようにコルーチンを実現しているのか
- Kotlin コルーチンの基本的な使い方

今日話すこと

- **コルーチンとはなにか**
- **Kotlinはどのようにコルーチンを実現しているのか**
- **Kotlin コルーチンの基本的な使い方**

今日話すこと

- コルーチンとはなにか
- Kotlinはどのようにコルーチンを実現しているのか
- Kotlin コルーチンの基本的な使い方

今日話すこと

- コルーチンとはなにか
- Kotlinはどのようにコルーチンを実現しているのか
- **Kotlin コルーチンの基本的な使い方**

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

- Cobolコンパイラを軽量に実装するためにコルーチンの概念を導入した
- ある条件下でプログラムを分離すると独立してそれぞれ動作できるというアイデア
- co-routine、つまり対等のルーチンという意味。お互いを呼び出して制御できる関係

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 SEARCH, ENTER, USE and INSPECT are accepted by 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. There is no formal 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

After 55 years…

Coroutines for Kotlinによると…

- A coroutine — is an instance of suspendable computation. It is conceptually similar to a thread, in the sense that it takes a block of code to run and has a similar life-cycle — it is created and started, but it is not bound to any particular thread. It may suspend its execution in one thread and resume in another one. Moreover, like a future or promise, it may complete with some result or exception.

https://github.com/Kotlin/kotlin-coroutines/blob/master/kotlin-coroutines-informal.md#terminology

Coroutines for Kotlinによると…

- コルーチンは、一時停止可能な計算のインスタンスです。 それは概念的にはスレッドに似ています。つまり、実行するコードブロックを持ち、同様のライフサイクルを持ち、作成され起動されますが、特定のスレッドに束縛されていません。 あるスレッドで実行を中断し、別のスレッドで再開することがあります。 さらに、FutureやPromissのように、何らかの結果や例外がある場合があります。

https://github.com/Kotlin/kotlin-coroutines/blob/master/kotlin-coroutines-informal.md#terminology

Coroutines for Kotlinによると…

- コルーチンは、**一時停止可能な計算のインスタンス**です。 それは概念的にはスレッドに似ています。つまり、**実行するコードブロックを持ち、同様のライフサイクルを持ち、作成され起動されますが、特定のスレッドに束縛されていません**。 あるスレッドで実行を中断し、別のスレッドで再開することがあります。 さらに、**FutureやPromissのように、何らかの結果や例外がある場合があります**。

https://github.com/Kotlin/kotlin-coroutines/blob/master/kotlin-coroutines-informal.md#terminology

Wikipediaによると…

- サブルーチンと異なり、状態管理を意識せずに行えるため、協調的処理、イテレータ、無限リスト、パイプなど、継続状況を持つプログラムが容易に記述できる。

https://ja.wikipedia.org/wiki/%E3%82%B3%E3%83%AB%E3%83%BC%E3%83%81%E3%83%B3

Wikipediaによると…

- サブルーチンと異なり、状態管理を意識せずに行えるため、協調的処理、イテレータ、無限リスト、パイプなど、**継続状況を持つプログラムが容易に記述できる**。

https://ja.wikipedia.org/wiki/%E3%82%B3%E3%83%AB%E3%83%BC%E3%83%81%E3%83%B3

コルーチンとは…

- 一時停止可能な計算のインスタンス
- スレッドのように実行するコードブロックを持ち、同様のライフサイクルを持ち、作成され起動されるが、特定のスレッドに束縛されない
- FutureやPromissのように、何らかの結果や例外がある場合がある
- 継続状況を持つプログラムが容易に記述できる

一時停止可能な計算のインスタンス

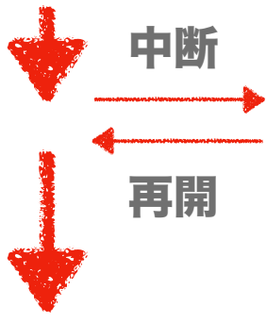
サブルーチン

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

開始からリターンまでが一つの処理単位

コルーチン

```
launch {
    val profile = getProfile(id).await()
    showProfile(profile)
}
```



処理の途中で一時停止が可能

スレッドとコルーチン

スレッド

```
Thread {
    val profile = getProfile(token)
    showProfile(profile)
}.start()
```

コードブロックとライフサイクルを持ち、作成と起動がされる

コルーチン

```
launch {
    val profile = getProfile(token).await()
    showProfile(profile)
}
```

スレッドに似ているが特定のスレッドに束縛されない

どのスレッドで実行されるかはコルーチン自身は気にしない

FutureやPromissのように値を返す場合がある

```
fun getProfile(id: Int) = async {
    //…
    return profile
}

launch {
        val profile = getProfile(id).await()
        showProfile(profile)
}
```

FutureやPromissのように値を返す場合がある

```
fun getProfile(id: Int) = async {
    //…
    return profile
}

launch {
    val profile = getProfile(id).await()
    showProfile(profile)
}
```

値を返すコルーチン

FutureやPromissのように値を返す場合がある

```
fun getProfile(id: Int) = async {
    //…
    return profile
}

launch {
        val profile = getProfile(id).await()
        showProfile(profile)
}
```

別のコルーチンを起動して自身は中断し、結果を受け取って再開する

継続状況を持つプログラムが
容易に記述できる

継続状況を持つプログラムとは

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

継続状況を持つプログラムとは

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

逐次実行

継続状況を持つプログラムとは

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

通信処理やDBアクセス、重たい計算などを
行う場合、ブロッキングしてしまう。

継続状況を持つプログラムとは

```
fun getProfile(id: Int, f: (Profile) -> Unit)
fun loadProfile(id: Int) {
  val profile = getProfile(id)
  showProfile(profil
}
```

コールバックを受け取る形にして…

継続状況を持つプログラムとは

```
fun getProfile(id: Int, f: (Profile) -> Unit)
fun loadProfile(id: Int) {
  getProfile(token) { profile ->
    showProgile(profile)
  }
}
```

ブロッキングしない形にする

継続状況を持つプログラムとは

```
fun getProfile(id: Int, f: (Profile) -> Unit)
fun loadProfile(id: Int) {
  getProfile(token) { profile ->
    showProgile(profile)
  }
}
```

お分かりいただけただろうか・・・

継続状況を持つプログラムとは

```
fun getProfile(id: Int, f: (Profile) -> Unit)
fun loadProfile(id: Int) {
  getProfile(token) { profile ->
    showProgile(profile)
  }
}
```

継続状況を持つプログラムとは

```
fun getProfile(id: Int, f: (Profile) -> Unit)
fun loadProfile(id: Int) {
  getProfile(token) { profile ->
    showProgile(profile)
  }
}
```

関数はここで一度中断している！

継続状況を持つプログラムとは

```
fun getProfile(id: Int, f: (Profile) -> Unit)
fun loadProfile(id: Int) {
  getProfile(token) { profile ->
    showProgile(profile)
  }
}
```

再開している…!

継続状況を持つプログラムとは

```
fun getProfile(id: Int, f: (Profile) -> Unit)
fun loadProfile(id: Int) {
  getProfile(token) { profile ->
    showProgile(profile)
  }
}
```

継続状況を持つプログラムっ

継続状況を容易に書く

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

継続状況を容易に書く

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

※擬似コードです

継続状況を容易に書く

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

※擬似コード

getProfile関数が返すコルーチンに制御を渡し自身は中断する

継続状況を容易に書く

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

※擬似コード

getProfile関数が返すコルーチンは別のスレッドで動作しているかもしれないししてないかもしれない

継続状況を容易に書く

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

getProfile関数が完了したら値を受け取るところから再開

す

継続状況を容易に書く

コールバックスタイル

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

コルーチン(async/await)

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

※擬似コードです

継続状況が増えても..

コールバックスタイル

```
fun loadProfile(id: Int) {
  getProfile(id) { profile ->
    getReport(profile.id) { report ->
      //..
          }
        }
      }
    }
  }
}
```

継続状況が増えても..

コルーチン(async/await)

```
fun loadProfile(id: Int) {
    val profile = getProfile(id).await()
    val report = getReport(profile.id).await()
    val a = getA().await()
    val b = getB().await()
    val c = getC().await()
    //…
}
```

※擬似コードです

継続状況を簡単に書けることで
実装が楽になるものたち

- ジェネレータ、ストリーム
- チャンネル
- アクターモデル
- パイプライン
- etc…

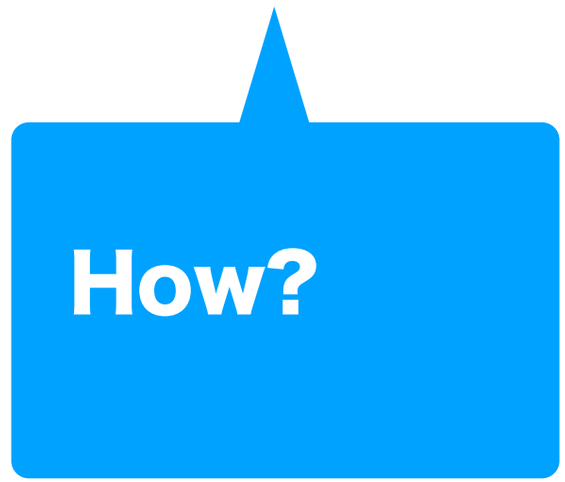
コルーチンとは…

- 一時停止可能な計算のインスタンス
- スレッドのように実行するコードブロックを持ち、同様のライフサイクルを持ち、作成され起動されるが、特定のスレッドに束縛されない
- FutureやPromissのように、何らかの結果や例外がある場合がある
- 継続状況を持つプログラムが容易に記述できる

Kotlinはどのように
コルーチンを実現しているのか？

Kotlinはどのように
コルーチンを実現しているのか？

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



KotlinはJVM言語

Kotlinコンパイラは Javaバイトコードを出力する

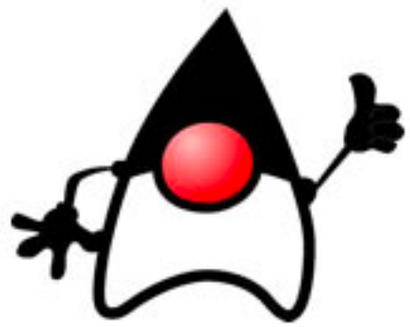
```
override fun onCreate(savedIns
    super.onCreate(savedInstar
    setContentView(R.layout.ac
    button.setOnClickListener

    }
    cancell_button.setOnClickL

    }
}
```

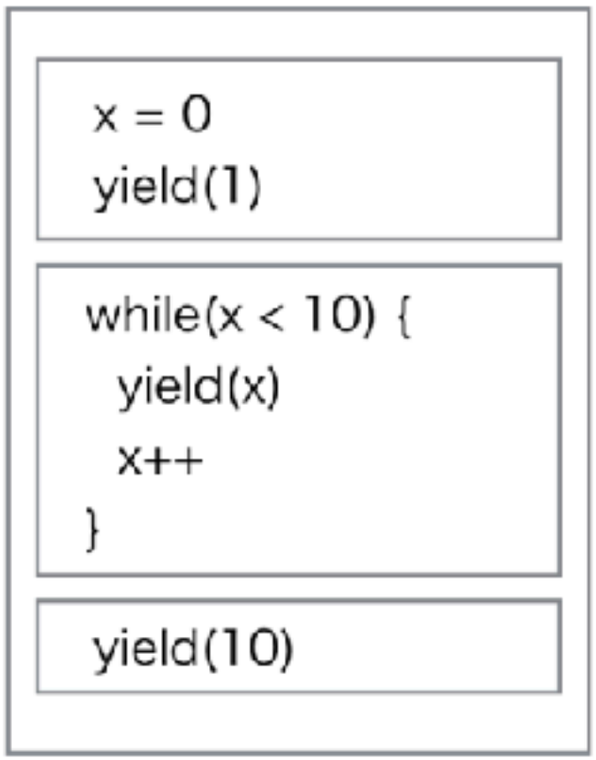
```
 INVOKESPECIAL androidx/
L1
 LINENUMBER 51 L1
 ALOAD 0
 LDC 2131296284
 INVOKEVIRTUAL com/sys1y
L2
 LINENUMBER 52 L2
 ALOAD 0
```

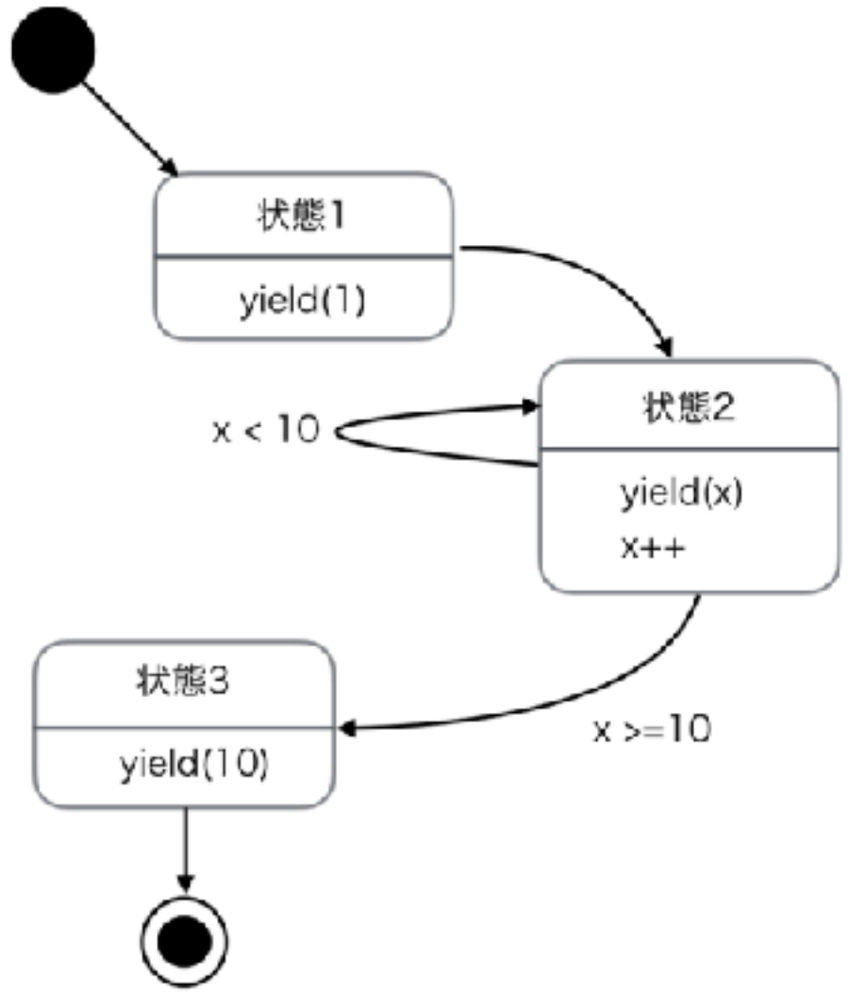
つまりKotlinのコルーチンはJavaで実装できる形になっているはず



Kotlinはどのように
コルーチンをJavaで実現して
いるのか？

コルーチンを
ステートマシンに変換している





https://github.com/Kotlin/kotlin-coroutines/blob/master/kotlin-coroutines-informal.md

コルーチンをステートマシンにする

```
fun simpleCoroutine() {
    val start = System.currentTimeMillis()
    println("start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}")
}
```

コルーチンをステートマシンにする

```
fun simpleCoroutine() {
    val start = System.currentTimeMillis()
    println("start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}")
}
```

ここで中断するとする

コルーチンをステートマシンにする

```
fun simpleCoroutine() {
    val start = System.currentTimeMillis()
    println("start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}")
}
```

２つの状態に分解できる

コルーチンをステートマシンにする

```
fun simpleCoroutine() {
    val start = System.currentTimeMillis()
    println("start")
    delay(1000)

    println("end ${System.currentTimeMillis() - start}")
}
```

横断的に利用する変数

コルーチンをステートマシンにする

```
class SimpleCoroutine {
    var label = 0
    var start = 0L
    fun resume() {
        when (label) {
            0 -> {
                start = System.currentTimeMillis()
                println(“start")
                label++
                delay(1000, this)
            }
            1 -> {
                println("end ${System.currentTimeMillis() - start}”)
                label++
            }
        }
    }
}
```

```
ss SimpleCoroutine {
  var label = 0
  var start = 0L
  fun resume() {
        when (label) {
              0 -> {
                    start = System.currentTi
                    println(“start")
                    label++
                    delay(1000, this)
              }
              1 -> {
                    println("end ${System.cu
                    label++
              }
        }
  }
}
```

```
ss SimpleCoroutine {
  var label = 0
  var start = 0L
  fun resume() {
      when (label) {
          0 -> {
              start = System.currentTi
              println(“start")
              label++
              delay(1000, this)
          }
          1 -> {
              println("end ${System.cu
              label++
          }
      }
  }
}
```

２つの状態を持つ

```
ss SimpleCoroutine {
  var label = 0
  var start = 0L
  fun resume() {
      when (label) {
          0 -> {
              start = System.currentTi
              println(“start")
              label++
              delay(1000, this)
          }
          1 -> {
              println("end ${System.cu
              label++
          }
      }
  }
}
```

現在の状態を表す

```
ss SimpleCoroutine {
  var label = 0
  var start = 0L
  fun resume() {
        when (label) {
              0 -> {
                    start = System.currentTi
                    println("start")
                    label++
                    delay(1000, this)
              }
              1 -> {
                    println("end ${System.cu
                    label++
              }
        }
  }
}
```

現在を変化させる

```
ss SimpleCoroutine {
  var label = 0
  var start = 0L
  fun resume() {
      when (label) {
          0 -> {
              start = System.currentTi
              println("start")
              label++
              delay(1000, this)
          }
          1 -> {
              println("end ${System.cu
              label++
          }
      }
  }
}
```

共通して使う変数

```
ss SimpleCoroutine {
  var label = 0
  var start = 0L
  fun resume() {
      when (label) {
          0 -> {
              start = System.currentTi
              println(“start")
              label++
              delay(1000, this)
          }
          1 -> {
              println("end ${System.cu
              label++
          }
      }
  }
}
```

処理を再開する関数

```
ass SimpleCoroutine {
  var label = 0
  var start = 0L
  fun resume() {
      when (label) {
          0 -> {
              start = System.currentTi
              println(“start")
              label++
              delay(1000, this)
          }
          1 -> {
              println("end ${System.cu
              label++
          }
      }
  }
}
```

```
fun delay(delayTime: Long, coroutine: SimpleCoroutine) {
  Thread {
    Thread.sleep(delayTime)
    coroutine.resume()
  }.start()
}
```

```
fun delay(delayTime: Long, coroutine: SimpleCoroutine) {
  Thread {
    Thread.sleep(delayTime)
    coroutine.resume()
  }.start()
}
```

再開のためにSimpleCoroutine
を受け取っておく

```
fun delay(delayTime: Long, coroutine: SimpleCoroutine) {
  Thread {
    Thread.sleep(delayTime)
    coroutine.resume()
  }.start()
}
```

指定された時間スリープして
SimpleCoroutineを再開する

```
ss SimpleCoroutine {
  var label = 0
  var start = 0L
  fun resume() {
        when (label) {
              0 -> {
                    start = System.currentTi
                    println(“start")
                    label++
                    delay(1000, this)
              }
              1 -> {
                    println("end ${System.cu
                    label++
              }
        }
  }
}
```

コルーチン

```
fun simpleCoroutine() {
    val start = System.currentTimeMillis()
    println("start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}")
}
```

ステートマシン

```
val simpleCoroutine = SimpleCoroutine()
simpleCoroutine.resume()
```

コルーチン

```
fun simpleCoroutine() {
    val start = System.currentTimeMillis()
    println("start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}")
}
```

start
end 1002

ステートマシン

```
val simpleCoroutine = SimpleCoroutine()
simpleCoroutine.resume()
```

start
end 1003

ステートマシンでコルーチンを実現する

- 中断と再開を状態遷移と見立てる
- 再開に必要な情報をステートマシン内にキャプチャする
- 内部状態を変化させながら実行する

どうやってコルーチンを
ステートマシンに変換するのか

suspend修飾子と継続

suspend修飾子

- コルーチンをマーキングするためにsuspend修飾子を導入
- ラムダ式にsuspend修飾子を付与するとコルーチン本体を表す(suspendラムダ)
- 関数に付与すると中断する場所を表す。その関数をsuspend関数と呼ぶ
- suspend関数はsuspend関数/ラムダからしか呼び出せない

継続

- コルーチン自身を継続オブジェクトにする
- suspend関数を継続渡しスタイル (Continuation Passing Style, CPS) に変換する
- suspend関数内で継続オブジェクトを取り出して任意に再開できるようにする

suspend関数とCPS

```
fun delay(delayTime: Long, coroutine: SimpleCoroutine) {
  Thread {
    Thread.sleep(delayTime)
    coroutine.resume()
  }.start()
}
```

suspend関数とCPS

```
suspend fun delay(delayTime: Long) {
  Thread {
    Thread.sleep(delayTime)
    coroutine.resume()
  }.start()
}
```

suspend関数とCPS

```
suspend fun delay(delayTime: Long) { // c: Continuation<T>
  Thread {
    Thread.sleep(delayTime)
    coroutine.resume()
  }.start()
}
```

コンパイル時に自動で
継続インタフェースの
引数が増える

suspend関数とCPS

```
suspend fun delay(delayTime: Long) { // c: Continuation<T>
  suspendCoroutine<Unit> { c ->
    Thread {
      Thread.sleep(delayTime)
      c.resume(Unit)
    }.start()
  }
}
```

suspend関数とCPS

```
suspend fun delay(delayTime: Long) { // c: Continuation<T>
  suspendCoroutine<Unit> { c ->
    Thread {
      Thread.sleep(delayTime)
      c.resume(Unit)
    }.start()
  }
}
```

継続インタフェースを取り出すために用意されている関数

suspend関数とCPS

```
suspend fun delay(delayTime: Long) { // c: Continuation<T>
  suspendCoroutine<Unit> { c ->
    Thread {
      Thread.sleep(delayTime)
      c.resume(Unit)
    }.start()
  }
}
```

継続インタフェースを使って再開処理を行う

suspendラムダとコルーチン

```
fun simpleCoroutine() {
  val start = System.currentTimeMillis()
  println("start")
  delay(1000)
  println("end ${System.currentTimeMillis() - start}")
}
```

suspendラムダとコルーチン

```
fun simpleCoroutine() {
  val f: suspend () -> Unit = {
    val start = System.currentTimeMillis()
    println(“start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}”)
  }

}
```

suspendラムダとコルーチン

```
fun simpleCoroutine() {
  val f: suspend () -> Unit = {
    val start = System.currentTimeMillis()
    println(“start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}”)
  }
}
```

suspendラムダによってコルーチンの範囲が決まる。

suspendラムダとコルーチン

```
fun simpleCoroutine() {
  val f: suspend () -> Unit = {
    val start = System.currentTimeMillis()
    println(“start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}”)
  }
  f.startCoroutine(NoOpCompletion)
}
```

suspendラムダとコルーチン

```
fun simpleCoroutine() {
  val f: suspend () -> Unit = {
    val start = System.currentTimeMillis()
    println(“start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}”)
  }
  f.startCoroutine(NoOpCompletion)
}
```

suspendラムダの拡張関数がいくつか用意されている。startCoroutine関数によってコルーチンを開始できる

suspendラムダとコルーチン

```
fun simpleCoroutine() {
  val f: suspend () -> Unit = {
    val start = System.currentTimeMillis()
    println(“start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}”)
  }
  f.startCoroutine(NoOpCompletion)
}
```

startCoroutine関数はContinuationを引数に取る。コルーチンが完了したら呼び出される

suspendラムダとコルーチン

```
fun simpleCoroutine() {
  val f: suspend () -> Unit = {
    val start = System.currentTimeMillis()
    println(“start")
    delay(1000)
    println("end ${System.currentTimeM
  }
  f.startCoroutine(NoOpCompletion)
}
```

冗長になるため便宜上用意したクラス

```
object NoOpCompletion<T> : Continuation<T> {
    override val context: CoroutineContext = EmptyCoroutineContext
    override fun resume(value: T) {
        // no op
    }
    override fun resumeWithException(exception: Throwable) {
        // no op
    }
}
```

suspendラムダとコルーチン

```
fun simpleCoroutine() {
  val f: suspend () -> Unit = {
    val start = System.currentTimeMillis()
    println(“start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}”)
  }
  f.startCoroutine(NoOpCompletion)
}
```

suspendラムダとコルーチン

```
fun simpleCoroutine() {
  val f: suspend () -> Unit = {
    val start = System.currentTimeMillis()
    println(“start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}”)
  }
  f.startCoroutine(NoOpCompletion)
}
```

start
end 1003

コルーチンビルダーと継続インターセプター

コルーチンを作るのはめんどくさい

```
fun simpleCoroutine() {
  val f: suspend () -> Unit = {
    val start = System.currentTimeMillis()
    println(“start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}”)
  }
  f.startCoroutine(NoOpCompletion)
}
```

コルーチンビルダー

```
fun simpleCoroutine() {
  launch {
    val start = System.currentTimeMillis()
    println(“start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}”)
  }
}
```

それぞれの性質をもつコルーチンビルダーがコルーチン標準ライブラリで提供されるので普段はそちらを使うことになる。

継続インターセプターと実行スレッド

```
launch(CommonPool) {
   // …
}
```

```
launch(UI) {
   // …
}
```

```
launch(JavaFx) {
   // …
}
```

```
launch(Swing) {
   // …
}
```

```
   async(CommonPool) {
      // …
   }
```

継続インターセプターと実行スレッド

```
launch(CommonPool) {
   // …
}
```

```
launch(UI) {
   // …
}
```

```
launch(JavaFx) {
   // …
}
```

```
launch(Swing) {
```

```
   async(CommonPool) {
      // …
   }
```

コルーチンビルダーに継続インターセプターを渡して実行スレッドをコントロールできる

Kotlinはどのように コルーチンをJavaで実現しているのか？

- コルーチンをステートマシンに変換している
- 変換のためにsuspend修飾子と継続を導入した
- 継続インターセプターにより、実行のスレッドを限定しない

Kotlin コルーチンの基本的な使い方

Kotlin1.1から登場

Kotlin 1.1 Released with JavaScript Support, Coroutines and more

Posted on March 1, 2017 by Roman Belov

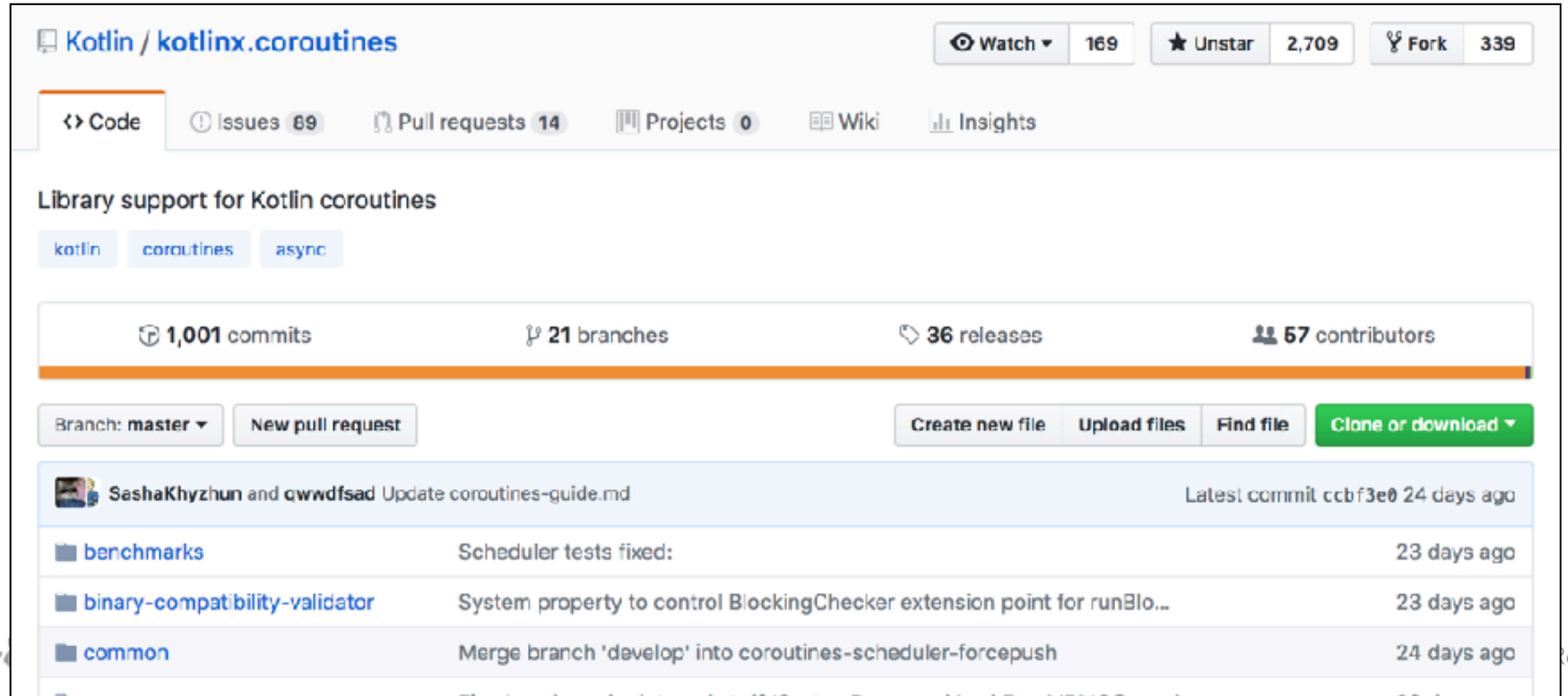
Members of our community have translated this blog post into several languages:

Today we release Kotlin 1.1. It's a big step forward enabling the use of Kotlin in many new scenarios, and we hope that you'll enjoy it.

各種機能はライブラリで提供

- core, reactive, native, ui, jsなどのモジュールに分かれている

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



特定の環境向けのライブラリもある

- Android, JavaFx, Swing向けが用意されている(継続インターセプターの実装)

https://github.com/Kotlin/kotlinx.coroutines/tree/master/ui

README.md

Coroutines for UI

This directory contains modules for coroutine programming with various single-threaded UI libraries. Module name below corresponds to the artifact name in Maven/Gradle.

Modules

- kotlinx-coroutines-android -- `UI` context for Android applications.
- kotlinx-coroutines-javafx -- `JavaFx` context for JavaFX UI applications.
- kotlinx-coroutines-swing -- `Swing` context for Swing UI applications.

experimental枠で提供

- kotlinx.coroutines.experimentalパッケージに関連するクラスがある
- 本番投入にはそれなりに覚悟が必要だったが..

```
package com.sys1yagi.kotlinfest2018.util

import kotlinx.coroutines.experimental.CommonPool
import kotlinx.coroutines.experimental.CoroutineScope
import kotlinx.coroutines.experimental.CoroutineStart
import kotlinx.coroutines.experimental.Job
import kotlinx.coroutines.experimental.launch
import kotlinx.coroutines.experimental.android.UI
import kotlin.coroutines.experimental.CoroutineContext


var EMPTY_JOB = Job()
var asyncContext: CoroutineContext = CommonPool
var uiContext: CoroutineContext = UI

fun <T> async(
        context: CoroutineContext = asyncContext,
        start: CoroutineStart = CoroutineStart.DEFAULT,
```

1.3でstableになる予定！！

See what's coming in Kotlin 1.3-M1

Posted on July 26, 2018 by Ilya Gorbunov

Today, after a long chain of incremental 1.2.X updates, it's time to see what's coming in Kotlin 1.3. We are happy to announce the first preview version of the new major release: Kotlin 1.3-M1.

Kotlin 1.3 brings many advancements including graduation of coroutines, new experimental unsigned arithmetic, and much more. Please read on for the details.

We'd like to thank our external contributors whose pull requests and commits were included in this release: Raluca Sauciuc, Toshiaki Kameyama, Leonardo Lopes, Jake Wharton, Jeff Wright, Lucas Smaira, Mon_chi, Nico Mandery, Oskar Drozda.

The complete list of changes in this release can be found in the changelog.

Coroutines are graduating to stable

Finally, coroutines will not be experimental anymore in 1.3. Both the syntax and the Standard Library APIs are stabilized and will remain backwards-compatible in the future.

Coroutines have been improved significantly since their introduction in 1.1. Notable features

※今回はまだexperimentalの環境の方で解説します

環境構築

app/build.gradle

```
kotlin {
    experimental {
        coroutines 'enable'
    }
}

dependencies {
  implementation”org.jetbrains.kotlin:kotlin-stdlib-jdk7:1.2.60"
  implementation “org.jetbrains.kotlinx:kotlinx-coroutines-core:0.24.0"

  // platformに合わせて追加
  implementation “org.jetbrains.kotlinx:kotlinx-coroutines-android:0.24.0"
}
```

環境構築

app/build.gradle

```
kotlin {
    experimental {
        coroutines 'enable'
    }
}

dependencies {
  implementation”org.jetbrains.kotlin:kotlin-stdlib-jdk7:1.2.60"
  implementation “org.jetbrains.kotlinx:kotlinx-coroutines-core:0.24.0"

  // platformに合わせて追加
  implementation “org.jetbrains.kotlinx:kotlinx-coroutines-android:0.24.0"
}
```

無くても動くけどIDE上とかで警告がでる

Coreライブラリが提供するコルーチンビルダー

Coroutine builder functions:

Name	Result	Scope	Description
launch	Job	CoroutineScope	Launches coroutine that does not have any result
async	Deferred	CoroutineScope	Returns a single value with the future result
produce	ReceiveChannel	ProducerScope	Produces a stream of elements
actor	SendChannel	ActorScope	Processes a stream of messages
runBlocking	T	CoroutineScope	Blocks the thread while the coroutine runs

Coreライブラリが提供するコルーチンビルダー

Coroutine builder functions:

Name	Result	Scope	Description
launch	Job	CoroutineScope	Launches coroutine that does not have any result
async	Deferred	CoroutineScope	Returns a single value with the future result
produce	ReceiveChannel	ProducerScope	Produces a stream of elements
actor	SendChannel	ActorScope	Processes a stream of messages
runBlocking	T	CoroutineScope	Blocks the thread while the coroutine runs

最初のコルーチン

```
launch {
    val start = System.currentTimeMillis()
    println("start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}")
}
```

最初のコルーチン

コルーチンビルダー

```
launch {
    val start = System.currentTimeMillis()
    println("start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}")
}
```

launch関数

```
public fun launch(
    context: CoroutineContext = DefaultDispatcher,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    parent: Job? = null,
    onCompletion: CompletionHandler? = null,
    block: suspend CoroutineScope.() -> Unit
): Job
```

- 結果を持たないコルーチンを作成するコルーチンビルダー関数

launch関数

```
public fun launch(
    context: CoroutineContext = DefaultDispatcher,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    parent: Job? = null,
    onCompletion: CompletionHandler? = null,
    block: suspend CoroutineScope.() -> Unit
): Job
```

コルーチン本体。CoroutineScopeの拡張suspendラムダ

launch関数

```
public fun launch(
    context: CoroutineContext = DefaultDispatcher,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    parent: Job? = null,
    onCompletion: CompletionHandler? = null,
    block: suspend CoroutineScope.() -> Unit
): Job
```

コルーチンの実行コンテキスト、共有データや継続インターセプターなどをここで指定できる

launch関数

```
public fun launch(
        context: CoroutineContext = DefaultDispatcher,
        start: CoroutineStart = CoroutineStart.DEFAULT,
        parent: Job? = null,
        onCompletion: CompletionHandler? = null,
        block: suspend CoroutineScope.() -> Unit
): Job
```

DefaultDispatcher

```
public actual val DefaultDispatcher: CoroutineDispatcher =
    if (useCoroutinesScheduler) ExperimentalCoroutineDispatcher() else CommonPool
```

DefaultDispatcher

```
tcher() else CommonPool
```

DefaultDispatcher

tcher() else CommonPool

スレッドプールを使ってコルーチンを実行する継続インターセプター。実行環境のCPUコア数に応じてスレッド数は変化する

launch関数

```
public fun launch(
    context: CoroutineContext = DefaultDispatcher,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    parent: Job? = null,
    onCompletion: CompletionHandler? = null,
    block: suspend CoroutineScope.() -> Unit
): Job
```

コルーチンの開始方法を指定する。デフォルトでは即座に開始する

launch関数

```
public fun launch(
    context: CoroutineContext = DefaultDispatcher,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    parent: Job? = null,
    onCompletion: CompletionHandler? = null,
    block: suspend CoroutineScope.() -> Unit
): Job
```

起動したコルーチンの完了を待ち合わせたりキャンセルできる

launch関数

```
public fun launch(
    context: CoroutineContext = DefaultDispatcher,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    parent: Job? = null,
    onCompletion: CompletionHandler? = null,
    block: suspend CoroutineScope.() -> Unit
): Job
```

親Jobを設定する。複数のコルーチンを一括でcancelしたい時などに使う

launch関数

```
public fun launch(
    context: CoroutineContext = DefaultDispatcher,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    parent: Job? = null,
    onCompletion: CompletionHandler? = null,
    block: suspend CoroutineScope.() -> Unit
): Job
```

コルーチンの完了をコールバックで受け取りたい場合に指定する

最初のコルーチン

```
launch {
    val start = System.currentTimeMillis()
    println("start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}")
}
```

最初のコルーチン

```
launch {
    val start = System.currentTimeMillis()
    println("start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}")
}
```

標準ライブラリが提供する
suspend関数

最初のコルーチン

```
launch {
    val start = System.currentTimeMillis()
    println("start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}")
}
```

start
end 1004

コルーチンと実行スレッド

```
launch {
    val start = System.currentTimeMillis()
    println("start")
    delay(1000)
    println("end ${System.currentTimeMillis() - start}")
}
```

どちらの状態もスレッドプールで実行される

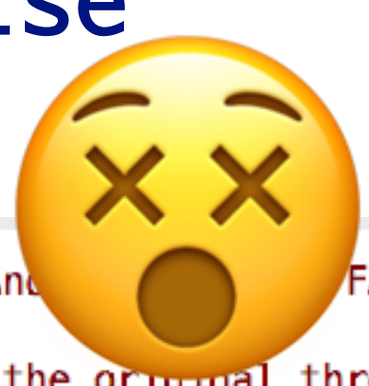
例えばUIを触るようなコルーチンだと…

```
launch {
  prograssBar.isVisible = true
  delay(1000)
  prograssBar.isVisible = false
}
```

例えばUIを触るようなコルーチンだと…

DefaultDispatcherはスレッドプールでコルーチンを実行する

```
launch {
    prograssBar.isVisible = true
    delay(1000)
    prograssBar.isVisible = false
}
```



```
--------- beginning of crash
2018-08-19 16:40:46.798 19431-20575/com.sys1yagi.kotlinfest2018 E/And                FATAL EXCEPTION: Fo
    Process: com.sys1yagi.kotlinfest2018, PID: 19431
    android.view.ViewRootImpl$CalledFromWrongThreadException: Only the original thread that created a
        at android.view.ViewRootImpl.checkThread(ViewRootImpl.java:7286)
        at android.view.ViewRootImpl.requestLayout(ViewRootImpl.java:1155)
        at android.view.View.requestLayout(View.java:21926)
        at android.view.View.requestLayout(View.java:21926)
        at android.view.View.requestLayout(View.java:21926)
        at android.view.View.requestLayout(View.java:21926)
        at android.view.View.requestLayout(View.java:21926)
        at android.view.View.requestLayout(View.java:21926)
        at android.view.View.requestLayout(View.java:21926)
        at androidx.constraintlayout.widget.ConstraintLayout.requestLayout(ConstraintLayout.java:3115
        at android.view.View.requestLayout(View.java:21926)
        at android.view.View.setFlags(View.java:13289)
        at android.view.View.setVisibility(View.java:9382)
        at com.sys1yagi.kotlinfest2018.MainActivity$simpleCoroutine$1.doResume(MainActivity.kt:206)
```

例えばUIを触るようなコルーチンだと…

```
launch(UI) {
    prograssBar.isVisible = true
    delay(1000)
    prograssBar.isVisible = false
}
```

例えばUIを触るようなコルーチンだと…

```
launch(UI) {
    prograssBar.isVisible = true
    delay(1000)
    prograssBar.isVisible = false
}
```

kotlinx-coroutines-androidが提供する継続インターセプター。常にUIスレッドで再開する

例えばUIを触るようなコルーチンだと…

```
launch(UI) {
  prograssBar.isVisible = true
  delay(1000)
  prograssBar.isVisible = false
}
```

UIスレッドで実行

async/await

```
launch(UI) {
  prograssBar.isVisible = true
  delay(1000)
  prograssBar.isVisible = false
}
```

async/await

```
class Profile(val name: String = "Jack")
fun loadProfile(): Profile {
    Thread.sleep(1000)
    return Profile()
}
```

```
launch(UI) {
  prograssBar.isVisible = true
  delay(1000)
  prograssBar.isVisible = false
}
```

async/await

```
class Profile(val name: String = "Jack")
fun loadProfile(): Profile {
    Thread.sleep(1000)
    return Profile()
}

launch(UI) {
  prograssBar.isVisible = true
  delay(1000)
  prograssBar.isVisible = false
}
```

擬似的な通信処理

async/await

```
class Profile(val name: String = "Jack")
fun loadProfile(): Profile {
    Thread.sleep(1000)
    return Profile()
}
```

```
launch(UI) {
  prograssBar.isVisible = true
  val profile = loadProfile()
  nameText.text = profile.name
  prograssBar.isVisible = false
}
```

async/await

```
class Profile(val name: String = "Jack")
fun loadProfile(): Profile {
    Thread.sleep(1000)
    return Profile()
}

launch(UI) {
  prograssBar.isVisible = true
  val profile = loadProfile()
  nameText.text = profile.name
  prograssBar.isVisible = false
}
```

ブロッキングする

async/await

```
class Profile(val name: String = "Jack")
fun loadProfile(): Profile {
    Thread.sleep(1000)
    return Profile()
}

launch(UI) {
  prograssBar.isVisible = true
  val profile = async { loadProfile() }.await()
  nameText.text = profile.name
  prograssBar.isVisible = false
}
```

async関数を使う

async関数

```
public fun <T> async(
    context: CoroutineContext = DefaultDispatcher,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    parent: Job? = null,
    onCompletion: CompletionHandler? = null,
    block: suspend CoroutineScope.() -> T
): Deferred<T>
```

- 結果を持つコルーチンを作成するコルーチンビルダー関数

Deferred<T>

```
public interface Deferred<out T> : Job {
    //…
    public suspend fun await(): T
    //…
}
```

中断して結果を待って再開する

async/await

```
class Profile(val name: String = "Jack")
fun loadProfile(): Profile {
    Thread.sleep(1000)
    return Profile()
}
```

```
launch(UI) {
  prograssBar.isVisible = true
  val profile = async { loadProfile() }.await()
  nameText.text = profile.name
  prograssBar.isVisible = false
}
```

async/await

```
class Profile(val name: String = "Jack")
fun loadProfile(): Profile {
    Thread.sleep(1000)
    return Profile()
}
```

```
launch(UI) {
  prograssBar.isVisible = true
  val profile = async { loadProfile() }.await()
  nameText.text = profile.name
  prograssBar.isVisible = false
}
```

Lang-8,Inc

async/await

```
class Profile(val name: String = "Jack")
fun loadProfile(): Profile {
    Thread.sleep(1000)
    return Profile()
}

launch(UI) {
  prograssBar.isVisible = true
  val profile = async { loadProfile() }.await()
  nameText.text = profile.name
  prograssBar.isVisible = false
}
```

継続を受け取って処理の完了を待つ

async/await

```
class Profile(val name: String = "Jack")
fun loadProfile(): Profile {
    Thread.sleep(1000)
    return Profile()
}

launch(UI) {
  prograssBar.isVisible = true
  val profile = async { loadProfile() }.await()
  nameText.text = profile.name
  prograssBar.isVisible = false
}
```

非同期処理が完了すると、UIスレッドで継続を再開する

async/await

```
class Profile(val name: String = "Jack")
fun loadProfile(): Profile {
    val profile = http.get(“http:/.”)
    return profile
}
```

```
launch(UI) {
  prograssBar.isVisible = true
  val profile = async { loadProfile() }.await()
  nameText.text = profile.name
  prograssBar.isVisible = false
}
```

コルーチンの基本的な使い方

- launch関数は結果が無いコルーチンを、async関数は結果があるコルーチンを作る
- デフォルトではCommonPoolが使われるので必要に応じて継続インターセプターをセットする
- launch関数とasync関数を組み合わせて使う(大体これで事足りる)

async/awaitと非同期処理

基本形

```
launch(UI) {
    prograssBar.isVisible = true
    val profile = async { loadProfile() }.await()
    nameText.text = profile.name
    prograssBar.isVisible = false
}
```

エラーハンドリング

```
launch(UI) {
  try{
     prograssBar.isVisible = true
     val profile = async { loadProfile() }.await()
     nameText.text = profile.name
     prograssBar.isVisible = false
  } catch (e: Exception) {
     showError(e)
  }
}
```

リクエストを直列で実行する

```
launch(UI) {
  try{
    prograssBar.isVisible = true
    val profile = async { loadProfile() }.await()
    nameText.text = profile.name
    prograssBar.isVisible = false
  } catch (e: Exception) {
    showError(e)
  }
}
```

リクエストを直列で実行する

```
launch(UI) {
  try{
    prograssBar.isVisible = true
    val token = async { getToken() }
    val profile = async { loadProfile(token.await()) }.await()
    nameText.text = profile.name
    prograssBar.isVisible = false
  } catch (e: Exception) {
    showError(e)
  }
}
```

リクエストを直列で実行する

```
launch(UI) {
  try{
    prograssBar.isVisible = true
    val token = async { getToken() }
    val profile = async { loadProfile(token.await()) }.await()
    nameText.text = profile.name
    prograssBar.isVisible = false
  } catch (e: Exception) {
    showError(e)
  }
}
```

ここでは中断せず開始だけしている

リクエストを直列で実行する

```
launch(UI) {
  try{
    prograssBar.isVisible = true
    val token = async { getToken() }
    val profile = async { loadProfile(token.await()) }.await()
    nameText.text = profile.name
    prograssBar.isVisible = false
  } catch (e: Exception) {
    showError(e)
  }
}
```

ここで結果を待っている

リクエストを並列で実行する

```
launch(UI) {
  try{
    prograssBar.isVisible = true
    val profile = async { loadProfile() }.await()
    val articles = async { loadArticles() }.await()
    show(profile, articles)
    prograssBar.isVisible = false
  } catch (e: Exception) {
    showError(e)
  }
}
```

リクエストを並列で実行する

```
launch(UI) {
  try{
    prograssBar.isVisible = true
    val profile = async { loadProfile() }.await()
    val articles = async { loadArticles() }.await()
    show(profile, articles)
    prograssBar.isVisible = false
  } catch (e: Exception) {
    showError(e)
  }
}
```

直列で実行されてしまう

リクエストを並列で実行する

```
launch(UI) {
  try{
    prograssBar.isVisible = true
    val profile = async { loadProfile() }
    val articles = async { loadArticles() }
    show(profile.await(), articles.await())
    prograssBar.isVisible = false
  } catch (e: Exception) {
    showError(e)
  }
}
```

リクエストを並列で実行する

```
launch(UI) {
  try{
     prograssBar.isVisible = true
     val profile = async { loadProfile() }
     val articles = async { loadArticles() }
     show(profile.await(), articles.await())
     prograssBar.isVisible = false
  } catch (e: Exception) {
     showError(e)
  }
}
```

並列で開始しておいて、待ち合わせるところで一気にawaitする

Cancel

```
launch(UI) {
  try{
     prograssBar.isVisible = true
     val profile = async { loadProfile() }.await()
     nameText.text = profile.name
     prograssBar.isVisible = false
   } catch (e: Exception) {
     showError(e)
   }
}
```

Cancel

```
val job = launch(UI) {
  try {
    prograssBar.isVisible = true
    val profile = async { loadProfile() }.await()
    nameText.text = profile.name
    prograssBar.isVisible = false
  } catch (e: CancellationException) {
    // cancel
  } catch (e: Exception) {
    showError(e)
  }
}

//…

job.cancel()
```

Cancel

```
val job = launch(UI) {
  try {
    prograssBar.isVisible = true
    val profile = async { loadProfile() }.await()
    nameText.text = profile.name
    prograssBar.isVisible = false
  } catch (e: CancellationException) {
    // cancel
  } catch (e: Exception) {
    showError(e)
  }
}

//…

job.cancel()
```

コルーチンを起動するときにJobを受け取っておけば任意のタイミングでキャンセルができる

Cancel

```
val job = launch(UI) {
  try {
    prograssBar.isVisible = true
    val profile = async { loadProfile() }.await()
    nameText.text = profile.name
    prograssBar.isVisible = false
  } catch (e: CancellationException) {
    // cancel
  } catch (e: Exception) {
    showError(e)
  }
}

//…

job.cancel()
```

キャンセルの例外が飛んでくるので
キャッチ

Cancel(Androidの例)

```
class MainActivity : AppCompatActivity() {
    var job: Job? = null
    //…
    fun doSomething(){
        job = launch(UI) { …
    }
    //…
    override fun onPause() {
        job?.cancel()
        super.onPause()
    }
}
```

Cancel(Androidの例)

```
class MainActivity : AppCompatActivity() {
    var job: Job? = null
    //…
    fun doSomething(){
        job = launch(UI) { …
    }
    //…
    override fun onPause() {
        job?.cancel()
        super.onPause()
    }
}
```

ライフサイクルに合わせてキャンセル

まとめてCancel

```
class MainActivity : AppCompatActivity() {
    var rootJob: Job? = null
```

まとめてCancel

```
class MainActivity : AppCompatActivity() {
    var rootJob: Job? = null
    override fun onResume() {
        super.onResume()
        rootJob = Job()
    }
```

まとめてCancel

```
class MainActivity : AppCompatActivity() {
    var rootJob: Job? = null
    override fun onResume() {
        super.onResume()
        rootJob = Job()
    }
    //…
fun doSomething(){
  launch(UI, parent = rootJob) { …
}
fun doSomething2(){
  launch(UI, parent = rootJob) { …
}
```

まとめてCancel

```
class MainActivity : AppCompatActivity() {
    var rootJob: Job? = null
    override fun onResume() {
        super.onResume()
        rootJob = Job()
    }
    //…
    fun doSomething(){
      launch(UI, parent = rootJob) { …
    }
    fun doSomething2(){
      launch(UI, parent = rootJob) { …
    }
    //…
    override fun onPause() {
        rootJob?.cancel()
        super.onPause()
    }
```

まとめてキャンセル

リトライ

```
launch(UI) {
    try {
        progressBar.isVisible = true
        val profile = async { loadProfile() }.await()
        nameText.text = profile.name
        progressBar.isVisible = false
    } catch (e: Exception) {
        showError(e)
    }
}
```

リトライ

```
launch(UI) {
    repeat(3) {
        try {
            progressBar.isVisible = true
            val profile = async { loadProfile() }.await()
            nameText.text = profile.name
            progressBar.isVisible = false
            return@launch
        } catch (e: Exception) {
            if (!shouldRetry(e)) {
                return@repeat
            }
        }
    }
    showError()
}
```

リトライ

```
launch(UI) {
    repeat(3) {
        try {
            progressBar.isVisible = true
            val profile = async { loadProfile() }.await()
            nameText.text = profile.name
            progressBar.isVisible = false
            return@launch
        } catch (e: Exception) {
            if (!shouldRetry(e)) {
                return@repeat
            }
        }
    }
    showError()
}
```

for文と一緒。これはコルーチンではない。

リトライ

```
launch(UI) {
    repeat(3) {
        try {
            progressBar.isVisible = true
            val profile = async { loadProfile() }.await()
            nameText.text = profile.name
            progressBar.isVisible = false
            return@launch
        } catch (e: Exception) {
            if (!shouldRetry(e)) {
                return@repeat
            }
        }
    }
    showError()
}
```

成功したらlaunchから抜ける

リトライ

```
launch(UI) {
    repeat(3) {
        try {
            progressBar.isVisible = true
            val profile = async { loadProfile() }.await()
            nameText.text = profile.name
            progressBar.isVisible = false
            return@launch
        } catch (e: Exception) {
            if (!shouldRetry(e)) {
                return@repeat
            }
        }
    }
    showError()
}
```

リトライすべきエラーでないならrepeatを抜ける

リトライ

```
launch(UI) {
    repeat(3) {
        try {
            progressBar.isVisible = true
            val profile = async { loadProfile() }.await()
            nameText.text = profile.name
            progressBar.isVisible = false
            return@launch
        } catch (e: Exception) {
            if (!shouldRetry(e)) {
                return@repeat
            }
        }
    }
    showError()
}
```

リトライが全て失敗したかリトライすべきでないエラーが発生場合

async/awaitと非同期処理

- try-catchでエラー処理
- 直列、並列は呼び出し方でコントロール
- Jobを使ってキャンセル
- リトライなどもループで処理できる

まだまだあるよコルーチン

RetrofitをDeferred化

```
@POST("/api/v1/articles/{id}/like")
fun like(@Path(“id") id: Long): Deferred<ResponseBody>

@DELETE(“/api/v1/articles/{id}/like")
fun likeCancel(@Path("id") id: Long): Deferred<ResponseBody>

@POST(“/api/v1/users/sign_up”)
fun signUp(@Body params: SignUpParameter): Deferred<SignInResponseEntity>

@POST("/api/v1/users/sign_in")
fun signIn(@Body params: SignInParameter): Deferred<SignInResponseEntity>
```

JakeWharton製。CallAdapterを実装してDeferred を返すようにしている。

https://github.com/JakeWharton/retrofit2-kotlin-coroutines-adapter

EventBusとChannel

```
class EventBus {
  private val channel = BroadcastChannel<Any>(1)
  fun send(event: Any, context: CoroutineContext = DefaultDispatcher) {
    launch(context) {
      channel.send(event)
    }
  }
  fun subscribe(): ReceiveChannel<Any> =
      channel.openSubscription()
  inline fun <reified T> subscribeToEvent() =
      subscribe().filter { it is T }.map { it as T }
}
```

https://gist.github.com/svenjacobs/57a21405b2dda4b62945c22235889d4a

EventBusとChannel

```
val bus: EventBus by inject()
override fun onResume() {
    super.onResume()
    launch(UI, parent = rootJob) {
        bus.subscribeToEvent<LogoutEvent>().consumeEach {
            // logged out
        }
    }
}
override fun onPause() {
    rootJob.cancel()
    super.onPause()
}

//...

bus.send(LogoutEvent)
```

onActivityResultをsuspend

```
launch(UI) {
  val result = activityResult(intent)
  if (result.isOk) {
    // ok
    val data: Intent? = result.flatMap()
  } else {
    // ng
  }
}
```

android-coroutinesというサードパーティライブラリが提供。requestPermissionもあるよ

https://github.com/pdvrieze/android-coroutines

Kotlinコルーチンまとめ

- 継続状況を持つプログラムが容易に記述できて便利
- Kotlinはコルーチンをステートマシンに変換して実現している
- async/awaitなどの各機能はKotlinの公式ライブラリが提供している
- launch関数とasync関数で大抵足りる
- 必要に応じて自分でコルーチンの仕組みを実装できる。すでにサードパーティライブラリがいくつかある

Enjoy Kotlin Coroutine!

Lang-8,Inc

ありがとうございました

Lang-8,Inc