Kotlin Coroutines を1年くらい サーバサイドで使ってみて

okue

Engineer Casual Technical Talks @ 2020/10/9

Target

- Kotlin Coroutines を(あんまり)使っていない Java/Kotlin の人に, Kotlin Coroutines をサーバサイドで使う雰囲気を伝える

そもそもコルーチンって?

- 中断可能な計算インスタンス

- 参考になりそうな資料
  - n月刊ラムダノート 「コルーチン」とは何だったのか
  - ALD Moura+, Revisiting Coroutines, 2009
    - コルーチンの分類についての話 (対称/非対称, stackful, delimited, one-shot, …)
  - Kotlin コルーチンを理解しよう (Kotlin Fest 2018)
  - Kotlin コルーチンを理解しよう 2019 (Kotlin Fest 2019)
  - KEEP にある Kotlin Coroutines のプロポーサル
  - Deep dive into Coroutines on JVM (KotlinConf 2017)
    - YouTube も
  - Suspend関数のはなし (60 pageの本, 100円)

Kotlin Coroutines って?

- 非同期/並行プログラミングの記述の簡単化
- Reactive Streams (Reactor, Rx), Completable/ListenableFuture などを統一的に記述

suspend が付いてる関数を suspend 関数と呼ぶ

suspending point

```
suspend fun getUserAddressAndPhoneNumber(): Pair<String, String> {
    val userId : String  = withContext(Dispatchers.IO) { this: CoroutineScope
        myRepository.getUserId() // blocking 用のスレッドプールで実行.
    }
    GlobalScope.launch { this: CoroutineScope
        kafkaClient.putEvent(userId) // 非同期になにかする. この実行の終了は待たない.
    }
    // getAddress と getPhoneNumber を並列に.
    val address : Deferred<String>  = GlobalScope.async { this: CoroutineScope
        myReactiveClient.getAddress(userId)
    }
    val phoneNumber : Mono<String!>  = myReactiveClient.getPhoneNumber(userId)
    return phoneNumber.awaitFirst() to address.await()
}
```

Outline

- 各種コルーチンの違い
  - 各種コルーチンビルダーや変換関数を雑観
  - async vs launch
  - flow vs sequence

- 各種ウェブフレームワークでの使い方
  - Armeria (Annotated services)
  - Armeria x grpc-kotlin
  - Spring WebFlux (Reactor)

- Summary

各種コルーチンの違い

各種コルーチンビルダーや変換関数

kotlinx-coroutines-core

- launch
  - 非同期に実行して値を返さない
- async
  - 非同期に実行して値を返す
- flow
  - 非同期に値の列を返す
- channel
  - メッセージの送受信を行うための
    キュー

kotlinx-coroutines-jdk8

- future (for CompletableFuture)
- CompletionStage.await

kotlinx-coroutines-reactor

- mono
- flux
- Flow.asFlux
- Deferred.asMono

kotlinx-coroutines-reacive

- publish
- Publisher.asFlow
- Flow.asPublisher
- Publisher.await{First, Single, Last}

async vs launch

- よく使うのはこの2つ
- 値を返す/返さないの違いに加えて, 例外ハンドリングが異なる

```
fun test(): Unit = runBlocking { this: CoroutineScope
    val job : Job  = GlobalScope.launch { this: CoroutineScope
        throw ArithmeticException("in launch")
    }
    job.join()
    val deferred : Deferred<Int>  = GlobalScope.async<Int> {
        throw IllegalArgumentException("in async")
    }
    try {
        deferred.await()
    } catch (e: IllegalArgumentException) {
        log.info { e.message }
    }
}
```

launch 内の例外は join 時に飛ばない.

Thread の uncaughtExceptionHandler が出力するのみで, エラーに気づけないことがある.

CoroutineExceptionHandler を ServiceLoader 経由で設定して, 自分の logger でログ出力するようにしておくとよい.

一方, async の場合は, await 時に例外が飛ぶ.

逆に, async に CoroutineExceptionHandler は効かないので注意.

ref: https://kotlinlang.org/docs/reference/coroutines/exception-handling.html

flow vs sequence

- どちらも値の列を作る関数で, 生成された列は map や filter といった演算を持つが...
- flow は非同期に消費される列を作るのに対し, sequence は同期的に消費される列を作る
  - i.e. sequence は CPU-bound な処理のときに使う
- sequence の中で, suspend 関数を呼べない

```
fun test2(): Unit = runBlocking { this: CoroutineScope
    val seq : Sequence<Int> = sequence { this: SequenceScope<Int>
        val deferred : Deferred<Int> = GlobalScope.async { 1 }
        yield(deferred.await())
        yield( value: 1)
    }
    val flo : Flow<Int> = flow { this: FlowCollector<Int>
        emit( value: 1)
        val deferred : Deferred<Int> = GlobalScope.async { 1 }
        emit(deferred.await())
        emit( value: 3)
    }
```

Restricted suspending functions can only invoke member or extension suspending functions on their restricted coroutine scope

ref: https://kotlinlang.org/docs/reference/coroutines/flow.html
https://medium.com/@elizarov/cold-flows-hot-channels-d74769805f9

各種ウェブフレームワークでの使い方

Coroutine Context

- Kotlin のコルーチンの実行は, スレッドに縛られない
- ウェブアプリケーションでは, Logback MDC, Zipkin trace context, Request context, Reactor context などコルーチンに伝えたいコンテキストを色々ある
- **Coroutine Context**
  - Job (失敗の伝搬のため)
  - CoroutineDispatcher (どのスレッドプールで実行するか)
  - CoroutineName (指定した文字列を Thread 名に追加する. デバッグ, ロギング用途)
  - ExceptionHandler (launch の投げる例外を拾う)
  - MDCContext (MDC を伝える)
  - ReactorContext (Reactor context を伝える)
  - ...

Armeria (Annotated services)

- Armeria 1.0.0 から, annotated service で suspend 関数を使えるようになった
- CoroutineContextService デコレータによって, メソッドやリクエストに応じたコンテキストを与えられる

```
Server.builder()
    .http(port)
    .annotatedService()
    .pathPrefix( pathPrefix: "/foo")
    .decorator(
        CoroutineContextService.newDecorator { ctx : ServiceRequestContext →
            CoroutineName( name: ctx.config().defaultServiceName() ?: "name")
        }
    )
```

- コルーチンは Armeria の context-aware event loop にディスパッチされる.

  @Blocking を付けると, context-aware blocking task executor にディスパッチされる.
  - Logback integration, Zipkin integration を使う場合は RequestContext の伝搬しないといけない => Armeria RequestContext を伝搬する CoroutineContext 例

ref: https://armeria.dev/docs/server-annotated-service/#kotlin-coroutines-support
https://github.com/line/armeria/tree/armeria-1.1.0/examples/annotated-http-service-kotlin

Armeria x grpc-kotlin

- grpc-kotlin の CoroutineContextServerInterceptor でコンテキストを与えられる

```
@Bean
fun myGrpcService(tracing: Tracing) = ArmeriaServerConfigurator { serverBuilder ->
    serverBuilder
        .service(
            GrpcService.builder()
                .addService(
                    ServerInterceptors.intercept(
                        GreeterImpl(),
                        coroutineContextInterceptor { _, _ ->
                            val ctx = ServiceRequestContext.current()
                            ctx.eventLoop().asCoroutineDispatcher() + ArmeriaRequestContext(ctx)
                        }
                    )
                )
                .supportedSerializationFormats(GrpcSerializationFormats.values())
                .enableUnframedRequests(true)
                .build(),
            BraveService.newDecorator(tracing),
```

ref: grpc-samples-in-kotlin

- Annotated services と同様, ユーザがディスパッチャを指定しなければ, コルーチンは context-aware な event loop/blocking task executor にディスパッチされる

WebFlux

- Spring WebFlux は coroutine support を謳っているが, Controller のメソッドに coroutine context を渡す術を提供していない
- そのため, メソッドの中身を, CoroutineContext を渡すためのボイラープレートで囲っていくことを強いられる (withDefaultContext {})

```
@GetMapping( ...value: "/api/bookings/{bookingId}")
suspend fun getBooking(@PathVariable bookingId: String): String = withDefaultContext {
    TODO()
}
```

- WebFilter などで, Reactor のコンテキストに値を突っ込んでおけば, コルーチンから参照できる (ref ReactorContext.kt)

ref: https://wiki.linecorp.com/pages/viewpage.action?pageId=2006025093

Summary?

最後に

- Kotlin Coroutines は便利でよい 😎

- 基本的には, 公式ガイド が充実している
  - 加えて, 作者 (Elizarov) のブログや KotlinConf の動画も