СПб АУ НОЦНТ РАН

Kotlin 05

27.11.2017

Sequence

```
// The elements might be calculated lazily
public interface Sequence<out T> {
    public operator fun iterator(): Iterator<T>
}
```

Sequence

```
val nums = generateSequence(1) { it + 1 }
println(nums.take(5).toList()) // prints [1, 2, 3, 4, 5]

// converting any iterable to Sequence
listOf(1, 2, 3).asSequence()
sequenceOf(1, 2, 3) // constant sequence
```

buildSequence

```
fun fibonacci() = buildSequence {
    var terms = Pair(0, 1)

    // this sequence is infinite
    while(true) {
        yield(terms.first)
        terms = Pair(terms.second, terms.first + terms.second)
    }
}

// [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
println(fibonacci().take(10).toList())
```

buildSequence

```
fun fibonacci() = buildSequence {
    var terms = Pair(0, 1)

    // this sequence is infinite
    while(true) {
        yield(terms.first)  // magic call
        terms = Pair(terms.second, terms.first + terms.second)
    }
}

// [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
println(fibonacci().take(10).toList())
```

Async pattern

```
// Abstract framework
while (true) {
    val task = tasks.poll()
    task.execute()
}
```

Async pattern

```
// Client task code
lightWeightOperation()
runSomeHeavyOperationWithCallBack {
    tasks.add {
        // continue in the framework thread
    }
}
```

UI

```
// Client task code
component.text = "Loading..." // lightweight
executorService.submit {
    val result = performHttpRequest()
    UIFramework.invokeLater {
        component.text = result
    }
}
```

Async Server

```
val ourResponse = Response()
startConnectionToAnotherServiceWithCallback {
    connection ->
        connection.sendData(data) {
            connection.receiveData {
                result ->
                ourResponse.result = result
                clientConnection.sendResponse(ourResponse)
            }
        }
}
```

Корутины/Continuation pattern

- Сложная языковая конструкция
- Дает возможность "замораживать"вычисления с возможностью их потом продолжать
- Очень просты в использовании с точки зрения пользовательского кода

suspend fun

```
suspend fun foo(): String {
    return "OK"
}

suspend fun bar() {
    println(foo())
}

fun baz() {
    // Error: suspend call from non-suspend function
    println(foo())
}
```

Continuation

```
public interface Continuation<in T> {
    public fun resume(value: T)
}
```

suspend fun

```
// Return type moved to Continuation
fun foo(c: Continuation<String>): Unit {
    c.resume("OK")
}

fun bar(c: Continuation<Unit>) {
    foo(object : Continuation<String> {
        override fun resume(value: String) {
            println(value)
        }
    })
}
```

suspendCoroutine

```
suspend fun <T> suspendCoroutine(
    block: (Continuation<T>) -> Unit
): T

suspend fun foo(): String {
    return suspendCoroutine {
        continuation -> continuation.resume("OK")
    }
}
```

suspendCoroutine

```
val task: CompletableFuture<String> = ..
suspend fun foo() = suspendCoroutine<String> {
    continuation ->
    task.whenComplete {
        result, throwable ->
        continuation.resume(result)
    }
}
```

Stack unwinding

- Когда futute заканчивается, вызывается continuation.resume("OK") из “foo”
- Но continuation там, это объект, который мы передаели в “bar”
- То есть в итоге мы возвращаемся к вызову “println“ из “bar”

startCoroutine

```
fun launch(suspendLambda: suspend () -> Unit) {
    // This call works until first suspend call
    suspendLambda.startCoroutine(object : Continuation<Unit> {
        override fun resume(value: Unit) {
            println("lambda has been completed")
        }
    })
}

fun main(args: Array<String>) {
    launch {
        bar()
    }
}
```

startCoroutine

```
fun launch(suspendLambda: suspend () -> Unit) {
    // This call works until first suspend call
    suspendLambda.startCoroutine(object : Continuation<Unit> {
        override fun resume(value: Unit) {
            println("lambda has been completed")
        }
    })
}

fun main(args: Array<String>) {
    launch {
        bar()
    }

    // Waiting for completable future to complete
    Thread.sleep(100000)
}
```

Создаем Future

```
fun launch(suspendLambda: suspend () -> Unit): Future<Unit> {
    val completableFuture = CompletableFuture<Unit>()
    suspendLambda.startCoroutine(object : Continuation<Unit> {
        override fun resume(value: Unit) {
            completableFuture.complete(Unit)
            println("lambda has been completed")
        }
    })

    return completableFuture
}

fun main(args: Array<String>) {
    launch {
        bar()
    }.get()
}
```

buildSequence

```
interface SequenceBuilder<in T> {
    suspend fun yield(value: T)
}
```

buildSequence

```
class IteratorImpl<T> : SequenceBuilder<T>, Iterator<T> {
    var nextContinuation: Continuation<Unit> = ...
    var nextResult = ..
    suspend override fun yield(value: T) {
        return suspendCoroutine {
            continuation ->
            nextResult = value
            nextContinuation = continuation
        }
    }

    // try run next piece of computation
    override fun hasNext() = ..
    override fun next(): T = nextResult
}
```

buildSequence

```
fun <T> buildIterator(
        x: suspend SequenceBuilder<T>.() -> Unit
): Iterator<T> {
    val iterator = IteratorImpl<T>()
    x.createCoroutine(iterator, object : Continuation<Unit> {
        override fun resume(value: Unit) {
            iterator.nextContinuation = null
        }
    })

    return iterator
}
```

Конечный автомат для корутины

```
suspend fun foo() {
    println(suspendCall())
}
```

Конечный автомат для корутины

```
fun foo(outerContinuation: Continuation<Unit>) {
    object : Continuation<Any?> {
        var label = 0
        override fun resume(value: Any?) {
            when (label) {
                0 -> {
                    label = 1
                    return suspendCall(this)
                }
                1 -> {
                    label = -1 // end of the function
                    println(value as String)
                    return
                }
            }
        }
    }.resume(null)
}
```

Обработка исключений

```
public interface Continuation<in T> {
    public fun resume(value: T)
    public fun resumeWithException(exception: Throwable)
}
```

Обработка исключений

```
val task: CompletableFuture<String> = ..
suspend fun foo() = suspendCoroutine<String> {
    continuation ->
    task.whenComplete {
        result, throwable ->
        if (throwable != null) {
            continuation.resumeWithException(throwable)
        } else {
            continuation.resume(result)
        }
    }
}
```

Быстрый возврат значения

```
suspend fun foo(): String
// Why Any?
fun foo(c: Continuation<String>): Any?
```

Быстрый возврат значения

```
// magic constant
val COROUTINE_SUSPENDED: Any = Any()

// magic intrinsic
suspend fun <T> suspendCoroutineOrReturn(
    block: (Continuation<T>) -> Any?
): T
```

Быстрый возврат значения

- Каждая suspend-функция может вернуть значение сразу
- Либо вернуть специальное значение – COROUTINE SUSPENDED, если все-таки "заморозка"произошла

Быстрый возврат значения

```
// Actually, it's even more complicated
fun <T> suspendCoroutine(
    block: (Continuation<T>) -> Unit
): T =  suspendCoroutineOrReturn { c ->
            block(c)
            COROUTINE_SUSPENDED
        }
```

Контекст корутины

```
public interface Continuation<in T> {
    // May be considered as a map of properties
    public val context: CoroutineContext
    public fun resume(value: T)
    public fun resumeWithException(exception: Throwable)
}
```

Интерсептор

- ContinuationInterceptor – один из "ключей"контекста
- Позволяет "оборачивать"исходные объекты continuation, потенциально изменяя семантику
- Одно из применений – вызов исходного continuation в рамках лямбды для “invokeLater” UI-фреймворка (Swing, Android)

Интерсептор

```
interface ContinuationInterceptor {
    fun <T> interceptContinuation(
            continuation: Continuation<T>
    ): Continuation<T>
}

object SwingInterceptor : ContinuationInterceptor {
    override fun <T> interceptContinuation(
            continuation: Continuation<T>
    ): Continuation<T> = object : Continuation<T> {
        override val context: CoroutineContext
            get() = continuation.context

        override fun resume(value: T) {
            SwingUtilities.invokeLater { continuation.resume(value) }
        }
        ...
    }
```

Интерсептор

```
// ‘launch‘/‘Swing‘ are from kotlinx.coroutines library
launch(Swing) {
    for (i in 1..10) {
        // ’append’ method and consequent
        // ’jProgressBar.setValue’ are called
        // within Swing event dispatch thread
        jTextArea.append(
                startLongAsyncOperation(i).await()
        )
        jProgressBar.value = i * 10
    }
}
```

kotlinx.coroutines

- https://github.com/Kotlin/kotlinx.coroutines
- Свои примитивы для многопоточного кода на корутинах: задачи/пулы задач и т.д.
- Расширения для популярных асинхронных библиотек (CompletableFutures, Rx, ..)
- Запуск асинхронных задач в рамках UI (Swing, Android)

Статус корутин

- Пакет kotlin.coroutines.experimental
- Возможно какие-то части немного изменятся
- По возможности будут предоставлены инструменты для миграции старого кода
- Уже используются преданными поклонниками Kotlin в своих проектах

Домашнее задание

- Ветка “04-fun-dbg”