

АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 8

ПЛАН

- Inline-функции

МОТИВАЦИЯ

- Лямбды, блочные конструкции - это очень красиво и идиоматично
- Но может дорого обходиться - по времени работы
- Особенно если используется где-то внутри вложенных циклов
- Яркий пример - что-то типа `synchronized`

INLINE

- `inline` перед функцией - функция вставляется прямо в место вызова
- По умолчанию `lambda`-параметры - тоже
- Можно индивидуально ограничить - через ключевое слово `noinline`
- Перед каждый параметром индивидуально

МОТИВАЦИЯ NOINLINE

- Для ситуаций, плохо совместимых с inline-значениями лямбд
- Пример: передача параметра в другую функцию - не inline
- Или сохранение в поле

ПРИМЕР NOINLINE

```
1 fun vvv2(action: () -> Unit) {  
2     println("before")  
3     action()  
4     println("after")  
5 }
```

ПРИМЕР NOINLINE

```
1 inline fun vvv(v: Int, noinline action: () -> Unit) {  
2     println("before")  
3     action()  
4     if (v > 0) {  
5         vvv2(action)  
6     }  
7     println("after")  
8 }  
9  
10 fun main() {  
11     vvv(10) {  
12         println("Hi!")  
13     }  
14 }
```

ПРИМЕР ОСМЫСЛЕННОГО RETURN

```
1 // важен не этот inline, а тот, который в fold
2 inline fun List<Int>.prod(): Int {
3     return this.fold(1) { curr, acc ->
4         println("curr: ${curr}")
5         if (curr == 0) return 0
6         curr * acc
7     }
8 }
```


PRO/CONTRA

- +: существенное ускорение
- -: увеличение байткода
- +/-: разрешается return из lambda
- Можно использовать для оптимизации, можно обжечься

REIFIED: МОТИВИРУЮЩИЙ ПРИМЕР

```
1 // не компилируется
2
3 fun<T> List<Any>.filterByType(): List<T> = filter {
4     it is T
5 }
6
7 fun main() {
8     println(listOf("123", 2322).findByType<String>())
9     println(listOf("123", 2322).findByType<Int>())
10 }
```

ИДЕЯ

- Если inline - мы знаем тип в данной точке вызова
- Можно сделать "подстановку"
- Решает только часть проблем

ЧАСТИЧНОЕ РЕШЕНИЕ

```
1 fun<T> List<Any>.findByType(): List<Any> = filter {  
2     it is T  
3 }  
4  
5 fun main() {  
6     println(listOf("123", 2322).findByType<String>())  
7     println(listOf("123", 2322).findByType<Int>())  
8 }
```

ПОКРАСИВЕЕ

```
1 inline fun<reified T> List<Any>.findByType(): List<T> {  
2     val result = mutableListOf<T>()  
3     forEach {  
4         if (it is T) {  
5             result.add(it)  
6         }  
7     }  
8     return result.toList()  
9 }
```

ТАК НЕЛЬЗЯ

```
1 inline fun<reified T> List<Any?>.findByType(): List<T?> {  
2     val result = mutableListOf<T?>()  
3     forEach {  
4         if (it is T || it == null) {  
5             result.add(it)  
6         }  
7     }  
8     return result.toList()  
9 }
```

ТАК МОЖНО

```
1 inline fun<reified T> List<Any?>.findByType(): List<T?> {  
2     val result = mutableListOf<T?>()  
3     forEach {  
4         if (it is T?) {  
5             result.add(it)  
6         }  
7     }  
8     return result.toList()  
9 }
```

И ТАК ТОЖЕ

```
1 inline fun<reified T> List<Any?>.findByType(): List<T> =
2 mutableListOf<T>().apply {
3     this@findByType.forEach {
4         if (it is T) {
5             add(it)
6         }
7     }
8 }.toList()
9 fun main() {
10     println(listOf("123", 2322, null)
11         .findByType<String?>())
12     println(listOf("123", 2322, null)
13         .findByType<String>())
14 }
```


МНОГОПОТОЧНОСТЬ В JVM

- Thread - понятие, узаконенное в языке и vm
- Есть формализованная модель памяти
- Есть механизм синхронизации и volatile-переменные
- Из коробки - много надстроек
- Пулы нитей, локи, семафоры, блокирующиеся очереди и т.п.

ПРИМЕРЫ ПРИЛОЖЕНИЙ

- Торрент-клиент
- Сервис-агретатор, делегирующий запросы партнерам
- Web-crawler
- Клиентское приложение мессенджера
- Прием данных из брокера сообщений

BASELINE-ПОДХОДЫ И ИХ ПРОБЛЕМЫ

- Чистые нити - недостаточно абстрактны
- Слишком дороги, чтобы бесконтрольно выделяться
- Коммуникация через wait-notity недостаточно абстрактна
- Реальный baseline: пулы нитей, атомики, коллекции из `java.util.concurrent`

BASELINE-ПОДХОДЫ И ИХ ПРОБЛЕМЫ

- Пулы добавляют сценарии для дедлоков
- Усложняется обработка ошибок
- Нет коробочного решения для базовых шаблонов типа "асинхронная цепочка"
- Нарушается принцип single-responsibility
 - Логика распараллеливания смешивается с бизнес-логикой

ПОДХОДЫ К РЕШЕНИЮ

- Future в Scala - решает задачу асинхронной цепочки
- Асинхронный контейнер, вписанный в структуру коллекций
- Можно делать отложенные map-ы, порождать производную Future по завершении текущей
- Можно унифицированно обрабатывать ошибки или делегировать обработку
- Легкий объект, но под капотом - конкурентная многопоточность

ПОДХОДЫ К РЕШЕНИЮ

- Функциональные фреймворки (Monix, Cat Effects) - обеспечивают кооперативную многопоточность
- Требуют освоения неочевидных сходупонятий
- RxJava/RxKotlin - реализация парадигмы реактивного программирования
- Фокусируются скорее на структурах
- Неплохо отделяют структуры от бизнес-логики

КОРУТИНЫ

- Корутины - механизм кооперативной многозадачности
- Классическая корутина - это кусок логики, предполагающий ожидание чего-либо
- Таймаута, асинхронного ввода, ответа на запрос, завершения другой корутины
- В момент ожидания она дает возможность поработать другим

ПРИМЕР

```
1 import kotlinx.coroutines.*
2
3 fun main() = runBlocking {
4     println("start")
5     launch {
6         delay(1000L)
7         println("World!")
8     }
9     println("Hello")
10 }
```


РАЗБЕРЕМ

- `runBlocking` - портал в мир корутин
- Создает контекст, запускает первую корутину, ожидает завершения
- `launch` запускает вторую корутину в контексте
- Вторая корутина засыпает "правильным" способом

РАЗБЕРЕМ

- Во время сна она не занимает нить
- Корутинный фреймворк освободившуюся нить может отдать кому-то
- В нашем случае - первой корутине
- По истечении таймаута - корутина снова займет какую-то нить

МНОГО КОРУТИН

```
1 import kotlinx.coroutines.*
2
3 fun main() = runBlocking {
4     repeat(50_000) {
5         launch {
6             delay(5000L)
7             print(".")
8         }
9     }
10 }
```

НИТИ

```
1 import kotlinx.coroutines.*
2
3 fun main() = runBlocking {
4     repeat(100) {
5         launch {
6             delay(5000L)
7             println(Thread.currentThread().id)
8         }
9     }
10 }
```

ДЕКОМПОЗИЦИЯ

- Засыпающие функции помечаются ключевым словом `suspend`
- Функция, вызывающая `suspend`-функции, обязана быть `suspend`-функцией
- Вызов `suspend`-функции вне корутинного контекста запрещен
- `launch` возвращает объект класса `Job`

JOB

- Над Job можно вызвать `join` - и дождаться завершения
- Можно вызвать `cancel` - в надежде прервать работу Job-а
- И это случится, как только Job войдет в `suspend`- режим
- Интересен эффект от брошенного исключения в рамках порожденной корутины

ПРИМЕР

```
1 import kotlinx.coroutines.*
2
3 fun main() = runBlocking {
4     val job = launch {
5         println("job-1")
6         delay(1000L)
7         //Thread.sleep(1000L)
8         println("job-2")
9     }
10 // .....
```

ПРИМЕР

```
1 // .....
2     println("out-1")
3     delay(10)
4     println("out-2")
5     job.join()
6     println("out-3")
7 }
```


ПРИМЕР

```
1 fun main() = runBlocking {  
2     val job = launch {  
3         println("job-1")  
4         delay(1000L)  
5         //Thread.sleep(1000L)  
6         println("job-2")  
7     }  
8     println("out-1")  
9     delay(10)  
10    println("out-2")  
11    job.cancel()  
12    println("out-3: " + job.isCancelled)  
13    delay(1)  
14 }
```

ПРИМЕР

```
1 fun main() = runBlocking {  
2     val job = launch {  
3         println("job-1")  
4         delay(1000L)  
5         throw RuntimeException()  
6         println("job-2")  
7     }  
8     // .....
```

ПРИМЕР

```
1 // .....
2     try {
3         println("out-1")
4         delay(10)
5         println("out-2")
6         Thread.sleep(500)
7         println("out-3: " + job.isCancelled)
8         delay(1000)
9     } catch (t: Throwable) {
10         println("GOT: " + t)
11         t.printStackTrace(System.out)
12     }
13 // .....
```

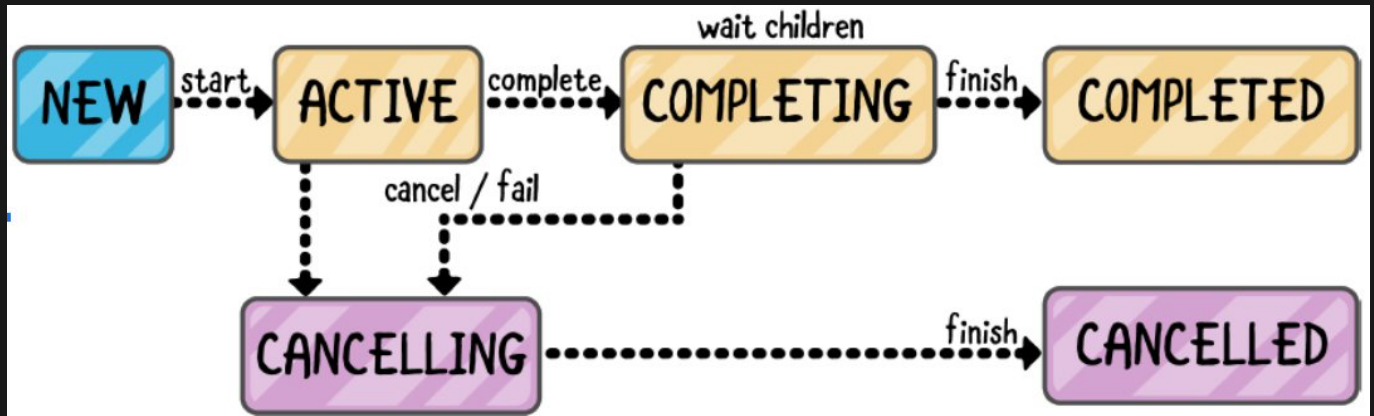
ПРИМЕР

```
1 // .....  
2     println("out-4")  
3     delay(1000)  
4     println("out-5")  
5 }
```

ПОЧЕМУ ТАК

- Концептуально корутина и порожденные ею корутины считаются частью единого решения
- И по умолчанию - если случается проблема в одном компоненте, это проблема всего решения
- Этого можно избежать, подробности попозже
- Если не избежать, корутина может еще доработать до первого suspend-a

ДИАГРАММА СОСТОЯНИЙ



ПРИМЕР

```
1 fun main() {  
2     runBlocking(Dispatchers.Default) {  
3         val job = Job()  
4         println(job)  
5         println(job.isActive)  
6         job.complete()  
7         println(job)  
8         println(job.isActive)  
9         println(job.isCompleted)  
10  
11 // .....
```

ПРИМЕР

```
1 // .....
2
3     val job2 = launch {
4         delay(2000)
5     }
6     println(job2)
7     println(job2.isActive)
8     job2.join()
9     println(job2)
10    println(job2.isActive)
11    println(job2.isCompleted)
12 }
13 }
```


ПРИМЕР

```
1 fun main() {  
2     runBlocking(Dispatchers.Default) {  
3         val job = launch(start=CoroutineStart.LAZY) {  
4             delay(2000)  
5         }  
6         println(job)  
7         println(job.isActive)  
8         job.start()  
9     }  
    // .....  
}
```

ПРИМЕР

```
1 // .....
2     println(job)
3     println(job.isActive)
4     job.join()
5     println(job)
6     println(job.isActive)
7     println(job.isCompleted)
8 }
9 }
```

ПРИМЕР С ДЕТЬМИ

```
1 fun main() {  
2     runBlocking(Dispatchers.Default) {  
3         val job = launch {  
4             launch {  
5                 delay(10000)  
6             }  
7             launch {  
8                 delay(20000)  
9             }  
10 // .....  
}
```

ПРИМЕР С ДЕТЬМИ

```
1 // .....
2         delay(1000)
3         println(
4             coroutineContext.job.children.toList()
5         )
6         delay(2000)
7     }
8
9     println(
10         "1: "
11         + coroutineContext.job.children.toList()
12     )
13     delay(500)
14 // .....
```

ПРИМЕР С ДЕТЬМИ

```
1 // .....
2     println("3: " + coroutineContext.job.isActive)
3     println(
4         "3: "
5         + coroutineContext.job.children.toList()
6     )
7     delay(8000)
8 // .....
```

ПРИМЕР С ДЕТЬМИ

```
1 // .....
2     println(
3         "4: "
4         + coroutineContext.job.isActive
5     )
6     println(
7         "4: "
8         + coroutineContext.job.children.toList()
9     )
10    delay(8000)
11 // .....
```

ПРИМЕР С ДЕТЬМИ

```
1 // .....
2     println(
3         "5: "
4         + coroutineContext.job.isActive
5     )
6     println(
7         "5: "
8         + coroutineContext.job.children.toList()
9     )
10 }
11 }
```

ПРИМЕР: CANCEL ИЗНУТРИ

```
1 fun main() {  
2     runBlocking(Dispatchers.Default) {  
3         launch {  
4             launch {  
5                 launch {  
6                     println("one")  
7                     delay(1000)  
8                     println("two")  
9                     cancel()  
10                    delay(1000)  
11                    println("three")  
12                }  
13 // .....  
}
```


ПРИМЕР: CANCEL ИЗНУТРИ

```
1 // .....
2
3         println("wait 10s")
4         delay(10000)
5         println("wait 10s done")
6     }
7     launch {
8         println("wait 20s")
9         delay(20000)
10        println("wait 20s done")
11    }
12 // .....
```

ПРИМЕР: CANCEL ИЗНУТРИ

```
1 // .....
2
3         delay(1000)
4         println(coroutineContext.job.children.toList())
5         delay(2000)
6     }
7 }
8 }
```

ПРИМЕР: CANCEL ИЗНУТРИ С ДЕТЬМИ

```
1 fun main() {  
2     runBlocking(Dispatchers.Default) {  
3         launch {  
4             launch {  
5                 launch {  
6                     println("one")  
7                     delay(1000)  
8                     println("two")  
9                     delay(1000)  
10                    println("three")  
11                }  
12            }  
13        }  
14    }  
15 }  
16 // .....  
17 }
```

ПРИМЕР: CANCEL ИЗНУТРИ С ДЕТЬМИ

```
1 // .....
2         println("wait 10s")
3         cancel()
4         delay(10000)
5         println("wait 10s done")
6     }
7     launch {
8         println("wait 20s")
9         delay(20000)
10        println("wait 20s done")
11    }
12 // .....
```

ПРИМЕР: CANCEL ИЗНУТРИ С ДЕТЬМИ

```
1 // .....
2         delay(1000)
3         println(
4             coroutineContext.job.children.toList()
5         )
6         delay(2000)
7     }
8 }
9 }
```

СВОИ SUSPEND-ФУНКЦИИ

- Пока мы вызывали `delay` и служебные методы
- Хочется создавать свои `suspend`-функции
- Чтобы они ожидали каких-то наших асинхронных событий

ИДЕЯ

- Есть интерфейс Continuation
- Каждой корутине соответствует объект, реализующий его
- Через него можно "разбудить" корутину
- Можно получить этот объект

ПРИМЕР

```
1 class CoroutineSocket() {  
2     val socket = AsynchronousSocketChannel.open()  
3     var isConnected: Boolean = false  
4     private set  
5     val isOpened: Boolean  
6     get() = socket.isOpen  
7     suspend fun connect(isa: InetSocketAddress) {  
8         suspendCoroutine {  
9             socket.connect(  
10                 isa, it, ContinuationHandler(this)  
11             )  
12         }  
13     }  
14 // .....
```


ПРИМЕР

```
1 // .....
2     companion object {
3         class ContinuationHandler<T>(
4             private val socket: CoroutineSocket
5             ): CompletionHandler<T, Continuation<T>> {
6             override fun completed(
7                 result: T, attachment: Continuation<T>
8             ) {
9                 socket.isConnected = true
10                attachment.resume(result)
11            }
12 // .....
```

ПРИМЕР

```
1 // .....
2         override fun failed(
3             exc: Throwable,
4             attachment: Continuation<T>
5         ) {
6             attachment.resumeWithException(exc)
7         }
8     }
9 }
10 }
11 // .....
```

ПРИМЕР

```
1 // .....
2 suspend fun main() {
3     val socket1 = CoroutineSocket()
4     println(socket1.isConnected)
5     socket1.connect(InetSocketAddress("www.ya.ru", 443))
6     println(socket1.isConnected)
7 // .....
```

ПРИМЕР

```
1 // .....
2     try {
3         val socket2 = CoroutineSocket()
4         println(socket2.isConnected)
5         socket2.connect(
6             InetAddress("www.yaya.ru", 443)
7         )
8         println(socket2.isConnected)
9     } catch (t: Throwable) {
10        println("done: " + t)
11    }
12 }
```

ПРИМЕР

```
1 object NetUtils {  
2     suspend fun isAlive(  
3         host: String, port: Int  
4     ): Boolean {  
5         return suspendCoroutine {  
6             val socket = AsynchronousSocketChannel.open()  
7             socket.connect(  
8                 InetSocketAddress(host, port),  
9                 it,  
10                ContinuationHandler(socket)  
11            )  
12        }  
13    }  
14    // .....
```

ПРИМЕР

```
1 // .....
2     private class ContinuationHandler(
3         private val socket: AsynchronousSocketChannel
4     ): CompletionHandler<Void, Continuation<Boolean>> {
5         override fun completed(
6             result: Void?,
7             attachment: Continuation<Boolean>
8         ) {
9             socket.close()
10            attachment.resume(true)
11        }
12 // .....
```

ПРИМЕР

```
1 // .....
2         override fun failed(
3             exc: Throwable,
4             attachment: Continuation<Boolean>
5         ) {
6             attachment.resume(false)
7         }
8     }
9 }
10
11 suspend fun main() {
12     println(NetUtils.isAlive("www.ya.ru", 443))
13     println(NetUtils.isAlive("www.yaya.ru", 443))
14 }
```

СЕРВЕРНАЯ СТОРОНА

- Тут отдельная проблема: в Java API нет callback-интерфейса на ассерт
- Решается через разные контексты
- Каждый контекст имеет свой пул нитей с подходящей политикой создания
- Есть отдельный диспетчер - Dispatchers.IO
- Подходит для блокирующего ввода-вывода или для интенсивных вычислений

СЕРВЕР НА МИНИМАЛКАХ

```
1 class ContinuationHandler<T>:
2     CompletionHandler<T, Continuation<T>> {
3     override fun completed(
4         result: T, attachment: Continuation<T>
5     ) {
6         attachment.resume(result)
7     }
8     override fun failed(
9         exc: Throwable, attachment: Continuation<T>
10    ) {
11        attachment.resumeWithException(exc)
12    }
13 }
14 // .....
```

СЕРВЕР НА МИНИМАЛКАХ

```
1 // .....
2 class CoroutineSocket(
3     val channel:AsynchronousSocketChannel =
4         AsynchronousSocketChannel.open()
5 ) {
6     var isConnected: Boolean = false
7     private set
8 // .....
```

СЕРВЕР НА МИНИМАЛКАХ

```
1 // .....
2     suspend fun connect(isa: InetAddress) {
3         suspendCoroutine {
4             channel.connect(isa, it, ContinuationHandler())
5             isConnected = true
6         }
7     }
8 // .....
```

СЕРВЕР НА МИНИМАЛКАХ

```
1 // .....
2     suspend fun read(
3         buffer: ByteBuffer
4     ): Int = suspendCoroutine {
5         channel.read(
6             buffer, it, ContinuationHandler<Int>()
7         )
8     }
9 // .....
```

СЕРВЕР НА МИНИМАЛКАХ

```
1 // .....
2     suspend fun write(
3         buffer: ByteBuffer
4     ): Int = suspendCoroutine {
5         channel.write(buffer, it, ContinuationHandler())
6     }
7 }
8 // .....
```

СЕРВЕР НА МИНИМАЛКАХ

```
1 // .....  
2 class CoroutineServerSocket {  
3     private val channel =  
4         AsynchronousServerSocketChannel.open()  
5     // .....
```

СЕРВЕР НА МИНИМАЛКАХ

```
1 // .....
2 suspend fun bindAndAccept(
3     port: Int
4 ): CoroutineSocket {
5     val result = suspendCoroutine {
6         channel.bind(InetSocketAddress(port))
7         val handler =
8             ContinuationHandler<AsynchronousSocketChannel>()
9         channel.accept(it, handler)
10    }
11    return CoroutineSocket(result)
12 }
13 }
```

ИСПОЛЬЗОВАНИЕ

```
1 fun main() {  
2     runBlocking {  
3         val server = CoroutineServerSocket()  
4         launch {  
5             val clientPeer = withContext(  
6                 Dispatchers.IO  
7             ) {  
8                 server.bindAndAccept(1234)  
9             }  
10  
11             val readBuffer = ByteBuffer.allocate(1024)  
12             val nRead = clientPeer.read(readBuffer)  
13 // .....  
}
```


ИСПОЛЬЗОВАНИЕ

```
1 // .....
2         readBuffer
3         .rewind()
4         .limit(nRead)
5
6         println(
7             "GOT: " +
8             Charset.defaultCharset()
9                 .decode(readBuffer)
10        )
11 // .....
```

ИСПОЛЬЗОВАНИЕ

```
1 // .....
2     val lines = listOf(
3         "HTTP/1.1 200 OK",
4         "Content-Type: text/plain",
5         "Content-Length: 5",
6         "",
7         "Hello",
8         ""
9     )
10    val answer = lines.joinToString("\r\n")
11    val writeBuffer =
12        Charset.defaultCharset()
13        .encode(answer)
14 // .....
```

ИСПОЛЬЗОВАНИЕ

```
1 // .....
2         val nWrite = clientPeer.write(writeBuffer)
3         println(nWrite)
4     }
5 }
6 }
```