

АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 11

ПЛАН

- Каналы, акторы
- Введение в Groovy

ПРИМЕР

```
1 val channel = Channel<Int>()
2
3 fun main() = runBlocking(Dispatchers.Default) {
4     launch {
5         for (x in 1..5) {
6             println("before send")
7             channel.send(x * x)
8             println("after send")
9         }
10    }
11    // .....
```

ПРИМЕР

```
1 // .....
2
3 repeat(5) {
4     delay(1000)
5     println(channel.receive())
6 }
7 }
```

ЗАКРЫТИЕ КАНАЛА

- Канал можно закрыть
- На читающей стороне это приведет к завершению итератора
- В блокирующих очередях подобное можно делать
- Через отдельные приседания в пользовательском коде

ПРИМЕР

```
1 fun main() = runBlocking {  
2     val channel = Channel<Int>()  
3     launch {  
4         for (x in 1..5) channel.send(x * x)  
5         channel.close()  
6     }  
7     for (y in channel) println(y)  
8     println("Done!")  
9 }
```

ПРИМЕР

```
1 fun CoroutineScope.produceSquares(): ReceiveChannel<Int> =  
2     produce {  
3         println(coroutineContext.job)  
4         for (x in 1..5) send(x * x)  
5     }  
6  
7 fun main() = runBlocking(Dispatchers.Default) {  
8     println(coroutineContext.job)  
9     val squares = produceSquares()  
10    squares.consumeEach { println(it) }  
11    println("Done!")  
12 }
```

ВАРИАНТЫ

- В нашем случае producer-корутина порождает данные из себя
- И здесь рандеву - идеальная конфигурация канала
- Но может быть так, что данные мы берем откуда-то еще
- Из внешнего мира или от другого producer-а

ВАРИАНТЫ

- Не факт, что всегда можем спокойно ждать, пока consumer прочитает
- Противоположная крайность - бесконечная очередь
- Все сможем буферизовать
- Одна беда - память конечна

ВАРИАНТЫ

- Промежуточный вариант - буфер конечного размера
- В канале можно сохранить данные, порожденные от вспышек активности
- В идеале, размер буфера должен быть рассчитан на них
- Но не должно быть принципиально дисбаланса скоростей записи и чтения

ВАРИАНТЫ

- Можем включить режим перезаписи старого
- Например, это данные каких-то замеров
- Старые не успели обработать - и уже новые подъехали
- А если при заполнении ждем освобождения - это механизм backpressure

LIVELOCK

```
1 val p1: CoroutineScope.() -> ReceiveChannel<Int> =  
2     fun CoroutineScope.(): ReceiveChannel<Int> =  
3         produce {  
4             println(coroutineContext.job)  
5             val c2 = p2()  
6             c2.consumeEach {  
7                 send(it)  
8             }  
9         }  
10 // .....
```

LIVELOCK

```
1 // .....
2 val p2: CoroutineScope.() -> ReceiveChannel<Int> =
3     fun CoroutineScope.(): ReceiveChannel<Int> =
4         produce {
5             println(coroutineContext.job)
6             val c1 = p1()
7             c1.consumeEach {
8                 send(it)
9             }
10 fun main() = runBlocking {
11     p1().consumeEach {
12         println("v: " + it)
13     }
14 }
```

ЛЕНИВЫЙ ДЕДЛОК

```
1 val p1: CoroutineScope.() -> ReceiveChannel<Int> =  
2     fun CoroutineScope.(): ReceiveChannel<Int> =  
3         produce {  
4             println(coroutineContext.job)  
5             c2Ref.get().consumeEach {  
6                 send(it)  
7             }  
8         }  
9 // .....
```

ЛЕНИВЫЙ ДЕДЛОК

```
1 // .....
2
3 val p2: CoroutineScope.() -> ReceiveChannel<Int> =
4     fun CoroutineScope.(): ReceiveChannel<Int> =
5         produce {
6             println(coroutineContext.job)
7             c1Ref.get().consumeEach {
8                 send(it)
9             }
10        }
11 // .....
```

ЛЕНИВЫЙ ДЕДЛОК

```
1 // .....
2 val c1Ref: AtomicReference<ReceiveChannel<Int>> =
3     AtomicReference(null)
4 val c2Ref: AtomicReference<ReceiveChannel<Int>> =
5     AtomicReference(null)
6
7 fun main() = runBlocking {
8     c1Ref.set(p1())
9     c2Ref.set(p2())
10
11     c2Ref.get().consumeEach {
12         println("v: " + it)
13     }
14 }
```


ЕЩЕ ВАРИАНТ

```
1  val p1: CoroutineScope.() -> ReceiveChannel<Int> =
2      fun CoroutineScope(): ReceiveChannel<Int> =
3          produce {
4              println(coroutineContext.job)
5              c1Ref.get().consumeEach {
6                  send(it)
7              }
8          }
9
10 val c1Ref: AtomicReference<ReceiveChannel<Int>> =
11     AtomicReference(null)
12
13 // .....
```

ЕЩЕ ВАРИАНТ

```
1 // .....
2
3 fun main() = runBlocking {
4     c1Ref.set(p1())
5     c1Ref.get().consumeEach {
6         println("v: " + it)
7     }
8 }
```

АКТОРЫ

- Внеязыковая абстракция
- Легкий объект с состоянием и поведением
- Может получать сообщения
- Поведение - только как реакция на сообщения

АКТОРЫ

- Можем изменить состояние
- Можем ответить отправителю
- Можем послать сообщения кому-то другому
- Все асинхронно, даже ответ отправителю

TIMELINE АКТОРА

- У каждого актора своя ось времени
- На оси строго упорядочены отрезки обработки сообщений
- Ненулевые, но и не большие
- Не должны произвольно растягиваться
- Разве что в рамках trade-of

ПАРАЛЛЕЛЬНОЕ ООП

- Актор можно считать объектом
- С полиморфизмом и наследованием - зависит от реализации
- Инкапсуляция - близка к идеалу
- Вместо методов - сообщения
- Сложность - только с гарантией доставки ответа

ПРОБЛЕМА КЛАССИЧЕСКОГО ООП

- Объект - как будто соответствует сущности реального мира
- Но в безнитевой модели исполнения - в любой момент времени активен один объект
- Что не очень соответствует реальности
- А в многонитевой - создается несколько временных осей

ПРОБЛЕМА КЛАССИЧЕСКОГО ООП

- На каждой такой оси воспроизводится схема с одним активный в моменте объектом
- Но на разных временных осях может активизироваться один и тот же объект
- В однопонитевой ситуации модель вынужденно упрощается
- А в многопонитевой - она "копируется" на несколько нитей

ПЛЮС АКТОРНОЙ МОДЕЛИ

- Акторная модель "правильнее" использует многопоточность
- Акторы-объекты реально параллельны
- И у них не двоится сознание
- Можно проектировать систему как набор акторов
- С состоянием и правилами коммуникации

КРАУЛИНГ

- Какие-то акторы непосредственно общаются с http-клиентом
- Какие-то - анализируют содержимое
- Какие-то сохраняют полезную информацию
- Какие-то - отслеживают очередь
- Какие-то - обрабатывают сбои и отказы

ЗАЧИТЫВАНИЕ ОБЪЕКТОВ ИЗ РАЗНЫХ ИСТОЧНИКОВ

- Сервис-агрегатор ежедневно зачитывает данные от партнеров
- От каждого партнера - в своем формате
- В любом случае - виртуальная лента из коротких объектов
- Заведем по актору на каждую ленту

ЗАЧИТЫВАНИЕ ОБЪЕКТОВ ИЗ РАЗНЫХ ИСТОЧНИКОВ

- По актору на разные преобразования -
унификация формата, обогащение, агрегация
- Получаем эффективное распараллеливание
- Интуитивно понятное

ГАРАНТИИ ДОСТАВКИ

- Exactly once - ценой блокировок или усложненных алгоритмов
- At most once - бесплатно
- At least once - нужны усилия
- Но сильно проще, чем exactly once
- В широком классе ситуаций - де-факто exactly once

ПРИМЕР

```
1 fun main() {  
2     runBlocking(Dispatchers.Default) {  
3         val actor = actor<String>() {  
4             for (data in channel) {  
5                 println(data)  
6             }  
7         }  
8  
9         actor.send("1234")  
10        actor.send("2345")  
11        actor.send("3456")  
12        actor.close()  
13    }  
14 }
```

ПРОТОКОЛ ОБЩЕНИЯ

- Тип актора - тип сообщений, которые он принимает
- Строка - это для примера
- В реальность там будет пачка sealed-классов с общим предком
- У каждого актора своя

КАК ОТВЕТИТЬ

- Надо в протоколе предусмотреть возможность передать `SendChannel`
- Отправитель передает себя (`channel`)
- Получатель туда отправляет сообщения
- Можно организовать более сложные схемы взаимодействия

ЧТО МОГЛО БЫТЬ ЛУЧШЕ

- Идентификация актора сводится к Java-объекту
- Актор может упасть
- В духе реактивного подхода он должен рестартовать
- Это можно сделать через супервизор
- Но это будет другой актор с другой идентичностью

ЛЕНИВЫЕ БЛОКИРОВКИ

- Канонический актер не должен делать блокирующих действий
- suspend-методы являются лечением многих проблем блокировок
- Но в акторах им тоже не место
- Или место - если знаем, что делаем
- Можем получить далеко идущие последствия

GROOVY

- Более динамический, чем Kotlin
- Но при этом компилируемый
- Это не интерпретатор, написанный на Java
- Есть свой синтаксис
- Но поддерживает Java как подмножество
- Как минимум в рамках устоявшихся версий

GROOVY

- Но многое заимствовано у Python
- С поправкой на бОльшую статичность
- Похожие понятия глобальных переменных
- Механизм создания динамических свойств и методов
- duck typing
- И даже ключевые слова

В ЧЕМ НЕ ОТЛИЧАЕТСЯ ОТ JAVA/KOTLIN

- `if/else`, `while`, `for`, `break`, `continue`
- Общее понятие классов и объектов
- Основная структура типов
- Общее понятие полей и методов
- Статика - близко к Java

ПРОСТЫЕ ПРОГРАММЫ

- Одноаргументные вызовы можно делать без скобок
- Код может быть вне функций
- Простые программы можно писать в скриптовом стиле
- Через `static main` - тоже можно

ПРИМЕР

```
1 static def f(v) {  
2     return v + 1  
3 }  
4  
5 static def f(int v1, v2) {  
6     return v1 + v2  
7 }  
8  
9 def nArgs() {  
10     return args.length  
11 }  
12 // .....
```

ПРИМЕР

```
1 // .....  
2 println "Hello world!"  
3 println this.args  
4 println(f 10)  
5 println(f(2, 3))  
6 println(nArgs())
```


СТРУКТУРА ИСХОДНИКОВ

- Пакеты и классы
- В конечном итоге все в классах
- Можно несколько классов в исходнике
- Но код глобального уровня попадает в класс, одноименный с файлом
- И тогда явного класса с таким именем быть не должно

СТРУКТУРА ИСХОДНИКОВ

- `def` глобального уровня превращается в методы неявного класса
- А исполняемые куски кода (`println` в примере) - оборачивается в метод `run` без аргументов
- Если попытаться определить метод `run` - будет конфликт с неявным

ДВА ФАЙЛА

- Можно написать два groovy-файла с кодом верхнего уровня
- Каждый из них можно отдельно запускать
- Но импортируются классы
- И неявный класс импортировать нельзя

МЕТОДЫ

- Методы можно определять в Java-стиле или в Groovy-стиле (def)
- Тип можно пропустить
- Groovy позволяет многое доопределять
- Все семантические ошибки будут пойманы во время исполнения

ПРИМЕР

```
1 def f(a, b) {  
2     return a + b  
3 }  
4  
5 println("hello")  
6 println(f("hello", 1).reverse())  
7 println(f(23, 1) + 5)
```

ПРИМЕР

```
1 def f(a, b) {  
2     return a + b  
3 }  
4  
5 println("hello")  
6  
7 //println(f("hello", 1).reverse())  
8 // сломаемся во время исполнения  
9  
10 println(f(23, 1) + 5)
```

ПРИМЕР

```
1 def f(int a, b) {  
2     return a + b  
3 }  
4  
5 def methodMissing(String name, args) {  
6     if (name == "f") {  
7         return f(Integer.parseInt(args[0]),  
8             args[1]).toString()  
9     }  
10 }  
11  
12 println("hello")  
13 println(f("123", 1).reverse())  
14 println(f(23, 1) + 5)
```

РАБОТА С ФАЙЛАМИ

- `java.io` импортируется автоматически
- В `File` добавлены разные полезные функции и свойства
- Лямбда блоки похожи на Kotlin
- И даже есть переменная `it`

ПРИМЕР: CAT

```
1 name = "/home/sreznick/work/db/" +  
2       "postgres/src/postgresql/README"  
3  
4 new File(name).eachLine {  
5     println $it  
6 }
```

ПРИМЕР: САТ ДЛЯ МАЛЕНЬКИХ ФАЙЛОВ

```
1 name = "/home/sreznick/work/db/"  
2       "postgres/src/postgresql/README"  
3 f = File(name)  
4 print(f.text)
```

КОЛЛЕКЦИИ

- Структуры берутся из Java
- Есть синтаксические упрощения
- Расширены методы
- duck typing

ПРИМЕР

```
1 stringData = ["hello", "world", "qwertyuiop"]
2 intData = [1, 5, 123, -10]
3 println(stringData)
4 println(intData)
5
6 println(stringData + intData)
7
8 allData = stringData + intData
9
10 println(allData.getClass())
```

ПРИМЕР

```
1 person = [name:"John", id: 1234567]
2
3 println(person.name)
4 println(person['name'])
5 println(person['id'])
6
7 for (e in person) {
8     println("${e.key}: ${e.value}")
9     print(e.getClass())
10 }
```

ООП

- В целом как Java
- Но можно писать `set/get`-методы
- Тогда обращение к свойству на чтение/запись направляется к методам

ПРИМЕР

```
1 class Person {
2     private final String name
3
4     Person(String name) {
5         this.name = name
6     }
7
8     String getName() {
9         println("getter")
10        return name
11    }
12 // .....
```

ПРИМЕР

```
1 // .....
2     void setName(String value) {
3         println("setter: '" + value)
4     }
5 }
6
7 person = new Person("john")
8 println(person.name)
9 person.name = "12343"
```


