АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 10

ПЛАН

- Каналы, акторы
- Введение в Groovy

FLOW

- Это контекст вычисления
- Его можно создать в обычной функции вне корутинного контекста
- И можно там же вызывать нетерминальные преобразования
- А вот терминальное - это suspend-функция

CHANNEL

- Это структура данных с suspend-методами
- Можно создать вне корутинного контекста
- send/receive - suspend-методы
- По умолчанию capacity - 0, политика при заполнении - SUSPEND

CHANNEL

- По умолчанию - режим рандеву
- Можно поставить capacity RENDEVOUZ, CONFLATED, BUFFERED или UNLIMITED
- Политики при заполнении: SUSPEND, DROP_OLDEST, DROP_LATEST
- Не все комбинации осмысленны

CHANNEL

- RENDEVOUZ + SUSPEND - дождались и передали
- CONFLATED + DROP_OLDEST - как ни странно, сломается
- CONFLATED + DROP_LATEST - тоже
- CONFLATED + SUSPEND - работает

CHANNEL

- CONFLATED + SUSPEND - с буфером размера 1
- По сути - как DROP_OLDEST/DROP_LATEST
- Почему так странно ?
- Костыль для переиспользования параметра по умолчанию

CHANNEL

- UNLIMITED игнорирует второй параметр
- BUFFERED + SUSPEND - буфер стандартного размера
- По умолчанию 64, конфигурируется через проперти
- BUFFERED + DROP_OLDEST/DROP_LATEST - эквивалент CONFLATED (+ SUSPEND)

CHANNEL

- По сути 4 режима - но это не все
- Символические значения отрицательны
- Положительные понимаются как длина буфера
- И с ними сочетается любой второй

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

```
    // ....................

    repeat(5) {
        delay(1000)
        println(channel.receive())
    }
}
```

ЗАКРЫТИЕ КАНАЛА

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

ПОТОЧНЕЕ

- close породит исключение на читающей стороне
- Типовая логика итерирования его обработает и примет как руководство к завершению итерирования
- Есть еще cancel со своим исключением
- Его `for` не обрабатывает

ПРИМЕР

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

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 расчитан на них
- Но не должно быть принципиально дизбаланса 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

ВЫБОР МЕНЬШЕГО ИЗ ЗОЛ

- Зло 1: задержки в обработке (глобальные)
- Зло 2: out of memory
- Зло 3: потери данных
- Зло 3.5: дупликация данных
- Возможны комбинации

ПРОБЛЕМЫ ПАРАЛЛЕЛИЗМА В КОНТЕКСТЕ КОРУТИН

- Livelocks
- Deadlocks
- Starvation

LIVELOCK

```
val p1: CoroutineScope.() -> ReceiveChannel<Int> =
    fun CoroutineScope.(): ReceiveChannel<Int> =
        produce {
            println(coroutineContext.job)
            val c2 = p2()
            c2.consumeEach {
                send(it)
            }
        }
// ....................
```

LIVELOCK

```
// ....................
val p2: CoroutineScope.() -> ReceiveChannel<Int> =
    fun CoroutineScope.(): ReceiveChannel<Int> =
        produce {
            println(coroutineContext.job)
            val c1 = p1()
            c1.consumeEach {
            send(it)
        }
}
fun main() = runBlocking {
   p1().consumeEach {
       println("v: " + it)
   }
}
```

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

```
// ....................

val p2: CoroutineScope.() -> ReceiveChannel<Int> =
    fun CoroutineScope.(): ReceiveChannel<Int> =
        produce {
            println(coroutineContext.job)
            c1Ref.get().consumeEach {
                send(it)
            }
        }
// ....................
```

ЛЕНИВЫЙ ДЕДЛОК

```
// ....................
val c1Ref: AtomicReference<ReceiveChannel<Int>> =
    AtomicReference(null)
val c2Ref: AtomicReference<ReceiveChannel<Int>> =
    AtomicReference(null)

fun main() = runBlocking {
   c1Ref.set(p1())
   c2Ref.set(p2())

   c2Ref.get().consumeEach {
       println("v: " + it)
   }
}
```

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 однонитевой ситуации модель вынужденно упрощается
- А в многонитевой - 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 реальности 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 организавать 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 актор 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
- Поддерживает многое из Java как подмножество

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

```
// ..........................
println "Hello world!"
println this.args
println(f 10)
println(f(2, 3))
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 имортировать 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

```
def f(a, b) {
  return a + b
}

println("hello")
println(f("hello", 1).reverse())
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

```
def f(int a, b) {
  return a + b
}

def methodMissing(String name, args) {
    if (name == "f") {
      return f(Integer.parseInt(args[0]),
               args[1]).toString()
    }
}

println("hello")
println(f("123", 1).reverse())
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: CAT 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

```
person = [name:"John", id: 1234567]

println(person.name)
println(person['name'])
println(person['id'])

for (e in person) {
    println("${e.key}: ${e.value}")
    print(e.getClass())
}
```

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

```
// ..................................
    void setName(String value) {
        println("setter: '" + value)
    }
}

person = new Person("john")
println(person.name)
person.name = "12343"
```