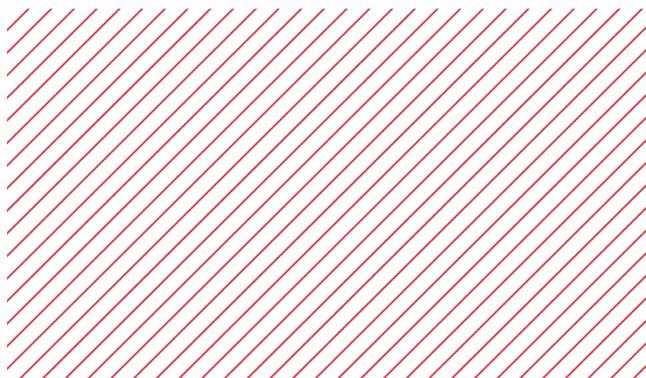


академия
больших
данных



Advanced Scala

Корольков Алексей



Структура курса

1. Введение в Scala. Основные конструкции языка



2. ООП в Scala. Pattern matching. Функциональные конструкции. Adt

3. Библиотека коллекций в Scala



4. Асинхронные операции, обработка исключений, неявные параметры

5. Параметрический полиморфизм. Имплиситы

6. Основы функционального программирования. Часть 1

7. Основы функционального программирования. Часть 2

8. ZIO, TF и основной ФП-стэк в Scala

9. Функциональные стримы на примере fs2, работа с бд

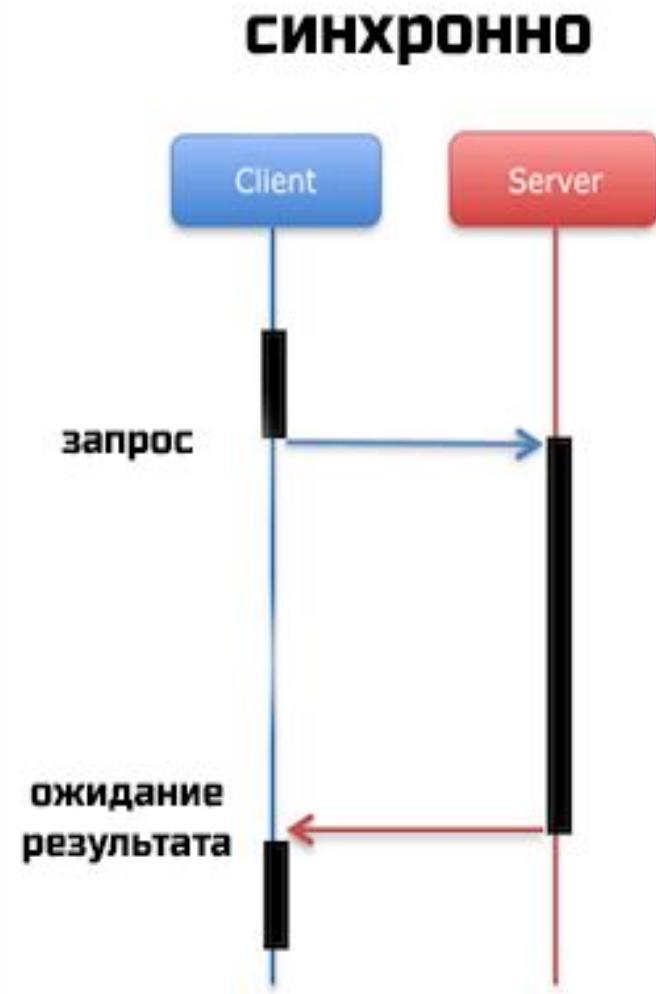
10. Акторы в Scala



Async

Async overview

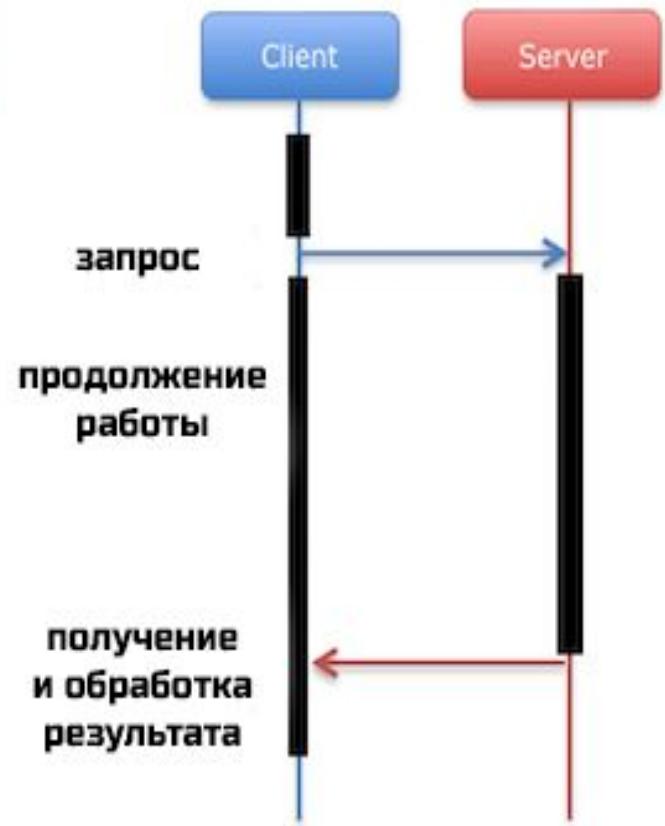
```
trait SocialService {  
    def getUserInfo(userId: Long): String  
}  
  
object Examples extends App {  
    val userId = 1L  
  
    val service: SocialService = ... // http impl  
  
    val userInfo = service.getUserInfo(userId)  
  
    println(userInfo)  
  
    ...  
}
```



Async callback

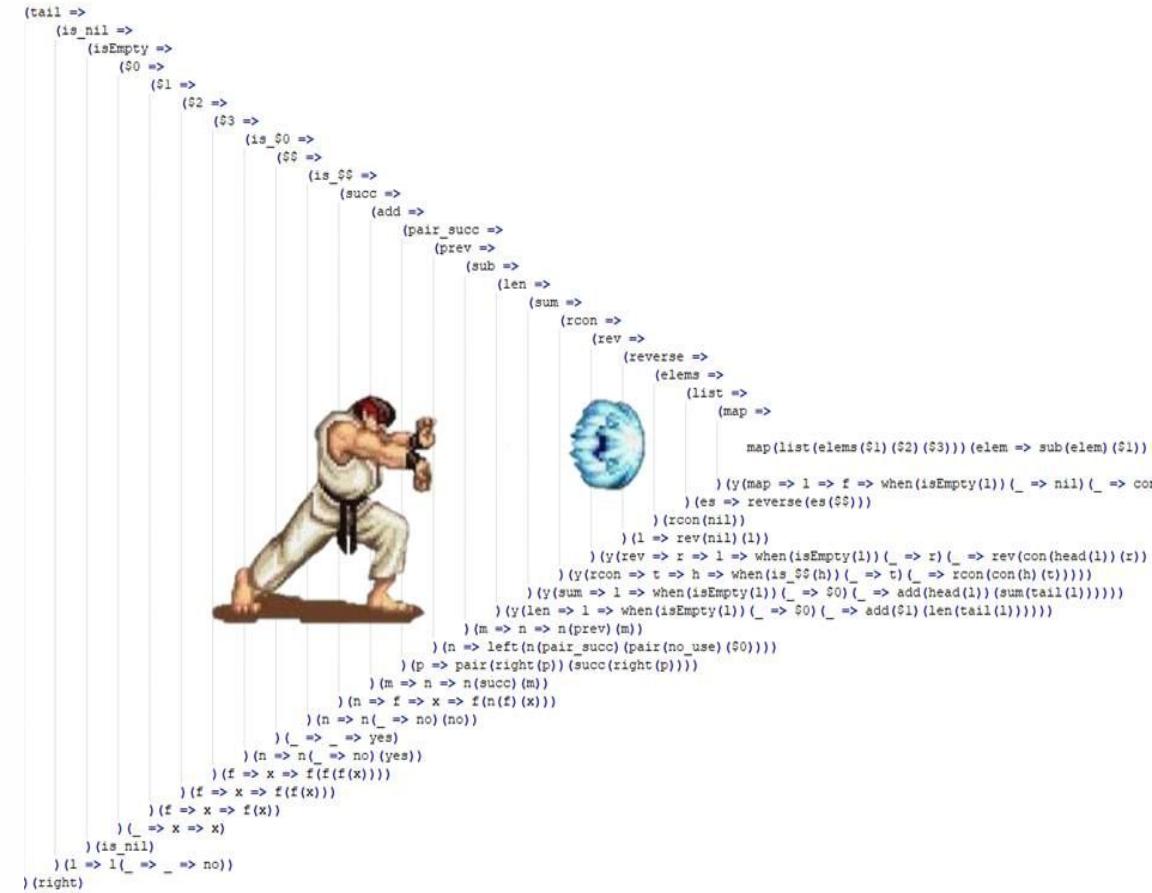
```
trait SocialServiceAsync {  
    def getUserInfoAsync(userId: Long, cb: Callback[String]): Unit  
}  
  
trait Callback[A] {  
    def onSuccess(payload: A): Unit  
    def onError(ex: Throwable): Unit  
}  
  
object ExamplesAsync extends App {  
    val userId = 1L  
    val service: SocialServiceAsync = ...  
  
    service.getUserInfoAsync(userId, new Callback[String] {  
        override def onSuccess(payload: String): Unit =  
            println(payload)  
        override def onError(ex: Throwable): Unit = println(ex)  
    })  
    ...  
}
```

асинхронно



Async callback

```
object ExamplesAsync extends App {  
    val userId = 1L  
    val service: SocialServiceAsync = ...  
    service.getUserInfoAsync(userId, new Callback[String] {  
        override def onSuccess(userInfo: String): Unit = {  
            service.getFriendsCountAsync(userId, new Callback[Long] {  
                override def onSuccess(friendsCount: Long): Unit =  
                    println(s"$userInfo + $friendsCount")  
                override def onError(ex: Throwable): Unit = println(ex)  
            })  
            override def onError(ex: Throwable): Unit = println(ex)  
        }  
    })  
    ...  
}
```



Monad

Монада - это параметрический тип данных, который реализует две операции: создание монады (иногда называют *unit*) — и функцию *flatMap* (иногда называют *bind*) и подчиняется некоторым правилам. Применяются они для реализации стратегии связывания вычислений:

```
object Monad {  
    def unit[T](x: T): Monad[T] = ???  
}  
  
trait Monad[T] {  
    def flatMap[U](f: T => Monad[U]): Monad[U]  
}
```



Monad. laws

Формальные законы, которым должна удовлетворять монада:

- Left unit law:

$$(unit(x) \text{ flatMap } f) == f(x)$$

- Right unit law:

$$(monad \text{ flatMap } unit) == monad$$

- Associativity law:

$$((monad \text{ flatMap } f) \text{ flatMap } g) == (monad \text{ flatMap } (x => f(x) \text{ flatMap } g))$$

Monad. laws

Проверим является ли тип **Option[T]** монадой, в качестве операции **unit(x)** используем конструктор **Some(x)**:

```
def squareFunction(x: Int): Option[Int] = Some(x * x) //f  
def incrementFunction(x: Int): Option[Int] = Some(x + 1) //g
```

Left unit law:

`Some(1).flatMap(squareFunction) == squareFunction(1)`

Right unit law:

`Some(1).flatMap(x => Some(x)) == Some(1)`

Associativity law:

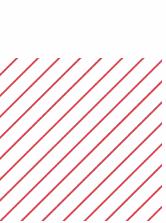
`val left = Some(1).flatMap(squareFunction).flatMap(incrementFunction)`

`val right = Some(1).flatMap(x => squareFunction(x).flatMap(incrementFunction))`

`left == right`



Future



Future

Future[A] - абстракция над произвольным вычислением, возвращающим значение типа А

- эффект асинхронного исполнения
- эффект возможного неудачного выполнения
- предоставляет методы композиции и модификации для работы с полученным значением **map/flatMap/...**



Future

Future[T] можно инициализировать разными способами:

используя метод ***successful***:

```
val f = Future.successful("Hello future")
```

используя метод ***apply***:

```
import scala.concurrent.ExecutionContext.Implicits.global
```

```
val f = Future("Hello future")
```



Implicit parameters

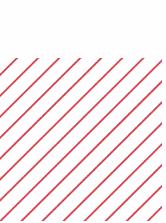
Implicit parameters (неявные параметры) — это параметры, которые могут быть автоматически переданы в функцию из контекста ее вызова. Для этого в нем должны быть однозначно определены и помечены ключевым словом `implicit` переменные соответствующих типов:

```
case class Context(message: String)
```

```
def printContext(implicit ctx: Context): Unit =  
  println(ctx.message)
```

```
implicit val ctx = Context("Hello implicit")
```

```
printContext //Hello implicit
```



Future. map

Как и в случае коллекций, метод **map** используется для изменения успешного результата **Future**, переданная функция применяется также асинхронно

```
def map[S](f: T => S)(implicit executor: ExecutionContext): Future[S]
```

```
import scala.concurrent.ExecutionContext.Implicits.global
```

```
Future.successful(1).map(_ + 1) //2
```

Future. flatMap

Метод **flatMap** используется для изменения успешного результата **Future**, когда функция выполняет новое асинхронное действие

```
def flatMap[S](f: T => Future[S])(implicit executor: ExecutionContext): Future[S]
```

```
import scala.concurrent.ExecutionContext.Implicits.global  
Future.successful(1).flatMap(x => Future.successful(x + 1))
```



Future. for comprehension

Для **Future** можно использовать конструкцию **for comprehension** также как и для коллекций, однако условия в виде *if* лучше не использовать, так как они будут выполняться в другом потоке и влекут ненужную нагрузку на сри

```
import scala.concurrent.ExecutionContext.Implicits.global  
for {  
    i <- Future.successful(1)  
    j <- Future.successful(2)  
} yield i + j
```

Future. for comprehension

Иногда из некоторого списка асинхронных задач с **Future** нужно сделать одну общую **Future** со списком результатов, пример:

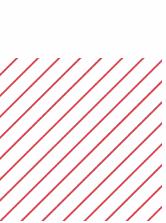
```
val seqF: Future[List[String]] = Future  
.sequence(List.fill(5)(Future("run")))
```

Если имеется не само вычисление в виде **Future**, а функция возвращающая **Future** и список значений из которых нужно произвести вычисление, то применяется метод **traverse**

```
val traverseF: Future[List[Int]] = Future  
.traverse(List.range(1, 10))(i => Future(i))
```



Future error handling



Future. recover

recover - метод у **future** для обработки исключительных ситуаций

```
def recover[U >: T](pf: PartialFunction[Throwable, U])  
  (implicit executor: ExecutionContext): Future[U]
```

```
Future(6 / 2).recover { case e: ArithmeticException => 0 } // result: 3
```

```
Future(6 / 0).recover { case e: ArithmeticException => 0 } // result: 0
```

```
Future(6 / 0).recover { case e: IllegalArgumentException => 0 }  
// result: throw exception
```

Try[T]

Try - это тип, который отражает возможность выполнения вычисления с исключением, является эффективным способом работы с исключениями в функциональном стиле с использованием комбинаторов **map/flatMap**

sealed abstract class Try[+T]

final case class Success[+T](value: T) extends Try[T]

final case class Failure[+T](exception: Throwable) extends Try[T]

- Success [T] – содержит результат успешного выполнения
- Failure [T] – содержит результат выполнения с ошибкой



Try[T]

Try("Hello try") //Success>Hello try

Try(new Exception("Boom")) //Success>java.lang.Exception: Boom

Try(throw new Exception("Boom")) //Failure>java.lang.Exception: Boom

Try(1).map(_ + 1) //Success>2

Try(1).flatMap(x => Try(x + 1)) //Success>2

*Try[Int](throw new Exception("Boom")).map(_ + 1)
//Failure>java.lang.Exception: Boom*

*Try(1).filter(_ % 2 == 0)
//Failure>java.util.NoSuchElementException: Predicate does not hold //for 1)*



Execution context

Execution context

Execution context – интерфейс для выполнения асинхронных задач, как правильно на пуле потоков. Обычно инициализируется из *java Executor*:

`ExecutionContext.fromExecutor(`

`new java.util.concurrent.ForkJoinPool(initialParallelism: Int)`

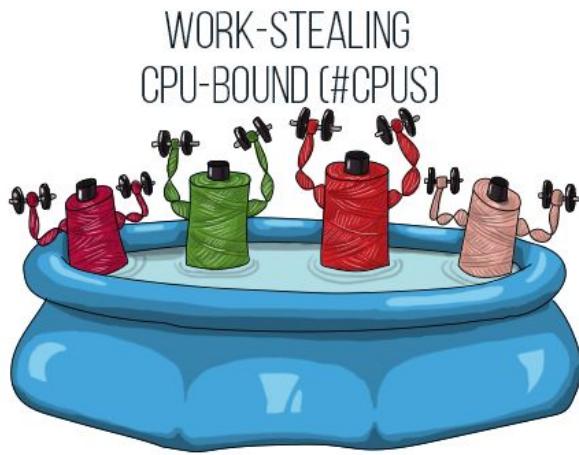
`)`

`ExecutionContext.fromExecutor(Executors.newFixedThreadPool(limit: Int)`

`ExecutionContext.fromExecutor(Executors.newSingleThreadExecutor())`

Выбор тред пулов

THREAD POOL BEST PRACTICES



COMPUTATION

FINITE RESOURCES
AVOID BLOCKING AT ALL COSTS



AVOID WORK AT ALL COSTS

EVENT DISPATCHER

CACHING
UNBOUNDED SIZE



BLOCKING IO

DISCLAIMER:
TEST AND MEASURE!
WHEN IT COMES TO CONCURRENCY,
NOBODY HAS IDEA WHAT THEY'RE DOING.

Спасибо за внимание