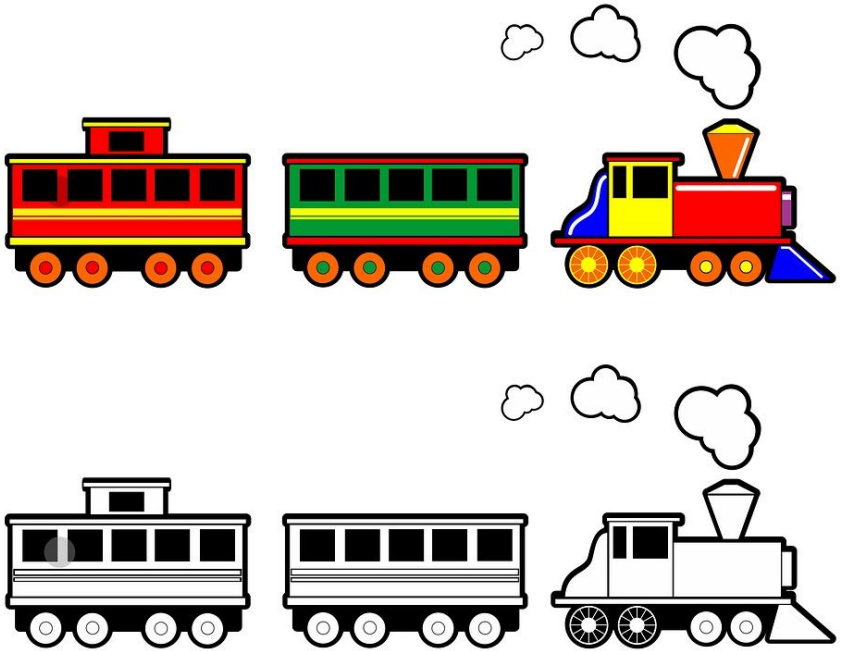
Scala School

Лекция 8: Асинхронные вычисления (продолжение)

План лекции

- “Традиционная” асинхронность в Scala
- Параллельные коллекции
- Java - коллекции в Scala
- Примеры Scala-библиотек



“Традиционная” асинхронность в Scala

Java-инструменты для многопоточности

Scala поддерживает весь традиционный для Java набор инструментов для работы с многопоточностью с небольшими отличиями

- Thread
- Volatile
- Synchronized
- Atomic
- Executor
- Lock
- ...

Thread

```
object ThreadExamples extends App {
  println(s"Hi! I am ${Thread.currentThread().getName}" )
  val t = new Thread() {
    override def run(): Unit = {
      println(s"Hello! My name is ${Thread.currentThread().getName}" )
      Thread.sleep(1000)
    }
  }
  t.start()
  t.join()
  println("New thread joined.")
}
```

Thread

Output:

```
Hi! I am main

Hello! My name is Thread-0
```

Thread

Output:

```
Hi! I am main

Hello! My name is Thread-0

<Waiting 1 sec>

New thread joined.


Process finished with exit code 0
```

Synchronized

В отличие от Java, в Scala synchonized - это метод класса AnyRef

```
class Person(var name: String) {
  def set(changedName: String) {
    this.synchronized {
      name = changedName
    }
  }
}
```

http://stackoverflow.com/questions/17627006/how-is-the-synchronized-method-on-anyref-implemented

Synchronized

```
class X {
    private val lock = new Object()

    lock.synchronized {
      println("wow")
    }
  }
```

Volatile

В отличие от Java, в Scala volatile - поле объявляется не с помощью ключевого слова, а с помощью аннотации @volatile

```
@volatile var running = true
```

Volatile

```
@volatile var running = true
for (i <- 1 to 5) {
  val t = new Thread {
    override def run(): Unit = while (running) {
        println(s"Still running ${Thread.currentThread().getName}" )
        Thread.sleep(1000)
        if (Random.nextDouble() < 0.05) running = false
    }
  }
  t.start() }
while (running) {}
```

Atomic, Executor, ...

Используются в Scala без всяких отличий от Java

```
val a = new AtomicInteger(10)
a.compareAndSet(10, 11)

val ex = Executors.newFixedThreadPool(10)
implicit val ec = ExecutionContext.fromExecutor(ex)
```

scala.collection.concurrent

Пакет `scala.collection.concurrent` содержит thread-safe коллекции

- `scala.collection.concurrent.Map` - базовый трейт для thread-safe Map
- `scala.collection.concurrent.TrieMap`

concurrent.TrieMap

`TrieMap` - thread-safe Map, построенная на основе Ctries.

Метод `.iterator` атомарно делает lazy snapshot всей коллекции и использует его для обхода элементов (consistent iterator).

```
scala> val m = new scala.collection.concurrent.TrieMap [String, Int]
m: scala.collection.concurrent.TrieMap [String,Int] = TrieMap ()
scala> m("a") = 2
scala> m("b") = 3
```

concurrent.TrieMap

```
scala> m.readOnlySnapshot()
res5: scala.collection.Map[String,Int] = TrieMap(a -> 2, b -> 3)

scala> m.snapshot()
res6: scala.collection.concurrent.TrieMap [String,Int] = TrieMap(a -> 2, b ->
3)

scala> m.iterator
res4: Iterator[(String, Int)] = non-empty iterator
```

TrieMap vs ConcurrentHashMap

TrieMap

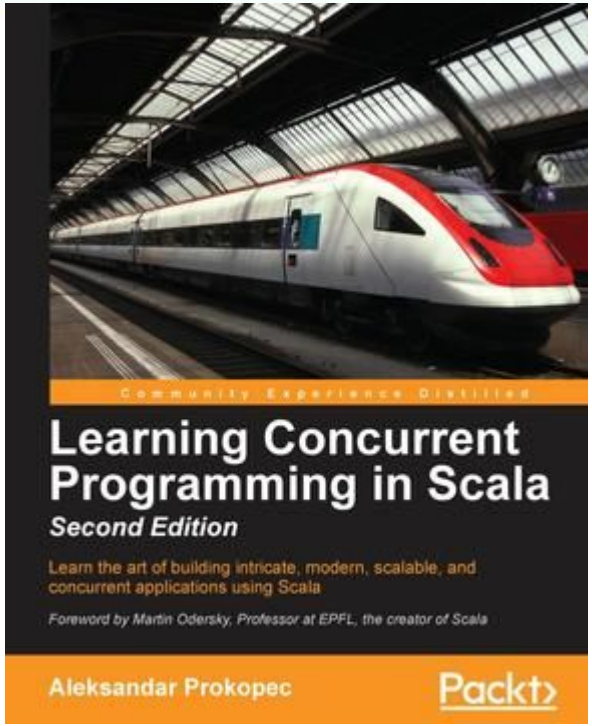
- Consistent Iterator

ConcurrentHashMap

- Быстрый lookup

Books

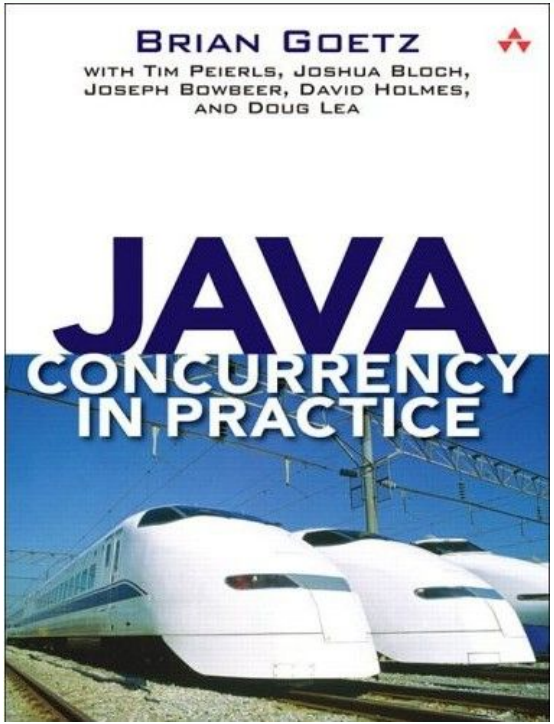
“Learning Concurrent Programming in Scala” - Aleksandar Prokopec



https://github.com/concurrent-programming-in-scala

Books

“Java Concurrency in Practice” - Brian Goetz



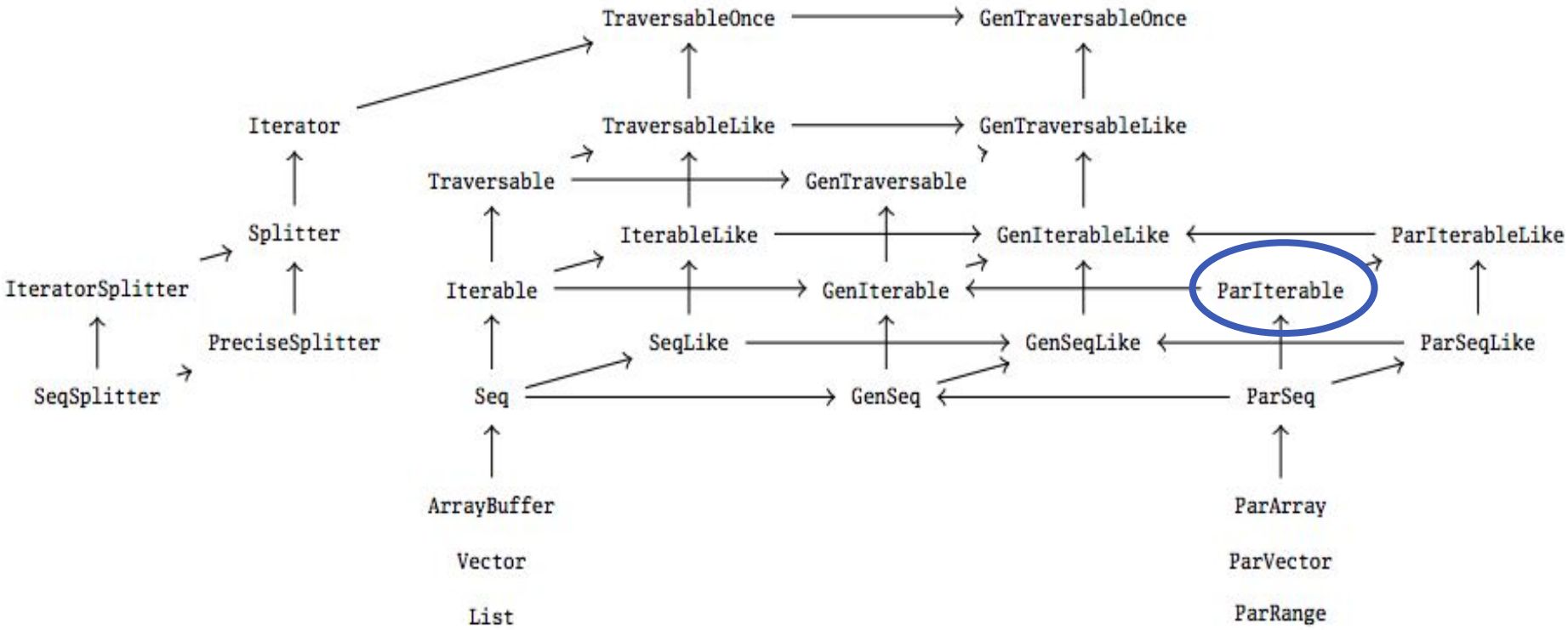
Параллельные коллекции

Parallel Collections

В Scala существует отдельный разряд коллекций, предназначенных для параллельного доступа к данным (Data Parallelism)
Они находятся пакете `scala.collection.parallel`

- `scala.collection.parallel.immutable`
- `scala.collection.parallel.mutable`

Иерархия параллельных коллекций (Seq)



GenIterable, GenSeq, GenSet, GenMap

Базовые трейты `GenIterable, GenSeq, GenSet, GenMap` наследуются как обычными (последовательными) коллекциями, так и параллельными

```
def doSmth(s: GenSeq[Int]) = s.map(_ * 2).sum


scala> doSmth(Vector(1,2,3))
res0: Int = 12


scala> doSmth(ParVector(1,2,3))
res3: Int = 12
```

ParIterable

`ParIterable[+T]` - базовый трейт для параллельных коллекций.
Параллельные операции осуществляются методом divide and conquer.

```
trait ParIterable[+T] extends GenIterable[+T] with ... {

  def splitter: IterableSplitter[T]
  def newCombiner: Combiner[T, ParIterable[T]]
  def taskSupport: TaskSupport
}
```

Parallel Collections

Чтобы получить параллельную коллекцию из непараллельной, нужно вызвать метод `.par (trait Parallelizable)`

```
scala> val imList = List(1, 2, 3, 4).par
imList: scala.collection.parallel.immutable.ParSeq [Int] = ParVector(1,2,3,4)

scala> val mutList = scala.collection.mutable.MutableList(1, 2, 3, 4).par
mutList: scala.collection.parallel.mutable.ParSeq [Int] = ParArray(1,2,3,4)
```

Parallel Collections

API для параллельных коллекций такое же, как и для непараллельных

Результат - параллельная коллекция

```
scala> imList.filter(_ > 2)
res17: scala.collection.parallel.immutable.ParSeq [Int] = ParVector(3, 4)

scala> mutList.filter(_ > 2)
res19: scala.collection.parallel.mutable.ParSeq [Int] = ParArray(3, 4)
```

scala.collection.mutable.Builder

`Builder` - объект, позволяющий инкрементально (с помощью метода +=) создавать новую коллекцию - используется в методах-трансформерах `(map, filter, groupBy, ...)`

```
trait Builder[-Elem, +To] {
  def +=(elem: Elem): this.type
  def result(): To
  def clear(): Unit
  def mapResult[NewTo](f: To => NewTo): Builder[Elem, NewTo] = ...
}
```

scala.collection.mutable.Builder

```
scala> val buf = new ArrayBuffer[Int]
buf: scala.collection.mutable.ArrayBuffer [Int] = ArrayBuffer()

scala> val bldr = buf mapResult (_.toArray)
bldr: scala.collection.mutable.Builder [Int,Array[Int]]
  = ArrayBuffer()
```

Combiner

`Combiner` - параллельная вариация `Builder`.

```
trait Combiner[-T, Repr+] extends Builder[T, Repr] {


  def combine[N <: T, NewRepr >: Repr](that: Combiner[N, NewRepr]):
Combiner[N, NewRepr]


}
```

Combiner

После выполнения метода `.combine` состояние исходных `Combiner`'ов неизвестно и они становятся невалидными и не должны больше использоваться.

НО разрешается в качестве результата возвращать один из исходных объектов

Настройка пула потоков

Для параллельный коллекций по умолчанию используется **ForkJoinPool** с количеством потоков равным **числу ядер процессора**. Его можно изменить с помощью объекта `TaskSupport`

```
val pool = new ForkJoinPool(4)
val myTaskSupport = new ForkJoinTaskSupport(pool)
val numbers = Vector.tabulate(5000)(i => i)
val parNumbers = numbers.par
parNumbers.tasksupport = myTaskSupport
```

Splitter

`Splitter[+T]` - “Усложненный” `Iterator`, обладающий методом `.split`, который разбивает исходный Splitter на несколько дочерних, каждый из которых обходит свою часть исходного. Позволяет нескольким процессорам обрабатывать коллекцию.

```
trait Splitter[+T] extends Iterator[T] {
  def split: Seq[Splitter[T]]
}
```

Splitter

Метод `.split` должен иметь низкую сложность (не более O(log(N)), т.к. в процессе выполнения он вызывается много раз.

Splitter

Метод `.split` должен иметь низкую сложность (не более O(log(N)), т.к. в процессе выполнения он вызывается много раз.

Параллелизуемые коллекции (существует эффективный `.split`):

- **Плоские:** `Array, ArrayBuffer, HashTable, ..`
- **Древовидные:** `Vector, immutable.Map, TrieMap`

Непараллелизуемые (нет эффективного `.split`)

- **Связные:** `List, Stream`

Параллелизуемые коллекции

immutable

```
Vector                -> ParVector
Range                 -> ParRange
HashMap / HashSet     -> ParHashMap / ParHashSet
```

mutable

```
Array                 -> ParArray
HashMap / HashSet     -> ParHashMap / ParHashSet
TrieMap               -> ParTrieMap
```

Параллелизуемые коллекции

Вызов метода `.par` создает параллельную коллекцию, ссылающуюся на элементы исходной, копирования **не происходит**, выполняется **быстро**

```
scala> Vector(1,2,3).par
res0: scala.collection.parallel.immutable.ParVector [Int] = ParVector(1, 2,
3)
```

Непараллелизуемые коллекции

При вызове `.par` элементы исходной коллекции копируются в новую.

Конвертация непараллелизируемой коллекции в параллелизируемую выполняется **не параллельно**

```
scala> Stream(1,2,3,4).par
res2: scala.collection.parallel.immutable.ParSeq [Int] = ParVector(1, 2, 3,
4)
```

Непараллелизуемые операции

- `foldLeft / foldRight`
- `reduceLeft / reduceRight`
- `reduceLeftOption / reduceRightOption`
- `scanLeft / scanRight`
- Методы, создающие непараллелизуемые коллекции: `toList, ...`

foldLeft

```
def foldLeft[B](z: B)(op: (B, A) => B): B
```

Элементы обходятся слева направо

```
scala> Vector(1, 2, 3).foldLeft("")((n, s) => n + s.toString)
res5: String = 123
```

aggregate

Вместо `.foldLeft` можно воспользоваться методом `.aggregate`, он принимает дополнительный параметр `combop` - способ объединения нескольких “агрегатов”

```
def aggregate[B](z: =>B)(seqop: (B, A) => B, combop: (B, B) => B): B
```

```
scala> Vector(1, 2, 3).aggregate("")((n, s) => n + s.toString, (s1, s2) =>
s1 + s2)
res6: String = 123
```

reduce, fold, scan

Для параллельных коллекций также можно использовать методы `.reduce`, `.fold`, `.scan` - они будут выполняться параллельно, в отличие от `Left` / `Right` версий

Для обычных коллекций `reduce == reduceLeft`, `fold == foldLeft`, `scan == scanLeft`
для параллельных `reduce`, `fold`, `scan` переопределены

reduce, fold

```
def foldLeft[B](z: B)(op: (B, A) => B): B
def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1

def reduceLeft[B >: A](op: (B, A) => B): B
def reduce[A1 >: A](op: (A1, A1) => A1): A1
```

scan

```
def scan[B >: A, That](z: B)(op: (B, B) => B)(implicit cbf:
CanBuildFrom[Repr, B, That]): That
def scanLeft[B, That](z: B)(op: (B, A) => B)(implicit bf: CanBuildFrom[Repr,
B, That]): That

scala> Vector(1, 2, 3, 4).scan(0)(_ + _)
res0: scala.collection.immutable.Vector [Int] = Vector(0, 1, 3, 6, 10)
```

Недетерминированный метод find

Параллельный метод `.find` возвращает первый подходящий объект, найденный каким-либо потоком.

Если в коллекции есть несколько элементов, удовлетворяющих предикату, результат может быть недетерминирован.

```
scala> Vector.tabulate(5000)(i => i).par.find(_ > 10)
res10: Option[Int] = Some(11)

scala> Vector.tabulate(5000)(i => i).par.find(_ > 10)
res11: Option[Int] = Some(2500)
```

Недетерминированный метод find

Если необходимо найти первый по порядку подходящий элемент, используется метод `.indexWhere`

```
scala> Vector.tabulate(5000)(i => i).par.indexWhere(_ > 10)
res15: Int = 11
```

Чистые функции и параллельные коллекции

Остальные параллельные методы **детерминированы**, если их операторы - **чистые функции**.

Если в метод передана **нечистая функция**, любой параллельный метод может стать **недетерминированным**.

Чистые функции и параллельные коллекции

Остальные параллельные методы **детерминированы**, если их операторы - **чистые функции**.

Если в метод передана **нечистая функция**, любой параллельный метод может стать **недетерминированным**.

```
scala> val uid = new AtomicInteger(0)
uid: java.util.concurrent.atomic.AtomicInteger  = 0

scala> (0 until 10).par.map(x => uid.incrementAndGet())
res18: scala.collection.parallel.immutable.ParSeq [Int] = ParVector(3, 6, 2,
5, 7, 1, 4, 8, 9, 10)
```

Коммутативность и ассоциативность

Методы `.aggregate, .reduce, .fold, .scan` принимают параметром бинарную функцию `op(a, b)`

- `op` не обязана быть **коммутативной**
- `op` обязана быть **ассоциативной**

Ассоциативность

Вычитание - не ассоциативная операция, при использовании параллельной коллекции получается неправильный результат

```
scala> Vector(1, 2, 3, 4).reduce(_ - _)
res19: Int = -8

scala> Vector(1, 2, 3, 4).par.reduce(_ - _) // Неправильный результат
res20: Int = 0
```

aggregate

```
def aggregate[B](z: =>B)(sop: (B, A) => B, cop: (B, B) => B): B
```

- cop - ассоциативный
- z - нейтральный элемент cop (cop(z, a) == a)
- cop(sop(z, a), sop(z, b)) == cop(z, sop((sop(z, a), b)

Side effects

При использовании с параллельными коллекциями нужно быть аккуратным с side effect - иначе могут появиться ошибки из-за многопоточного доступа

```
scala> val buf = new mutable.MutableList[Int]()
scala> for (i <- (0 until 50).par) buf += i

scala> buf
res34: scala.collection.mutable.MutableList[Int] = MutableList(0, 1)
```

Side effects

Если во время обхода параллельной коллекции необходимо модифицировать объект, нужно позаботиться о потокобезопасности

```
scala>  val safeBuf = new CopyOnWriteArrayList[Int]()
safeBuf: java.util.concurrent.CopyOnWriteArrayList[Int] = []

scala>   for (i <- (0 until 10).par) safeBuf.add(i)

scala> safeBuf
res44: java.util.concurrent.CopyOnWriteArrayList[Int] = [0, 1, 2, 3, 4, 7,
6, 8, 9, 5]
```

ParTrieMap

У `ParTrieMap` метод `.foreach` не обходит элементы, добавленные после начала параллельной операции (при начале параллельной операции атомарно создается snapshot)

```
val cache = new concurrent.TrieMap[Int, String]()
for (i <- 0 until 100) cache(i) = i.toString
for ((num, str) <- cache.par) cache(-num) = s "-$str"
```

Java-коллекции в Scala

JavaConverters

Позволяет удобно конвертировать Scala и Java коллекции друг в друга с помощью implicit

```
import scala.collection.JavaConverters._

scala> val jList = new util.ArrayList[String](util.Arrays.asList("scala",
"java"))
jList: java.util.ArrayList[String] = [scala, java]

scala> jList.asScala
res4: scala.collection.mutable.Buffer[String] = Buffer(scala, java)
```

JavaConverters

Позволяет удобно конвертировать Scala и Java коллекции друг в друга с помощью implicit

```
import scala.collection.JavaConverters._

scala> val list = List("scala", "java")
list: List[String] = List(scala, java)

scala> list.asJava
res6: java.util.List[String] = [scala, java]
```

collection.immutable asJava

При оборачивании immutable-коллекций в Java, при выполнении update-методов выбрасывается исключение

```
scala> val jList = List(1,2,3,4).asJava
jList: java.util.List[Int] = [1, 2, 3, 4]

scala> jList.add(5)
java.lang.UnsupportedOperationException
  at java.util.AbstractList.add(AbstractList.java:148)
```

JavaConverters

```
scala.collection.Iterable              <=> java.lang.Iterable
scala.collection.Iterator              <=> java.util.Iterator
scala.collection.mutable.Buffer        <=> java.util.List
scala.collection.mutable.Set           <=> java.util.Set
scala.collection.mutable.Map           <=> java.util.Map
scala.collection.mutable.concurrent.Map <=>
java.util.concurrent.ConcurrentMap
```

JavaConverters - что внутри?

`scala.collection.convert.Wrappers` содержит классы-обертки для Scala и Java коллекций

```
case class JIteratorWrapper[A](underlying: ju.Iterator[A]) extends
AbstractIterator[A] with Iterator[A] {

    def hasNext = underlying.hasNext
    def next() = underlying.next
  }
```

JavaConverters - что внутри?

```
case class IteratorWrapper[A](underlying: Iterator[A]) extends ju.Iterator[A]
with ju.Enumeration[A] {

    def hasNext = underlying.hasNext
    def next() = underlying.next()
    def hasMoreElements = underlying.hasNext
    def nextElement() = underlying.next()
    override def remove() = throw new UnsupportedOperationException
  }
```

ConcurrentHashMap

```
import scala.collection.JavaConverters._
import java.util.concurrent.ConcurrentHashMap


scala> val map = new ConcurrentHashMap[String, String] asScala
map: scala.collection.concurrent.Map [String,String] = Map()
```

JavaConversions

Как JavaConverters, но не надо вызывать `.asScala`, `.asJava`

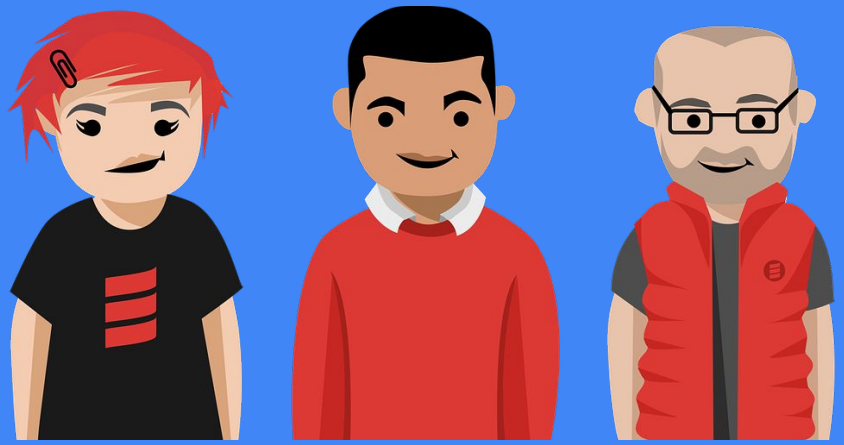
```
import scala.collection.JavaConversions._

scala> val jList = new util.ArrayList[String](util.Arrays.asList("scala",
"java"))
jList: java.util.ArrayList[String] = [scala, java]

scala> jList.map(_.toUpperCase)
res3: scala.collection.mutable.Buffer[String] = ArrayBuffer(SCALA, JAVA)
```

Deprecated since 2.12

Примеры Scala-библиотек



Awesome Scala

https://github.com/lauris/awesome-scala

Репозиторий со списком популярных библиотек на Scala

Spray Json

Работа с Json через case - классы

https://github.com/spray/spray-json

Spray Json

Создаем case class для Json структуры

```
case class ServerMessage(stack: Option[List[Int]] = None,
                         currentOffer: Option[Offer] = None,
                         moneyWon: Option[Int] = None,
                         isError: Boolean = false)

case class Offer(banknotes: List[Int],
                 offerNumber: Int)
```

Spray Json

Создаем object с implicit форматтером

```
object ServerMessageJsonProtocol extends DefaultJsonProtocol {
  implicit val offerJsonProtocol = jsonFormat2(Offer.apply)
  implicit val serverMessageJsonProtocol =
jsonFormat4(ServerMessage.apply)
}
```

Spray Json - сериализация

```
import spray.json._
import ServerMessageJsonProtocol._

val offer = Offer(List(1, 2), 4)
val msg = ServerMessage(Some(List(4, 5)), Some(offer), None, false)

val json = msg.toJson.toString
println(json)
```

Spray Json - сериализация

```
import spray.json._
import ServerMessageJsonProtocol._


val offer = Offer(List(1, 2), 4)
val msg = ServerMessage(Some(List(4, 5)), Some(offer), None, false)


val json = msg.toJson.toString
println(json)
```

```
{
        "stack": [4, 5],
        "currentOffer": {
                "banknotes": [1, 2],
                "offerNumber": 4
        },
        "isError": false
}
```

Spray Json - десериализация

```
val jsonStr =
  """{
    |"stack":[4,5],
    |"currentOffer":{"banknotes":[1,2],"offerNumber":4},
    |"isError":false
    |}""".stripMargin
val deserialized = jsonStr.parseJson.convertTo[ ServerMessage ]
println(deserialized)
```

Spray Json - десериализация

```
val jsonStr =
   """{
     |"stack":[4,5],
     |"currentOffer":{"banknotes":[1,2],"offerNumber":4},
     |"isError":false
     |}""".stripMargin
 val deserialized = jsonStr.parseJson.convertTo[ ServerMessage]
 println(deserialized)
```

```
ServerMessage(Some(List(4, 5)),Some(Offer(List(1, 2),4)),None,false)
```

Spray Json - что еще?

- Поддержка типизированных классов
- Кастомные сериализаторы / десериализаторы
- Форматтеры для рекурсивных типов
- ...

Спасибо