Scala

Василий Купоросов

4 ноября 2018 г.

Содержание

1. Основы

1.1. Немного истории

В Java до версии 5 не было `generic`'ов. Затем с участием Мартина Одерски, исследователя в области языков программирования, дженерики добавили. Тем не менее, стало понятно, что в рамках обратной совместимости нормально это сделать не получится. Тогда Одерски решил создать свой язык, с нормальной системой типов. Изначально Scala была академическим языком (для исследований в области ЯП), но постепенно она переросла в промышленный. Как академический язык, Scala развивалась силами студентов, которые пилили и добавляли свои фичи. Это привело к тому, что фич стало очень много, некоторые из них позволяют делать одно и то же действие разными способами. В связи с этим существуют разные мнения, как надо писать на этом языке. Однако есть официальный стайлгайд, которого желательно придерживаться.

Scala совмещает в себе функциональную и объектно-ориентированную парадигмы. Начнем мы с ООП, так как это более привычно читателям. Многие вещи уже знакомы нам из Котлина и Java, на них подробно останавливаться не будем.

1.2. Классы

1.2.1. `class`

Обычный класс, как в Java. Внутри можно заводить поля и методы.

```
class Foo {
  val bar: Int = 42
  def foo(): Int = bar + 1
}
```

У класса бывают примарный и побочный конструкторы, побочными пользуются редко. Побочный конструктор объявляется ключевым словом `this`.

```
class Bar(innerFoo: Int) {
  this(doubleFoo: Double) = // побочный конструктор
    this(doubleFoo.toInt)   // вызывает примарный

  def foo(): Int = innerFoo // используем значение из примерного конструктора
}
```

1.2.2. `trait`

`trait` – это интерфейс. Интерфейсы в скале поддерживают дефолтную реализацию методов. Также в интерфейсе можно завести немутабельное поле без значения.

```
trait Bar {
  val bar: Double
  def foo(): Int = 42
}
```

1.2.3. `object`

Объекты и `companion`-объекты в Scala есть. Они мощнее чем в Котлине, но об этом позже. Также надо заметить, то статических полей и методов в классах и интерфейсах нет, статичность достигается как раз за счет `object`'ов. Чтобы объявить `companion object`, надо написать объект с именем нужного класса.

```
object Foo // синглтон Foo
class Bar // класс Bar
object Bar // companion object для класса Bar
```

1.3. Наследование

Наследоваться, как и в Java, можно от одного класса и любого количества интерфейсов, но ключевое слово для них одно – `extends`. для множественного наследования используется ключевое слово `with`. Если в множественном наследовании участвует класс, он должен стоять на первом месте.

```
class Foo
trait Bar
trait Baz

class MyClass extends Foo with Bar with Baz
```

При наследовании как у полей, так и у методов надо писать модификатор `override`.

```
class Foo extends Bar {
  override val bar: Double = 42
  override def foo(): Int = bar.toInt
}
```

1.3.1. Наследование в `generic`'ах

Вместо `extends` и `super` в Scala используются специальные значки. Это пришло из теории типов (как мы помним, Scala изначально академический язык).

```
class GenericFoo[F <: Foo] // <F extends Foo>
class GenericFoo[F >: Foo] // <F super Foo>
```

Символ `<:` также можно использовать для наследования вместо `extends`.

```
trait Foo <: Serializable // trait Foo extends Serializable
```

1.4. Оператор `new` и функциональный метод `apply()`

В отличие от Котлина, в Scala надо писать `new` при инстанцировании объекта. В Scala это пришло из Java, а туда – из C++, в котором это важно – создется ли объект в куче или на стеке.

Есть способ, как не писать `new`. Для этого заведем `companion object` для нашего класса и реализуем в нем метод `apply()`.

```
class Bar
object Bar {
  def apply(): Bar = new Bar
}

val bar = Bar() // теперь можем не писать new
val bar = Bar.apply() // аналогичный вызов
```

Метод `apply()` можно определять с разными аргументами, то есть это что-то вроде перегрузки круглых скобок (`operator ()` в C++).

Также `apply()` может заменить вторичный конструктор. Недостаток только в том, что при объявлении класса-наследника можно использовать только вызовы конструкторов.

```
class Bar(foo: Int) { ... }
object Bar {
  def apply(doubleFoo: Double) = new Bar(doubleFoo.toInt)
}

class Baz extends Bar(42.0) // ошибка
```

1.5. Скобки

1.5.1. Писать – не писать

В Scala есть конвенция, что если какой-то код можно не писать, его писать не надо. Если у класса нет тела, то фигурные скобки не пишутся, то же и с вызовами функций. Если в функцию не передаются аргументы, и она не имеет побочных эффектов, скобки принято опускать. Если же побочные эффекты есть, скобки все же пишут, чтобы подчеркнуть, что мы вызываем функцию. При объявлении метода без побочных эффектов скобки тоже опускаются.

1.5.2. Скобки для `generic`'ов

В Scala для `generic` параметров пишутся квадратные скобки, а не угловые. Так сделано, чтобы избежать неоднозначности при парсинге. Пример, когда угловые скобки распарсить не получается:

```
f1(f2<A, B>(x + 1))
```

1.6. Массивы и доступ к элементам

В качестве массивов используется класс `Array`. Поскольку квадратные скобки заняты под `generic`'и, для доступа к элементам они не используются. Зато для `Array` и для некоторых других коллекций определены волшебные методы – уже знакомый нам `apply()` и `update()`. Они позволяют обращаться к элементам через круглые скобки.

```
val array: Array[Int](3)
array(0) = 42 // array.update(0, 42), присваиваем первый элемент
println(array(0)) // array.apply(0), получаем первый элемент
```

1.7. Модификаторы доступа

По-умолчанию все члены класса публичные. Для всего остального есть модификаторы `private` и `protected`. Вместо `package private` модификатора используются модификаторы с усилением.

```
package ru.spbau.jvm.scala

class Foo {
  private var foo: Int         // доступ только внутри класса
  private[scala] var bar: Int // доступ только в пакете ru.spbau.jvm.scala
  private[jvm] var baz: Int    // доступ только в пакете ru.spbau.jvm
  private[this] var bus: Int  // доступ только внутри экземпляра

  def fun(other: Foo): Unit = {
    bus = 42       // ок, доступ изнутри экземпляра
    other.bus = 42 // ошибка, доступ из другого экземпляра
  }
}
```

Примарный конструктор тоже можно сделать защищенным или приватным

```
class Foo protected()
class Bar private()
class Baz private[this]()
```

Так как `companion object`'ы имеют доступ к приватным полям своих классов, единственный способ что-то от них скрыть (в т.ч. конструктор) – это `private[this]`. При этом, так как в Java такой фичи нет, в ней этот модификатор воспринимается как просто `private`.

1.8. Properties

Они есть. Примерно как в Котлине. **TODO:** об этом позже

1.9. Hello world!

Есть несколько способов написать запускающуюся программу на Scala. Самый простой и привычный – это объявить метод `main`.

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello world!")
  }
}
```

Другой способ – отнаследоваться от трейта `App` и написать всё прямо в теле класса. Но таким способом пользоваться не рекомендуется.

```
class HelloWorld extends App {
  println("Hello world!")
}
```

1.10. Пакеты, `import`'ы и коллизии

В Scala, как и в Котлине, есть свои коллекции, например `List`. Если мы хотим использовать `java.util.List`, надо либо явно писать весь пакет, либо заимпортить его. Но при импорте Scala'вский `List` перекроется, что не очень хорошо. Поэтому можно импортить прямо пакеты.

```
import java.util
var list: util.List[Int]
```

Но такие названия как `util` тоже могут встречаться часто и конфликтовать. Чтобы этого избежать, можно создавать синонимы для пакетов. В основном это используют именно для `java.util`, для чего-то другого так делают редко.

```
import java.{util => ju}
var list: ju.List[Int]
```

1.11. Избавляемся от лишнего

Рассмотрим следующий код и поймем, что в нем можно сократить.

```
def foo(x: Boolean): Int = {
  if (x) return 42
  return 43
}
```

Во-первых, последний `return` можно не писать, потому что результат последнего выражения и так возвращается по-умолчанию. Во-вторых, можно объединить две ветки условия в `if else` и избавиться от `return` совсем.

```
def foo(x: Boolean): Int = {
  if (x) 42 else 43
}
```

Теперь у нас функция состоит из одного выражения, поэтому можно убрать скобки и возвращаемый тип, так как он выведется.

```
def foo(x: Boolean) = if (x) 42 else 43
```

1.12. Рекурсия

Напишем функцию, считающую факториал числа.

```
def fac(n: Int): BigInt =
  if (n == 1) 1
  else n * fac(n - 1)
```

Проблема такой реализации в том, что при вызове `fac(10000)` случится переполнение стека. Хочется сделать хвостовую рекурсию, а затем развернуть ее в цикл.

```
@tailrec
def facTailRec(n: Int, accumulator: BigInt = 1): BigInt = n match {
  case 1 => accumulator
  case _ => facTailRec(n - 1, n * accumulator)
}
```

Здесь мы видим сразу две фичи – аннотацию `@tailrec` и матчер. Матчер работает примерно как `when` в Котлине. Аннотация `@tailrec` нужна только для того, чтобы удостовериться, что в методе используется хвостовая рекурсия. В противном случае код не скомпилируется. Компилятор же **всегда** разворачивает хвостовую рекурсию в цикл.

1.13. Метод в методе

В нашей реализации факториала есть одна концептуальная проблема. Мы можем передать туда любой аккумулятор, и это повлияет на результат. Чтобы этого избежать, можно обернуть наш метод в другой метод, таким образом спрятав его от нерадивых пользователей.

```
def facTailRec(n: Int): BigInt = {
  @tailrec
  def facTailRecInner(n: Int, accumulator: BigInt = 1): BigInt = n match {
    case 1 => accumulator
    case _ => facTailRec(n − 1, n * accumulator)
  }

  facTailRecInner(n)
}
```

1.14. `try`

Блок `catch` очень похож на `match` тем, что внутри мы матчим исключения, чтобы по-разному их обрабатывать. Рассмотрим пример.

```
try {
  fac(100000)
} catch {
  case e: StackOverflowError => facTailRec(100000)
  case _: IOException | _: FileException => ... // комбинируем случаи
  _: // матчим всё что не сматчилось ранее
}
```

TODO: подробнее будет позже, вместе с `match`

1.15. Алгебраические типы данных

Хотим написать список в функциональном стиле (как в Haskell).

```
sealed trait IntList

final class IntNil extends IntList
final class IntCons(val head: Int, val tail: IntList) extends IntList
```

Теперь можем заводить списки.

```
val list: IntList = new IntCons(1, new IntCons(2, new IntCons(3, new IntNil)))
```

А потом матчить.

```
list match {
  case cons: IntCons => println(cons.head) // теперь cons = list.asInstanceOf[IntCons]
  case _: IntNil => // ничего не делаем
}
```

Поскольку трейт `IntList` мы объявили `sealed`, у нас есть гарантия, что больше никаких наследников кроме этих двух нет, то есть всегда что-то заматчится. Иначе мы могли бы завести инстанс анонимного класса, и матчер бы свалился, потому что никакая ветка бы не подошла.

1.16. `case` классы

В нашей реализации списка тоже есть лишний код, от которого лучше избавиться. Для начала поймем, что `IntNil` бывает всего один, поэтому его можно сделать `object`'ом. А теперь добавим к наследникам сключевое слово `case`. Оно создает для класса `companion object`, определяет для него методы `apply()` и `thenApply()` (о нем позже). Сделает неизменяемые параметры примарного конструктора полями (не надо будет писать в конструкторе `val`) и сгенерирует методы `equals()`, `hashCode()` и `toString()`. В Котлине очень похожее поведение имеют `data`-классы.

```
1 sealed trait IntList
2
3 case object IntNil extends IntList
4 final case class IntCons(head: Int, tail: IntList) extends IntList
5
6 val list: IntList = IntCons(1, IntCons(2, IntCons(3, IntNil)))
```

Более того, теперь в матчере для `IntNil` не надо писать плейсхолдер, а `IntCons` теперь можно матчить по его параметрам. Теперь можем рекурсивно вывести список на экран.

```
1 def printList(list: IntList): Unit = {
2   list match {
3     case IntNil =>
4     case IntCons(head, tail) =>
5       println(head)
6       printList(tail)
7   }
8 }
```

2. Функциональное программирование на Scala

Как мы помним, Scala – симбиоз функционального и объектно-ориентированного языков. Так как в JVM не предусмотрены фичи функциональной парадигмы, всё сделано синтетически.

2.1. Функциональные типы

Рассмотрим следующий метод.

```
def increase(x: Int): Int = x + 1
```

В байт коде это будет обычная функция, увеличивающая свой аргумент на единицу.

Можно написать то же самое в функциональном стиле.

```
val increase: Int => Int = x => x + 1
```

Здесь `Int => Int` – это функциональный тип, а `x => x + 1` – объявление функции. Можно написать еще более функционально.

```
val increase: Int => Int = _ + 1
```

В этих примерах надо обязательно явно указывать тип, потому что иначе компилятор будет думать, что аргумент имеет тип `Any`, и у него нет оператора `+`.

Наиболее интересная версия. Тут можно без явного указания типа.

```
val increase = (_: Int) + 1
```

Запись `Int => Int` – это синтаксический сахар для `Function1[Int, Int]`. То есть с точки зрения Java, это будет просто класс функция, у которой можно вызвать метод `apply()`.

Функциональные классы и другие подобные штуки в Scala написаны вручную и существуют лишь до 22, то есть могут принимать максимум 22 аргумента. В Scala 3 их немного переделают, и это ограничение уйдет.

Рассмотрим теперь функцию с двумя аргументами.
У нее будет тип `Function2[Int, Int, Int]`.

```
val sum: (Int, Int) => Int = _ + _
```

Если хочется как-то различать аргументы, например, если хотим сложить их в другом порядке, можно применить pattern matching.

```
val sum: (Int, Int) => Int = {
  case (left, right) => right + left
}
```

2.2. Частичное применение

С точки зрения компилятора предыдущий пример – это частичная функция. Она позволяет матчить аргументы и проверять частные случаи.

```
val weirdSum: PartialFunction[(Int, Int), Int] = {
  case (0, _) | (_, 0) => 0 // если хотя бы один ноль, то сумма тоже ноль
  case (left, right) => left + right
}
```

Если мы в первом кейсе заменим ноль на плейсхолдер, то получим частично примененную функцию.

```
val weirdSum: PartialFunction[(Int, Int), Int] = {
  case (0, _) | (_, 0) => _
}
```

Если оба аргумента будут ненулевые, то pattern matching не пройдется, и выпадет ошибка `MatchError`.

Вернемся к нормальной реализации `sum()` и вызовем её.

```
val sum: PartialFunction[(Int, Int), Int] = {
  case (left, right) => left + right
}

sum(1, 2);
```

Внимательный читатель найдет здесь две особенности. Во-первых, мы вызвали поле `val sum` как функцию. В этом нам помог уже известный нам метод `apply()`, который можно вызывать, просто приписав скобки к переменной. Во-вторых, наша функция должна принимать tuple с двумя `Int`'ами, а мы передали просто два числа. Автоматический тюплинг преобразовал наши аргументы в одну пару и передал в функцию.

То же самое можно написать в виде функции.

```
val sum: (Int, Int) => Int = {
  case (left, right) => left + right
}

def sum(left: Int, right: Int): Int = (left, right) match {
  case (left, right) => left + right
}

sum.apply(1, 2); // вызовется val sum
sum(1, 2); // вызовется def sum
```

Здесь случилась коллизия имен, но компилятор поймет, что если мы вызываем что-то как функцию, то надо вызвать именно функцию, а если через `apply()`, то это переменная.

2.3. Каррирование

Каррирование – преобразование функции из `(Int, Int) => Int` в `Int => (Int => Int)`. В Scala как такового каррирования нет, но есть способ его имитировать. Это позволит нам реализовать частичное применение.

2.3.1. Placeholder

Самый бессмысленный и беспощадный пример – вызов функции, с заменой всех аргументов на плейсхолдеры.

```
def sum(left: Int, right: Int): Int = left + right
sum(_, _) // выражение имеет тип (Int, Int) => Int
sum(_, _)(1, 2) // и это можно даже вызвать
```

Более содержательный пример – вызов с заменой некоторых аргументов на плейсхолдеры.

```
sum(42, _) // получили выражение с типом Int => Int
```

2.3.2. Parameter scope

Можно объявлять функцию с несколькими скоупами параметров, тогда и при вызове придется писать аргументы в разных скобках.

```
def sum(left: Int)(right: Int): Int = left + right
sum(1)(2) // просто вызываем sum
sum(1) _ // частичное применение sum
```

2.4. Вариантность

Вернемся к нашему инкременту и подумаем, какой наиболее общий тип можно приписать нашей функциональной переменной.

```
val increase: ? => ? = (x: Int) => x + 1
```

Принять мы можем любого наследника `Int`, а вернуть любого его потомка. Это называется умными словами – **контрвариантность** аргумента и **ковариантность** возвращаемого значения. Таким образом наиболее общий тип будет такой.

```
val increase: Nothing => Any = (x: Int) => x + 1
```

`Any` – это "Top type" – тип, от которого наследуются все остальные, а `Nothing` – "Bottom type" – является чем-то вроде наследника всех остальных классов.

Для generic параметров тоже можно задавать вариантность. Посмотрим на объявление `Function1`.

```
trait Function1[−T1, +R]
```

В Java эму соответствует такая запись.

```
interface Function1<? extends T1, ? super R> { }
```

2.5. Кортежи

Кортежи – это синтаксический сахар для `Tuple1`, `Tuple2`, ... `Tuple22`. В Scala 3 ограничение в 22 элемента тоже уйдет.

К элементам кортежа можно обращаться по номерам (`_1`, `_2`, ...). Элементы нумеруются с единицы.

```
val tuple: (Int, String) = (42, ''str'')
println(tuple._1)
println(tuple._2)
```

Но так писать крайне не рекомендуется. Вместо этого элементы лучше именовать и далее использовать как переменные.

```
val (index, text) = (42, ''str'')
println(index)
println(text)
```

Рассмотрим функцию суммы, принимающая tuple из двух слагаемых.

```
val sum: Function1[Tuple2[Int, Int], Int] = tuple => tuple._1 + tuple._2
```

В Haskell-стиле правильнее было бы писать так.

```
val sum: Function1[Tuple2[Int, Int], Int] = (left, right) = left + right
```

Но в Scala так писать пока нельзя, но скоро будет можно. Пока что приходится писать через `case`.

В Котлине нет тюплов. Там можно распаковывать списки и коллекции, пользуясь похожим на тюплы синтаксисом. Недостаток списков в том, что нельзя во время компиляции (на уровне типов) проверить количество элементов, и в рантайме наш `sum()` может сломаться.

Вообще система типов нужна, чтобы отлавливать ошибки как можно раньше. Поэтому в промышленных языках чаще используется статическая типизация.