Основные структуры данных

В Scala существует несколько прекрасных коллекций.

**Смотрите также:** В Effective Scala описывается как использовать [Коллекции](https://twitter.github.com/effectivescala/index-ru.html#%D0%9A%D0%BE%D0%BB%D0%BB%D0%B5%D0%BA%D1%86%D0%B8%D0%B8).

Списки

scala> val numbers = List(1, 2, 3, 4)

numbers: List[Int] = List(1, 2, 3, 4)

Наборы

Наборы не содержат одинаковых элементов

scala> Set(1, 1, 2)

res0: scala.collection.immutable.Set[Int] = Set(1, 2)

Кортеж

Кортеж объединяет простые логические элементы коллекции без использования классов.

scala> val hostPort = ("localhost", 80)

hostPort: (String, Int) = (localhost, 80)

В отличие от case классов, у него нет именованных функций доступа, вместо этого у него есть функции доступа, которые носят названия по номеру позиции элемента кортежа и они нумеруются с 1, а не с 0.

scala> hostPort.\_1

res0: String = localhost

scala> hostPort.\_2

res1: Int = 80

Кортежи прекрасно могут использоваться вместе с сопоставлением с образцом.

hostPort match {

case ("localhost", port) => ...

case (host, port) => ...

}

Кортежи имеет специальный “соус”, который позволяет сделать Кортеж 2 значений: ->

scala> 1 -> 2

res0: (Int, Int) = (1,2)

**Смотрите также:** В Effective Scala описывается [разрушение связей](https://twitter.github.com/effectivescala/index-ru.html#%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5%20%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5-%D0%A0%D0%B0%D0%B7%D1%80%D1%83%D1%88%D0%B5%D0%BD%D0%B8%D0%B5%20%D1%81%D0%B2%D1%8F%D0%B7%D0%B5%D0%B9%20(Destructuring%20bindings)) (“распаковка” кортежа).

Карты

Они могут содержать в себе основные типы данных.

Map(1 -> 2)

Map("foo" -> "bar")

Выглядит словно это особый синтаксис, но вспомните наше обсуждение в разделе Кортеж, что с помощью -> можно создавать Кортежи.

Map() тоже использует синтаксис, который мы изучили ранее на Уроке №1: Map(1 -> "one", 2 -> "two"), который раскрывается в Map((1, "one"), (2, "two")), где первый элемент является ключом, а вторым элементом является некое значение.

Карты внутри себя могут содержать другие Карты или даже функции.

Map(1 -> Map("foo" -> "bar"))

Map("timesTwo" -> { timesTwo(\_) })

Опция

Опция представляет собой контейнер, который хранит какое-то значение или не хранит ничего совсем.

Основной интерфейс Опции выглядит следующим образом:

trait Option[T] { def isDefined: Boolean def get: T def getOrElse(t: T): T }

Опция сам по себе это обобщенный тип и он имеет два подкласса: Some[T] и None

Давайте взглянем на пример того, как Опция может использоваться:

Map.get использует Option для возврата собственных значений. Опция говорит вам, что метод может не вернуть того значения, которое мы запросили.

scala> val numbers = Map("one" -> 1, "two" -> 2)

numbers: scala.collection.immutable.Map[java.lang.String,Int] = Map(one -> 1, two -> 2)

scala> numbers.get("two")

res0: Option[Int] = Some(2)

scala> numbers.get("three")

res1: Option[Int] = None

Теперь наши данные отлавливаются с помощью Option. Как мы можем это использовать?

Первое, что приходит на ум, это использовать условие, опираясь на метод isDefined.

// Мы хотим умножить число на 2 или возвратить 0. val result = if (res1.isDefined) { res1.get \* 2 } else { 0 }

Мы предполагаем, что вы будете использовать getOrElse или сопоставление с образцом для результата.

getOrElse дает вам легкий способ объявить стандартное значение.

val result = res1.getOrElse(0) \* 2

Option прекрасно работает вместе с сопоставлением с образцом.

val result = res1 match { case Some(n) => n \* 2 case None => 0 }

**Смотрите также:** В Effective Scala описываются [Опции](https://twitter.github.com/effectivescala/index-ru.html#%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5%20%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5-%D0%9E%D0%BF%D1%86%D0%B8%D1%8F).

Функциональные комбинаторы

Комбинаторы называются так потому, что они созданы, чтобы объединять результаты. Результат одной функции часто используется в качестве входных данных для другой.

Наиболее распространенным способом, является использование их со стандартными структурами данных.

map

Применяет функцию к каждому элементу из списка, возвращается список с тем же числом элементов.

scala> numbers.map((i: Int) => i \* 2)

res0: List[Int] = List(2, 4, 6, 8)

или передается частично вызываемая функция

scala> def timesTwo(i: Int): Int = i \* 2

timesTwo: (i: Int)Int

scala> numbers.map(timesTwo \_)

res0: List[Int] = List(2, 4, 6, 8)

foreach

foreach похож на map, но ничего не возвращает. foreach используется для создания побочных эффектов.

scala> numbers.foreach((i: Int) => i \* 2)

Он ничего не возвращает.

Вы можете попробовать сохранить результат в переменную, но она будет иметь тип Unit (другими словами void)

scala> val doubled = numbers.foreach((i: Int) => i \* 2)

doubled: Unit = ()

filter

Данный комбинатор удаляет любой элемент, если функция, применяемая к этому элементу, возвращает ложь. Функции, которые возвращают Boolean, часто называются функциями-предикатами

scala> numbers.filter((i: Int) => i % 2 == 0)

res0: List[Int] = List(2, 4)

scala> def isEven(i: Int): Boolean = i % 2 == 0

isEven: (i: Int)Boolean

scala> numbers.filter(isEven \_)

res2: List[Int] = List(2, 4)

zip

zip объединяет содержимое двух списков в один парный список.

scala> List(1, 2, 3).zip(List("a", "b", "c"))

res0: List[(Int, String)] = List((1,a), (2,b), (3,c))

partition

partition разделяет список, в зависимости от результата, который возвращает функция-предикат.

scala> List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).partition(\_ % 2 == 0)

res0: (List[Int], List[Int]) = (List(2, 4, 6, 8, 10),List(1, 3, 5, 7, 9))

find

find возвращает первый элемент коллекции, который удовлетворяет функции-предикату.

scala> numbers.find((i: Int) => i > 5)

res0: Option[Int] = Some(6)

drop и dropWhile

drop удаляет первые i элементов

scala> numbers.drop(5)

res0: List[Int] = List(6, 7, 8, 9, 10)

dropWhile удаляет первый элемент, который не удовлетворяет функции-предикату. Например, если мы применим dropWhile к нечетным числам из нашего списка, то 1 будет удалена (но не 3, которая стоит за 2).

scala> numbers.dropWhile(\_ % 2 != 0)

res0: List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9, 10)

foldLeft

scala> numbers.foldLeft(0)((m: Int, n: Int) => m + n)

res0: Int = 55

0 – это начальное значение (Не забывайте, что мы используем числа из List[Int]), где m  
работает как аккумулятор.

Взгляните сами:

scala> numbers.foldLeft(0) { (m: Int, n: Int) => println("m: " + m + " n: " + n); m + n }

m: 0 n: 1

m: 1 n: 2

m: 3 n: 3

m: 6 n: 4

m: 10 n: 5

m: 15 n: 6

m: 21 n: 7

m: 28 n: 8

m: 36 n: 9

m: 45 n: 10

res0: Int = 55

foldRight

Этот комбинатор похож на foldLeft, за исключением того, что он работает с противоположной стороны.

scala> numbers.foldRight(0) { (m: Int, n: Int) => println("m: " + m + " n: " + n); m + n }

m: 10 n: 0

m: 9 n: 10

m: 8 n: 19

m: 7 n: 27

m: 6 n: 34

m: 5 n: 40

m: 4 n: 45

m: 3 n: 49

m: 2 n: 52

m: 1 n: 54

res0: Int = 55

flatten

flatten сжимает вложенные структуры.

scala> List(List(1, 2), List(3, 4)).flatten

res0: List[Int] = List(1, 2, 3, 4)

flatMap

flatMap это часто используемый комбинатор, который объединяет map и flatten. flatMap берет функцию, которая работает с вложенными списками и объединяет результаты.

scala> val nestedNumbers = List(List(1, 2), List(3, 4))

nestedNumbers: List[List[Int]] = List(List(1, 2), List(3, 4))

scala> nestedNumbers.flatMap(x => x.map(\_ \* 2))

res0: List[Int] = List(2, 4, 6, 8)

Думайте об этом, как о коротком способе использования map, а затем применения flatten к результату:

scala> nestedNumbers.map((x: List[Int]) => x.map(\_ \* 2)).flatten

res1: List[Int] = List(2, 4, 6, 8)

в этом примере вызывается map, а позднее flatten, как пример “комбинаторной” природы этих функций.

**Смотрите также:** В Effective Scala описывается [flatMap](https://twitter.github.com/effectivescala/index-ru.html" \l "%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5%20%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5-%60flatMap%60).

Обобщенные функциональные комбинаторы

Теперь мы узнали о множестве функции для работы с коллекциями.

Что нам может понадобиться, чтобы иметь возможность написать свои функциональные комбинаторы?

Интересно, что каждый функциональный комбинатор показанный выше, может быть написан поверх fold. Рассмотрим несколько примеров.

def ourMap(numbers: List[Int], fn: Int => Int): List[Int] = {

numbers.foldRight(List[Int]()) { (x: Int, xs: List[Int]) =>

fn(x) :: xs

}

}

scala> ourMap(numbers, timesTwo(\_))

res0: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)

Почему именно List[Int]()? Scala не достаточно умна, чтобы реализовать пустой список, для хранения целочисленных значений.

А как же Map?

Все функциональные комбинаторы прекрасно работают и с Картами. Карты можно рассматривать как список пар, так что функции, которые вы пишете работают с парами ключей и значений Карты.

scala> val extensions = Map("steve" -> 100, "bob" -> 101, "joe" -> 201)

extensions: scala.collection.immutable.Map[String,Int] = Map((steve,100), (bob,101), (joe,201))

теперь выберем каждую запись, у которой телефонный код меньше 200.

scala> extensions.filter((namePhone: (String, Int)) => namePhone.\_2 < 200)

res0: scala.collection.immutable.Map[String,Int] = Map((steve,100), (bob,101))

Т.к. в результате вы получаете кортеж, то вам приходится тащить ключи и значения с их позиционными функциями доступа. Да уж!

К счастью, мы можем использовать сопоставление с образцом, чтобы извлечь ключ и значение.

scala> extensions.filter({case (name, extension) => extension < 200})

res0: scala.collection.immutable.Map[String,Int] = Map((steve,100), (bob,101))

Объединение функций

Давайте создадим две полезные функции:

scala> def addUmm(x: String) = x + " umm"

addUmm: (x: String)String

scala> def addAhem(x: String) = x + " ahem"

addAhem: (x: String)String

compose

compose создает новую функцию, которая является объединением других функций f(g(x))

scala> val ummThenAhem = addAhem \_ compose addUmm \_

ummThenAhem: (String) => String = <function1>

scala> ummThenAhem("well")

res0: String = well umm ahem

andThen

andThen похожа на  compose, но сначала вызывается первая функция, затем вторая g(f(x))

scala> val ahemThenUmm = addAhem \_ andThen addUmm \_

ahemThenUmm: (String) => String = <function1>

scala> ahemThenUmm("well")

res1: String = well ahem umm

Каррирование и Частичный вызов функций

case утверждения

Что же такое case утверждения?

Это подкласс функций, который называется PartialFunction.

Что представляет собой совокупность множества case утверждений?

Это множество объединений PartialFunctions вместе.

Понимание PartialFunction

Функция работает для каждого аргумента определенного типа. Другими словами, функция объявляется как (Int) => String, принимающая любой Int и возвращающая строку.

Частичная функция определена только для определенных значений определенного типа. Частичные функции (Int) => String не может принимать любой Int.

isDefinedAt это метод PartialFunction, который может использоваться, чтобы определить, будет ли PartialFunction принимать данный аргумент.

*Заметьте* PartialFunction не связана с частично вызываемыми функциями, о которых мы говорили ранее.

**Смотрите также:** В Effective Scala описывается [PartialFunction](https://twitter.github.com/effectivescala/index-ru.html" \l "%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5%20%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5-%D0%A7%D0%B0%D1%81%D1%82%D0%B8%D1%87%D0%BD%D1%8B%D0%B5%20%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8).

scala> val one: PartialFunction[Int, String] = { case 1 => "one" }

one: PartialFunction[Int,String] = <function1>

scala> one.isDefinedAt(1)

res0: Boolean = true

scala> one.isDefinedAt(2)

res1: Boolean = false

Вы можете использовать частичную функцию.

scala> one(1)

res2: String = one

PartialFunctions может быть объединена с чем-то новым, называемым OrElse, который показывает, определен ли PartialFunction с передаваемым аргументом.

scala> val two: PartialFunction[Int, String] = { case 2 => "two" }

two: PartialFunction[Int,String] = <function1>

scala> val three: PartialFunction[Int, String] = { case 3 => "three" }

three: PartialFunction[Int,String] = <function1>

scala> val wildcard: PartialFunction[Int, String] = { case \_ => "something else" }

wildcard: PartialFunction[Int,String] = <function1>

scala> val partial = one orElse two orElse three orElse wildcard

partial: PartialFunction[Int,String] = <function1>

scala> partial(5)

res24: String = something else

scala> partial(3)

res25: String = three

scala> partial(2)

res26: String = two

scala> partial(1)

res27: String = one

scala> partial(0)

res28: String = something else

Тайна о case.

 Мы уже видели case утверждение там, где обычно использовалась функция.

scala> case class PhoneExt(name: String, ext: Int)

defined class PhoneExt

scala> val extensions = List(PhoneExt("steve", 100), PhoneExt("robey", 200))

extensions: List[PhoneExt] = List(PhoneExt(steve,100), PhoneExt(robey,200))

scala> extensions.filter { case PhoneExt(name, extension) => extension < 200 }

res0: List[PhoneExt] = List(PhoneExt(steve,100))

Почему это работает?

filter принимает функцию. В этом случае используется функция-предикат (PhoneExt) => Boolean.

PartialFunction — это подтип Function, поэтому filter может принимать PartialFunction в качестве аргумента!