Дата последней сборки документа:
26 октября 2020 г. в 19:43

Заметки. Практика использования и наиболее полезные конструкции языка `Scala`

Подвойский А.О.

Здесь приводятся заметки по некоторым вопросам, касающимся машинного обучения, анализа данных, программирования на языках `Scala` и прочим сопряженным вопросам так или иначе, затрагивающим работу с данными.

Содержание

1. Установка SDK

SDKMAN (Software Development Kit Manager) https://sdkman.io/ – Очень полезная утилита для работы scala-средой.

```
curl -s "https://get.sdkman.io" | bash
source "$HOME/.sdkman/bin/sdkman-init.sh"
sdk version
```

2. Компиляция программ на Scala

Пусть есть программа такая программа

```
object Hello extends App {
  println("Hello, world")
}
```

Скомпилировать эту программу можно с помощью утилиты командной строки `scalac`

```
scalac Hello.scala
```

Затем можно запустить программу с помощью утилиты командной строки `scala`

```
scala Hello
```

После этого в рабочей директории появятся файлы с расширениями `Hello.class`, `'Hello$.class'`, `'Hello$delayedInit$body.class'`

3. Краткая справка по особенностям языка

3.1. Часто используемые типы

Все типы в Scala являются классами, поэтому нет никакой разницы между простым типом и классом. Можно вызывать метод непосредственно у числа, например

```
var float_var = 1.toFloat
```

Грубо можно выделить 10 простых классов:

- `Byte`: целое число от -128 до 127,
- `Short`: целое число от -32768 до 32767,
- `Int`: целое число от -2147483648 до 2147483647,

- `Long`: целое число от -9223372036854775808 до 9223372036854775807,
- `Float`: десятичное число от -3.4028235-e38 до 3.4028235+e38,
- `Double`: десятичное число от -1.7976931348623157+e308 до 1.7976931348623157+e308,
- `Char`: символ; литералами являются одинарные кавычки,
- `String`: строка; простейшими литералами являются двойные кавычки. Использование:
  - `"`: можно использовать специальные символы переноса строки и прочих,
  - `"""`: многострочная строка,
  - `s"` или `s"""`: строка, в которой можно подставлять переменные через `$varName`, или выражения `${1 + $varName}`, например,

```
import scala.io.StdIn.readLine
import scala.io.StdIn.readInt

val name = readLine("Your name: ")
print("Enter your age: ")
val age = readInt()
val result = if (age < 18) "child" else "adult"
print(s"Hello, $name! You are $result.") // Hello, Leor! You are adult.
```

  - `f"` или `f"""`: строка, в которую можно подставлять переменные через `$varName%format`, или через `${varName}%format`, например,

```
print(f"Hello, ${name}! You are ${result}.") // Hello, Leor! You are adult.
print(f"${age/scala.math.cos(4) + 1}%.3e") // -4,049e+01
```

  - `raw"` или `raw"""`: сырая строк, которая не интерпретирует специальные символы.
- Так же можно реализовать свои специальные литералы через implicit-классы.

3.2. Методы

Scala позволяет использовать в качестве имен любые символы, поэтому «+» это имя метода. Основные правила вызова функций и методов:

- Если аргумент один, то `()` можно заменить на `{}`,
- Если аргумент один и используется точечная нотация, то `()` не использовать,
- Если функция не принимает аргументов, то `()` можно не использовать. Общее правило здесь такое: если метод изменяет объект, то скобки нужны, иначе нет особенной необходимости,
- Если имя метода заканчивается на «`:`», то метод правоассоциативен, то есть параметр будет слева.

В Scala нет операторов `++` и `--`, вместо них используются `+=1` и `-=1`.

3.3. Управляющие конструкции и функции

Все управляющие конструкции в Scala возвращают значение, как следствие, имеют тип и могут быть присвоены переменной. Например

```
val x: Int = 1; val y: Int = 2
val z = if (x > 1) "Yes" else if (y > 2) "Oh, yea!" else "No" // z: String = No
```

В иерархии типов Scala общим родительским является тип Any, общим дочерним Nothing, а за тип void отвечает Unit, который имеет значение `()`.

3.4. Циклы

Самое первое, что необходимо узнать о циклах в Scala, здесь нет инструкций `break` и `continue`. В Scala есть стандартные циклы `while` и `do/while`

```
while (true) {
  // doSomething
}

do {
  // Something
} while (true)
```

Есть цикл `for`

```
for (i <- 1 to 5) print(i) // 12345
for (i <- 1 until 5) print(i) // 1234
```

Допускается использовать несколько генераторов, определений и ограничителей, разделенных между собой точкой с запятой или определенных с новой строки. При этом генераторы будут выполняться как вложенные циклы слева-направо

```
for (i <- 1 to 3; j <- 1 to 5) print(s"$i-$j\t")
```

В определениях переменные являются var. Ограничители выглядят как `if booleanExpression`, итерация запуститься только в случае выполнения условий. Каждый ограничитель следует за генератором, которому он принадлежит, и не должен содержать переменные, находящиеся в объявлении цикла после него

```
for (i <- 1 to 3 if i % 2 == 0; j <- 1 to 5 if i != j) print(s"$i-$j")
```

Если тело цикла начинается с ключевого слова `yield`, то цикл вернет *коллекцию*

```
for (i <- 1 to 3; j <- 1 to 3) yield i + j
//val res IndexedSeq[Int] = Vector(11, 12, 13, 12, ..., 17, 18)
```

3.5. Функции

Функция определяется с помощью ключевого слова `def`. Тип возвращаемого объекта обязательно указывать только в случае процедур (функций, которые ничего не возвращают) и рекурсивных функций, однако нет ничего плохого в том, чтобы всегда указывать тип возвращаемого объекта

```
// это процедура, поэтому тип Unit обязательно указывать!
def foo(s: String, status: Float = 0.835.toFloat): Unit = {
  print(f"--- ${s} [${status}%.3e] ---")
}

// это обычная функция, возвращающая объект целочисленного типа данных
def prod(a: Int, b: Int = 10): Int = {
  // нет необходимости использовать return!!!
  a*b
}

// это рекурсивная функция
def fact(n: Int): Int = {
  if (n <= 0) 1 else {
    n*fact(n-1)
```

```
  }
}

// функция, принимающая переменное число аргументов
def varargs(args: Int*): Unit = {
  print(args.length)
}
varargs(1, 4, 10, 2) // 4
varargs(1 to 5: _*)  // 5
```

3.6. Каррирование

Функцию нескольких аргументов можно превратить в каррированный вариант

```
val plus3 = (x: Int, y: Int, z: Int) => {
    x + y + z  // здесь не нужно использовать return
}
val plus3c = plus3.curried // вызван метод curried
plus3c(10)(20)(30)  // 60
```

На Python эта задача была бы решена так

```
def plus3(x: int) -> int:
    def layer1(y: int) -> int:
        def layer2(z: int) -> int:
            return x + y + z
        return layer2
    return layer1
plus3(10)(20)(30)  # 60
```

3.7. Композиция

Цепочки можно вычислять с помощью методов `andThen` и `compose`

```
val plus1 = (x: Int) => x + 1
val mul3 = (x: Int) => x*3

val plusThenMul = plus1 andThen mul3
plusThenMul(5)  // mul3(plus1(5))

val plusBeforeMul = plus1 compose mul3
plusBeforeMul(5) // plus1(mul3(5))
```

3.8. Массивы

Массивы бывают *фиксированной* и *переменной* длинны, они типизированы типом, которых их наполняет (тип указывается в квадратных скобках). Массивом фиксированной длины является тип `Array`. Для инициализации пустого массива определенной длины используется конструкция `new Array[T](length: Int)`, а для инициализации наполненного массива – конструкция `Array(value1, value2, ..., valueN)`

```
// при создании пустого массива необходимо указывать new, в противном случае будет создан
// массив, состоящий из указанного объекта
val arr = new Array[Float](3)  // val res50: Array[Float] = Array(0.0, 0.0, 0.0)
Array(1.0, 2.5, 3.75, 4)  // val res51: Array[Double] = Array(1.0, 2.5, 3.75, 4.0)
```

Для доступа к элементам массива используются круглые скобки

```
val a = new Array[Int](5)  // Array[Int] = Array(0, 0, 0, 0, 0)
a(2) = 10 // Array[Int] = Array(0, 0, 10, 0, 0)
val a0f2 = a(2) // a0f2: Int = 10
```

Массивом переменной длины является тип ArrayBuffer из пакета `scala.collection.mutable`. Для создания пустого массива можно воспользоваться `ArrayBuffer[T]()` с `new` или без. Основные методы массивов:

- `+=(n: T)` – добавить элемент в конец,
- `++=` – добавить другую коллекцию в конец,
- `trimEnd(n: Int)` – удалить n-элементов с конца,
- `insert(position: Int, values: T*)` – вставить на позицию position значения values,
- `remove(position: Int, length: Int = 1)` – удалить с позиции position значения.

Пример

```
import scala.collection.mutable.ArrayBuffer
// массив переменной длины
val arr = new ArrayBuffer[Int]()
arr += 10 // arr.type = ArrayBuffer(10)
arr += 20 // arr.type = ArrayBuffer(10, 20)
// массив фиксированной длины
val arrint = new Array[Int](2) // arrint: Array[Int] = Array(0, 0)
arrint(0) = 100
arrint(1) = 200
arr ++= arrint // arr.type = ArrayBuffer(10, 20, 100, 200)
arr += (1000: Int) // arr.type = ArrayBuffer(10, 20, 100, 200, 1000)
```

Заполнить массив фиксированной длины в цикле

```
val arr = new Array[Float](3)
for (i <- 0 until arr.length) arr(i) = 10.toFloat*(i + 1)
```

Для преобразования массива одного типа в другой используются методы `toArray/toBuffer`. Обойти массивы можно несколькими способами

```
val arr: Array[Int] = Array(4, 0, 10) // или просто val arr = Array(4, 0, 10)
for (i <- 0 until arr.length) println(i, arr(i))
// (0,4)
// (1,0)
// (2,10)

for (elem <- arr) println(elem)
//  4
//  0
// 10
// то же самое
arr.foreach(println(_))
```

Можно использовать `yield` для создания новых коллекций

```
import math._

val arr = Array(1 to 5: _*) // val arr: Array[Int] = Array(1, 2, 3, 4, 5)
val res = for (elem <- arr) yield pow(elem, 2) + 10 // val res4: Array[Double] = Array(11.0,
    14.0, 19.0, 26.0, 35.0)
```

В Python это выглядело было так

```
import numpy as np

arr = np.array(range(1,5+1))
res = arr**2. + 10
```

Есть несколько типичных методов

```
val arr = Array(3.23, 5.3, 4.2) // val arr: Array[Double] = Array(3.23, 5.3, 4.2)
arr.mkString(";")
val res27: String = 3.23;5.3;4.2
```

Матрицу можно создать так

```
val m = Array.ofDim[Int](2,2)
```

3.9. Ассоциативные массивы

Ассоциативные массивы в Scala представлены классами `Map[KeyType, ValueType]` для неизменяемых массивов, и `scala.collection.mutable.Map[KeyType, ValueType]` для изменяемых. Ассоциативные массивы представляют собой коллекцию пар. Пару можно создать через `(Value1, Value2)` или `Value1 -> Value2`.

Неизменяемые массивы можно создать только одним способом, через вызов `apply` объекта-компаньона

```
// неизменяемый ассоциативный массив
val m = Map[Int, Int](1 -> 2, 3 -> 4)
// val m: scala.collection.immutable.Map[Int,Int] = Map(1 -> 2, 3 -> 4)
```

По-аналогии можно создать наполненный изменяемый массив

```
// изменяемый ассоциативный массив
val mutable_arr = scala.collection.mutable.Map[String,String](
  "package name" -> "Ansys",
  "solver type" -> "iterative"
)
// обращение по ключу
mutable_arr("package name") // val res2: String = Ansys
mutable_arr("solver type") // val res3: String = iterative
```

А вот чтобы создать пустой изменяемый ассоциативный массив необходимо воспользоваться классом `scala.collection.mutable.HashMap`

```
val hash_arr = scala.collection.mutable.HashMap[String,String]()
// или так
import scala.collection.mutable.HashMap
val hash_arr = HashMap[String,String]()  // пустой изменяемый массив
hash_arr("package name") = "Ansys" // создать пару
hash_arr("solver type") = "direct" // еще одну
hash_arr("package name") // val res7: String = Ansys
```

Проверить существование ключа можно с помощью метода `contains(key: T)`. И чтобы избавится от постоянных проверок на существование был добавлен метод `getOrElse(key: T1, defaultValue: T2)`

```
val m = Map[String,Int]("key1" -> 10, "key2" -> 20) // неизменяемый массив
m.contains("key1") // val res31: Boolean = true
m.getOrElse("key1", false) // val res33: AnyVal = 10 (ключ есть)
m.getOrElse("key10", false) // val res36: AnyVal = false, потому что такого ключа нет
```

Метод `getOrElse` в Scala похож на метод словарей `get` в Python. Они оба возвращают значение по ключу (если ключ есть в ассоциативном массиве) или значение по умолчанию (если ключ отсутствует)

```
# Python
d = dict([
    ('key1', 10),
    ('key2', 20)
])
d.get('key1', False) # 10
d.get('key10', False) # False
```

У ассоциативных массивов есть метод `get(key: T)`, который возвращает объект типа `Option`, который представлен типами `Some(Type)` и `None`. Так, если ключ не будет найден, то метод `get` вернет `None`, иначе `Some(value)`

```
val m = Map[String,String](
  "package name" -> "Ansys",
  "solver type" -> "iterative"
)
m.get("package name") // val res8: Option[String] = Some(iterative)
m.get("language") // val res9: Option[String] = None
```

Изменять значения можно только в изменяемых ассоциативных массивах

```
val m = scala.collection.mutable.Map[Char,Int](
  'a' -> 10, 'b' -> 20
) // scala.collection.mutable.Map[Char,Int] = HashMap(a -> 10, b -> 20)
m('a') // val res11: Int = 10
m += ('c' -> 31)  // добавить пару
```

Несколько методов для изменяемых массивов:

- `+=`: добавить одну или несколько пар к массиву,
- `-=`: удалить пару из массива.

Чтобы обойти ассоциативные массивы необходимо использовать цикл `for`

```
val m = scala.collection.mutable.HashMap[String,String]()
m += ("package name" -> "Nastran")
m += ("solver type" -> "iterative")
for ((k, v) <- m) println(f"${k} : ${v}")
// package name : Nastran
// solver type : iterative
```

По умолчанию внутреняя реализация массива представляет собой хеш-таблицу. Чтобы создать отсортированный массив необходимо создать его на основе сбалансированного дерева с помощью класса `SortedMap`.

3.10. Кортежи

Пара – это простейший случай кортежа. Кортеж (Tuple) создается с помощью круглых скобок, а его типом будет `TupleN[Type1, Type2 ... TypeN]`

```
Tuple3(1, 'e', true) // val res10: (Int, Char, Boolean) = (1,e,true)
def foo(a: Tuple4[Int, Int, Char, Boolean]): Unit = { println("Yes") }
foo(Tuple4(1, 10, 'r', true)) // Yes
foo((2, 15, 'w', false)) // Yes
```

Обратиться к элементам кортежа можно с помощью методов `_n`, где `n` – номер элемента в кортеже *начиная с 1*

```
val t = (1, true, 'F', "Fortran") // val t: (Int, Boolean, Char, String) = (1,true,F,Fortran)
t._4 // val res20: String = Fortran
```

Также удобно использовать сопоставление с образцом

```
val (id, isMan, class, name) = t
// val id: Int = 1
// val isMan: Boolean = true
// val mark: Char = F
// val name: String = Fortran
```

В Python можно не использовать скобки при распаковке

```
t = (1, True, 'F', 'Fortran')
(x, y, z, d) = t # можно просто x, y, z, d = t
for var in list('xyzd'):
    obj = eval(var)
    print(type(obj), obj)
# <class 'int'> 1
# <class 'bool'> True
# <class 'str'> F
# <class 'str'> Fortran
```

У индексного массива есть метод `zip(another: Array[T])`, который возвращает массив пар

```
val a = Array(1 to 6: _*)
val b = = Array("python": _*)
a.zip(b) // val res33: Array[(Int, Char)] = Array((1,p), (2,y), (3,t), (4,h), (5,o), (6,n))
```

4. Классы

Классы в Scala объявляются подобно классам в других языках, с помощью ключевого слова `class`. Свойства и методы объявляются внутри класса также как и переменные функции

```
class Person {
  val age = 10
  def say(phrase: String): Unit = {
    print(phrase)
  }

  def sayHi(): Unit = {
    this.say("Hi\n")
  }
}
val john = new Person()
// даже если метод не принимает аргументов, его следует вызывать с пустыми скобками, т.е. как
    sayHi()
john.sayHi() // Hi
```

Для создания экземпляра класса используется ключевое слово `new`. По умолчанию все свойства и методы являются *публичными*, а ключевого слова `public` нет в принципе. Все поля должны быть инициализированы. Обратиться к свойству или методу можно через точку, а к своим методам изнутри как с использованием `this`, так и напрямую как к функции или переменной.

Если хочется вызывать метод, который не принимает аргументов, без явного указания пустых скобок, то их не следует указывать при определении метода внутри класса

```
...
  def sayHi: Unit = { // нет скобок у метода
    this.say("Hi\n")
  }
```

4.1. Методы доступа

Свойства и методы можно сделать недоступными из вне. Для этого их необходимо сделать приватными с помощью ключевого слова `private` или защищенными (доступными в наследниках класса) `protected`. При этом можно оставить доступ через методы доступа. При объявлении свойств публичными Scala неявно создает два метода `valName` – getter и `valName_=(value: Type)` – setter. В этом можно убедиться, если скомпилировать класс через scalac и посмотреть байт код через `javap -private`.

Когда поле объявляется как `private`, то поле будет доступно только внутри класса. Именно класса, то есть экземпляры одного класса будут иметь доступ к приватным методам и свойствам друг друга. Чтобы ограничить видимость *внутри экземпляра*, следует объявить метод или свойство как `private[this]`

```
class Person {
  private[this] var name: String = "John"
  private val age = scala.util.Random.nextGaussian()

  def name_=(value: String) = name = value
  def age_?(person: Person) = person.age
}
```

4.2. Конструкторы

5. Вызов функций и методов

Математические функции определены в пакете `scala.math`. Их можно импортировать инструкцией

```
import scala.math._ // импорт всех элементов пакета
```

Здесь символ «_» – «групповой» символ, аналог «*» в Python: `from math import *`.

Замечание

При использовании пакета, имя которого начинается с префикса `scala.`, этот префикс можно опустить. Например, инструкция `import math._` эквивалентна инструкции `import scala.math._`, а вызов `math.sqrt(2)` эквивалентен вызову `scala.math.sqrt(2)`

---

6. Метод apply

В языке принято использовать синтаксис, напоминающий вызовы функций. Например, если `s` – это строка, тогда выражение `s(i)` вернет i-ый символ строки

```
"Fortran"(4)  // вернет ‘r‘ как 4-ый символ строки
// тоже самое с использованием метода ‘apply‘ "Fortran".apply(4)
```

Функции часто передаются методам в очень компактной форме записи. Например, чтобы вернуть количество символов верхнего регистра в строке можно воспользоваться конструкцией

```
// --- Scala ---
val s: String = "PythonTheBestLanguage"
s.count(_.isUpper)  // 4
```

На Python эту задачу можно решить так

```
# --- Python ---
# с помощью генератных выражений и генераторов списков
In[]: %timeit -n10 len(list(char for char in s if char.isupper()))
Out[]: 7.62 milis +/- 383 ns per loop (mean +/- std. dev. of 7 runs, 10 loops each)
In []: %timeit -n10 len([char for char in s if char.isupper()])
Out[]: 5.04 milis +/- 406 ns per loop (mean +/- std. dev. of 7 runs, 10 loops each)
# с помощью теоретико-множественных операций
In[]: from string import ascii_uppercase
In []: %timeit -n10 len(set(s).intersection(set(ascii_uppercase)))
Out[]: 7.5 milis +/- 812 ns per loop (mean +/- std. dev. of 7 runs, 10 loops each)
```

7. Управляющие структуры и функции

В Java или C++ мы различаем *выражения*, такие как `3 + 4`, и *инструкции*, например `if`. Выражение имеет значение; инструкция выполняет действие. В Scala практически все конструкции имеют значения, то есть являются *выражениями*. Это позволяет писать более короткие и более удобочитаемые программы.

7.1. Условные выражения

В Scala `if/else` возвращает значение, а именно значение выражения, следующего за `if` или `else`. Например,

```
val x: Int = 10
val n = if (x > 0) 1 else -1  // тернарное выражение
```

В Python это выглядело бы так

```
x = 10
n = 1 if (x > 0) else -1
```

В Scala *каждое выражение имеет тип*. Например, выражение `if (x > 0) 1 else -1` имеет тип `Int`, потому что обе ветви имеют тип `Int`. Типом выражения, способного возвращать значения разных типов, такого как `if (x > 0) "positive"else -1`, является супертип для обеих ветвей. В данном примере одна ветвь имеет тип `java.lang.String`, а другая – тип `Int`. Их общий супертип называется `AnyVal`.

Может получиться так, что инструкция `if` не будет иметь значения. Например, в случае, когда `if (x > 0) 1`, а `x` отрицательный. Однако в Scala каждое выражение предполагает наличие какого-либо значения. Эта проблема была решена введением класса `Unit`, единственное значение которого записывается как `()`[1]. Инструкция `if` без ветви `else` эквивалентна инструкции

```
if (x > 0) 1 else ()
```

[1]Эту комбинацию можно воспринимать как пустое значение и считать тип `Unit` аналогом типа `void` в Java или C++. Но, строго говоря, `void` означает отсутствие значения, тогда как `Unit` имеет единственное значение, означающее «нет значения»

Многострочное выражение в интерактивной оболочке можно заключить в фигурные скобки

```
{
  if (x > 0) 1
  else
    if (x == 0) 0
    else -1
}
```

Если потребуется перенести длинную строку на другую строку, первая строка должна оканчиваться символом, который не может интерпретироваться как конец инструкции. Для этого подойдет любой оператор

```
s = s0 + (v - v0)*t + // оператор + сообщает парсеру, что это не конец
0.5*(a - a0)*t*t
```

На практике длинные строки можно обрамлять фигурными скобками

```
if (n > 0) {  // открывающая скобка в конце строки явно свидетельствует,
              // что инструкция будет продолжена на следующих строках
  r = r*n
  n -=1 // ‘-=1‘ следует писать слитно
}
```

В языке Scala блок `{}` содержит последовательность *выражений* и сам считается выражением, результатом которого является результат последнего выражения.

Это может пригодиться для инициализации значений `val`, когда требуется выполнить более одного действия. Например

```
import scala.math
val x: Int = 10
val x0: Int = 1
val y: Int = 24
val y0:Int = 50
val distance = { val dx = x - x0; val dy = y - y0; scala.math.sqrt(dx*dx + dy*dy) }
// вернет 27.51363298439521
```

Поскольку инструкции присвоения возвращают значение `Unit`, их нельзя объединять в цепочки

```
// -- Scala
x = y = 1 // Неправильно!
```

В Python можно

```
# здесь просто x и y ссылаются на один и тот же объект целочисленного типа данных со значением 1
x = y = 1
```

8. Ввод и вывод

Чтобы вывести значение, используйте функцию `print` или `println`. Последняя добавляет символ перевода строки в конце. Имеется также функция `printf`, принимающая строку описания формата в стиле языка C

```
printf("CAE-package %s has %d cores", "Nastran", 32)
```

Прочитать строку, введенную в консоли с клавиатуры, можно с помощью функции `readLine`. Чтобы прочитать число, логическое или символьное значение, используйте `readInt`, `readDouble`, `readByte`, `readShort`, `readLong`, `readFloat`, `readBoolean` или `readChar`.

Метод `readLine`, в отличие от других, принимает строку приглашения к вводу

```
import scala.io.StdIn.readLine
import scala.io.StdIn.readInt
val name = readLine("Your name: ") // ввод имени
print("Your age: ")
val age = readInt()  // ввод возраста
printf("Hello, %s! Next year, you will be %d.", name, age+1)
// Hello, Leor! Next year, you will be 33.
```

9. Циклы

В Scala отсутствует прямой аналог цикла `for (инициализация;проверка;обновление)`. Если такой цикл потребуется, у вас есть два варианта на выбор – использовать цикл `while` или инструкцию `for`, как показано ниже

```
val n:Int = 10 // константа
var r: Int = 1 // переменная со значением по умолчанию
for (i <- 1 to n)
    r = r*i
printf("Result: %d", r) // Result: 3628800
```

В Python эта задача решалась бы так

```
n = 10
r = 1
for i in range(1,n+1):
    r *= i
print(f'Result: {r}')  # Result: 3628800
```

Вызов `1 to n` вернет объект `Range`, представляющий числа в диапазоне от 1 до `n` (включительно).

Конструкция

```
for (i <- expr)
```

обеспечивает последовательное присваивание переменной `i` всех значений выражения `expr` справа от `<-`. Порядок присвоения зависит от типа выражения. Для коллекций, таких как `Range`, присвоит переменной `i` каждое значение по очереди.

Перед именем переменной в цикле `for` не требуется указывать `val` или `var`. Тип переменной соотвествует типу элементов коллекции. Область видимости переменной цикла ограничивается телом цикла.

Для обхода элементов строки или массива зачастую нужно определить диапазон от 0 до $n-1$. В этом случае используйте метод `until` вместо `to`. Он возвращает диапазон, не включающий верхнюю границу. Например

```
val s: String = "Hello"
var sum: Int = 0
for (i <- 0 until s.length)
  sum += s(i)
```

В действительности в данном примере нет необходимости использовать индексы. Цикл можно выполнить непосредственно по символам

```
var sum: Int = 0
for (ch <- "Hello")
  sum += ch
```

Как и в Python. В Python тоже можно перебирать элементы последовательности прямо в цикле без индексов. В Scala циклы используются те так часто, как в других языках. Значения в последовательностях зачастую можно обрабатывать, применяя функцию сразу ко всем элементам, для чего достаточно произвести единственный вызов метода.

В Scala нет инструкций `break` или `continue` для преждевременного завершения цикла. Но как же быть, если это потребуется? Есть несколько вариантов:

1. Использовать логическую переменную управления циклом,
2. Используйте вложенные функции – при необходимости можно выполнить инструкцию `return` в середине функции,
3. Используйте метод `break` объекта `Breaks`

```
// здесь передача управления за пределы цикла выполняется путем возбуждения и перехвата и
   склюючения, поэтому избегайте пользоваться этим механизмом, когда скорость выполнения кр
   итична
import scala.util.control.Breaks._
breakable {
  for (...) {
    if (...) break; // выход из прерываемого блока
      ...
  }
}
```

9.1. Расширенные циклы for и for-генераторы

В предыдущем разделе была представлена базовая форма цикла `for`. Однако эта конструкция намного богаче, чем в Java или C++. В заголовке цикла `for` допускается указывать несколько *генераторов* в форме *переменная* <- *выражение*, разделяя их «;»

```
// --- Scala
// несколько генераторов
for (i <- 1 to 3; j <- 1 to 5)
  print(f"${10*i + j} ")
```

```
# --- Python
# вложенные циклы
for i in range(1, 3+1):
    for j in range(1, 5+1):
        print(f'{10*i+j} ')
```

Что касается форматирования, то можно использовать конструкции с f-строками. Пример

```
import scala.io.StdIn.readLine
import scala.io.StdIn.readInt

val package_name = readLine("Enter package's name: ")
print("Enter number of cores: ")
val n_cores = readInt()
print(f"CAE-package: ${package_name}, number of cores: ${n_cores}") // f-строка
```

То есть, чтобы подставить в основную строку другую строку или какой-либо другой объект, используется конструкция `${}`.

Управлять форматом вывода чисел можно так

```
val n: Double = 0.345345345345
print(f"This is a number: ${n}%.3f") // This is a number: 0,345
```

Каждый генератор может иметь *ограничитель* – логическое условие с предшествующим ему ключевым словом `if`

```
// --- Scala
for (i <- 1 to 3; j <- 1 to 3 if i != j)
  print(f"${10*i + j} ")
```

```
# --- Python
for i in range(1,3+1):
    for j in range(1,3+1):
        if i != j:
            print(f'{10*i + j} ', end=' ')
```

Допускается любое количество *определений*, вводящих переменные для использования внутри цикла

```
// --- Scala
for (i <- 1 to 3; from = 4 - i; j <- from to 3)
  print(f"${10*i + j} ")
  // 13 22 23 31 32 33
```

На Python эта задача могла бы быть решена так

```
for i in range(1,3+1):
    frm = 4 - i
  for j in range(frm,3+1):
      print(f'{10*i + j} ')
# 13 22 23 31 32 33
```

Когда тело цикла начинается с инструкции `yield`, цикл будет конструировать коллекцию, добавляя в нее по одному элементу в каждой итерации

```
for (i <- 1 to 10)
  yield i % 3
// IndexedSeq[Int] = Vector(1, 2, 0, 1, 2, 0, 1, 2, 0, 1)
```

Такого рода циклы называют `for`-генераторами (for-comprehension)

```
// в Scala оператор yield может существовать вне контекста функции
// в Python yield имеет смысл только в контексте функции
for (c <- "Hello"; i <- 0 to 1)
  yield (c + i).toChar
// IndexedSeq[Char] = Vector(H, e, l, l, o, I, f, m, m, p)
```

При желании генераторы, ограничители и определения цикла `for` можно заключить в фигурные скобки и вместо точек с запятой использовать переводы строки

```
for { i <- 1 to 3
      from <- 4 - i
      j <- from to 3 }
  print(f"${10*i + j} ")
```

10. Функции

В дополнение к методам в Scala имеются функции. Метод оперирует объектом, а функции – нет. Чтобы определить функцию, нужно указать имя функции, параметры и тело, как показано ниже

```
def abs(x: Double) = if (x >= 0) x else -x
// или
// def abs(x: Double) = {if (x >= 0) x else -x}
```

Требуется определить типы всех параметров. Однако, если функция не рекурсивная, определять тип возвращаемого значения не требуется. Компилятор Scala определяет тип возвращаемого значения по типу выражения справа от символа `=`.

В Python эту задачу можно было бы решить так

```
def _abs(x: float) -> float:
    return x if (x >= 0) else -x
```

или с помощью анонимных функций

```
_abs = lambda x: x if (x >= 0) else -x
_abs(10) # 10
_abs(-20) # 20
```

Если тело функции содержит более одного выражения, используйте блок. *Последнее* выражение в блоке определяет значение, возвращаемое функцией. Например, следующая функция вернет значение `r` после цикла `for`

```
// --- Scala
def fact(n: Int) = {
  var r: Double = 1
  for (i <- 1 to n)
    r *= i
  r  // возвращаем значение r
}
fact(100) // 9.33262154439441E157
```

```
# --- Python
def fact(n: int) -> int:
    r = 1
    for i in range(1,n+1):
        r *= i
    return r
fact(100)
```

В данном случае нет необходимости явно использовать ключевое слово `return`. В Scala допускается использовать `return` для *немедленного* выхода из функции, как в Java или C++, но такой способ редко используется.

Замечание

Даже при том, что нет ничего особенного в том, чтобы использовать `return` в именованных функциях, лучше все-таки стараться привыкать жить без `return`. В анонимных функциях `return` не возвращает значение вызывающей программе, а выполняет выход во вмещающую, именованную функцию. Инструкцию `return` можно интерпретировать как своеобразную инструкцию `break` для функций и использовать ее только для преждевременного прерывания выполнения функций

В *рекурсивных* функциях *тип* возвращаемого значения должен указываться *обязательно*. Например

```
def fact(n: Int): Int = {
  // не нужен return!!!
  if (n <= 0) 1 else n*fact(n - 1)
}
```

Без определения типа возвращаемого значения компилятор Scala не сможет убедиться, что выражение `n*fact(n - 1)` имеет тип `Int`.

На Python эта задача решалась бы так

```
def fact(n: int) -> int:
```

```
    # оператор return нужен обязательно!
    return 1 if (n <= 0) else n*fact(n - 1)
```

К слову, в Scala нет ветки elif как в Python, поэтому приходится использовать вложенные условия. Например для функции sign

$$\text{sign}(x) = \begin{cases} 1, & x > 0, \\ 0, & x = 0, \\ -1, & x < 0 \end{cases}$$

```
def sign(x: Float) = {
  if (x > 0) 1 else {
    if (x == 0) 0 else -1
  }
}
```

10.1. Аргументы по умолчанию и именованные аргументы

Существует возможность определять аргументы по умолчанию для функций, чтобы при вызове можно было не указывать их значения явно. Например:

```
// в начале указвыаются позиционные аргументы, потом именованные
def decorate(str: String, left: String = "[", right: String = "]") = {
  // return не нужен!!!
  left + str + right
}
```

На Python

```
def decorate(strr: str, left: str = '[', right: str = ']') -> str:
    return '{}{}{}'.format(left, strr, right)
    # или так
    # return f'{left}{strr}{right}'
```

Если при вызове число аргументов оказывается меньше числа параметров, применяются аргументы по умолчанию, начиная с конца. При передаче аргументов можно также указывать имена параметров. Именованные аргументы необязательно должны следовать в том же порядке, что и параметры. Например

```
decorate(left="<<<", str="Hello", right=">>>")  // "<<<Hello>>>"
```

10.2. Переменное количество аргументов

Иногда бывает удобно реализовать функцию, способную принимать переменное чило аргументов. Следующий пример демонстрирует синтаксис объявления таких функций

```
// --- Scala
def sum(args: Int*) = {
  var result = 0
  for (arg <- args)
    result += arg
  result
}
val s = sum(1, 4, 9, 16, 25)
```

Функция принимает единственный параметр типа `Seq`. Такой функции нельзя передавать уже имеющуюся последовательность значений. Следующий вызов оформлен неверно `val s = sum(1 to 5) // Ошибка!!!`. Если вызывать функцию с одним аргументом, это должно быть единственное целое число, а не диапазон целых чисел. Чтобы исправить ошибку, необходимо сообщить компилятору, что параметр должен интерпретироваться как последовательность аргументов. Добавьте в конец «`: _*`»

```
val s = sum(1 to 5: _*) // интепретировать 1 to 5 как последовательность аргументов
```

```
# --- Python
def summ(*args) -> int:
    result = 0
    for arg in args:
        result += arg
    return result
# позиционные аргументы собираются в кортеж
s = summ(1, 4, 9, 16, 25)
lst = [1, 4, 9, 16, 25]
s = summ(*lst)  # нужно распаковать список перед передачей
# можно воспользоваться генераторными выражениями
s = summ(*list(i**2 for i in range(1,5+1)))
```

Без этого синтаксиса не обойтись при определении рекурсивной функции с переменным числом аргументов

```
def recursiveSum(args: Int*): Int = {
  if (args.head == 0) 0
  else args.head + recursiveSum(args.tail: _*) // функция recursiveSum на каждой итерации вызыва
    ется с усеченной последовательностью args
}
```

Здесь `head` – начальный элемент последовательности, а `tail` – *все остальные* элементы последовательности. Это также объект `Seq`, поэтому необходимо использовать «`: _*`» для преобразования его в последовательность аргументов.

На Python

```
# функция будет вызваться рекурсивно до тех пор, пока не выполнится
# условие останова, то есть пока длина последовательности не станет равной 0
def recursiveSum(*args) -> int:
    print(f'type of args: {args}({type(args)})')
    # переданная последовательность чисел собирается в кортеж args
    if len(args) == 0:  # условие останова
        return 0
    else:
        # кортеж args перед передачей функции следует распаковать
        return args[0] + recursiveSum(*args[1:])
# ----------------------------------------
# type of args: (1, 5, 10)(<class 'tuple'>)
# type of args: (5, 10)(<class 'tuple'>)
# type of args: (10,)(<class 'tuple'>)
# type of args: ()(<class 'tuple'>)
```

11. Процедуры

Раньше в Scala была специальная форма записи для объявления функций, не возвращающих значений. Если тело функции заключено в фигурные скобки *без предшествующего символа* =,

тогда возвращаемое значение будет иметь тип `Unit`. Такие функции называются *процедурами*. Процедура не возвращает значение и вызывается исключительно ради побочного эффекта.

Замечание

Сейчас (версия 2.13.3) этот процедурный синтаксис отменен и для объявления процедуры требуется *явно* указывать тип возвращаемого объекта (`Unit`)

Например, следующая процедура выводит строку в рамке

```
def box(s: String): Unit = { // явно указывается Unit = {...
  val border = f"${"-"*s.length}--\n"
  println(f"${border}|${s}|\n${border}")
}
box("test")
//-------
//|Scala|
//-------
```

На Python

```
from typing import NoReturn

def box(s: str) -> NoReturn:
    border = f'{"-"*len(s)}--\n'
    print(f'{border}|{s}|\n{border}')
#--------
#|Python|
#--------
```

12. Ленивые значения

Когда значение `val` объявляется как `lazy` (ленивое), его инициализация откладывается до первого обращения к нему. Например

```
// читает файл в строку
lazy val words = scala.io.Source.fromFile("./nonexistent_file_name.txt").mkString
```

Если программа никогда не обратиться к значению `words`, файл никогда не будет открыт. Ленивые значения удобно использовать, чтобы отложить дорогостоящие операции инициализации. В Python механизм отложенных вычислений можно реализовать с помощью карированных функций, генераторных функций, сопрограмм и т.д.

Ленивые значения можно считать чем-то средним между `val` и `def`

```
// вычисляется немедленно, в момент определения words
val words = scala.io.Source.fromFile("./file_name.txt").mkString
// вычисляется при первом обращении к words
lazy val words = scala.io.Source.fromFile("./file_name.txt").mkString
// вычисляется всякий раз, когда происходит обращение к words
def words = scala.io.Source.fromFile("./file_name.txt").mkString
```

Прочитать файл построчно можно так

```
import scala.io.Source.fromFile

lazy val fo = fromFile("./test_args.py").mkString // или просто fromFile("...")
for (line <- fo)
  print(line)
```

На Python

```
with open('test_args.py', 'r') as tf:
    for line in tf:
        print(line.rstrip())
```

Замечание

Откладывание вычислений не дается бесплатно. Всякий раз, когда выполняется обращение к ленивому значению, вызывается метод, который проверяет, потокобезопасным способом, было ли инициализированно значение

---

13. Исключения

В языке Scala, в отличие от Java отсутствуют «контролируемые» исключения – вам никогда не придется объявлять, что некоторый метод или функция может возбуждать исключение. Разработчики Scala решили отказаться от контролируемых исключений, признавая, что полная проверка на этапе компиляции не всегда оправдана.

Выражение `throw` имеет специальный тип `Nothing`, что может пригодиться в выражениях `if/else`. Если одна из ветвей имеет тип `Nothing`, типом всего выражения `if/else` становится тип другой ветки. Например

```
import scala.math._
if (x >= 0) { sqrt(x) } else throw new IllegalArgumentException("x should not be negative")
```

Первая ветвь имеет тип `Double`, вторая – тип `Nothing`. Таким образом, все выражение `if/else` имеет тип `Double`.

В Python это можно решить так

```
import math
def f(x):
    if x >= 0:
        return math.sqrt(x)
    else:
        assert x >= 0, 'x should not be negative'
```

Конструкции `try/catch` и `try/finally` имеют разные цели. Инструкция `try/catch` обрабатывает исключения, а инструкция `try/finally` предпринимает некоторые действия (обычно – освобождение ресурсов), если исключение не будет обработанно. Их можно объединить в одну инструкцию `try/catch/finally`

```
try { ... } catch { ... } finally { ... }
```

Замечание

Если возникает необходимость использовать в качестве имени переменной зарезервированное слово, то его необходимо обернуть обратными кавычками, например, `var ‘do‘: Int = 10`

Список литературы

1. *Хостаманн К.* Scala для нетерпеливых. – М.: ДМК Пресс, 2013. – 408 с.