

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	1
2	Установка библиотек для Scala	1
3	Базовый sbt-файл	2
4	Компиляция программ на Scala	3
5	Сборка jar-файлов в Spark-проектах	3
6	Работа с языком Scala в JupyterLab	4
7	Работа со Scala через Ammonite	5
8	Приемы использования библиотеки Breeze	5
9	Использование ключевых <code>private</code> и <code>protected</code>	7
	Список литературы	9

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. Установка библиотек для Scala

Например, библиотеку `breeze` <https://github.com/scalanlp/breeze/wiki/Installation>, близкую по своему функционалу к библиотеке `numpy` языка Python, можно установить с помощью `sbt` следующим образом

build.sbt

```
name := "project_name"
version := "0.1"
scalaVersion := "2.13.3"

libraryDependencies += Seq(
  // Last stable release
  "org.scalanlp" %% "breeze" % "1.1",
  // Native libraries are not included by default. add this if you want them
  // Native libraries greatly improve performance, but increase jar sizes.
  // It also packages various blas implementations, which have licenses that may or may not
  // be compatible with the Apache License. No GPL code, as best I know.
  "org.scalanlp" %% "breeze-natives" % "1.1",
  // The visualization library is distributed separately as well.
  // It depends on LGPL code
  "org.scalanlp" %% "breeze-viz" % "1.1"
)
```

Если используется IDE IntelliJ IDEA, то файл `build.sbt` должен располагаться в директории проекта, например, `C:\Users\ADM\IdeaProjects\project_name`. Тогда после запуска сессии IDEA будут доступны все библиотеки.

Теперь можно запустить сессию в директории с файлом `build.sbt` командной

```
$ sbt console
scala> import breeze.linalg._
scala> val v = DenseVector(1.0, 2.0, 3.0)
```

К слову, есть полезная шпаргалка по breeze <https://github.com/scalanlp/breeze/wiki/Linear-Algebra-Cheat-Sheet>

3. Базовый sbt-файл

Зависимости проекта описываются в файле `build.sbt`

build.sbt

```
scalaVersion := "2.13.3"

libraryDependencies += Seq(
  "org.scalanlp" %% "breeze" % "1.1",
  "org.scalanlp" %% "breeze-natives" % "1.1",
  "org.scalanlp" %% "breeze-viz" % "1.1",
  "org.plotly-scala" %% "plotly-render" % "0.8.0",
  // "org.apache.spark" %% "spark-core" % "2.2.3" % "provided",
  // "org.apache.spark" %% "spark-sql" % "2.2.3" % "provided",
  // "org.vegas-viz" %% "vegas" % "0.3.11"
)
```

При работе со Scala-проектом с помощью `sbt` или IntelliJ IDEA версия языка определяется параметром `scalaVersion` в файле сборки `build.sbt`, например

```
scalaVersion := "2.12.12"
...
```

Остается только при запуске сессии в REPL набрать `sbt console` (а не `scala`), чтобы загрузить указанную версию Scala и все зависимости проекта.

То есть для того чтобы использовать разные версии Scala в разных проектах нужно указать нужную версию Scala в файле сборки `build.sbt`.

К слову, узнать версию Scala в сессии можно так

```
scala.util.Properties.versionNumberString // String = 2.12.12
scala.util.Properties.versionMsg // String = Scala library version 2.12.12 -- Copyright
2002-2020, LAMP/EPFL and Lightbend, Inc.
```

На MacOS для того чтобы выяснить доступные версии Scala можно воспользоваться менеджером пакета `brew`

```
brew search scala
```

4. 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'`

5. Сборка jar-файлов в Spark-проектах

Создадим простое Spark-приложение

SimpleApp.scala

```
import org.apache.spark.sql.SparkSession

object SimpleApp {
  def main(args: Array[String]) = {
    val logFile = "./req.txt"
    // создаем Spark-сессию
    val spark = SparkSession.builder.appName("Simple Application").getOrCreate()
    val logData = spark.read.textFile(logFile).cache()
    val numsA = logData.filter(line => line.contains("a")).count()
    val numsB = logData.filter(line => line.contains("b")).count()
    println(s"--> Lines with a: $numsA, Lines with b: $numsB")
    spark.stop() // <-- NB
  }
}
```

В файл сборки `build.sbt` следует добавить следующие строки

Пример файла `build.sbt`

```
name := "SparkML"
```

```

version := "1.0"

scalaVersion := "2.12.12"

libraryDependencies += Seq(
  "org.apache.spark" %% "spark-sql" % "3.0.1" % "provided",
  "org.apache.spark" % "spark-mllib_2.12" % "3.0.1" % "provided" // в строке используется один
  "%"!!!
)

```

Для того чтобы sbt работал корректно, требуется разместить SimpleApp.scala и build.sbt следующим образом:

- о файл build.sbt должен лежать в корне проекта,
- о а scala-скрипт – по пути src/main/scala/SimpleApp.scala.

Теперь можно упаковать приложение

```
sbt package
```

Для запуска scala-приложения используется spark-submit

```

spark-submit \
--class "SimpleApp" \
--master local \
target/scala-2.12/simple-project_2.12-1.0.jar

```

Для запуска python-сценария следует набрать

```

spark-submit \
--master local \
SimpleApp.py

```

6. Работа с языком Scala в JupyterLab

Для того, чтобы JupyterLab поддерживал код на Scala требуется установить ядро spylon-kernel

```

# Step 1: Install spylon kernel
pip install spylon-kernel

# Step 2: create a kernel spec
python -m spylon_kernel install

# Step 3: start jupyter notebook
jupyter notebook

```

Посмотреть установленные ядра можно так

```
jupyter kernelspec list
```

В некоторых случаях удобнее работать с almondd <https://almond.sh/docs/try-docker> – это Scala-ядро для Jupyter. Проще всего воспользоваться docker-образом

```
docker run -it --rm -p 8888:8888 almonddsh/almond:latest
```

Можно указать конкретную версию almondd или Scala

```

docker run -it --rm -p 8888:8888 almonddsh/almond:0.10.9
docker run -it --rm -p 8888:8888 almonddsh/almond:0.10.9-scala-2.12.8

```

Затем нужно будет открыть в браузере вкладку с адресом, который будет указан в логах. Для примера начнем работу с библиотекой plotly <https://github.com/alexarchambault/plotly-scala>

в сеансе Jupyter

```
import $ivy.`org.plotly-scala::plotly-almond:0.8.0` // <-- NB: динамическое подключение библиотеки
import plotly._
import plotly.element._
import plotly.layout._
import plotly.Almond._

val (x, y) = Seq(
  "Banana" -> 10,
  "Apple" -> 8,
  "Grapefruit" -> 5
).unzip

Bar(x, y).plot()
```

Узнать домашнюю директорию Spark можно так

```
echo `sc.getConf.get("spark.home")` | spark-shell
```

Для поддержки Spark в almond необходимо добавить следующие строки (ВАЖНО: ядро и кластер Spark должны использовать одну и ту же версию Scala) <https://github.com/almond-sh/almond/blob/master/docs/pages/usage-spark.md>

в сеансе Jupyter. Для поддержки Spark

```
import $ivy.`org.apache.spark::spark-sql:2.4.0` // Or use any other 2.x version here
import $ivy.`sh.almond::almond-spark:@VERSION@` // Not required since almond 0.7.0 (will be automatically added when importing spark)
```

В самом простом случае можно воспользоваться web-интерфейсом binder <https://mybinder.org/>, доступного на странице проекта almond (по состоянию на 02.12.20 Spark совместим только с версией Scala 2.11).

Подключить breeze в сеансе JupyterLab на базе binder можно такой конструкцией

```
import $ivy.`org.scalanlp::breeze:1.0`

import breeze.linalg.{DenseMatrix, DenseVector}
import breeze.stats.regression.{leastSquares, lasso}
...
```

7. Работа со Scala через Ammonite

Ammonite <https://ammonite.io/> на высоком уровне абстракции (очень грубо) представляет собой командную REPL-оболочку для Scala (на самом деле возможности гораздо шире).

8. Приемы использования библиотеки Breeze

Быстрое введение в библиотеку Breeze можно найти здесь <https://github.com/scalanlp/breeze/wiki/Quickstart>, а шпаргалку по работе с инструментами линейной алгебры по адресу <https://github.com/scalanlp/breeze/wiki/Linear-Algebra-Cheat-Sheet>.

Чтобы создать полносвязанный вектор или матрицу можно воспользоваться следующими приемами

```
import breeze.linalg._

DenseVector.ones[Double](5)
// np.ones(5) <-- numpy

DenseVector.fill(3){5} // или просто DenseVector.fill(3)(5)
// np.ones(3)*5 <-- numpy

DenseVector.fill(3){scala.math.sin(10)}
// np.ones(3)*np.sin(10)

DenseMatrix.ones[Double](3,2)
// np.ones((3,2)) <-- numpy

DenseMatrix((1.0, 2.0), (3.0, 4.0))
// np.array([[1.0, 2.0], [3.0, 4.0]]) <-- numpy
```

Для диапазона

```
linspace(1,5,10)
// np.linspace(1,5,10 <-- numpy)
```

Транспонирование векторов и матриц

```
DenseVector(1.to(5): _*).t
// np.array(range(1,6)).reshape(-1, 1)

DenseMatrix((10, 20, 30), (40, 50, 60)).t
// np.array([[10, 20, 30], [40, 50, 60]]).T
```

Можно использовать приемы генерации значений таблицы на лету

```
DenseMatrix.tabulate(3, 2){ case (i, j) => i + j}
```

На ванильном Python генерацию на лету можно было бы реализовать так

```
def tabulate(n: int, m: int) -> np.array:
    arr = []
    for i in range(n):
        row = []
        for j in range(m):
            row.append(i + j)
        arr.append(row)
    return np.array(arr)
```

Создать матрицу на базе массива можно так

```
val mtx = new DenseMatrix(2, 3, Array[Int](10, 20, 30, 40, 50, 60)) // обязательно new!
```

Для создания векторов и матриц, заполненных равномерно распределенными псевдослучайными числами используется метод `rand`

```
DenseVector.rand(3)
// np.random.rand(3)
// или
DenseMatrix.rand(3, 2)
// np.random.rand(3, 2)
```

Чтение и запись векторов и матриц

```

val mtx = DenseMatrix.rand(3, 2)
mtx(1, 1) // 1-ая строка и 1-ый столбец

val v = DenseVector.rand(10)
v.slice(1,5) // или a(1 to 4) или a(1 until 5)
// v[1:5] <-- пипру; правая граница не включается!

v(5 to 0 by -1)
// v[5::-1]

v(1 to -1) // v[1:]

v(-1) // v[-1] последний элемент

v(:, 2) // v[:, 2]

```

Примеры других манипуляций

```

mtx.reshape(3, 2) // как и в пипру

// разворачивание матрицы в вектор
mtx.toDenseVector // mtx.flatten()

// копирование нижнего треугольника данных
lowerTriangular(mtx) // np.tril(mtx)

// копирование верхнего треугольника данных
upperTriangular(mtx) // np.triu(mtx)

// верхнеуровневое копирование
mtx.copy // np.copy(mtx)

// выбрать диагональные элементы матрицы
diag(mtx) // np.diagonal(mtx)

```

9. Использование ключевых private и protected

Если элемент класса объявлен с префиксом `private val/var`, то доступ к этому элементу можно будет получить *только в теле класса*, содержащего объявление этого закрытого элемента. Смысл в том, чтобы закрытый/приватный элемент не был доступен из вне, но в теле класса, содержащего объявление закрытого элемента, всегда можно получить доступ к элементу объявленному как `private val/var`, даже через экземпляр класса.

Если же элемент объявлен с префиксом `private[this] val/var`, то доступ к элементу возможен *только из-под текущего экземпляра класса*.

В самом простом случае, если поле объявлено как *публичное* (то есть без пометок `private` и пр.), то к этому полю можно получить доступ как из-под дочерних классов, так и из-под переменной, связанной с экземпляром класса (или из-под анонимного экземпляра класса, вида `(new MyClass).value`)

```

import scala.math.{Pi}

class Test {
  val a = 10 // общедоступное поле
  val b = Pi // общедоступное поле

```

```

def readValueA = a
def readValueB = b
}

val t = new Test()
t.a // val res0: Int = 10
t.b // val res1: Double = 3.141592653589793
t.readValueA // val res2: Int = 10
t.readValueB // val res3: Double = 3.141592653589793

```

Если поле объявлено как **private**, то будут созданы *приватные* методы доступа, а это значит, что получить доступ к такому полю через переменную, связанную с экземпляром класса или из-под дочернего класса (т.е. подкласса) не удастся (НО есть очень важный нюанс, о котором написано ниже).

```

import scala.math.{Pi}

class Test { // родительский класс
  private val a = 10
  val b = Pi

  def readValueA = a
  def readValueB = b
}

class DaughterTest extends Test { // дочерний класс
  //val readValueAfromSubClass = a // (-) нельзя получить доступ к "a" (private a)
  val readValueBfromSubClass = b
}

val d = new DaughterTest() // экземпляр-подкласс
d.readValueBfromSubClass // val res2: Double = 3.141592653589793

```

Замечание

Если поле публичное (без **private** и пр.), то создаются публичные методы доступа. Если поле приватное, то, соответственно, создаются приватные методы доступа. Причем в случае приватного поля получить доступ к нему из-под экземпляра класса или из-под дочернего класса не удастся

ОЧЕНЬ ВАЖНОЕ ЗАМЕЧАНИЕ: если элемент класса объявлен как закрытый (приватный), то доступ к нему действительно нельзя получить из-под экземпляра класса или из-под дочернего класса. Но если обращение к этому закрытому элементу класса выполняется из любой точки тела самого объявляемого класса (класса, который содержит объявление закрытого элемента), то доступ возможен даже из-под экземпляра этого объявляемого класса. Пример

```

class MyClassPrivate( // параметры первичного конструктора
  private val name: String, // будет создан приватный/закрытый метод чтения name
  private val age: Int // будет создан приватный/закрытый метод чтения age
) {
  def call =
    // обращение в области видимости текущего класса
    (new MyClassPrivate("Leor", 34)).name // находясь в теле класса, в котором поле name объявлялось закрытым, можно получить доступ к этому закрытому элементу даже через экземпляр класса
}

```



```
// внешняя область видимости
// обращение вне области видимости класса MyClassPrivate
(new MyClassPrivate("Alex", 35)).name // ERROR: value name in class MyClassPrivate cannot be
    accessed as a member of MyClassPrivate from class
(new MyClassPrivate("Alex", 35)).call // Leor

class MyClassPrivateThis( // параметры первичного конструктора
    name: String,          // тоже что и private[this] val name: String,
    age: Int               // тоже что и private[this] val age: Int
) {
    def call = (new MyClassPrivateThis("Leor", 34)).name // ОШИБКА!!! Т.к. в данном случае к name
        можно получить доступ только из-под текущего экземпляра класса, а мы пытаемся получить досту
        п из-под нового экземпляра класса
}
```

Однако, если поле объявить как **protected**, то доступ к этому полю можно будет получить из-под дочерних классов, но по-прежнему нельзя получить из-под экземпляра класса.

```
import scala.math.{Pi}

class Test { // родительский класс
    protected val a = 10 // <-- NB
    val b = Pi

    def readValueA = a
    def readValueB = b
}

class DaughterTest extends Test { // дочерний класс
    val readValueAfromSubClass = a // обращение к защищенному полю "a"
    val readValueBfromSubClass = b
}

val t = new Test()
//t.a // нельзя обратиться из-под экземпляра
t.b
//t.readValueA
t.readValueB

val d = new DaughterTest()
d.readValueAfromSubClass // val res4: Int = 10
d.readValueBfromSubClass // val res5: Double = 3.141592653589793
```

Список литературы

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