好玩的数据开发：使用Scala和Spark

（第2版）

目 录

前 言

Scala是基于JVM的语言，它混合了函数式编程和对象编程的特点。这语言本身挺好玩的，它的操作数据集合的方法（高阶函数）非常丰富，写起来灵活而又高效。其他语言包括Ruby, Python也有类似的方法，不过Scala基于函数式编程的特点，实现得更纯粹一些。

如果只是入门的话，Scala本身并不难，它甚至可以当作脚本一样运行。本书对Scala的介绍主要以数据操作为目的，并且假设读者有一定的编程功底。

众所周知，Spark是大数据处理领域的明星产品，它主要用Scala写的。Spark虽然也提供了其他语言的API，比如Java, Python, R，但使用原生的Scala API更为方便。Spark底层的RDD API，大多数是Scala的高阶函数的分布式版本，它们的使用方式甚至高度类似。

本书主要介绍Scala面向数据开发所需要的基本技能，以及联合Scala和Spark进行开发的基本技巧。Scala与Spark都是很专业的话题，每个话题认真讨论的话，都可以写一本几百页的书。很显然本书的目的只是帮助读者快速入门。如果要深入了解这两个领域，在书里我也推荐了其他资源，大家可以自行阅读。

我在2022年3月份写了本书第一版，发布后陆续收到一些读者的反馈。于是在5月份进行改进，写了第二版。第二版主要对排版格式进行调整，另外增加了一些Spark的使用案例。

Wes Peng
2022年5月

第一章 Scala介绍

第一节. Scala简介

Scala是在Java虚拟机上运行的语言，它的历史其实很早，早在2004年，初始版本就面世了。作者是洛桑联邦理工学院的Martin Odersky教授。

Scala混合了对象编程和函数式编程的特点，语法极其灵活。它是一门静态、强类型的语言，通过编译时检查，保证代码的安全性和一致性。

Scala与Java可以互相调用，这意味着它可以使用Java的丰富的类库。如果你要用Scala开发一个完整的应用系统，我认为首先要懂点Java。因为Scala原生的类库并不是很完善，很多场景下，需要用到Java的一些知识。

比如在大数据开发时，不可避免要访问Hadoop生态的东西，Hive、Hbase之类的，而这些类库都是Java的，Scala直接拿来用了。

不过不用担心，Scala自身的数据操作API极为强大。哪怕与Spark交互，也可以使用原生的API，不需要了解多少Java的东西。而且，只是操作大数据的话，甚至面向对象的编程接口都用不上，只需要了解下函数式编程就够了。

第二节. 函数式编程

函数式编程代表一个技术流派，它甚至发展成一门哲学，有自己的原则和信仰。面向对象的编程，与函数式编程，各自的风格看起来格格不入。

我们无需了解太多思想上的东西，仅仅知道函数式编程在Scala里的特点就够了。简言之，Scala的函数式编程具备两大特点：

1. 不可变的值
2. 纯函数

所谓不可变的值，是指Scala的默认变量类型就是不可变的，它用val来声明一个不可变的变量。这种方式保证了数据安全，在多个状态切换下，变量的值始终不变。

Scala采用“update by copy”的原则来更新变量值。如果要修改某个变量，就产生一份修改后的拷贝，并将该拷贝赋值给一个新的变量。原始变量雷打不动的不变。

纯函数其实也是一个函数，不过它具备如下特点：

1. 函数没有副作用
2. 函数的返回值由输入参数唯一决定

其一，没有副作用，是指函数不跟外部环境交互，不依赖于外部变量（比如全局变量），不改变外部环境的值。

每个人学习新的编程语言时，都会首先写个hello world。如下是scala的hello world函数实现：

```
scala> def hello() =
     |   println("hello world")
def hello(): Unit
```

这个函数就不是一个纯函数，因为它打印字符到终端，与外部环境发生了交互，改变了外部环境的值（终端产生了IO行为）。

同理，如果在函数里要读入一个外部文件、读数据库、读取网络socket，这些都参与了外部环境的交互，它们就都不是纯函数。

纯函数还不能访问外部变量，更不能改变任何外部变量的值。比如你在函数里调用全局的系统时间：

```
scala> time.LocalDateTime.now().toString
```

那这就不是一个纯函数了，因为它访问了函数外部的值。

其二，唯一决定纯函数返回值的，是输入参数。如果输入参数不变，那么返回值一定不变，变了就不是纯函数了。

比如，如下函数就不是纯函数：

```
scala> def randNum(s:Int) =
     |   Random.between(0,s)
def randNum(s: Int): Int
```

因为你每次调用它，哪怕输入相同的参数，它的返回值也变了。示例如下：

```
scala> randNum(3)
val res2: Int = 1

scala> randNum(3)
val res3: Int = 2
```

真正的纯函数是，只要输入值相同，哪怕你调用它千万次，输出值也是相同的。

上述不可变的值，以及纯函数，是scala函数式编程的核心原则，也是scala操作大数据的核心思想。它保证了数据在并发操作下的安全性，用户基于此可以编写高度可靠的程序。这大概也是Spark采用Scala开发的原因吧。

关于scala函数式编程，我就简单介绍到这里。如果你想深入了解，建议认真阅读如下书：

Functional Programming in Scala by Paul Chiusano and Runar Bjarnason

第三节. 安装Scala

Scala目前主流分支是1.3版本和3.1版本。虽然后者版本号更高，但是Spark主要是用前者开发的，所以我这里主要介绍1.3版本。

虽然在本地也可以安装Scala，比如Windows或者Mac系统都可以。不过作为新手练习，我建议在云主机上安装。这样你就可以随时清除旧环境，不会因为装了太多版本，而整的环境混乱。

Scala对于系统要求并不高，1核1GB内存的KVM虚拟机就足够了。不过考虑到后续要安装Spark，建议还是2核2GB内存的KVM虚拟机。很多平台比如Linode, Digitalocean, Vultr都提供这种开箱即用的虚拟机，按秒计费，不用了就把它取消掉就好。

我在一个云主机提供商那里开启了一个KVM虚拟机，它包括2个AMD核心、1GB内存、35GB高速磁盘，作为练手用足够了。

拿到虚拟机后，先安装操作系统。操作系统当然是Linux。有些大数据软件，比如Apache Drill，只能跑在Linux上。所以熟悉Linux是数据开发人员的必备技能。

Linux系统可以选择Ubuntu系统或者Centos系统，因为这两者比较主流。我个人熟悉前者，就安装了Ubuntu 20.0 x64系统。系统选择64位的，都大数据时代了，就不要用32位了。

系统安装好后，使用ssh登陆系统，先运行如下命令，把系统更新一遍：

```
# apt update
# apt upgrade
```

然后创建一个普通用户，使用普通用户权限再次登陆系统。因为使用root去操作有一定风险。

Scala官方推荐使用sdkman这个工具去安装程序，请见这个页面的说明。我自己也是用sdkman来安装scala的，下面是步骤。

首先，系统里要有Java JDK环境，并且是版本8以上。Ubuntu安装OpenJDK很方便，直接用apt就可以搞定，如下所示。

```
$ sudo apt install default-jdk
```

请注意我的系统是Ubuntu 20.0，安装完后得到的JDK是openjdk 11.0版本。

```
$ java -version
openjdk version "11.0.14" 2022-01-18
OpenJDK Runtime Environment (build 11.0.14+9-Ubuntu-0ubuntu2.20.04)
OpenJDK 64-Bit Server VM (build 11.0.14+9-Ubuntu-0ubuntu2.20.04, mixed mode, sharing)
```

接着，安装sdkman，运行如下命令：

```
$ sudo apt install zip
$ sudo apt install unzip
$ curl -s "https://get.sdkman.io" | bash
```

上述先安装两个系统命令，zip和unzip，用于压缩和解压缩。第三句用来安装sdkman工具。

安装完后，运行如下命令：

```
$ source "$HOME/.sdkman/bin/sdkman-init.sh"
$ sdk version
```

上述第一句导入sdkman需要的环境变量，第二句打印sdkman的版本号。

如果你不想每次都手工导入环境变量，那么可以如下设置：

```
$ vi .bash_profile
```

这里打开你的家目录下的.bash_profile文件（如没有就创建一个），写入如下内容：

```
. .bashrc
```

也就是在.bash_profile里加载.bashrc的内容，后者包括了sdkman的环境变量设置。

安装完sdkman后，接着就是运用这个工具来安装scala:

```
$ sdk install scala 2.13.8
```

请注意安装scala并不需要root权限，程序文件安装到了你的家目录下。如下查看scala程序的版本号和路径：

```
$ scala -version
Scala code runner version 2.13.8 -- Copyright 2002-2021, LAMP/EPFL and Lightbend, Inc.

$ which -a scala
/home/pyh/.sdkman/candidates/scala/current/bin/scala
```

如果你运行”sdk list scala”这个命令，就可以看到系统里所有已安装的scala版本，例如：

```
================================================================================
Available Scala Versions
================================================================================
    3.1.1           2.12.14         2.11.12         2.10.3
    3.1.0           2.12.13         2.11.11         2.10.2
    3.0.2           2.12.12         2.11.8          2.10.1
    3.0.1           2.12.11         2.11.7
    3.0.0           2.12.10         2.11.6
 > * 2.13.8          2.12.9          2.11.5
   * 2.13.7          2.12.8          2.11.4
    2.13.6          2.12.7          2.11.3
    2.13.5          2.12.6          2.11.2
    2.13.4          2.12.5          2.11.1
    2.13.3          2.12.4          2.11.0
    2.13.2          2.12.3          2.10.7
    2.13.1          2.12.2          2.10.6
    2.13.0          2.12.1          2.10.5
   * 2.12.15         2.12.0          2.10.4

================================================================================
+ - local version
* - installed
> - currently in use
================================================================================
```

最后，在命令行里输入scala命令，打开scala的REPL（交互式shell），就可以开始练习了。

```
$ scala
Welcome to Scala 2.13.8 (OpenJDK 64-Bit Server VM, Java 11.0.14).
Type in expressions for evaluation. Or try :help.

scala>
```

这里再说明下在spark大数据开发时，scala的版本问题。截至目前，spark最新的版本主要还是用scala 2.12开发的，但部分兼容scala 2.13的语法。如果在spark-shell里，使用2.12与2.13都没有问题。但使用spark-submit提交代码时，用2.13版本会有问题。

Scala 2.13相对于2.12版本，多了一些新的函数，数据操作的API也更为强大。但是也多了一些约束，比如执行不带参数的函数时，2.13要求带上圆括号，2.12则不必。

在scala 2.12里，这样调用函数是合法的：

```
scala> def mytest() = println("hello world")
mytest: ()Unit

scala> mytest
hello world
```

但是在scala 2.13里，空参数调用函数要带上圆括号，否则会触发警告。

```
scala> def mytest() = println("hello world")
def mytest(): Unit

scala> mytest
       ^
       warning: Auto-application to `()` is deprecated. Supply the empty argument list `()` explicitly to invoke method
mytest,
       or remove the empty argument list from its definition (Java-defined methods are exempt).
       In Scala 3, an unapplied method like this will be eta-expanded into a function.
hello world

scala> mytest()
hello world
```

Spark的API方法里，有大量的空参数方法，如果你在2.13里调用它们，记得带上圆括号。

第二章 Scala基本语法

第一节. 变量和数据类型

scala使用val关键字来声明一个不可变的变量。相反的，使用var关键字来声明一个可变值的变量。如下所示。

```
scala> val str = "hello world"
val str: String = hello world

scala> var num = 123
var num: Int = 123
```

上述str是一个不可变的变量，意味着你不能改变它，包括重新赋值：

```
scala> str = "greetings"
           ^
       error: reassignment to val
```

这就触发了一个重复赋值的错误。

但上述num变量是可变的，你可以重新赋值它，如下所示：

```
scala> num = 456
// mutated num

scala> println(num)
456
```

前面我们说过，scala操作大数据，用到了函数式编程的核心思想。而函数式编程的一大特性是值不可变，所以在实际数据开发中，我们基本上只使用val来声明变量，很少使用var。这也保证了数据的安全性。

前文也说过，scala采用”update by copy”的思想来改变一个值。也就是说，通过copy产生一个新的值，并且将它赋值给新的变量。如下所示。

```
scala> val str2 = str + " buddies"
val str2: String = hello world buddies
```

上述将str拷贝一份，并且加上新的字串，赋值给str2变量。在赋值的结果里，也说明了变量的类型。比如，str是一个串（String），num是一个整数（Int）。scala还有其他的数据类型，如下表所示。

Data Type	Possible Values
Boolean	true or false
Byte	8-bit signed two’s complement integer (-2^7 to 2^7-1, inclusive) -128 to 127
Short	16-bit signed two’s complement integer (-2^15 to 2^15-1, inclusive) -32,768 to 32,767
Int	32-bit two’s complement integer (-2^31 to 2^31-1, inclusive) -2,147,483,648 to 2,147,483,647
Long	64-bit two’s complement integer (-2^63 to 2^63-1, inclusive) (-2^63 to 2^63-1, inclusive)
Float	32-bit IEEE 754 single-precision float 1.40129846432481707e-45 to 3.40282346638528860e+38
Double	64-bit IEEE 754 double-precision float 4.94065645841246544e-324d to 1.79769313486231570e+308d
Char	16-bit unsigned Unicode character (0 to 2^16-1, inclusive) 0 to 65,535
String	a sequence of Char

这里稍微注意下的是Char类型，它表示单个unicode字符。Char类型用单引号来定义，双引号就是String类型了。

```
scala> val char = 'a'
val char: Char = a

scala> val char = '1'
val char: Char = 1
```

scala是强类型的，比如两个异构类型不能直接相加：

```
scala> "123" + 456
```

这会触发类型错误。但是，可以进行类型转换后，再相加：

```
scala> "123".toInt + 456
val res2: Int = 579
```

上述将“123”字串，转成了Int整型。反过来，也可以将整数转换成字串，如下所示。

```
scala> "123" + 456.toString
val res3: String = 123456
```

字串的相加，实际是concat操作，也就是拼接起来。

在scala里，一切都是对象。数字是对象，字串也是对象。比如，我们想知道数字对象有哪些转换方法，那么可以在scala shell里输入一个数字，加上”.to”关键字，然后按tab键，REPL就会自动提示你有哪些转换方法可用。如下所示。

```
scala> 123.to
to(                toChar             toFloat            toLong             toShort
toBinaryString     toDegrees          toHexString        toOctalString      toString() (universal)
toByte             toDouble           toInt              toRadians
```

如何打印一个变量呢？很简单，使用println()内置函数。

```
scala> val name = "John"
val name: String = John

scala> val age = 23
val age: Int = 23

scala> println(name)
John
```

可以使用s”expression”来进行变量内插置换，使用f”expression”来格式化输出（类似于printf函数）。例如：

```
scala> println(s"$name is $age years old")
John is 23 years old

scala> println(f"$name is $age%.2f years old")
John is 23.00 years old
```

变量置换是支持表达式的，例如：

```
scala> println(s"$name is ${age+1} years old next year")
John is 24 years old next year
```

上述{}里的就是表达式，age + 1是一个表达式操作。

在REPL里，scala还使用resX来存储临时变量，X是一个数字。你可以引用这个临时变量。

```
scala> "john" + " doe"
val res2: String = john doe

scala> println(res2)
john doe
```

上述产生一个临时变量res2，我们调用println打印这个变量。

第二节. 控制语句

scala的控制语句并不复杂，跟其他语言差不太多。比如一个条件控制语句：

```
scala> import scala.util.Random
import scala.util.Random

scala> if (Random.between(0,10) > 5) {
     |   true
     | } else {
     |   false
     | }
val res15: Boolean = true
```

上述if..else是一个选择执行的条件控制语句。紧跟着的true或者false，是scala里的布尔值。

scala不支持其他语言常见的”?:”操作符号，比如这种写法行不通：

```
scala> val x = 2
val x: Int = 2

scala> val y = x > 1 ? true : false
                     ^
       error: value ? is not a member of Int
```

你只要改成基本的if..else陈述就可以了：

```
scala> val y = if (x > 1) true else false
val y: Boolean = true
```

scala的代码块使用花括号{}引用起来。不过如果紧接着控制语句的只有一行代码，就不需要花括号。

```
scala> if (x > 1)
     |   println("got x greter than 1")
got x greter than 1
```

请注意scala不支持if倒置的方式：

```
scala> println("I meet you") if (name == "John")
                              ^
       error: ';' expected but 'if' found.
```

scala每行结尾是不需要使用分号“;”的。很多Java程序员喜欢在行后面加一个分号，其实是没有必要的。

scalar支持for循环控制语句：

```
scala> val li = List(3,2,1,4)
val li: List[Int] = List(3, 2, 1, 4)

scala> for (x <- li)
     |   println(x)
3
2
1
4
```

scala的for与众不同的是，可以yield返回值，并且带上guard条件。

```
scala> for {
     |   x <- li
     |   if x > 2
     | } yield x
val res19: List[Int] = List(3, 4)
```

上述for循环就在循环过程中，产生了一个新的List，它由原List的元素，经过条件筛选后yield过来。那个if x > 2的条件筛选，就是guard语句。

请注意：这种for循环方式，在scala语法里相当常见。因为scala不鼓励使用可变值，那么要想从数组里筛选出一个新数组，不使用临时变量的话，就只能这样去yield了。

用其他语言对比，在如下ruby程序里，筛选出一个新数组的常见操作是：

```
> li = [3,2,1,4]
=> [3, 2, 1, 4]

> newli = []
=> []

> li.each do |s|
    newli.push(s) if s > 2
> end
=> [3, 2, 1, 4]

> newli
=> [3, 4]
```

但是scala既不使用可变数组，又不鼓励在代码块里访问外部变量，那么要实现上述目的，最好的方式就是这种有点奇葩的for循环了。

一个更复杂的for + guard语句的案例，请见我的一篇博客里的代码。

scala同样也支持while循环，比如：

```
scala> var i = 0
     | while (i < 3) {
     |     println(i)
     |     i += 1
     | }
0
1
2
```

scala生成数列有to和until两种方式，它们稍有不同，请见如下示例：

```
scala> for (x <- 1 to 5) println(x)
1
2
3
4
5

scala> for (x <- 1 until 5) println(x)
1
2
3
4
```

显而易见，to生成的数列包含了最后一位数，而until生成的数列，不包含最后一位数。

第三节. 正则表达式和pattern matching

scala使用如下方式来定义一个正则表达式：

```
scala> val regex = """(\w+)\s+(\d+)""".r
val regex: scala.util.matching.Regex = (\w+)\s+(\d+)
```

请注意上述三引号是必要的，它有对正则表达式的特殊字符自动转义的功能。

如何匹配呢？可以使用matches关键字（请注意这个方法在2.13版本里才引入，2.12及之前的版本并没有）。

```
scala> val str = "hello 999"
val str: String = hello 999

scala> if ( regex.matches(str) )
     |   println("matched")
matched
```

还要特别注意一个坑是，scala正则表达式是从头到尾自动锚定的，等于是自动加上了^$这两个符号（一个表示开头，一个表示结尾）。所以这种情况不会matches:

```
scala> val regex = """^\w+""".r
val regex: scala.util.matching.Regex = ^\w+

scala> val str = "hello 999"
val str: String = hello 999

scala> if ( regex.matches(str) )
     |   println("matched")

scala>
```

上述代码稍微改一下，对正则表达式加上unanchored属性，就可以去掉自动锚定了。

```
scala> val regex = """^\w+""".r.unanchored
val regex: scala.util.matching.UnanchoredRegex = ^\w+

scala> if ( regex.matches(str) )
     |   println("matched")
matched
```

请注意：scala的正则表达式是直接拿来Java的，语法跟Java完全一样。当然，性能上也一般，远赶不上C或Perl的正则表达式的性能。

正则表达式大量用在pattern matching语句里，用来捕获变量，如下所示。

```
scala> val str = "John is 23 years old"
val str: String = John is 23 years old

scala> val regex = """(\w+) is (\d+).*""".r
val regex: scala.util.matching.Regex = (\w+) is (\d+).*
```

```
scala> str match {
    |   case regex(name,age) => println(s"name:$name, age:$age")
    | }
name:John, age:23
```

正则表达式在处理字符上非常有用。当然，scala本身的字符处理方式很强大，随便输入一个字符，跟上一个”.”，再敲tab键，可以看到针对字符串的操作方法提示，如下所示：

```
scala> "str".
!=(                (universal)   equals(                maxOption(                stripLineEnd                ##
(universal)   equalsIgnoreCase(          min(                stripMargin                *(
exists(                minBy(                stripPrefix(                +(                filter(
minByOption(                stripSuffix(
…
```

字符串方法非常多，我这里只截取了一部分。比如split方法：

```
scala> val str = "hello world buddies"
val str: String = hello world buddies

scala> str.split("""\s+""")
val res10: Array[String] = Array(hello, world, buddies)
```

这个split的参数，就是一个正则表达式。它按照正则表达式设置的条件（这里是空格），把字符串拆开，产生一个字符数组。

当然，我们可以直接使用pattern matching语句，来代替这个split函数。

```
scala> val regex = """(\w+)\s+(\w+)\s+(\w+)""".r
val regex: scala.util.matching.Regex = (\w+)\s+(\w+)\s+(\w+)

scala> str match { case regex(x,y,z) => Array(x,y,z) }
val res12: Array[String] = Array(hello, world, buddies)
```

综合看，上述match { case … }陈述，是scala里常见的代码结构。跟前面提到的for { …} yield陈述一样，pattern matching是scala独有的特点。它可以用来代替switch陈述，也可以更优雅的替换掉if else陈述。它被大量使用，以至于看起来无处不在。

```
scala> val li = List(("apple",3),("orange",2),("cherry",5))
val li: List[(String, Int)] = List((apple,3), (orange,2), (cherry,5))

scala> li.map { case (fruit,num) => num }
val res0: List[Int] = List(3, 2, 5)
```

上述li是一个List，它的元素是一个二元tuple，代表水果名，与水果数量。

我们使用map函数来遍历这个List，提取水果数量，并得到一个新的List。在遍历的时候，就使用了pattern matching语句，来提取水果名（放在fruit变量里），以及数量（放在num变量里）。当然在map里只返回了数量，结果是产生一个新的List。

```
scala> val misc = List("abc",123,'A',0.99)
val misc: List[Any] = List(abc, 123, A, 0.99)

scala> for (x <- misc) {
     |   x match {
     |     case i:Int => println("got Int ",i)
     |     case c:Char => println("got Char ",c)
     |     case f:Float => println("got float ",f)
     |     case _ => println("got others ",x)
     |   }
     | }
(got others ,abc)
(got Int ,123)
(got Char ,A)
(got others ,0.99)
```

再来看上述代码的pattern matching部分。通过遍历List，分别match到List的元素是整数、unicode字符、浮点数，还是其他。像这种case语句的设置，一定要覆盖到所有的输入情况，否则就会抛错。最后一个case里，占位符_表示匹配任意其他情况。这个_占位符要放到最后，让它的优先级最低。

scala还有个知名的case class功能，用来定义数据结构，它在spark编程里用的较多。请见如下代码。

```
scala> case class Fruit (name: String, num: Int)
class Fruit

scala> val li = List(("apple",3),("orange",4),("cherry",2),("plum",3))
val li: List[(String, Int)] = List((apple,3), (orange,4), (cherry,2), (plum,3))

scala> val structed = li.map { case (fruit,num) => Fruit(fruit,num) }
val structed: List[Fruit] = List(Fruit(apple,3), Fruit(orange,4), Fruit(cherry,2), Fruit(plum,3))
```

如上，首先声明了一个case class，名字为Fruit，里面定义了数据的结构。接着声明了一个含有水果名和数量的List。然后，使用pattern matching语句来捕获水果名和数量，并把这两个变量传给case class，用来生成强类型约定的数据结构。

如下pattern matching语句用来捕获程序的命令行参数。如果没带参数，就将count初始化为10，否则就使用命令行输入的参数。

```
scala> def main(args:Array[String]):Unit = {
     |
     |   val count = args.size match {
     |     case 0 => 10
     |     case _ => args(0).toInt
     |   }
     |   // ...
     | }
def main(args: Array[String]): Unit
```

跟前面的for {...} yield陈述类似，pattern matching陈述也可以带guard语句，用来进行筛选。

```
scala> for ( x <- 1 to 10 ) {
     |   x match {
     |     case i:Int if i % 2 == 0 => println(s"got $i")
     |     case _ => println("skip..")
     |   }
     | }
skip..
got 2
skip..
got 4
skip..
got 6
skip..
got 8
skip..
got 10
```

上述代码遍历1到10，挑选出整数并且能被2整除的，打印出来。case后面的if语句就是guard条件，用来进一步筛选匹配到的元素。

第四节. 函数与匿名函数

scala定义函数使用def关键字，这点跟其他语言比如ruby类似。如下是定义main函数（又叫做入口函数）的通用写法：

```
scala> def main(args:Array[String]):Unit =
     |   println("hello world")
def main(str: Array[String]): Unit
```

上述args通常表示命令行参数，它的数据类型是一个字符数组。Unit表示返回值的类型。因为我们没有返回值（只是打印了hello world），所以类型是Unit，对应于C语言里的void.

我们可以省略返回类型的定义，scala会自动推导出类型。比如：

```
scala> def mytest(s:String) = {
     |   s.size
     | }
def mytest(s: String): Int

scala> mytest("hello world")
val res0: Int = 11
```

上述代码里，scala自动推导出返回值类型是Int，这显然是正确的。

但有些情况下，scala是无法自动推导返回类型的，这个时候就要显式去定义好返回类型。比如递归的情况。

```
scala> def myrec(x:Int) = {
```

```
     |   x match {
     |     case 1 => 1
     |     case _ => x + myrec(x-1)
     |   }
     | }
         case _ => x + myrec(x-1)
                       ^
On line 4: error: recursive method myrec needs result type
```

上述是一个递归函数，它没有加上返回类型定义，结果就抛错了。改正的方式是，加上返回类型的定义。

```
scala> def myrec(x:Int):Int = {
     |   x match {
     |     case 1 => 1
     |     case _ => x + myrec(x-1)
     |   }
     | }
def myrec(x: Int): Int

scala> myrec(3)
val res1: Int = 6
```

这是一个非常简单的递归，它对于数字n，计算的是1 + 2 + 3 + … n的结果。

scala自身的代码实现里包含了大量递归，什么左递归、右递归、尾递归之类的，它们很复杂。如果要想深入了解，推荐阅读Programming in Scala这本书，它是scala的作者写的，包含了深度的原理。

不过我们在数据操作场景里，不需要那么高级的知识。只要用好递归实现的作品即可，比如后面谈到的高阶函数。

虽然scala可以自动推导返回类型，但许多scala开发者坚持加上明确的返回类型。这样一是让函数声明更清晰，自成文档。二是返回数据更加安全可控。

不管怎样，传入参数的类型，是要显式定义清楚的。scala无法猜测传入参数的类型。除了String, Int这些类型外，参数类型还可以是Array, List, case class等结构。

```
scala> def myfilter(li:List[Int]) = {
     |   for {
     |     x <- li
     |     if x % 2 == 0
     |   } yield x
     | }
def myfilter(li: List[Int]): List[Int]

scala> myfilter(List(3,2,1,4,0,9))
val res3: List[Int] = List(2, 4, 0)
```

上述代码，接受一个List作为参数，然后遍历该List，查找能被2整除的元素，并且返回新的List。它的返回值也是一个List，scala自动推导出来的。

scala支持可变参数，比如传入多个整数，使用如下方式来定义函数：

```
scala> def myint(x:Int*) = {
     |   x.foreach {println}
     | }
def myint(x: Int*): Unit

scala> myint(3,2,1,4)
3
2
1
4
```

scala也支持所谓的泛型，即不确定类型的参数。我曾经写过一个函数，用来对输入List进行采样，返回随机子List集合。代码如下：

```
scala> import scala.util.Random
import scala.util.Random

scala>  def subset[A](x:List[A],y:Int):List[A] = {
     |
     |   if (x.size < y) {
     |     return x
     |   }
     |
     |   val set = scala.collection.mutable.Set[Int]()
     |
     |   while ( set.size < y  ) {
     |     val idx = Random.between(0,x.size)
     |     set += idx
     |   }
     |
     |   set.toList.map(i => x(i) )
     | }
def subset[A](x: List[A], y: Int): List[A]
```

这个subset函数的输入参数，包含一个不确定类型的List，以及采样的元素个数。对于不确定的类型，可以用任意一个大写字母来表示，A, B, C, D都可以。scala解析器会自动计算参数的类型。

调用subset函数的方式如下：

```
scala> val li = List(3,2,1,4,3,2,5,6)
val li: List[Int] = List(3, 2, 1, 4, 3, 2, 5, 6)

scala> subset(li,3)
val res1: List[Int] = List(3, 4, 2)

scala> subset(li,3)
val res2: List[Int] = List(2, 5, 6)
```

参数定义还可以用多个括号分开写，请见如下示例。

```
scala> def mytest(s:String)(x:Int) =
     |   s.size + x
def mytest(s: String)(x: Int): Int

scala> mytest("hello")(11)
val res7: Int = 16
```

这种写法叫做Currying函数，它的用途之一如下：

```
scala> val mytest2 = mytest("hello") _
val mytest2: Int => Int = $Lambda$1155/0x00000008010c2c78@3f67f5ff

scala> mytest2(15)
val res8: Int = 20
```

这里mytest2是一个函数值，scala里是这么叫的。函数值跟python的lambda，以及其他语言的匿名函数，用途是一样的。

匿名函数是高阶函数的基础，而高阶函数在scala的各种数据操作里，用的如此广泛。所有的高阶函数，要么接受一个匿名函数作为参数，要么返回一个匿名函数。

定义一个匿名函数，有好多方法，不过大道至简，我们采用最通用的方法即可。

```
scala> val f:String=>Int = (x) => x.size
val f: String => Int = $Lambda$1159/0x0000000801047800@6c04310f

scala> f("hello")
val res10: Int = 5
```

上述f是一个函数值，本质上是个匿名函数。它的输入参数类型是String，返回值是Int.

(x) => x.size是函数体定义，其中x是输入参数，x.size是针对参数的执行陈述。只有一个参数的话，x外面的括号是可以省略的。如果是多个参数，则必须加上括号。

在scala里，输入参数、返回类型唯一表示一个函数，至于函数有没有名字，并不重要。

匿名函数有什么用呢？最主要用途就是传递给其他函数，作为参数使用。请见如下代码。

```
scala> def mytest(s:String,f:String=>Int) =
     |   f(s)
def mytest(s: String, f: String => Int): Int

scala> mytest("hello", x => x.size)
val res12: Int = 5
```

如上，mytest函数是一个命名函数，但它接受一个String参数，以及一个匿名函数参数。

这个匿名函数的输入类型是String，输出类型是Int。用户传入的匿名函数，必须严格遵循这个类型定义。

接着调用mytest函数，输入一个hello字符串，以及一个匿名函数主体：x => x.size。然后函数成功执行，返回输入字串的长度。

这种匿名函数的使用方式，在scala里用的如此广泛，所以务必理解它的原理。

再看两个例子：

```
scala> def mystr(s:String,f:String=>String) = f(s)
def mystr(s: String, f: String => String): String

scala> mystr("world",x => x.reverse)
val res15: String = dlrow

scala> mystr("world",x => x.substring(0,2))
val res16: String = wo
```

```
scala> def myagg(li:List[Int], f:List[Int]=>Int) = f(li)
def myagg(li: List[Int], f: List[Int] => Int): Int

scala> myagg(List(3,2,1,4,0), x => x.max)
val res19: Int = 4

scala> myagg(List(3,2,1,4,0), x => x.min)
val res20: Int = 0
```

它们原理都是差不多的，将匿名函数作为参数传递给命名函数。在函数声明时，定义好匿名函数的输入值、返回值。在函数执行时，按照事先定义好的类型，传入一个匿名函数主体。

最后，我们可以通过implicit class，对某个对象（比如String对象）来增加一个自定义方法，这个方法接受一个匿名函数作为参数。

```
scala> implicit class foo(s:String) {
     |   def strops(f:String=>String) = f(s)
     | }
class foo

scala> "hello".strops ( x => x.reverse )
val res21: String = olleh

scala> "hello".strops ( x => x.substring(0,2) )
val res22: String = he
```

上述strops方法，就是对String类新增的成员方法，它接受一个匿名函数作为参数。这其实也就是所谓高阶函数的实现，下一节重点介绍。

第三章 Scala高阶函数和数据结构

第一节. 高阶函数

所谓高阶函数，是指要么函数的输入值是一个函数，要么函数的返回值是一个函数。很显然，前一章我们讨论的接受匿名函数作为参数的函数，就是一个高阶函数。

Scala在大数据编程时，主要就是操作不同的数据集合，比如List或者Map。针对这些集合有很多操作方法，它们大部分都是高阶函数。

比如，针对List的map方法（这里指的是map这个成员方法，而不是Map这种数据结构），就是一个高阶函数。因为它接受一个匿名函数作为参数。

假设有如下List，它包含元素1到10，我们想对每个元素加1，那么可以使用map方法：

```
scala> val li = (1 to 10).toList
val li: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> li.map ( x => x+1)
val res11: List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
```

上述map遍历list，然后使用用户自定义的匿名函数，对每个元素加1，并且返回一个新的List.

x => x+1就是一个匿名函数，这点在前一章节已多次讨论。如果不清晰，请回头再看过。

map的实现等同于如下for循环：

```
scala> for {
     |  x <- li
     | } yield x + 1
val res12: List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
```

由于map操作支持用户自定义匿名函数，它的功能就十分强大，可以实现多种形式的数据transform。

比如，我们可以用map来做各种数据抽取和转换。

```
scala> val fruit = List(("orange",3),("apple",2),("cherry",5))
val fruit: List[(String, Int)] = List((orange,3), (apple,2), (cherry,5))
```

```
scala> fruit.map { case (x,y) => y }
val res13: List[Int] = List(3, 2, 5)
```

上述我们用map抽取出水果列表的数量部分，并且产生一个新的List.

在Spark编程里，经常需要将一系列的特征词汇，转换成RDD（弹性分布式数据集）。此时也需要用到map。

```
scala> val words = List("scala","perl","ruby","c++","python","java","lisp","erlang")
val words: List[String] = List(scala, perl, ruby, c++, python, java, lisp, erlang)

scala> words.map( x => (x,1) )
val res14: List[(String, Int)] = List((scala,1), (perl,1), (ruby,1), (c++,1), (python,1), (java,1), (lisp,1), (erlang,1))
```

上述map运行的结果是产生一个新的List，每个元素是一个二元tuple，包含了物品的名字和物品出现的次数。

请注意：每次map运行都产生一个新的数据集合，这也符合函数式编程的风格：update by copy。它不会修改原始的数据集合。在实际工作中，这些高阶函数都是链式运作，即一大堆高阶函数拼接在一起，但每个函数都不会修改输入值，而只是拷贝数据并且返回转换后的对象。由此产生一个新的对象，原来的输入对象不会被修改，这保证了数据的安全性。

在使用高阶函数时，有两个语法细节稍微提一下。

第一，高阶函数的参数的包围形式，既可以是圆括号，也可以是花括号。case语句除外，这里必须使用花括号。如下所示。

```
scala> (1 to 5).map(x => x*2)
val res20: IndexedSeq[Int] = Vector(2, 4, 6, 8, 10)

scala> (1 to 5).map{ x => x*2 }
val res21: IndexedSeq[Int] = Vector(2, 4, 6, 8, 10)

scala> (1 to 5).map { case i:Int => i*2 }
val res22: IndexedSeq[Int] = Vector(2, 4, 6, 8, 10)
```

第二，scala有一个语法糖，可以使用占位符“_”来代表传入的参数。它的写法如下：

```
scala> (1 to 5).map( _*2 )
val res23: IndexedSeq[Int] = Vector(2, 4, 6, 8, 10)

scala> List("some","buddies","here").map( _.size )
val res24: List[Int] = List(4, 7, 4)
```

这里的“_”看起来有些奇怪，不过不重要，如果你不熟悉，那么用经典的匿名函数写法就好了。它只是语法糖而已，背后的东西没变。

map经常和reduce结合在一起用，用来实现统计之类的目的。如下是一个常见案例。

```
scala> val words = List("scala","perl","ruby","perl","scala","python")
val words: List[String] = List(scala, perl, ruby, perl, scala, python)

scala> words.map( x => (x,1) ).groupMapReduce(_._1)(_._2)(_+_).toList
val res19: List[(String, Int)] = List((python,1), (scala,2), (perl,2), (ruby,1))
```

上述给定一个字串列表，要统计每个字串出现的次数，可以用groupMapReduce，它对应的Spark RDD方法是reduceByKey()。这个方法看起来很复杂，但背后的本质就是group、map和reduce。map已经讲过了，再讲讲reduce和group。

reduce就是把上下文按照用户给定的方式汇聚起来。比如用户指定两数相加，那就把输入对象的每个元素执行相加。

```
scala> val li = (1 to 10).toList
val li: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> li.reduce( (x,y) => x + y)
val res25: Int = 55

scala> li.reduce( (x,y) => x * y)
val res26: Int = 3628800
```

这里(x,y) => x + y是用户自定义的匿名函数。x,y两个输入参数表示上下文相邻的两个元素。

当然，按照scala的语法糖，上述代码也可以简写如下：

```
scala> li.reduce( _ + _)
val res27: Int = 55

scala> li.reduce( _ * _)
val res28: Int = 3628800
```

那么group呢？它在scala里对应的语法是groupBy，即按照用户给定的条件，对对象进行分组。用法如下。

```
scala> words
val res39: List[String] = List(scala, perl, ruby, perl, scala, python)

scala> words.map( (_,1) ).groupBy(_._1)
val res38: scala.collection.immutable.Map[String,List[(String, Int)]] = HashMap(perl -> List((perl,1), (perl,1)),
scala -> List((scala,1), (scala,1)), python -> List((python,1)), ruby -> List((ruby,1)))
```

上述words对象，在map执行后，产生一个新的List。新List的元素是一个二元tuple，包含了字串名，和字串本次的出现次数（为1）。然后，groupBy再访问这个新的List，按照用户给定的条件，根据tuple的第一个元素（即字串名）进行分组，并且返回分组后的新对象。

scala访问tuple使用的下标是从1开始的，这点要特别注意。

```
scala> (1,2,3)._1
val res40: Int = 1

scala> (1,2,3)._2
val res41: Int = 2
```

综上所述，把group, map, reduce结合起来，就产生了groupMapReduce函数，它的用法就是按照key分组，对输入对象进行汇聚统计。groupMapReduce(_._1)(_._2)(_+_)这里带了三个参数，第一个参数是指定groupBy的对象，第二个参数是指定reduce的对象，第三个参数是指定reduce的方式。

这个函数在spark的RDD编程里用的非常多，所以这里花了较多文字介绍。至于如何根据group, map, reduce来实现groupMapReduce，这个算法并不难。比如ruby没有这个方法，我就自己实现了一个，请见我的Github项目。如果你对此还疑惑不解，请看后一节groupMapReduce详解里的描述。

filter也是scala里常用的方法，它的用法跟map差不多，如下。

```
scala> val li = (1 to 10).toList
val li: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> li.filter( _ > 5)
val res49: List[Int] = List(6, 7, 8, 9, 10)
```

上述对1到10这个List执行过滤，找出大于5的，产生一个新的List.

```
scala> List("hello","world","lofter","soft","leom").filter( _.contains("lo") )
val res52: List[String] = List(hello, lofter)
```

上述对一个字符列表执行过滤，找出包含“lo”的字符，返回一个新的List.

filter对应的for循环实现如下：

```
scala> for {
     |   x <- li
     |   if x > 5
     | } yield x
val res53: List[Int] = List(6, 7, 8, 9, 10)
```

scala还有个flatten方法，对数组中的数组，或者列表中的列表，进行打散操作。例如：

```
scala> List(List(1,2), List(3,4)).flatten
val res63: List[Int] = List(1, 2, 3, 4)
```

对应的有个flatMap方法，也就是对数据结构，先map再flatten。flatMap常用在外部读入文件的行处理上，比如分词行为。对读入的每行执行一个map，得到一个数组。再调用flat把所有数组打散成一个大数组。用法示例如下。

```
scala> import scala.io.Source
import scala.io.Source

scala> val lines = Source.fromFile("/etc/lsb-release").getLines().toList
val lines: List[String] = List(DISTRIB_ID=Ubuntu, DISTRIB_RELEASE=18.04, DISTRIB_CODENAME=bionic,
DISTRIB_DESCRIPTION="Ubuntu 18.04.6 LTS")

scala> lines.flatMap(x => x.split("="))
val res7: List[String] = List(DISTRIB_ID, Ubuntu, DISTRIB_RELEASE, 18.04, DISTRIB_CODENAME, bionic,
DISTRIB_DESCRIPTION, "Ubuntu 18.04.6 LTS")
```

sort函数也是大数据运算里常见的方法。scala的默认sort方法如下：

```
scala> List(3,2,1,4,0).sorted
val res13: List[Int] = List(0, 1, 2, 3, 4)
```

如上，sorted方法返回一个排序后的新列表。

如果要自己指定规则排序，那么请使用sortBy方法：

```
scala> List(3,2,1,4,0).sortBy(-_)
val res14: List[Int] = List(4, 3, 2, 1, 0)
```

上述sortBy方法，接受的参数看起来有点奇怪，实际上就是"_"占位符（前文已解释），加上一个"-"表示取反，也就是说这里指定按照逆序进行排列。

如下，如果sortBy只接受一个参数identity，那它就是默认排序，相当于sorted函数。

```
scala> List(3,2,1,4,0).sortBy(identity)
val res17: List[Int] = List(0, 1, 2, 3, 4)

scala> List(3,2,1,4,0).sorted
val res18: List[Int] = List(0, 1, 2, 3, 4)
```

如下的sortBy排序，依照tuple的第二个元素，进行降序排列。

```
scala> val li = List(("apple",10),("orange",2),("cherry",5))
val li: List[(String, Int)] = List((apple,10), (orange,2), (cherry,5))

scala> li.sortBy(-_._2)
val res20: List[(String, Int)] = List((apple,10), (cherry,5), (orange,2))
```

sortBy经常结合groupMapReduce一起使用，来对大型数据集合进行分组排序处理。请见如下案例。

```
scala> import scala.io.Source
import scala.io.Source

scala> val lines = Source.fromFile("test.txt").getLines().toList
```

```
scala> lines.size
val res0: Int = 100000

scala> val sorted = lines.map( (_,1) ).groupMapReduce(_._1)(_._2)(_+_).toList.sortBy(-_._2)

scala> sorted.size
val res3: Int = 9462

scala> sorted.take(20).foreach(println)
(the,4995)
(to,3777)
(a,2000)
(and,1837)
(is,1799)
(i,1538)
(in,1503)
(on,1463)
(at,1455)
(of,1439)
(this,1334)
(you,1295)
(that,1174)
(for,1147)
(it,1020)
(be,767)
(not,761)
(from,755)
(with,721)
(are,708)
```

如上代码是典型的字数统计案例。输入文件有10万行，每行一个字符。文件读入到List数据结构中，List的元素就是行的内容。然后，对这个List先用map进行转换，将每个元素转换为二元tuple，得到一个新的List。再对这个新List运用groupMapReduce操作，得到分组汇总的元素。对分完组的元素，运用sortBy进行逆序排序，得到字符数统计结果。最后打印统计结果的前20名。

如下是distinct函数，用来取列表的唯一值。如果你熟悉SQL，那么就知道这个distinct是差不多的。

```
scala> List(3,2,1,4,3,2).distinct
val res28: List[Int] = List(3, 2, 1, 4)
```

而scala另外还支持一个distinctBy操作，它是一个高阶函数，支持按用户自定义的条件进行distinct。如下是一个示例。

```
scala> List(3,2,1,4,3,2).distinctBy( _ % 2 == 0)
val res29: List[Int] = List(3, 2)
```

上述表示对每个元素按能否被2整除来作唯一性区分。只有两个可能：要么能整除，要么不能。所以就得到了列表开头的两个元素。

fold函数，也是常见的数列汇聚操作，它的用法跟reduce差不多，不过加上了一个初始值。下面对比一下你就知道了。

```
scala> (1 to 5).toList.reduce( _+_ )
val res31: Int = 15

scala> (1 to 5).toList.fold(10)( _+_ )
val res32: Int = 25
```

如上，reduce按照用户指定的方式，对列表里的元素进行相加汇聚操作。而fold稍有不同的是，传递多了一个参数（10），这个参数表示初始值，在汇聚的时候，要算上这个初始值。

第二节. 详解groupMapReduce

groupMapReduce是scala原生的函数，对应的spark里的RDD函数叫做reduceByKey，我在ruby里也扩展了一个数组方法叫做reduceByKey。三者的用法高度类似，我也写过一篇博客介绍，有兴趣请点击浏览。

为了方便读者进一步了解这个函数，我引用自己实现的ruby的reduceByKey说明下原理。这个函数的全部代码也就如下几行。

```
class Array
    def reduceByKey
        if block_given?
            group_by {|x| x[0]}.map {|x,y| y.reduce {|x,y| [ x[0], yield(x[1],y[1]) ]}}
        else
            raise "no block given"
        end
    end
end
```

真正有效的就是这么一行：

```
group_by {|x| x[0]}.map {|x,y| y.reduce {|x,y| [ x[0], yield(x[1],y[1]) ]}}
```

它是怎么实现的呢？我结合一个具体的数据讲解下。

比如，有如下ruby二维数组，每个元素都是k/v形式的子数组。

```
list = [["apple",3],["orange",2],["apple",4],["plum",3],["orange",5]]
```

运行reduceByKey后，得到如下结果：

```
> list.reduceByKey {|x,y| x+y}
```

```
=> [["apple", 7], ["orange", 7], ["plum", 3]]
```

它按照key（水果名）分组，把value（水果数量）按照用户自定义的函数进行汇聚操作。

用户在调用这个方法时，要传一个代码块：{|x,y| x+y}。它的作用等同于用户自定义的匿名函数。这个匿名函数传递给了reduceByKey，用来指定reduce的处理方式（累加）。

上述那行有效的ruby代码，首先执行了一个group_by操作，按照key进行分组。执行完后，得到如下结果。

```
> grouped = list.group_by {|x| x[0]}
=> {"apple"=>[["apple", 3], ["apple", 4]], "orange"=>[["orange", 2], ["orange", 5]], "plum"=>[["plum", 3]]}
```

你看，group_by就是按照水果名进行分组，把该水果名下所有的数据条目，集合到一起，放在一个数组里。

接下来就是要map上述产生的数据结构，对水果条目的数组进行进一步分析。这个map后，我们实际上只要value的部分，也就是按名字聚集的水果条目的数组。

```
> mapped = grouped.map {|x,y| y}
=> [[["apple", 3], ["apple", 4]], [["orange", 2], ["orange", 5]], [["plum", 3]]]
```

上述结果是一个大数组，每个子数组包含了按名字聚集的水果条目。只要对每个子数组再执行一次reduce操作，就可以了。比如对第一个子数组reduce如下。

```
> mapped[0].reduce {|x,y| [x[0], x[1]+y[1]]}
=> ["apple", 7]
```

如果全部reduce完，就是我们想要的结果：

```
[["apple", 7], ["orange", 7], ["plum", 3]]
```

上述分拆代码用来说明，所谓groupMapReduce函数，也就是真正执行了group, map, reduce三个函数来进行计算。再看一眼关键的代码：

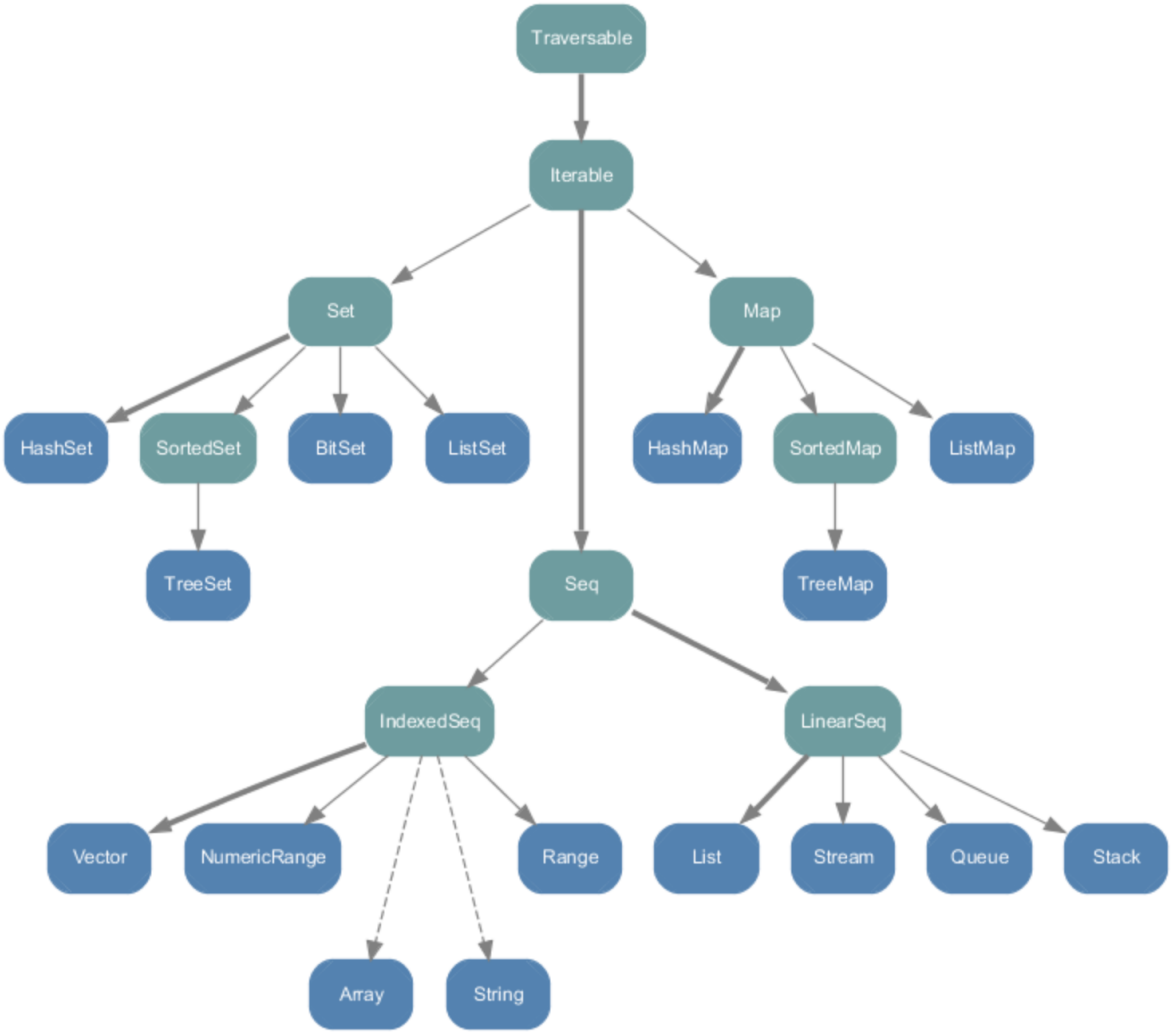
```
group_by {|x| x[0]}.map {|x,y| y.reduce {|x,y| [ x[0], yield(x[1],y[1]) ]}}
```

实际上就是三个函数联合操作罢了。上述reduce是map的子函数，map遍历时把所有元素的reduce都执行完了。另外，yield那句表示将参数传给用户的函数，用户的函数拿到这两个参数后，自己决定怎么汇聚（比如相加或者相乘）。

第三节. 数据结构

scala的数据结构叫做collection，十分丰富多彩。总体上分为mutable（可变）和不可变（immutable）两大家族。出于函数式编程的特点，scala默认导入的是immutable的数据结构，即不可变的集合。如果你需要mutable的结构，就需要手工导入。

scala数据结构的体系很庞大，如下是scala.collection.immutable家族的血脉图。



它总体上分为三部分：

1. Set：数据集合，里面的值唯一
2. Map：数据映射，是k/v这种成对的形式
3. Seq：数据序列，又分为线性的，和索引的。前者代表是List，后者代表是Vector。

而scala.collection.mutable家族的成员就更复杂，它们的示意图请见scala官网这个页面，我不在此赘述。

考虑到函数式编程的特点，出于数据安全性考虑，大部分情形下，我们只需要用到不可变的数据结构。不可变的集合是默认导入的，比如我们常见的List声明：

```
scala> val li = List(3,2,1,4)
val li: List[Int] = List(3, 2, 1, 4)
```

数组声明：

```
scala> val arr = Array(3,2,1,4)
val arr: Array[Int] = Array(3, 2, 1, 4)
```

Set声明：

```
scala> val set = Set(3,2,1,4)
val set: scala.collection.immutable.Set[Int] = Set(3, 2, 1, 4)
```

Map声明：

```
scala> val map = Map(1->2, 3->4)
val map: scala.collection.immutable.Map[Int,Int] = Map(1 -> 2, 3 -> 4)
```

对于不可变的数据结构，你是不能修改它的，比如不能增加或删除值。

```
scala> val li = List(1,2,3,4)
val li: List[Int] = List(1, 2, 3, 4)

scala> li(0) = 3
       ^
       error: value update is not a member of List[Int]
       did you mean updated?
```

如上试图修改一个不可变集合的List值会引发错误。

但是不可变集合的Array类型，却可以修改它的值：

```
scala> val arr = Array(1,2,3,4)
val arr: Array[Int] = Array(1, 2, 3, 4)

scala> arr(3) = 9

scala> arr
val res16: Array[Int] = Array(1, 2, 3, 9)
```

这是因为虽然值修改了，但是指向array成员的指针地址并没有改变。尽管它是用val声明的不可变结构，修改了值但是指针没变，因此不冲突。

而可变结构需要手工导入，比如可变数组使用ArrayBuffer，它的使用方式如下：

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

```
scala> buffer += 1
val res0: buffer.type = ArrayBuffer(1)

scala> buffer += 2
val res1: buffer.type = ArrayBuffer(1, 2)
```

可变哈希映射的使用：

```
scala> val hash = scala.collection.mutable.Map[String,Int]()
val hash: scala.collection.mutable.Map[String,Int] = HashMap()

scala> hash += ("str" -> 1)
val res6: hash.type = HashMap(str -> 1)
```

实际上，在大数据操作中，我们主要使用不可变集合中的List、Set、Map这几个结构。其他的很少使用。

比如大家都认为数组（Array）很有用，查找起来很快，但scala的开发人员自己都不推荐使用数组。他们在邮件列表里如下回复：

> *Never use Array unless you need it for Java interop, or unless you are writing very-high-performance code and you are absolutely convinced that the only way to make it fast enough is to use Array.*
>
> *Array never appears at all in normal Scala code.*
>
> *You should prefer to use immutable collections such as List. But even if you decide you need a mutable collection, Array isn't the first one you reach for. Because it comes directly from the JVM, it isn't a proper Scala collection and as a result, has multiple peculiarities.*

所以简言之，没必要使用数组。我自己也是这么干的。

但是Set和Map还是有用的。前者保存不重复的数据，后者采用k/v的方式来保存数据，都各自有其使用场景。

scala的数据结构的API，我不打算在本书详细讲解。如果你想了解每个数据结构如何使用的，那么推荐阅读Scala Cookbook by Alvin Alexander这本书，书里将每个数据结构的操作方法都举例演示了一遍。

数据结构的使用案例，我们再来看如下程序及运行结果。

```
scala> import scala.io.Source
     |
     | val li = Source.fromFile("words.txt").getLines()
     | val set_sw = Source.fromFile("stopwords.txt").getLines().toSet
     | val hash = scala.collection.mutable.Map[String,Int]()
     |
     |
     | for (x <- li) {
```

```
    |     if ( ! set_sw.contains(x) ) {
    |       if (hash.contains(x)) hash(x) += 1 else hash(x) = 1
    |     }
    | }
    |
    | val sorted = hash.toList.sortBy(-_._2)
    | sorted.take(20).foreach {println}

(message,28796)
(send,25714)
(unsubscribe,20906)
(2022,19696)
(file,19650)
(list,19502)
(mailing,15676)
(2021,15589)
(jan,15365)
(100644,14259)
(mail,14189)
(diff,14173)
(email,14148)
(return,13829)
(code,13198)
(pm,13172)
(data,12785)
(group,12566)
(feb,12483)
(problem,11961)
```

上述代码也用来做字符统计。它读取原始文件（每行一个字符），放在li数据结构里。读取stopwords（关键字过滤），放在set_sw这个Set结构里。然后声明了一个可变结构的哈希Map。接着在for循环里，首先过滤掉stopword。接着判断，如果Map包含了字符key，就把值增加1，否则就将值初始化为1。这样统计的结果就位于Map里，最后排序打印统计列表。

请注意上述li对应的数据结构，既不是List也不是Set，它是另一种结构，叫做Iterator。Iterator是独立的数据结构，它不属于任何collection。

```
scala> li
val res1: Iterator[String] = <iterator>
```

Iterator的特性之一是惰性执行，之二是只呈现一次。比如上述程序执行完了，再查看li结构，就发现已经清空了。

```
scala> li.foreach(println)

scala>
```

所谓惰性执行是指在需要的时候才执行。对Iterator进行map的时候不会马上执行，只有在需要map结果的时候，才会真正执行。我们看如下示例。

```
scala> import scala.io.Source
import scala.io.Source
```

```
scala> val fd = Source.fromFile("/etc/lsb-release").getLines()
val fd: Iterator[String] = <iterator>

scala> fd.map( _.size )
val res0: Iterator[Int] = <iterator>

scala> res0.toList
val res1: List[Int] = List(17, 21, 23, 40)

scala> res0.foreach(println)
```

如上，fd这个Iterator，执行了一次map，得到的还是一个Iterator。这个时候map并没有真正执行。直到执行toList操作，map才执行了。同时原来的Iterator也被清空了，foreach遍历时返回空。这就是惰性执行。Spark里大量使用了这种原则，以提高计算效率。

为什么读取文件得到的是一个Iterator？这主要出于效率的考虑。在大文件情况下，如果将这个Iterator转换成List，马上内存就溢出了。这是Iterator在效率上的优势。

但是Iterator对涉及到shuffle的操作，是不能执行的。这是因为shuffle要将数据打散再重新排列组合，涉及到多次访问数据的操作。而Iterator的元素只能present once.

groupBy和sortBy这些都是shuffle操作，它们不能针对Iterator对象执行。

在实际大数据开发中，我们主要从外部来加载数据。不管spark、hive，还是drill、impala，这些工具都是计算和分析引擎，本身并不存储数据。我们往往需要从外部数据源加载数据，比如HDFS, S3, Mysql, 甚至是流(streaming)。数据格式多种多样，既有文本文件、CSV的，又有JSON、Parquet的，等等。

这些数据加载到内存，就放在一个数据结构里，其中List是用的最多的，Set和Map也有使用。数据放到数据结构里，就可以运用各种高阶函数，对数据进行计算，如过滤、汇总、分组、排序等。

如果文件内容太大，那么单机的scala就处理不了，此时就需要spark这种分布式计算引擎。spark的RDD计算引擎，也使用跟scala类似的高阶函数。不过是把这种函数计算，由单机环境搬到分布式环境上罢了。所以掌握了原理，由scala开发环境切换到spark开发环境，就是顺其自然的事。

第四节. 编译和构建

scala既可以作为脚本运行，又可以编译成JVM需要的Jar包。对于长时间运行的任务，无疑编译后执行的效率更高。scala的默认编译和构建工具是sbt。下面介绍如何安装和使用sbt。

sbt同样使用sdkman来进行安装。关于sdk工具的使用，在安装scala那一节已经有介绍，这里不再赘述。安装了sdk工具后，运行如下命令安装sbt:

```
$ sdk install sbt
```

请注意sdk安装的软件，都不需要root权限。

安装完后，在命令行输入sbt就可以进入交互式界面：

```
$ sbt
copying runtime jar...
[info] [launcher] getting org.scala-sbt sbt 1.6.2  (this may take some time)...
[info] [launcher] getting Scala 2.12.15 (for sbt)...
[info] Updated file /home/pyh/project/build.properties: set sbt.version to 1.6.2
[info] welcome to sbt 1.6.2 (Ubuntu Java 11.0.14)
[info] loading project definition from /home/pyh/project
[info] Updating
https://repo1.maven.org/maven2/jline/jline/2.14.6/jline-2.14.6.pom
  100.0% [##########] 19.4 KiB (88.3 KiB / s)
[info] Resolved  dependencies
[info] Fetching artifacts of
[info] Fetched artifacts of
[info] set current project to pyh (in build file:/home/pyh/)
[info] sbt server started at local:///home/pyh/.sbt/1.0/server/60070a2b98c3d69a98e6/sock
[info] started sbt server
sbt:pyh>
```

在交互式shell里输入help可以看到命令帮助：

```
sbt:pyh> help

  <command> (; <command>)*                Runs the provided semicolon-separated commands.
  about                                   Displays basic information about sbt and the build.
  tasks                                   Lists the tasks defined for the current project.
…
```

我们既可以在sbt shell里运行sbt命令，比如输入compile进行编译。也可以在外部运行sbt命令，比如在bash里输入sbt compile进行编译。

```
sbt:pyh> compile
[success] Total time: 1 s, completed Mar 15, 2022, 8:50:41 AM
```

下面使用sbt构建一个测试项目，步骤细分如下：

1. 在Linux系统里创建项目目录，比如test
2. 进到test目录，运行mkdir -p src/main/scala创建源代码目录
3. 在test目录里，创建build.sbt文件，里面放置的内容在后面描述
4. 在test目录的src/main/scala下，创建test.scala程序文件，内容如后描述
5. 在test目录里，运行sbt compile进行编译
6. 如果编译通过，再运行sbt run执行程序
7. 可选使用sbt package来打包程序

按照上述步骤，我们先在系统里创建test目录。然后进到这个目录，编辑build.sbt文件，输入内容如下：

```
name := "test"

version := "0.1"

scalaVersion := "2.13.8"

libraryDependencies +=  "org.scalanlp" %% "breeze" % "2.0.1-RC1"
```

这个文件指定scala项目的构建环境，比如包依赖等。

请注意：上述各行，务必用空行分开。也就是说，两行配置之间，隔着一个空行。

上述配置文件说明如下：

1. 第一行：指定项目名字
2. 第二行：指定项目版本
3. 第三行：指定本次构建所使用的scala版本
4. 第四行：指定本次项目所需要的包依赖

前面三行都一眼明了，最后一行指定包依赖的格式，请参考这个链接的说明。

简言之，用百分号"%"分开的三个域，分别表示组织id、工件id、修订版本号。这里表示加载了breeze这个依赖，它的主版本号跟随scala的版本（使用两个百分号"%%"表示跟随scala的版本），修订版本号是2.0.1-rc1。

创建完build.sbt后，接着创建源代码。源代码通常位于src/main/scala子目录下，如果没有就先手工创建这个目录。

在src/main/scala目录下，创建test.scala文件，内容如下：

```
import breeze.linalg._

object Mytest {
  def main(args:Array[String]):Unit = {
    val m = DenseMatrix.zeros[Int](5,5)
    println(s"colums: ${m.cols}, rows: ${m.rows}")
  }
}
```

在程序里，首先加载breeze这个科学计算库。接着定义一个单态对象，名字为Mytest。在对象里，定义入口函数，名字为main。

main主函数只做了一件事，创建一个5x5的稠密矩阵，初始值为0，然后打印矩阵的列数、行数，就退出了。

代码写完后，再回到test目录，运行sbt compile进行编译：

```
$ sbt compile
...
[info]   Compilation completed in 18.843s.
[success] Total time: 31 s, completed Mar 15, 2022, 9:13:08 AM
```

编译过程中，会自动下载所需要的依赖包。编译完成后，如果成功会有如上类似的提示。接着，运行sbt run来执行程序。

```
$ sbt run
…
colums: 5, rows: 5
[success] Total time: 2 s, completed Mar 15, 2022, 9:13:34 AM
```

如上，程序执行完成，打印了行列数。最后一行的success表示执行成功。

此时，我们运行如下命令，看看sbt自动生成的目录结构：

```
$ tree -L 3
.
├── build.sbt
├── project
│   ├── build.properties
│   └── target
│       ├── config-classes
│       ├── scala-2.12
│       └── streams
├── src
│   └── main
│       └── scala
└── target
    ├── bg-jobs
    ├── global-logging
    ├── scala-2.13
    │   ├── classes
    │   ├── sync
    │   ├── test_2.13-0.1.jar
    │   ├── update
    │   └── zinc
    ├── streams
    │   ├── _global
    │   ├── compile
    │   └── runtime
    └── task-temp-directory
```

在目录结构里，除了build.sbt和src/main/scala是我们手工创建的外，其他都是sbt自动生成的目录。主要包括project目录，用来存放项目配置；以及target目录，用来存放目标文件。其中，target/scala-2.13/test_2.13-0.1.jar就是生成的jar文件。

至此，我们的项目就构建完了。关于sbt的更多功能，请参考其官方文档。

第四章 Spark简介

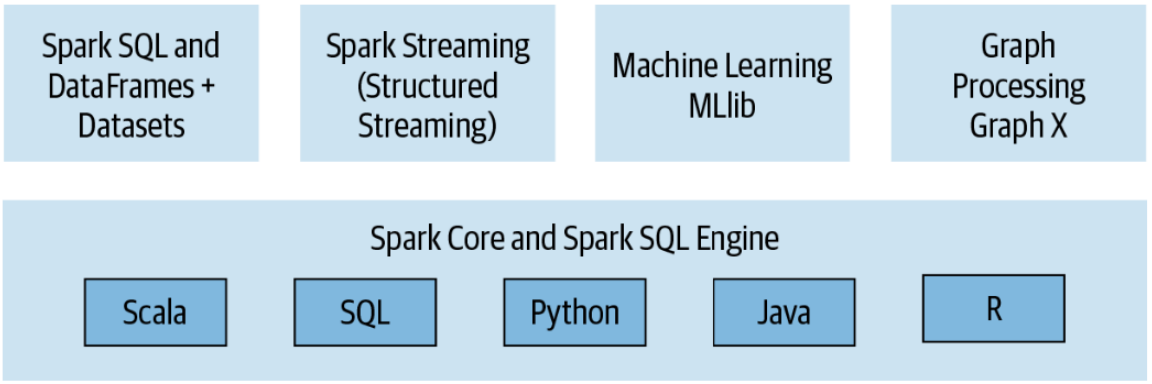
第一节. 关于Apache Spark

Spark是databricks公司维护的分布式计算软件，是Apache名下的开源项目。这个软件主要用于分布式计算功能。前面我们已经提了scala的高阶函数用来进行数据计算，但scala是单机运行的，数据量大的话，它就会内存溢出。而Spark解决了这个问题，它在设计上就是为分布式计算任务服务的。

首先，安利下databricks公司的绝对经典的书《Learning Spark, Second Edition》。这本书居然是在线免费的，而且写的非常好。如果你对Spark不熟悉，那么看这本书入门就足够了。

Spark最初的目的是为了取代Hadoop的map/reduce（简称MR）。我们知道Hadoop默认的MR非常慢，它在执行过程中涉及到的数据交换都要落盘到HDFS。而Spark优化了这个问题，它全部在内存中完成计算任务，性能相对于Hadoop的MR，大幅提升数十倍不止。

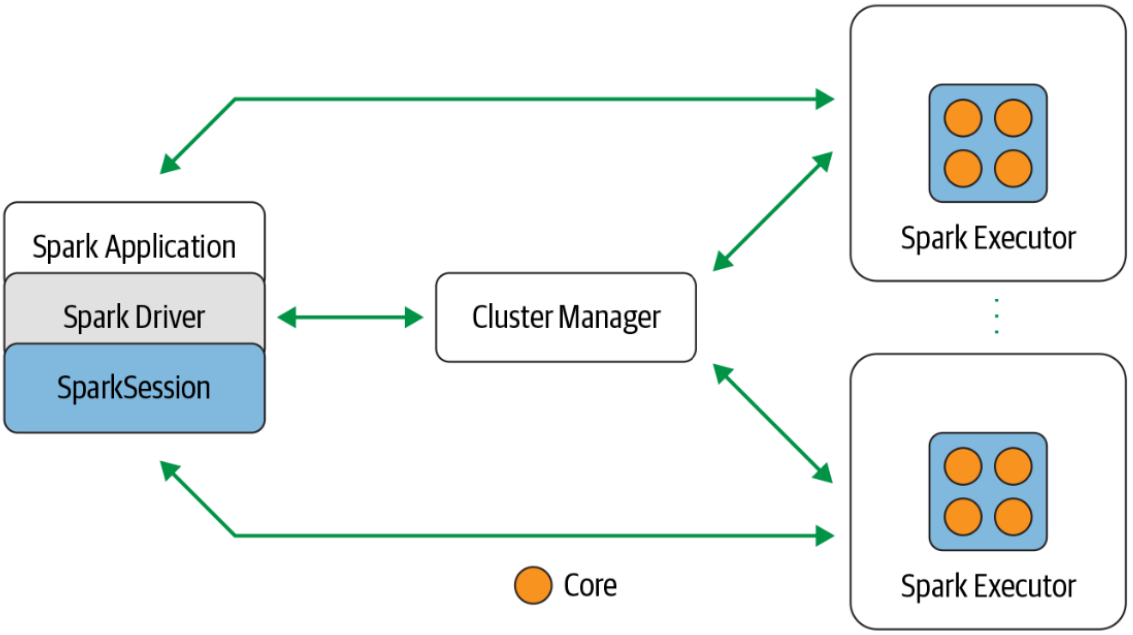
Spark当前发展到3.2版本，这个版本的功能和稳定性都有了很大改善。它的功能结构图如下：



最底层是Spark Core引擎（也就是RDD计算）以及SQL引擎。在此基础上，有如下四大部分面向应用的API：

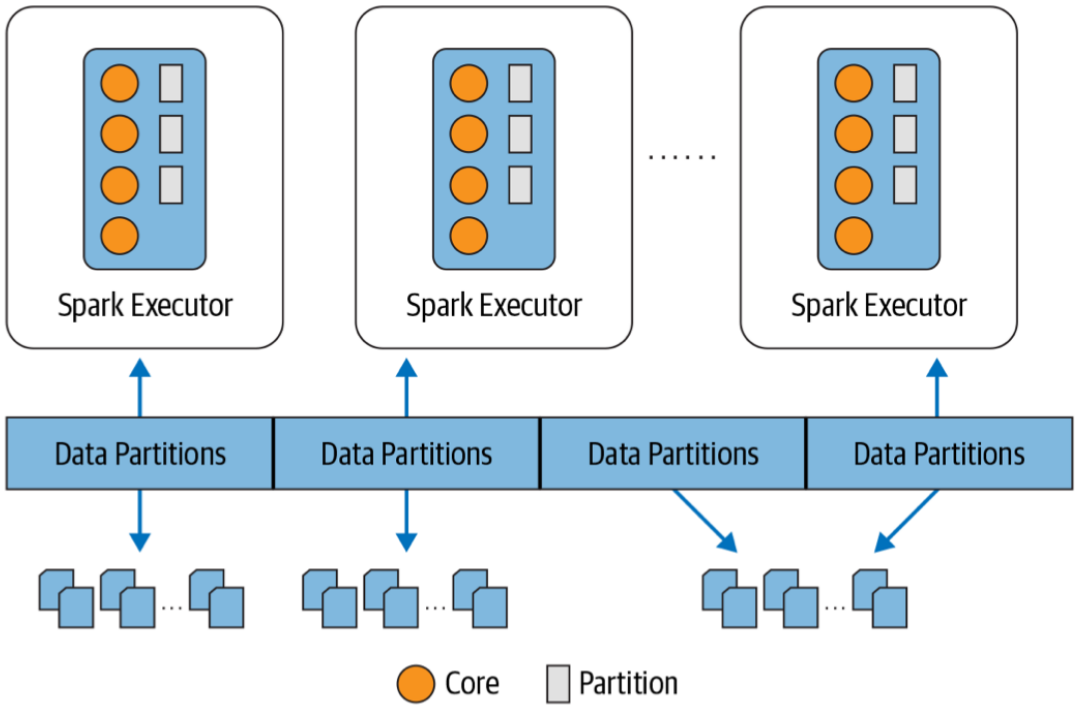
1. Spark SQL和dataframe, datasets API
2. Spark流API，包含DStream和结构化流
3. 机器学习MLlib API
4. 图计算API Graph X

Spark的技术执行架构如下图所示。



用户的应用代码在Spark Driver这里，在代码里定义的任务，通过Spark Session提交到Spark Executor那里执行。Executor是一个集群，可以由Cluster Manager来进行管理。

Spark采用分区的方式，将数据分散到多个物理机器上执行并行运算，这也是Spark大数据处理的核心优势。如下图所示。



第二节. 安装Spark

我们在scala的同一台机器上安装spark。如前文所叙，我们已经准备了一台新的虚拟机，配置是2个AMD核，1GB内存，安装ubuntu 20.0 64位操作系统。

spark的安装有几种模式可以选择，包括本地模式、standalone模式、集群模式等。由于我们这里只有一个单独测试机，因此选择本地模式。

首先，下载Spark最新版本，当前版本是3.2.1 ,[下载页面](#)在此。

在Linux机器上，输入如下命令来下载spark-3.2.1.

```
$ wget https://dlcdn.apache.org/spark/spark-3.2.1/spark-3.2.1-bin-hadoop3.2.tgz
```

下载完后，使用tar命令来解压文件包。

```
$ tar zxvf spark-3.2.1-bin-hadoop3.2.tgz
```

解开后得到一个新目录，将这个目录移动到路径/opt/spark:

```
$ sudo mv spark-3.2.1-bin-hadoop3.2 /opt/spark
```

编辑用户家目录的.bash_profile文件，新增如下内容：

```
export JAVA_HOME=/usr
export SPARK_HOME=/opt/spark
export PATH=$PATH:/opt/spark/bin:/opt/spark/sbin
export PYSPARK_PYTHON=/usr/bin/python3
```

然后，退出当前shell，再重新登陆，以使得上述环境变量生效。

重新登陆后，运行spark-shell进入交互式编程环境：

```
$ spark-shell
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.2.1
      /_/

Using Scala version 2.12.15 (OpenJDK 64-Bit Server VM, Java 11.0.14)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

这就可以了，本地安装的环境就这么简单。看到上述scala提示符，你是不是有很熟悉的感觉呢？没错，这本质上就是scala的REPL，不过加载了spark的一些内置对象。

Spark本身使用scala开发，在spark-shell里，你可以使用熟悉的scala语句来编写程序。

请注意spark的官网写道：

```
Note that Spark 3 is pre-built with Scala 2.12 in general and Spark 3.2+ provides additional pre-built distribution with Scala 2.13.
```

也就是说spark 3.2还是用scala 2.12开发，但提供了scala 2.13的部分附加功能。通常在spark-shell里使用scala 2.13是没有问题的。但如果要用spark-submit提交任务，则需要用回scala 2.12版本。

为方便起见，我们在系统里安装上scala 2.12.15版本：

```
$ sdk install scala-2.12.15
```

安装完后，使用如下命令在当前shell里切换到2.12.15:

```
$ sdk use scala 2.12.15

Using scala version 2.12.15 in this shell.
```

跟scala一样，spark既可以在交互式shell（REPL）里使用编程环境，也可以将客户端代码编译打包后，通过spark-submit提交到集群进行计算。我在随后到章节里，主要讲解在交互式shell里使用scala进行编程。

第三节. 为什么使用Apache Spark

使用spark的理由有很多，包括但不限于：

- 使用spark的RDD API来清洗和转换数据
- 使用spark的dataframe API对结构化数据进行高效查询
- 使用spark的stream API来集成流查询，等等

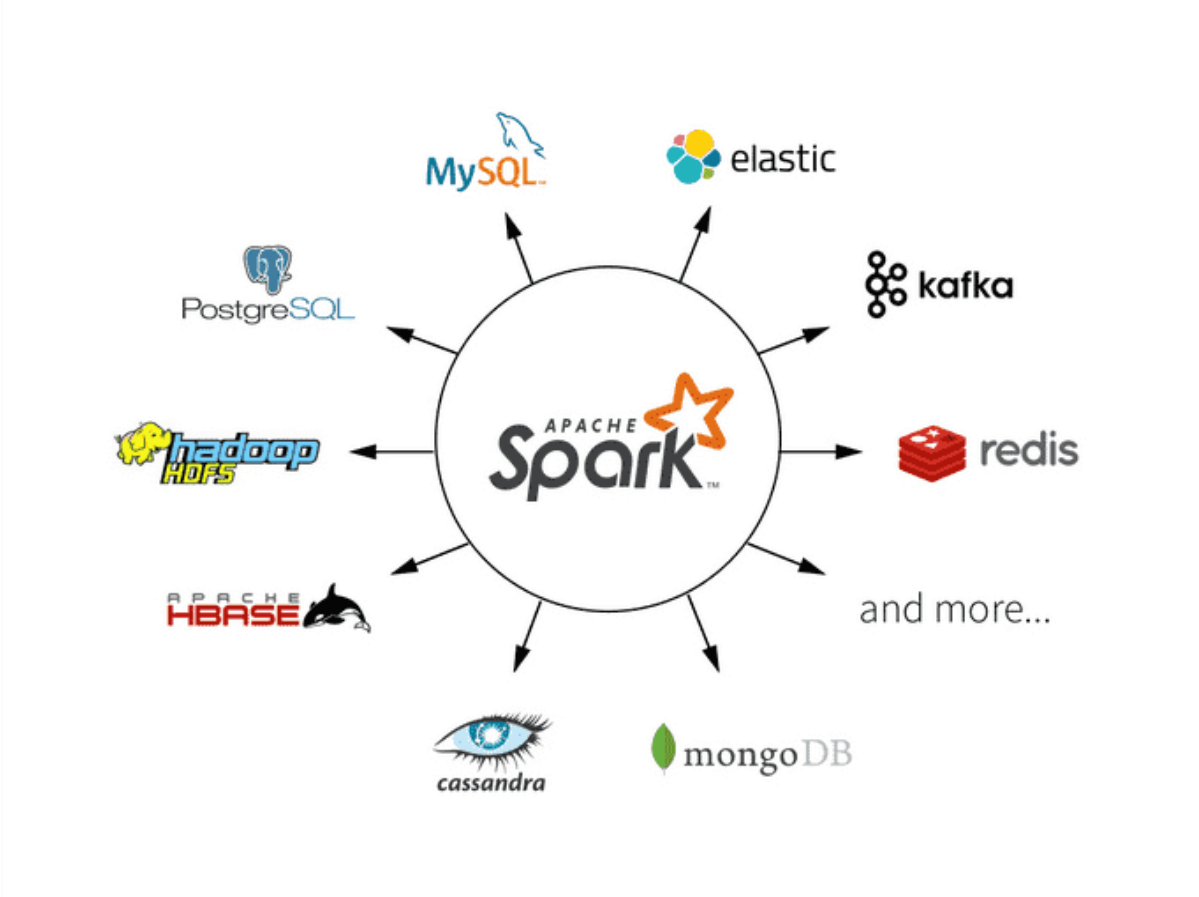
然而，在大数据领域，spark的竞争者也很多。比如同为Apache旗下的开源产品flink, impala, drill等，都有spark类似的功能。那么spark为何能独树一帜呢？

这其中有很多原因。最重要的一点是spark是最早开始做内存计算引擎的。在2010年代，基于文件系统的分布式计算引擎Hadoop还很慢，运行一个长查询甚至需要几个小时的时间。于是spark横空出世，利用分布式内存计算，大幅提升了计算引擎的效率，使过去需要几个小时的查询，在几分钟内就能完成。spark最早成为分布式内存计算的标杆，它在业内得到广泛认可，几乎成为行业标准，后来出现的其他产品都借鉴学习了spark的东西。

此外，spark与scala真是相辅相成，互惠互利。scala的数据操作API很强大，高阶函数很强大，流API很强大。然而，scala摆脱不了JVM内存的限制，稍大点的文件，动不动就让scala内存溢出。对于shuffle函数的操作，scala内置的文件流处理机制毫无作用。而spark全面改善了内存管理的问题。哪怕在一个只有2G物理内存的机器上，也能轻松处理4G的文件，所有group，sort查询毫无压力。这是原生的scala不可想象的。关于小内存处理大文件的性能情况，我曾经写过一篇博客，可以点此浏览。

scala带给spark的好处也显而易见，比如丰富的高阶函数完善了数据操作手段，函数式编程保证数据的高可靠性，强大的类型处理机制保证数据安全等。scala的所有特性，自定义函数也好，数据结构也好，正则表达式也好，case class也好，implicit class也好，在spark里都可以无缝的集成和使用。这个优势是其他语言API所不可比拟的。

一个真实的场景是，分析web服务器的日志。这些访问日志动不动就几十个G，用scala原生语言来分析它们会很费劲。但是，把它们扔到spark集群里，就可以利用所有scala的数据操作方法，和spark的分布式内存计算效率，来查询这些数据了。spark帮助我们做了内存管理和优化，不会因为文件太大而导致内存溢出。在本书后面的章节介绍了如何在spark里查询web服务器日志。

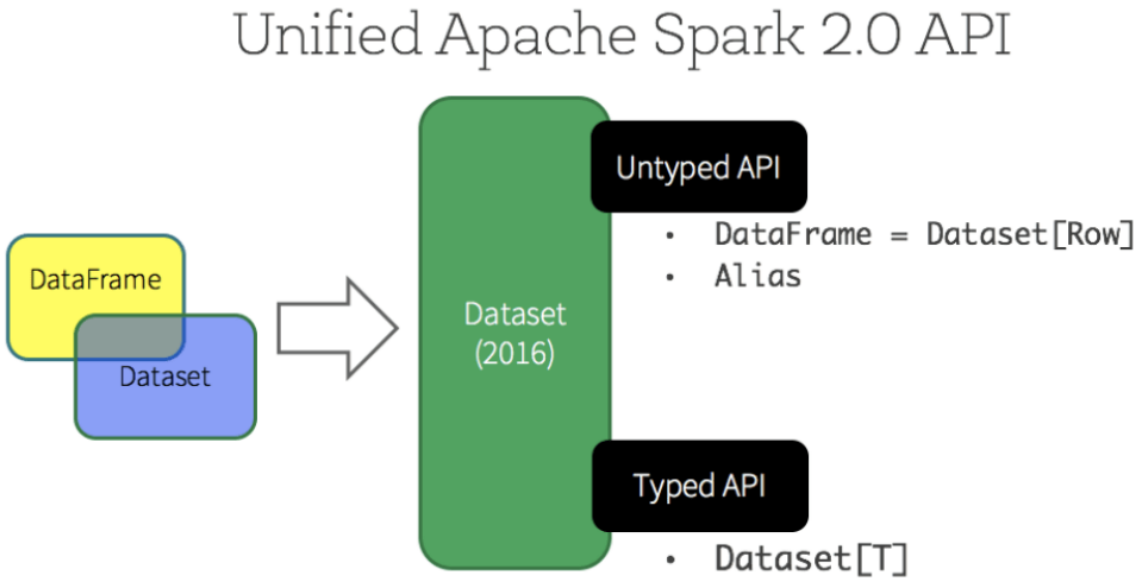


Spark发展这么多年，还有个优势是生态上的广度。在大数据领域，spark独树一帜，成为分布式内存计算的标杆。这个领域的其他产品，或主动或被动的都向spark看齐，从而使得spark集成别家的产品非常容易。如上图所示，各种SQL、NoSQL引擎，Hadoop产品，流式引擎，都与spark良好的集成。这减轻了用户的工作量，降低了项目开发的风险。

第五章 Spark编程环境

第一节. 使用RDD编程

RDD代表弹性分布式数据集，它是spark并行运行的基础，也是spark的core API。spark最早只提供了RDD API，后来的SQL引擎都是在此基础上发展起来的。你看如下图就明白了。



如果是针对结构化数据的查询任务，当前版本的spark里，已经没有必要使用RDD API。有两点理由：

1. RDD API比较慢，而SQL引擎是高度优化过的。基于高度结构化数据的大型统计任务，使用SQL引擎的API，比如dataframe，效率要高得多。
2. RDD API写起来不太直观，不像SQL引擎，可以直接写SQL来查询。而SQL语句大家都很熟悉。

不过，使用RDD也有自己的好处，比如数据清洗时的过滤和transform，用RDD会方便很多。另外，SQL引擎是针对结构化数据的，而RDD却不分数据类型，结构化的、半结构化的、非结构的数据，都可以处理。

实际情况是经常需要在这两个引擎之间切换。比如原始数据在RDD里清洗完成，就转换成dataframe对象，再切换成dataframe API去完成统计。

RDD的编程接口文档请见[这个链接]。它可以使用我们熟悉的高阶函数。这些执行函数又分为两部分：transformation和action。前者是惰性执行的，关于惰性执行我在scala那一部分已经讲过。简言之，前者的函数定义后并不会马上执行，直到遇到后者，才真正落地执行。

让我们开始RDD编程之旅。

首先输入spark-shell，打开spark的交互式shell：

```
$ spark-shell
Spark context available as 'sc' (master = local[*], app id = local-1647321502887).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.2.1
      /_/

Using Scala version 2.12.15 (OpenJDK 64-Bit Server VM, Java 11.0.14)
Type in expressions to have them evaluated.
Type :help for more information.
```

如同你在上述提示里看到的一样，有两个关键变量内置了：

1. sc：代表spark上下文对象，面向RDD编程接口
2. spark：代表spark会话对象，面向SQL引擎的编程接口

在RDD里，我们主要使用sc这个内置变量。比如：

```
scala> val rdd = sc.parallelize(List(1,2,3,4))
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:23
```

如上List(1,2,3,4)是输入数据，经过sc对象的parallelize方法转换后，就成为一个RDD对象。

这个RDD对象可以使用transformation和action两大类里的方法来操作它。这些方法大多数是高阶函数。

让我们看一个简单的map:

```
scala> rdd.map(_+1)
res0: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[1] at map at <console>:24
```

map完后并没有立刻打印结果，这是因为transformation是惰性执行的。需要调用一个action函数，才能使它真正执行和落地。collect就是一个action类型的函数。

```
scala> rdd.map(_+1).collect
res1: Array[Int] = Array(2, 3, 4, 5)
```

take也是action类型的，比如取前2个元素：

```
scala> rdd.map(_+1).take(2)
res2: Array[Int] = Array(2, 3)
```

reduce是action类型的，比如：

```
scala> rdd.reduce(_+_)
res3: Int = 10
```

我们来看一个实际案例，统计水果的数量。

假设有一个数据文件，共1000行，每行包括一个水果名字。我们统计下水果的数量和排序。这个数据文件我已经生成好了，在此链接下载。

在spark-shell里，首先加载这个文件到RDD对象：

```
scala> val rdd = sc.textFile("data/fruits.txt")
rdd: org.apache.spark.rdd.RDD[String] = data/fruits.txt MapPartitionsRDD[5] at textFile at <console>:23

scala> rdd.count
res10: Long = 1000
```

然后执行如下语句进行统计分析：

```
scala> rdd.map( (_,1) ).reduceByKey( _+_ ).sortBy(-_._2).collect
res6: Array[(String, Int)] = Array((Longan,47), (Watermelon,45), (plum,44), (carambola,39), (pear,37), (fig,36),
(strawberry,35), (betelnut,35), (apple,34), (mango,34), (banana,34), (muskmelon,33), (tangerine,33), (Lemon,32),
(orange,31), (loquat,31), (Waxberry,31), (rambutan,31), (peach,31), (persimmon,30), (Cherry,30), (juice,29),
(honey,29), (papaya,28), (coconut,28), (pineapple,27), (durian,27), (Kiwi,26), (lichee,25), (olive,25), (grape,23))
```

我们看到统计的结果是一个数组，包含了水果的分组数量统计，以及按数量从多到少排序。

这个统计过程中，使用了如下高阶函数：

1. map: 将输入的水果名，转换为(apple, 1)这种k/v形式的tuple。在各种RDD计算任务里，数组元素都是成对出现的。
2. reduceByKey: 这个对应scala原生语言的groupMapReduce，我在前面的章节已详细讲过，不确认的话请回头再看一看。这个函数的作用是，按照水果名分组，将水果数量进行相加汇总。
3. sortBy: 按照tuple的第二个元素（水果数量）进行降序排列。

最后一个collect不是高阶函数，它是action函数，负责将之前的transform落地执行。

再看一眼reduceByKey函数，这个函数用的很广泛，它是reduce的改进版本，用于分组统计。参考如下示例。

```
scala> val fruits = List(("apple",1),("orange",2),("apple",3),("orange",4))
fruits: List[(String, Int)] = List((apple,1), (orange,2), (apple,3), (orange,4))

scala> val rdd = sc.parallelize(fruits)
```

```
rdd: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[0] at parallelize at <console>:24

scala> rdd.reduceByKey(_ + _).collect
res1: Array[(String, Int)] = Array((orange,6), (apple,4))
```

reduceByKey执行后，就按照水果名进行分组，并且把每组的数量进行相加。这个函数要求输入是k/v这种形式，即一个二元tuple。_+_看起来花里胡俏，其实是一个匿名函数，对应如下实现，这点在之前的scala介绍里反复提起过。

```
scala> rdd.reduceByKey{ (x,y) => x+y }.collect
res2: Array[(String, Int)] = Array((orange,6), (apple,4))
```

sortBy高阶函数，跟scala里原生的sortBy用法几乎一样，指定一个排序的条件。如下所示。

```
scala> rdd.reduceByKey(_ + _).sortBy( -_._2 ).collect
res3: Array[(String, Int)] = Array((orange,6), (apple,4))
```

这个排序条件就是，按照tuple的第二个元素（数量），进行降序排列。

跟scala原生语言一样，spark的RDD也有distinct函数：

```
scala> val rdd = sc.textFile("data/fruits.txt")
rdd: org.apache.spark.rdd.RDD[String] = data/fruits.txt MapPartitionsRDD[10] at textFile at <console>:23

scala> rdd.count
res14: Long = 1000

scala> rdd.distinct.count
res15: Long = 31
```

它的filter高阶函数也跟scala的基本一样：

```
scala> rdd.filter( _ == "apple").count
res17: Long = 34
```

再接着看一个函数groupByKey:

```
scala> val rdd2 = rdd.map( (_,1) )
rdd2: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[16] at map at <console>:23

scala> rdd2.groupByKey.map{ case(x,y) => (x,y.size) }.sortBy(-_._2).collect
res24: Array[(String, Int)] = Array((Longan,47), (Watermelon,45), (plum,44), (carambola,39), (pear,37), (fig,36),
(strawberry,35), (betelnut,35), (apple,34), (mango,34), (banana,34), (muskmelon,33), (tangerine,33), (Lemon,32),
(orange,31), (loquat,31), (Waxberry,31), (rambutan,31), (peach,31), (persimmon,30), (Cherry,30), (juice,29),
(honey,29), (papaya,28), (coconut,28), (pineapple,27), (durian,27), (Kiwi,26), (lichee,25), (olive,25), (grape,23))
```

这个函数其实不是高阶函数，它没有输入的匿名函数。它只作用在二元tuple上。它针对输入的k/v数据，根据key进行分组。简单看看如下示例。

```
scala> val tmp = sc.parallelize(List(("apple",1),("orange",2),("apple",3)))
tmp: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[29] at parallelize at <console>:23

scala> tmp.groupByKey.collect
res25: Array[(String, Iterable[Int])] = Array((orange,CompactBuffer(2)), (apple,CompactBuffer(1, 3)))
```

检查下返回的数据结构，就知道groupByKey是如何工作的了。

再看一个sortByKey函数，它只接受一个参数，指定是否为升序排列，排列依据也是key。

```
scala> tmp.sortByKey(true).collect
res3: Array[(String, Int)] = Array((apple,1), (apple,3), (orange,2))

scala> tmp.sortByKey(false).collect
res4: Array[(String, Int)] = Array((orange,2), (apple,1), (apple,3))
```

如上，参数为true的话采用升序排列，参数为false的话采用降序排列。

RDD的countByKey方法，是根据key来汇总个数的。因为count是个action方法，所以countByKey也是acton方法。请见如下示例。

```
scala> val tmp = sc.parallelize(List(("apple",1),("orange",2),("apple",3)))
tmp: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[2] at parallelize at <console>:23

scala> tmp.countByKey
res1: scala.collection.Map[String,Long] = Map(orange -> 1, apple -> 2)
```

RDD API还有一个比较奇怪的函数，aggregateByKey，它接受三个参数：第一个是初始值，第二个是分区内的汇聚函数，第三个是分区之间的汇聚函数。如果初始值是0的话，它的作用跟reduceByKey是一样的。stackoverflow上有一个非常好的解释，我引用如下。

```
scala> val pairs = sc.parallelize(Array(("a", 3), ("a", 1), ("b", 7), ("a", 5)))
pairs: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[10] at parallelize at <console>:23

scala> val resReduce = pairs.reduceByKey(_ + _)
resReduce: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[11] at reduceByKey at <console>:23

scala> resReduce.collect
res5: Array[(String, Int)] = Array((b,7), (a,9))

scala> //0 is initial value, _+_ inside partition, _+_ between partitions

scala> val resAgg = pairs.aggregateByKey(0)(_+_,_+_)
resAgg: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[12] at aggregateByKey at <console>:23

scala> resAgg.collect
res6: Array[(String, Int)] = Array((b,7), (a,9))
```

RDD的flatMap跟scala原生的flatMap用法也几乎一样：

```
scala> val rdd = sc.textFile("/etc/lsb-release")
```

```
val rdd: org.apache.spark.rdd.RDD[String] = /etc/lsb-release MapPartitionsRDD[65] at textFile at <console>:1

scala> rdd.flatMap(x => x.split("=")).collect()
val res21: Array[String] = Array(DISTRIB_ID, Ubuntu, DISTRIB_RELEASE, 18.04, DISTRIB_CODENAME, bionic,
DISTRIB_DESCRIPTION, "Ubuntu 18.04.6 LTS")
```

除了map跟flatMap外，RDD还有个特殊的mapValues，它针对k/v结构的，并且只对值进行map。让我们看如下示例。

```
scala> val rdd = sc.parallelize( List( ("apple",2),("orange",3),("cherry",4) ) )
rdd: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[0] at parallelize at <console>:23

scala> rdd.mapValues(_ + 1).collect
res0: Array[(String, Int)] = Array((apple,3), (orange,4), (cherry,5))
```

上述mapValues等同于如下map:

```
scala> rdd.map { case(k,v) => (k,v+1) }.collect
res0: Array[(String, Int)] = Array((apple,3), (orange,4), (cherry,5))
```

对于大型数据集，经常有采样需求，takeSample函数被用来实现这一目的。

```
scala> val rdd = sc.parallelize( (1 to 10).toList )
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[5] at parallelize at <console>:23

scala> rdd.takeSample(false,5)
res2: Array[Int] = Array(5, 4, 8, 9, 1)

scala> rdd.takeSample(false,5)
res3: Array[Int] = Array(8, 2, 4, 3, 10)
```

这个函数第一个参数指定是否允许重复采样，上述false是一个布尔值，表示不重复采样。第二个参数是采样的个数。

第二节. 使用RDD来查询半结构化数据

Spark RDD的强大能力之一是处理多种数据格式。既可以是结构化的数据，又可以是半结构化、甚至非结构化的数据。在如下案例里，我们使用RDD来查询web服务器的访问日志，这是一种典型的半结构化数据。

web服务器的访问日志，动辄若干个G，这样的日志用传统方式分析起来费时费力。幸运的是，大数据时代有spark这样的工具可以对它运行有效查询。除了spark外，还有Apache Drill也支持web服务器日志的存储插件，可以用来查询和分析日志。

我们有一个日志片段文件，名为access.log，它包含了用户访问Apache httpd服务器的日志。一个最常用的功能是IP统计，总共有多少唯一IP访问了我们的网站。这个统计可以用如下RDD查询来实现。

```
scala> val rdd = sc.textFile("tmp/access.log")
val rdd: org.apache.spark.rdd.RDD[String] = tmp/access.log MapPartitionsRDD[70] at textFile at <console>:1

scala> rdd.map(x => x.split("""\s+""")(0)).distinct().count()
val res22: Long = 189
```

上述代码的作用是，首先加载日志文件到RDD对象。接着执行一个map，把每行的内容，按照空格分隔开来，得到一个数组。这个数组的第一个元素就是IP地址，我们单独提取出IP地址。然后对IP地址数组，运行distinct方法，得到唯一IP。再运行count方法，得到唯一IP的个数，这就是想要的结果。

如何查看访问IP的前10名呢？运行如下查询。

```
scala> rdd.map(x => (x.split("""\s+""")(0),1)).reduceByKey(_+_).sortBy(-_._2).take(10).foreach(println)
(141.101.105.219,656)
(172.70.142.3,263)
(162.158.159.91,47)
(172.68.110.165,47)
(172.70.90.139,47)
(172.70.162.61,47)
(141.101.99.236,47)
(172.70.147.105,28)
(172.70.54.225,24)
(172.70.142.237,16)
```

上述代码解释如下：

- 首先对日志逐行运行map，取出IP地址，加上一个数字1，组成一个二元tuple并返回。
- 对包含二元tuple的数组，运行reduceByKey方法，这里的key是IP地址，对value执行累加操作，得到了根据IP地址汇总的数量。
- 再运行sortBy方法，按照汇总的数量，进行降序排序。
- 取出结果的前10名，循环进行打印。

如果我们想查看有哪些浏览器访问了网站，那么运行如下查询。

```
scala> val regex = """.*\"(.*)\"""".r
val regex: scala.util.matching.Regex = .*\"(.*)\"

scala> rdd.map { case regex(bot) => bot }.distinct().count()
val res25: Long = 73

scala> rdd.map { case regex(bot) => (bot,1) }.reduceByKey(_+_).sortBy( -_._2).take(5).foreach(println)
(Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/85.0.4183.102
Safari/537.36,655)
(Mozilla/5.0 (Windows NT 10.0; rv:78.0) Gecko/20100101 Firefox/78.0,351)
(Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4240.193
Safari/537.36,262)
(Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/74.0.3729.169
Safari/537.36,188)
```

```
(WordPress/5.9.3; https://xxx.com,27)
```

上述代码解释如下：

- 第一行定义了一个正则表达式，用于匹配最后一个双引号里的内容，就是UserAgent。
- 第二行在map里用case语句提取了浏览器，并通过distinct和count方法，得到访问的唯一浏览器的数量。
- 第三句跟上述查询访问IP前10的原理是一样的，这里查询访问的浏览器前5的型号和数量。

只要你了解web服务器日志的格式，那么可以用spark的RDD来查询任何想要的因素。这就是RDD的一个实用案例。

第三节. RDD查询的性能

Spark的RDD很强大，它支持各种数据类型的查询，还可以方便的对数据做transform。但是，RDD的查询性能，相对于高度结构化的dataframe API，却稍有逊色。dataframe是在spark的SQL引擎上运行的，而SQL引擎是高度优化过的。关于dataframe与SQL引擎，这里暂时提及一下，在后文会详细描述。

我们通过一个示例看两者的性能对比。有如下数据集，它是一份虚构的个人信息数据，数据规模和字段类型如下。

```
scala> df.count()
val res28: Long = 1000000

scala> df.printSchema()
root
 |-- NAME: string (nullable = true)
 |-- SEX: string (nullable = true)
 |-- BORN: string (nullable = true)
 |-- ZIP: integer (nullable = true)
 |-- EMAIL: string (nullable = true)
 |-- JOB: string (nullable = true)
 |-- SALARY: double (nullable = true)
```

如上，有100万条数据，包含了名字、性别、出生日期、邮政编码、电子邮件、工作、薪水7个字段。

如果我们想查询每个工种的平均薪水，并且按从高到低打印前10名，那么在spark RDD里如下查询。

```
scala> rdd.map { x => x.split(",") }.
     |       map{ x => (x(5), x(6).toDouble) }.
     |       groupByKey().mapValues(x => x.sum/x.size).
     |       sortBy(-_._2).take(10).
     |       foreach(println)
```

```
(Financial Advisor,17627.49984729716)
(Veterinary Technologist & Technician,17626.99676080575)
(Veterinarian,17626.10807045477)
(Radiologic Technologist,17625.788032656314)
(Bus Driver,17625.104965243296)
(Physical Therapist,17609.883656224236)
(Architect,17608.53486480992)
(Construction Worker,17598.863951644867)
(IT Manager,17591.957864038617)
(Security Guard,17591.273364018412)
```

而对应的dataframe查询方法（后文会介绍）如下：

```
scala> df.groupBy("job").agg(avg("salary").alias("avg_salary")).orderBy(desc("avg_salary")).show(10,false)
+-----------------------------------+------------------+
|job                                |avg_salary        |
+-----------------------------------+------------------+
|Financial Advisor                  |17627.49984729716 |
|Veterinary Technologist & Technician|17626.99676080575 |
|Veterinarian                       |17626.10807045477 |
|Radiologic Technologist            |17625.788032656314|
|Bus Driver                         |17625.104965243296|
|Physical Therapist                 |17609.883656224236|
|Architect                          |17608.53486480992 |
|Construction Worker                |17598.863951644867|
|IT Manager                         |17591.957864038617|
|Security Guard                     |17591.273364018412|
+-----------------------------------+------------------+
only showing top 10 rows
```

上述两段查询，结果是一样的，但dataframe要快一些。可惜spark没有提供查询耗时统计，否则就可以看到对比很明显。尤其对于体量非常大的数据，比如条目过亿的数据集，dataframe的性能优势更是明显，要甩开RDD一大截。

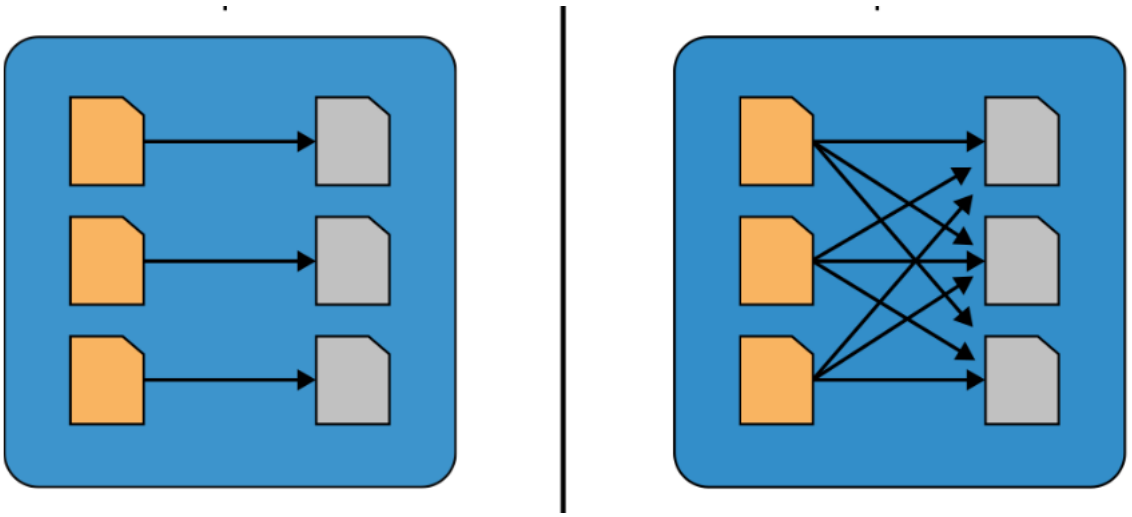
所以，对于高度结构化的数据查询，建议使用后述介绍的dataframe API来进行。spark官方也是这么建议的。

第四节. 关于shuffle与数据分区

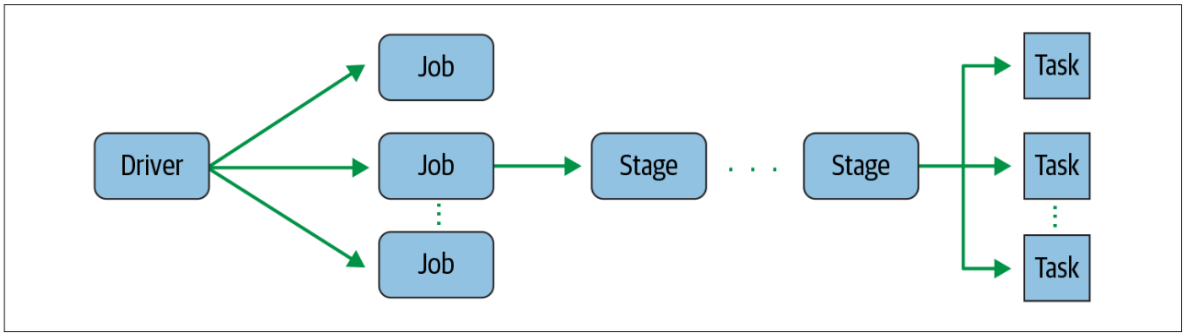
spark的RDD API里，有些非常昂贵的操作，它涉及到shuffle操作。我们在scala的内容里已经介绍过，Iterator是惰性执行的，它没有shuffle相关的方法。因为shuffle涉及到对数据集打散再重新排列组合。这个操作在spark里也是非常昂贵的，数据要在集群之间重新balance，性能上会受比较大的影响。

涉及到shuffle的操作包括所有的group操作和sort操作。很显然这两个操作要对数据重新洗牌，所以它们很是消耗性能。

如下第一张图是不需要shuffle的操作，第二张图是shuffle操作，它将数据打散重排了。



通常来说，spark的每个应用就是一个job。每个job在执行时，会被拆分成多个stage。一般属于shuffle的操作，会单独拆分成stage。其他不属于shuffle的操作，可能放在同一个stage里。而同一个stage，可能又位于多个数据分区上，每个分区都有一个对应的task。虽然stage是按照图的方式，从前往后执行的。但是task是并行执行的。这个示意图如下。



多个task之间如果有数据交互，比如shuffle操作，就会带来比较大的网络IO和延时。所以数据洗牌往往认为是非常昂贵的操作。

对于每个RDD对象，究竟有多少个数据分区？这取决于数据量有多大，以及cpu的个数。

如果数据size小于128M（这是HDFS文件系统的默认block大小），那么就按照cpu的个数来确定分区数量。比如我的虚拟机是两个cpu，那么它的数据分区就是2.

```
scala> rdd.getNumPartitions
res5: Int = 2
```

如果数据size大于128M，那么分区数量通常是数据大小除以128M得到的个数。比如我从磁盘加载一个大文件，那么分区数就显然上去了。

```
scala> val rdd = sc.textFile("data/words.txt")
rdd: org.apache.spark.rdd.RDD[String] = data/words.txt MapPartitionsRDD[9] at textFile at <console>:23

scala> rdd.count
```

```
res6: Long = 205157060

scala> rdd.getNumPartitions
res7: Int = 33
```

对于涉及到shuffle的操作，比如所有的group、sort函数，还有distinct，都可以手工指定分区数量。比如：

```
scala> rdd.distinct(50).count
res8: Long = 132789
```

第五节. 使用Dataframe编程

dataframe API是基于SQL引擎的。它相对于RDD API来说，的确改进了很多，更直观，性能也更高。关于dataframe与RDD的计算性能对比，我写过一篇博客，在这个链接，有兴趣可以浏览。

dataframe的函数多了很多，而且还支持用户自定义的函数（UDF）。

比如，求一个数列的平均数，RDD要自己去算，但dataframe可以用自带的函数解决。

```
scala> val rdd = sc.parallelize( (1 to 10).toList )
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[14] at parallelize at <console>:23

scala> val sum = rdd.reduce(_+_)
sum: Int = 55

scala> val count = rdd.count
count: Long = 10

scala> val mean = sum.toFloat/count.toFloat
mean: Float = 5.5
```

上述是RDD的实现。

```
scala> val df = rdd.toDF
df: org.apache.spark.sql.DataFrame = [value: int]

scala> df.select(avg("value")).show
+----------+
|avg(value)|
+----------+
|       5.5|
+----------+
```

上述是dataframe的实现。可见后者要方便的多。

dataframe是跟SQL一样高度结构化的数据组织形式，所有又叫做结构化API。它有行有列，在数据处理方式上，类似于数据库的table。dataframe对象也可以跟spark table非常方便的互相转换。

dataframe处理高度结构化的数据，这是优点也是缺点。优点是数据更直观，效率也高。缺点是对于半结构化，或者非结构化的数据，dataframe处理不了，此时还得借助于RDD。在机器学习的数据处理过程中，往往还是需要RDD来处理半结构化的数据。

前文也说过，spark跟hive、drill、impala这类引擎一样，都是偏重计算和分析，并不负责存数据，不像Mysql或者Hbase一样，有自己专有的存储引擎。所以在spark计算任务中，dataframe的内容往往从外部读取。spark支持非常多的外部数据源读取接口，外部存储系统包括本地文件系统、HDFS分布式文件系统、S3对象存储系统、Mysql关系型数据库等。存储格式包括JSON, CSV, Parquet, Avro等多种格式。

dataframe所支持的数据源接口，请参考这个文档。

当然，出于测试目的，也可以手工创建dataframe，如下所示。

```
scala> val df = spark.createDataFrame(List(("apple",1),("orange",2)) ).toDF("fruit","number")
df: org.apache.spark.sql.DataFrame = [fruit: string, number: int]

scala> df.show
+------+------+
| fruit|number|
+------+------+
| apple|     1|
|orange|     2|
+------+------+


scala> val df2 = sc.parallelize(List(("apple",1),("orange",2)) ).toDF("fruit","number")
df2: org.apache.spark.sql.DataFrame = [fruit: string, number: int]

scala> df2.show
+------+------+
| fruit|number|
+------+------+
| apple|     1|
|orange|     2|
+------+------+
```

如上提供了两种创建方式。一种是从spark session对象创建，spark session是启动spark-shell后，内置的变量（名字为spark），可以直接访问spark的dataframe方法。

另一种是从传统的spark context对象创建，spark context也是启动spark-shell后，内置的变量（名字为sc），可以访问spark的RDD相关属性。

toDF()里的参数，指定创建的dataframe的列的名字。关于列的数据类型，可以用printSchema方法来查看。

```
scala> df.printSchema
```

```
root
 |-- fruit: string (nullable = true)
 |-- number: integer (nullable = false)


scala> df2.printSchema
root
 |-- fruit: string (nullable = true)
 |-- number: integer (nullable = false)
```

如上，我们看到fruit列的数据格式是string，number列的数据格式是integer。这些都是spark猜出来的，没错，就是猜的。不过绝大部分情况下，spark猜的很准，所以不用担心。

如果是从scala内部数据结构转换成spark的结构，schema就不用担心，它们继承的很好。当然如果你想在转成dataframe的时候，不使用原来的数据结构，那也可以做，请参考如下。

```
scala> val fruits = List(("apple",2),("orange",3),("cherry",4))
fruits: List[(String, Int)] = List((apple,2), (orange,3), (cherry,4))

scala> case class Fruits(name:String,number:Float)
defined class Fruits

scala> val df_fruits = fruits.map{ case (x,y) => Fruits(x,y) }.toDF("fruit","count")
df_fruits: org.apache.spark.sql.DataFrame = [fruit: string, count: float]

scala> df_fruits.show
+------+-----+
| fruit|count|
+------+-----+
| apple|  2.0|
|orange|  3.0|
|cherry|  4.0|
+------+-----+

scala> df_fruits.printSchema
root
 |-- fruit: string (nullable = true)
 |-- count: float (nullable = false)
```

如上，fruits这个List的原有数据结构是List(String,Int)。我们使用case class来定义一个名字为Fruits的class，它的数据结构是(String,Float)。接下来，我们使用map将fruits的成员转成case class，然后再用toDF转成dataframe，这样数据结构也转换了。

从上述示例也可以看到，scala的List数据结构，可以直接转成dataframe，这是非常方便的。请见如下示例。

```
scala> val li = List("apple","orange","cherry")
li: List[String] = List(apple, orange, cherry)

scala> li.toDF("fruit").show
+------+
| fruit|
+------+
| apple|
```

```
|orange|
|cherry|
+------+
```

请注意只有spark的scala API才能这样用，pyspark不行。这是因为scala有一个implicit class的功能，之前我们已经介绍过。利用这个implicit class，spark可以偷偷的往scala的List class增加实例方法。例如：

```
scala> implicit class Opslist(li:List[Int]) {
     |   def shift = li.head
     | }
class Opslist

scala> val li = List(3,2,1,4)
val li: List[Int] = List(3, 2, 1, 4)

scala> li.shift
val res0: Int = 3
```

上述代码，通过implicit class，往List这个class里增加了一个实例方法shift。同理，spark的toDF直接将scala的List转换成dataframe，也是这么实现的。

如果我们从外部文件加载内容，转成spark的dataframe，那确实有必要担心数据类型的问题。这个问题说起来挺复杂的，因为在实际操作中，外部的数据源存在各种不规整，每个列都有不同的异常值存在。可以说，一份构造好的数据源，是一笔宝贵的财富，可惜这样的财富在现实中并不多见。

由于有异常值的存在，本来是数字类型的列，可能变成了字串。关于对异常值的处理，后文我们再详细说明。

对于外部加载的数据源，如何确定它的格式？spark提供了一些技巧。比如，你可以自定义schema，实现对数据格式的精准控制。也可以通过spark的infer schema技能，让它自动猜测格式。下面从一个实际案例来看这种处理方式，以及了解如何进行dataframe查询。

我们有如下dataset，它是一份真实的数据，数据内容与风暴监测相关。数据在此下载。

这是一个csv文件，数据各列的说明如下：

col 1	col2	col3	col4	col5	col6	col7	col8	col9	col10	col11	col12
ID	名字	年	月	日	小时	纬度	经度	状态	分类	风力	气压

spark可以直接从外部加载csv文件的内容到dataframe。关于csv的格式，spark可以自动infer schema，也可以手工控制schema。如果我们不放心，就手工定义schema，并对读入的数据源指定这个schema。请参考如下实现。

```
scala> val schema = """id INT, name STRING, year INT, mon INT, day INT,
     |                 hour INT, lat FLOAT, lon FLOAT, status STRING,
```

```
     |                    cate INT, wind INT, pressure INT"""
schema: String =
id INT, name STRING, year INT, mon INT, day INT,
                    hour INT, lat FLOAT, lon FLOAT, status STRING,
                    cate INT, wind INT, pressure INT

scala> val storms = spark.read.schema(schema).csv("data/storm-data.csv")
storms: org.apache.spark.sql.DataFrame = [id: int, name: string ... 10 more fields]
```

如上，我们在scala shell里做了两件事，第一件是通过DSL定义好了数据的schema，包括列的名字和列的格式。第二件是通过spark.read接口，读取外部数据源（csv文件）到spark，得到一个dataframe对象。在读取的时候，我们指定了schema属性。

再看看schema是否如同我们所设置：

```
scala> storms.printSchema
root
 |-- id: integer (nullable = true)
 |-- name: string (nullable = true)
 |-- year: integer (nullable = true)
 |-- mon: integer (nullable = true)
 |-- day: integer (nullable = true)
 |-- hour: integer (nullable = true)
 |-- lat: float (nullable = true)
 |-- lon: float (nullable = true)
 |-- status: string (nullable = true)
 |-- cate: integer (nullable = true)
 |-- wind: integer (nullable = true)
 |-- pressure: integer (nullable = true)
```

如果我们不指定schema的话，spark也会自己猜，在读取的时候设置inferSchema的属性为true。请看如下示例。

```
scala> val st2=spark.read.option("inferSchema",true).csv("data/storm-data.csv")
st2: org.apache.spark.sql.DataFrame = [_c0: int, _c1: string ... 10 more fields]

scala> st2.printSchema
root
 |-- _c0: integer (nullable = true)
 |-- _c1: string (nullable = true)
 |-- _c2: integer (nullable = true)
 |-- _c3: integer (nullable = true)
 |-- _c4: integer (nullable = true)
 |-- _c5: integer (nullable = true)
 |-- _c6: double (nullable = true)
 |-- _c7: double (nullable = true)
 |-- _c8: string (nullable = true)
 |-- _c9: integer (nullable = true)
 |-- _c10: integer (nullable = true)
 |-- _c11: integer (nullable = true)
```

这就是在读取csv的时候，让spark自己去infer schema。大多数时候这种猜测也挺准的，比如上述就是把float猜成double而已。float可以看成double的子集，所以这也没有问题。

如上，有了storms这个dataframe后，我们可以基于此做统计分析工作。

```
scala> storms.count
res13: Long = 10010

scala> storms.agg(countDistinct("name")).show
+-----------+
|count(name)|
+-----------+
|        198|
+-----------+

scala> storms.groupBy("name").count.orderBy(desc("count")).show
+---------+-----+
|     name|count|
+---------+-----+
|    Emily|  207|
|   Bonnie|  185|
|Claudette|  180|
|    Felix|  178|
|  Alberto|  170|
| Danielle|  157|
|  Isidore|  156|
|  Edouard|  149|
|    Danny|  146|
|     Ivan|  144|
|Josephine|  143|
|     Erin|  141|
|   Gordon|  140|
|     Lili|  136|
|   Gustav|  136|
|   Jeanne|  136|
|     Gert|  133|
|Gabrielle|  130|
|    Debby|  130|
|     Dean|  128|
+---------+-----+
only showing top 20 rows
```

如上三句是标准的dataframe API方法，它们的含义如下：

1. 查询dataframe总的行数
2. 查询”name”列的唯一出现的行数
3. 按”name”列分组查询，统计每组的个数，并且按降序排列

你看，groupBy和orderBy这类语句，都是高度类似SQL的。如果你熟悉SQL，那么dataframe API用起来也很顺手。再看如下语句，按年份统计风暴出现的次数。

```
scala> storms.groupBy("year").count.orderBy(desc("count")).show
+----+-----+
|year|count|
+----+-----+
|1995|  660|
|2005|  498|
|2012|  454|
|2003|  422|
|1998|  413|
```

```
|2004|  410|
|2010|  402|
|2001|  370|
|1989|  356|
|1990|  354|
|2008|  335|
|2011|  323|
|2000|  318|
|1996|  315|
|1979|  301|
|2002|  285|
|1985|  263|
|1988|  259|
|1984|  236|
|2015|  220|
+----+-----+
only showing top 20 rows
```

上述查询，还可以引入Window函数，带上风力排名。Window函数放在withColumn方法里，这个方法用来增加或者修改列。

```
scala> import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.expressions.Window

scala> storms.groupBy("year").count().withColumn("ranking",rank().over(Window.orderBy(desc("count")))).show()
+----+-----+-------+
|year|count|ranking|
+----+-----+-------+
|1995|  660|      1|
|2005|  498|      2|
|2012|  454|      3|
|2003|  422|      4|
|1998|  413|      5|
|2004|  410|      6|
|2010|  402|      7|
|2001|  370|      8|
|1989|  356|      9|
|1990|  354|     10|
|2008|  335|     11|
|2011|  323|     12|
|2000|  318|     13|
|1996|  315|     14|
|1979|  301|     15|
|2002|  285|     16|
|1985|  263|     17|
|1988|  259|     18|
|1984|  236|     19|
|2015|  220|     20|
+----+-----+-------+
only showing top 20 rows
```

如下是按年份聚合，计算每年的平均风力：

```
scala> storms.groupBy("year").agg(avg("wind")).orderBy(desc("year")).show
+----+-----------------+
|year|        avg(wind)|
+----+-----------------+
```

```
|2015| 49.40909090909091|
|2014|63.992805755395686|
|2013| 41.50990099009901|
|2012|52.940528634361236|
|2011|  52.6625386996904|
|2010| 55.87064676616915|
|2009|54.150326797385624|
|2008|55.298507462686565|
|2007|53.497652582159624|
|2006| 49.68421052631579|
|2005| 59.36746987951807|
|2004|62.048780487804876|
|2003| 58.69668246445497|
|2002| 46.80701754385965|
|2001|  51.2027027027027|
|2000| 54.65408805031446|
|1999| 61.38095238095238|
|1998| 61.58595641646489|
|1997| 48.05194805194805|
|1996| 57.73015873015873|
+----+------------------+
only showing top 20 rows
```

上述查询，同样可以引入Window函数，按照年份的风力平均值排名。

```
scala>
storms.groupBy("year").agg(avg("wind").alias("avg_wind")).withColumn("ranking",rank().over(Window.orderBy(desc("
avg_wind")) )).show()
+----+------------------+-------+
|year|          avg_wind|ranking|
+----+------------------+-------+
|2014|63.992805755395686|      1|
|2004|62.048780487804876|      2|
|1998| 61.58595641646489|      3|
|1999| 61.38095238095238|      4|
|1976| 59.90384615384615|      5|
|2005| 59.36746987951807|      6|
|2003| 58.69668246445497|      7|
|1996| 57.73015873015873|      8|
|1989| 57.65449438202247|      9|
|1992|57.108108108108105|     10|
|1981| 56.61585365853659|     11|
|1995| 55.88636363636363|     12|
|2010| 55.87064676616915|     13|
|2008|55.298507462686565|     14|
|2000| 54.65408805031446|     15|
|2009|54.150326797385624|     16|
|1977|  53.9622641509434|     17|
|1980| 53.72670807453416|     18|
|2007|53.497652582159624|     19|
|2012|52.940528634361236|     20|
+----+------------------+-------+
only showing top 20 rows
```

dataframe当然也支持select和where语句：

```
scala> storms.select("name","wind","cate").where($"cate" === -1).show
```

```
+--------+----+----+
|    name|wind|cate|
+--------+----+----+
|     Amy|  25|  -1|
|     Amy|  25|  -1|
|     Amy|  25|  -1|
|     Amy|  25|  -1|
|     Amy|  25|  -1|
|     Amy|  25|  -1|
|     Amy|  25|  -1|
|     Amy|  30|  -1|
|Caroline|  25|  -1|
|Caroline|  25|  -1|
|Caroline|  25|  -1|
|Caroline|  25|  -1|
|Caroline|  25|  -1|
|Caroline|  25|  -1|
|Caroline|  25|  -1|
|Caroline|  25|  -1|
|Caroline|  25|  -1|
|Caroline|  25|  -1|
|Caroline|  25|  -1|
|Caroline|  25|  -1|
+--------+----+----+
only showing top 20 rows
```

请特别注意在dataframe API里等于和不等于的表达式，它们看起来有点奇怪。等于是三个等号”===”，不等于是两个等号中间夹着叹号”=!=”。

```
scala> storms.select("*").where($"name" =!= "Amy").count
res16: Long = 9980

scala> storms.select("*").where($"name" === "Amy").count
res17: Long = 30
```

在查询里，如何引用一个列的值？这有好几种方法，示例如下。最简单的，就是用如下$”year”这种形式，来引用年份的值。

```
scala> storms.select("year","status","wind").where($"year" === 2000).show(5)
+----+------------------+----+
|year|            status|wind|
+----+------------------+----+
|2000|tropical depression|  25|
|2000|tropical depression|  25|
|2000|tropical depression|  25|
|2000|tropical depression|  25|
|2000|tropical depression|  25|
+----+------------------+----+
only showing top 5 rows
```

其次，可以用col()函数，来获取列的值。

```
scala> storms.select("year","status","wind").where(col("year") === 2000).show(5)
+----+------------------+----+
|year|            status|wind|
```

```
+----+------------------+----+
|2000|tropical depression|  25|
|2000|tropical depression|  25|
|2000|tropical depression|  25|
|2000|tropical depression|  25|
|2000|tropical depression|  25|
+----+------------------+----+
only showing top 5 rows
```

另外，还可以直接访问dataframe对象的key，以列名作为key来获取到列的值。

```
scala> storms.select("year","status","wind").where(storms("year") === 2000).show(5)
+----+------------------+----+
|year|            status|wind|
+----+------------------+----+
|2000|tropical depression|  25|
|2000|tropical depression|  25|
|2000|tropical depression|  25|
|2000|tropical depression|  25|
|2000|tropical depression|  25|
+----+------------------+----+
only showing top 5 rows
```

最后说明下，dataframe还支持SQL里标准的expr表达式，比如case when语句。

```
scala> storms.select("wind").withColumn("level",expr("case when wind > 50 then 'big' else 'small' end")).show
+----+-----+
|wind|level|
+----+-----+
|  25|small|
|  25|small|
|  25|small|
|  25|small|
|  25|small|
|  25|small|
|  25|small|
|  30|small|
|  35|small|
|  40|small|
|  45|small|
|  50|small|
|  50|small|
|  55|  big|
|  60|  big|
|  60|  big|
|  60|  big|
|  60|  big|
|  60|  big|
+----+-----+
only showing top 20 rows
```

如上表达式的意思是，新增一列“level”（使用withColumn函数），它的值是一个表达式。在表达式里，如果风力大于50，则设置为”big”，否则就是”small”。

第六节. Spark对异常数据的处理

上一节提及了在现实工作中，外部读入的数据源，经常面临各种异常值，比如数值列里包含了字符串。大多数情况下，spark能对异常值自我适应，比如它自动筛选掉不符合条件的行，以及自动对列的类型进行格式转换。

这听起来有点不合常规，但实际上spark真有这么强大的能力来处理各种异常数据。我们先看一个简单的示例。

```
scala> val df = spark.read.format("csv").option("inferSchema",true).option("header",true).load("tmp/sample.csv")
val df: org.apache.spark.sql.DataFrame = [name: string, sex: string ... 1 more field]

scala> df.printSchema()
root
 |-- name: string (nullable = true)
 |-- sex: string (nullable = true)
 |-- age: string (nullable = true)


scala> df.show()
+----------+------+---+
|      name|   sex|age|
+----------+------+---+
|    John M|  male| 30|
|    Lily X|female| 22|
|   Peter Z|  male| 34|
|Jennifer C|female| xx|
|  Rebeca Q|female| 18|
|  Howard P|  male| 24|
+----------+------+---+
```

如上spark从外部读取一个csv文件，这个csv包含三列：name, sex, age。我们看到age那一列有一个异常值，Jennifer的age本应该是一个数值，这里却显示为xx，是一个字符串。这就导致spark不能正确的infer schema，它把age的类型设置为string（本应该是int）。

如果我们对age进行汇聚计算，会发生什么情况呢？请参考如下代码。

```
scala> df.agg(avg("age")).show()
+--------+
|avg(age)|
+--------+
|    25.6|
+--------+
```

如上，我们对所有人求平均年龄，尽管有异常值存在，age的列类型也不正确，但结果却是正确的。这说明了两件事：

- 对于这个汇聚计算，spark自动过滤掉了异常值，比如Jennifer这一行就忽略了。
- 对于剩下的行，spark自动进行类型转换，将字串转换成数值类型，然后再进行计算。

在异常值的处理方面，spark的适应性很强大。其他的开源数据分析工具比如Apache Drill，就不能做出这样的处理。

然而，这并非说明spark总是能适应各种数据异常情况，并且做出恰当的处置。有的时候，我们不得不对数据源进行自定义的清洗和转换。请看如下真实的IMDB影视标题的数据，在此下载原始数据，是一份csv文件。

下载完后，我们使用标准的spark read方法加载这个csv。

```
scala> val df = spark.read.format("csv").option("inferSchema",true).option("header",true).load("skydrive/imdb.csv")
val df: org.apache.spark.sql.DataFrame = [tconst: string, titleType: string ... 7 more fields]

scala> df.printSchema()
root
 |-- tconst: string (nullable = true)
 |-- titleType: string (nullable = true)
 |-- primaryTitle: string (nullable = true)
 |-- originalTitle: string (nullable = true)
 |-- isAdult: string (nullable = true)
 |-- startYear: string (nullable = true)
 |-- endYear: string (nullable = true)
 |-- runtimeMinutes: string (nullable = true)
 |-- genres: string (nullable = true)
```

加载完后，打印schema看一下，全是string类型，这就说明数据存在异常。对于其中的一些字段比如isAdult, startYear等，明显应该是数值类型。

这份真实数据集，规模还比较大，有800多万行记录，分成6个分区。

```
scala> df.count()
val res5: Long = 8549346

scala> df.rdd.getNumPartitions
val res6: Int = 6
```

我们检验下数据：

```
scala> df.select("startYear").filter($"startYear".rlike("[^0-9]+")).show(3)
+---------+
|startYear|
+---------+
|       \N|
|       \N|
|       \N|
+---------+
only showing top 3 rows


scala> df.select("isAdult").filter($"isAdult".rlike("[^0-9]+")).show(3)
+--------------------+
|             isAdult|
+--------------------+
|"12th Anniversary...|
|        Robert Klein|
|"Host: Spike Jone...|
+--------------------+
```

```
only showing top 3 rows
```

如上所示，不出我们所料，startYear和isAdult列都含有异常数据。它们本应该是数值，却包含很多奇怪的字符。在检验方法里，使用了两个比较特殊的dataframe方法。一个是filter，它用来进行条件筛选。一个是rlike，类似于Mysql的rlike函数，用来进行正则表达式匹配。这个filter和rlike结合起来的意思是，筛选出非数字的值。

如果我们要按startYear分组，统计出哪一年出产的影视作品最多，那么如下编写查询。

```
scala> df.filter($"startYear".rlike("^[0-9]+$")).groupBy("startYear").count().orderBy(desc("count")).show()
+---------+------+
|startYear| count|
+---------+------+
|     2018|408237|
|     2017|405951|
|     2019|389812|
|     2016|385868|
|     2015|364535|
|     2020|357613|
|     2021|348330|
|     2014|347493|
|     2013|327611|
|     2012|306525|
|     2011|269609|
|     2010|238265|
|     2009|209788|
|     2008|198017|
|     2007|183607|
|     2006|165042|
|     2005|148430|
|     2004|134854|
|     2003|116679|
|     2002|104397|
+---------+------+
only showing top 20 rows
```

首先对dataframe目标运行filter和rlike组合查询，过滤出纯数值的年份。然后，按年份进行分组计数，并且按照计数从高到低排列，打印出结果。这里的关键在于filter做数据清洗，过滤掉异常值。

如果我们要统计成人影片与非成人影片的数量，那么如下查询。

```
scala> df.filter($"isAdult".isin("0","1")).groupBy("isAdult").count().show()
+-------+-------+
|isAdult|  count|
+-------+-------+
|      0|8284336|
|      1| 263411|
+-------+-------+
```

因为isAdult只有两个值，0或者1，其他都是异常。所以我们这里照样用filter先过滤掉异常，再运行后面的分组计数查询。

如果我们要得到一个干净彻底的dataframe要如何做呢？请参考如下执行。

```
scala> val df2 = df.filter($"isAdult".rlike("^[0-9]+$")).filter($"startYear".rlike("^[0-9]+$"))
val df2: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [tconst: string, titleType: string ... 7 more
fields]

scala> val df3 = df2.withColumn("isAdult",$"isAdult".cast("int")).withColumn("startYear",$"startYear".cast("int"))
val df3: org.apache.spark.sql.DataFrame = [tconst: string, titleType: string ... 7 more fields]

scala> df3.printSchema()
root
 |-- tconst: string (nullable = true)
 |-- titleType: string (nullable = true)
 |-- primaryTitle: string (nullable = true)
 |-- originalTitle: string (nullable = true)
 |-- isAdult: integer (nullable = true)
 |-- startYear: integer (nullable = true)
 |-- endYear: string (nullable = true)
 |-- runtimeMinutes: string (nullable = true)
 |-- genres: string (nullable = true)
```

如上三句代码的意思是：

1. 对df目标连续运行2个filter，过滤掉isAdult和startYear的异常值，结果产生一个新的dataframe，赋值给df2。
2. 对df2的两个列，isAdult和startYear，使用withColumn方法修改它的列类型。执行类型修改的是cast方法，它将新的列类型广播为int类型。修改的结果是产生一个新的dataframe，赋值给df3。
3. 打印df3的schema，我们可以看到isAdult和startYear两个列，已经修改为整数类型。

这就是一个标准的数据清洗的过程。最后我们确认下产生的干净dataframe的数据情况。

```
scala> df3.agg(countDistinct("isAdult")).show()
+--------------+
|count(isAdult)|
+--------------+
|             2|
+--------------+
```

我们看到，新的dataframe的isAdult列，没有包含乱七八糟数据了，它只应有0和1两个值。

再说一下spark对空值（null）的处理。我们查看如下数据。

```
scala> df.show()
+-----------+------+----+
|       name|   sex| age|
+-----------+------+----+
|    John M|  male|  30|
|    Lily X|female|  22|
|   Peter Z|  male|  34|
|Jennifer C|female|null|
|  Rebeca Q|female|  18|
|  Howard P|  male|  24|
+-----------+------+----+
```

这个dataframe现在包含了一个空值，Jennifer的age为空（null）。

如何过滤掉空值呢？请使用isNull和isNotNull方法。注意跟Mysql的空值一样，空值不能直接用来比较。比如，如下查询是有问题的。

```
scala> df.filter($"age" === null).show()
+----+---+---+
|name|sex|age|
+----+---+---+
+----+---+---+
```

要筛选出空值，正确的查询是：

```
scala> df.filter($"age".isNull).show()
+----------+------+----+
|      name|   sex| age|
+----------+------+----+
|Jennifer C|female|null|
+----------+------+----+
```

对应的，如下筛选非空值：

```
scala> df.filter($"age".isNotNull).show()
+--------+------+---+
|    name|   sex|age|
+--------+------+---+
|  John M|  male| 30|
|  Lily X|female| 22|
| Peter Z|  male| 34|
|Rebeca Q|female| 18|
|Howard P|  male| 24|
+--------+------+---+
```

总之，在实际情况中，你可能会碰到多种数据异常情况。这些情况要分开来看。有的spark能自适应处理，有的并不能。你可能要手工对数据进行清洗，以及对数据类型进行转换。得到干净有效的数据，永远是数据工作中最重要的一个环节。

第七节. 自定义函数与Mysql数据源

Spark的SQL引擎支持自定义函数（UDF），用法很简单，只要写好一个函数，并且注册到SQL引擎的函数空间就好。请见如下示例。

```
scala> val f:String=>Int = x => x.size
f: String => Int = $Lambda$4580/0x0000000841df7840@188da0e2

scala> val strSize = udf(f)
```

```
strSize: org.apache.spark.sql.expressions.UserDefinedFunction =
SparkUserDefinedFunction($Lambda$4580/0x0000000841df7840@188da0e2,IntegerType,List(Some(class[value[0]:
string])),Some(class[value[0]: int]),None,false,true)

scala> val df = sc.textFile("/etc/lsb-release").toDF("line")
df: org.apache.spark.sql.DataFrame = [line: string]

scala> df.select("line").withColumn("line_size",strSize($"line")).show(false)
+----------------------------------------+---------+
|line                                    |line_size|
+----------------------------------------+---------+
|DISTRIB_ID=Ubuntu                       |17       |
|DISTRIB_RELEASE=20.04                   |21       |
|DISTRIB_CODENAME=focal                  |22       |
|DISTRIB_DESCRIPTION="Ubuntu 20.04.4 LTS"|40       |
+----------------------------------------+---------+
```

如上各行的操作说明如下：

1. 编写一个匿名函数，赋值给f，这个匿名函数的作用就是算字串的长度
2. 将匿名函数f注册到spark的SQL函数空间
3. 读取一个文本文件，转换成dataframe，df的每个元素就是文本文件的一行
4. 从df里select出行的内容，并增加一列，该列的值由自定义函数算出来

然后我们看到的结果就是每行的内容，以及每行的字符数。

之前我们说过，spark的dataframe支持非常多的外部数据源导入，例如csv, json, s3等。这里再演示下如何导入Mysql的数据源。

首先，到Mysql的官网下载mysql-connector-java.

选择ubuntu操作系统，下载后是一个.deb文件。使用如下命令安装这个package：

```
$ sudo dpkg -i mysql-connector-java_8.0.28-1ubuntu20.04_all.deb
```

然后，使用如下命令查看package文件的安装路径：

```
$ dpkg --listfiles mysql-connector-java
/.
/usr
/usr/share
/usr/share/doc
/usr/share/doc/mysql-connector-java
/usr/share/doc/mysql-connector-java/CHANGES.gz
/usr/share/doc/mysql-connector-java/INFO_BIN
/usr/share/doc/mysql-connector-java/INFO_SRC
/usr/share/doc/mysql-connector-java/LICENSE.gz
/usr/share/doc/mysql-connector-java/README
/usr/share/doc/mysql-connector-java/changelog.Debian.gz
/usr/share/doc/mysql-connector-java/copyright
/usr/share/java
/usr/share/java/mysql-connector-java-8.0.28.jar
```

我们需要手工将上述列出来的最后一个文件，拷贝两份，一份放到spark的jars目录，一份放到你启动spark-shell时的本地目录。

```
$ cp /usr/share/java/mysql-connector-java-8.0.28.jar /opt/spark/jars/
$ cp /usr/share/java/mysql-connector-java-8.0.28.jar .
```

然后，按如下方式启动spark-shell:

```
$ spark-shell --jars mysql-connector-java-8.0.28.jar
```

现在，假设你在本机已经安装了Mysql，并且创建了库和表，设置了正确的用户权限。那么就可以从spark-shell里加载Mysql的数据源了。

```
scala> val mysql_df = (spark
     |       .read
     |       .format("jdbc")
     |       .option("url", "jdbc:mysql://127.0.0.1:3306/fruits")
     |       .option("driver", "com.mysql.jdbc.Driver")
     |       .option("dbtable", "myfruits")
     |       .option("user", "wes")
     |       .option("password", "fruit123")
     |       .load() )
mysql_df: org.apache.spark.sql.DataFrame = [id: int, name: string ... 1 more field]

scala> mysql_df.select("*").show
+---+------+------+
| id|  name|number|
+---+------+------+
|  1| apple|    20|
|  2|orange|    12|
|  3|banana|    10|
|  4|cherry|    15|
|  5|  plum|     9|
|  6| lichi|    30|
|  7| mongo|     5|
|  8| berry|    28|
|  9| grape|     5|
| 10|tomato|    25|
+---+------+------+
```

上述我们使用spark.read的方式读取Mysql数据源，它的用法其实跟我们前面提到的加载csv的方法差不多，只不过选项各有不同。

上述选项里，第一个option指定jdbc的路径，第二个option指定驱动，第三个option指定表名，第四个option指定用户名，第五个option指定密码。最后调用load方法，加载数据。

数据加载到mysql_df这个dataframe对象，然后就可以用之前我们介绍的datafraem API方法来操作数据了。

最后说明下，spark的dataframe方法非常多，体系很庞大，请用google搜索“spark sql functions”来获取全部函数列表。

第八节. Spark与Hive集成

我个人使用Spark与Hive集成的开发环境并不多。不过按照官网的示例，很顺畅就能在Spark里访问Hive的数据，所以这里也记录一下。

首先，我在本机安装了Spark、Hadoop、Hive的完整环境。Hadoop生态的版本号挺重要的。因为版本问题导致各软件协作不通，在实际中挺常见的。

我部署的各个软件的版本如下：

```
$ java -version
java version "1.8.0_321"

$ hadoop version
Hadoop 3.3.2

$ hive --version
Hive 3.1.2

mysql> select version();
+-------------------------+
| version()               |
+-------------------------+
| 5.7.37-0ubuntu0.18.04.1 |
+-------------------------+
1 row in set (0.00 sec)

$ spark-submit --version
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.2.1
      /_/

Using Scala version 2.13.5, Java HotSpot(TM) 64-Bit Server VM, 1.8.0_321
```

我这里的操作系统是ubuntu 18.04 x64。上述Mysql只用来做Hive的metastore的存储服务，没有其他作用。

Hive和Spark都部署为local模式，HDFS采用伪分布式模式。它们都没有真正的集群，是在单机下运行。

我的用户目录下的.bash_profile环境变量设置如下：

```
export JAVA_HOME=/opt/jdk/jdk1.8.0_321
```

```
export HADOOP_HOME=/opt/hadoop
export PATH=$HADOOP_HOME/bin:$PATH

export SPARK_HOME=/opt/spark
export PATH=$PATH:/opt/spark/bin:/opt/spark/sbin
export PYSPARK_PYTHON=/usr/bin/python3

export HIVE_HOME=/opt/hive
export PATH=$HIVE_HOME/bin:$PATH
```

它们也没有特殊的意义，就是指定了java, hadoop, spark, hive的路径环境变量。

如下命令是必须要执行的：

```
$ cp mysql-connector-java-8.0.27.jar /opt/hive/lib
$ cp mysql-connector-java-8.0.27.jar /opt/spark/jars
```

这里将Mysql的JDBC驱动放到相应的库目录下。因为Hive和Spark都要访问Mysql的metastore服务，所以这个驱动必不可少。驱动文件在Mysql官网下载。

如果你之前没有配置过Hive，那么可以参考如下我的配置。

Hive的配置文件是/opt/hive/conf/hive-site.xml，它的内容如下：

```
<configuration>
    <property>
        <name>javax.jdo.option.ConnectionURL</name>
        <value>jdbc:mysql://localhost/spark?createDatabaseIfNotExist=true</value>
        <description>metadata is stored in a MySQL server</description>
    </property>
    <property>
        <name>javax.jdo.option.ConnectionDriverName</name>
        <value>com.mysql.jdbc.Driver</value>
        <description>MySQL JDBC driver class</description>
    </property>
    <property>
        <name>javax.jdo.option.ConnectionUserName</name>
        <value>spark</value>
        <description>user name for connecting to mysql server</description>
    </property>
    <property>
        <name>javax.jdo.option.ConnectionPassword</name>
        <value>spark123</value>
        <description>password for connecting to mysql server</description>
    </property>

</configuration>
```

上述配置的意思是，使用Mysql JDBC来管理metastore存储服务，存储路径是Mysql数据库里名为spark的库。Mysql用户名是spark，密码是spark123。当然，在Mysql里这几个属性要事先配置好。

Hadoop的核心配置文件是/opt/hadoop/etc/hadoop/core-site.xml，它的内容如下：

```
<configuration>
   <property>
      <name>fs.defaultFS</name>
      <value>hdfs://localhost:9000</value>
   </property>

  <property>
    <name>hadoop.proxyuser.pyh.groups</name>
    <value>*</value>
  </property>

  <property>
    <name>hadoop.proxyuser.pyh.hosts</name>
    <value>*</value>
  </property>
</configuration>
```

上述指定了HDFS的namenode的地址和端口，以及两个安全配置。这两个针对本机登陆用户（名为pyh）的proxyuser配置必须设置好，否则Hive访问HDFS存在权限问题。

配置完上述后，运行如下命令来初始化metastore服务：

```
$ schematool -dbType mysql -initSchema
```

这里要注意下是否成功，不成功的话检查是啥原因。比如没有拷贝JDBC驱动，或者Mysql权限不对。

为了验证Hive使用Mysql作为metastore是否成功，可以运行hive命令进入交互式shell，创建一个表，比如hivemysql。然后登陆Mysql，运行如下查询，看这个表是否存在：

```
mysql> select TBL_NAME,TBL_TYPE from TBLS;
+-----------+---------------+
| TBL_NAME  | TBL_TYPE      |
+-----------+---------------+
| hivemysql | MANAGED_TABLE |
| ppl       | MANAGED_TABLE |
+-----------+---------------+
2 rows in set (0.01 sec)
```

如上，就说明Mysql管理了Hive的元数据存储服务。

最后，将如下几个文件，拷贝到Spark安装目录的conf子目录下：

```
/opt/hive/conf/hive-site.xml
/opt/hadoop/etc/hadoop/core-site.xml
/opt/hadoop/etc/hadoop/hdfs-site.xml
```

运行spark-shell进入Spark的交互式编程环境，输入如下代码：

```
import java.io.File
import org.apache.spark.sql.{Row, SaveMode, SparkSession}
val warehouseLocation = new File("spark-warehouse").getAbsolutePath

val spark = SparkSession
  .builder()
  .appName("Spark Hive Example")
  .config("spark.sql.warehouse.dir", warehouseLocation)
  .enableHiveSupport()
  .getOrCreate()

import spark.implicits._
import spark.sql
```

这就初始化好了Spark SQL与Hive Session的集成环境。

我们运行如下查询，来获取到Hive里存储的数据。

```
scala> sql("use default")
val res0: org.apache.spark.sql.DataFrame = []

scala> sql("show tables").show()
+---------+---------+-----------+
|namespace|tableName|isTemporary|
+---------+---------+-----------+
|  default|hivemysql|      false|
|  default|      ppl|      false|
+---------+---------+-----------+


scala> sql("desc ppl").show()
+--------+---------+-------+
|col_name|data_type|comment|
+--------+---------+-------+
|    name|   string|   null|
|     sex|   string|   null|
|    born|     date|   null|
|     zip|      int|   null|
|   email|   string|   null|
|     job|   string|   null|
|  salary|    float|   null|
+--------+---------+-------+


scala> sql("select job,avg(salary) as avg_sa from ppl group by job order by avg_sa desc").show(10,false)
+-----------------------------------+-----------------+
|job                                |avg_sa           |
+-----------------------------------+-----------------+
|IT Manager                         |17682.772872552938|
|Computer Systems Administrator     |17663.631755044105|
|Cashier                            |17620.562191510366|
|Plumber                            |17612.91147476341 |
|Veterinary Technologist & Technician|17608.890145072535|
|Fabricator                         |17604.09839886502 |
|Computer Programmer                |17591.551812450747|
|Middle School Teacher              |17590.884146341465|
|Physical Therapist                 |17590.864870299618|
```

```
|Bus Driver                         |17581.65017168249 |
+-----------------------------------+------------------+
only showing top 10 rows
```

上述SQL显而易见，跟标准Mysql的操作差不多。第一句选择数据库，第二句展示库里有哪些表，第三句描述表的结构，第四句对表执行一个标准的SQL查询语句。

这样，我们就完成了在Spark里查询Hive数据的任务。

第九节. 使用spark-submit提交任务

上面我们介绍的操作都是在spark-shell里的。下面演示从客户端应用程序提交job到spark。

首先，创建一个项目目录，比如spark，然后进入这个目录。

```
$ mkdir spark
$ cd spark
```

创建build.sbt文件，这个文件的格式，我们已在之前scala的章节里讲过。文件内容如下。

```
name := "Simple Project"

version := "1.0"

scalaVersion := "2.12.15"

libraryDependencies += "org.apache.spark" %% "spark-sql" % "3.2.1"
```

请注意：目前客户端提交的程序，尚不能用2.13版本的scala，请用scala 2.12。

创建src/main/scala子目录，在子目录里创建SimpleApp.scala代码文件，内容如下：

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

object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "/etc/sysctl.conf"
    val spark = SparkSession.builder.appName("Simple Application").getOrCreate()
    val logData = spark.read.textFile(logFile).cache()
    val numAs = logData.filter(line => line.contains("a")).count()
    val numBs = logData.filter(line => line.contains("b")).count()
    println(s"Lines with a: $numAs, Lines with b: $numBs")
    spark.stop()
  }
}
```

在项目根目录下，我们看到的路径结构应该是这样的：

```
$ tree
.
├── build.sbt
└── src
    └── main
        └── scala
            └── SimpleApp.scala
```

在项目根目录里，运行sbt package打包和编译：

```
$ sbt package
…
[info]   Compilation completed in 25.413s.
[success] Total time: 34 s, completed Mar 16, 2022, 4:17:47 PM
```

看到上述提示，就是编译成功了。下面使用spark-submit来提交任务：

```
$ spark-submit --class "SimpleApp" --master local[2] target/scala-2.12/simple-project_2.12-1.0.jar
```

spark-submit参数解释如下：

1. –class: class名，就是源代码里用object关键字定义的那个单态类。
2. –master: 指定spark的主服务器地址，我们采用本地部署，所以是local。后面的local[2]根据cpu的个数来配置，因为我的虚拟机有两个cpu，所以设置为2。
3. 接着是编译生成的jar文件，在这里指定它的路径。

有一大堆的日志打印到屏幕，不用理它们，这些日志可以配置关掉的。看任务的输出：

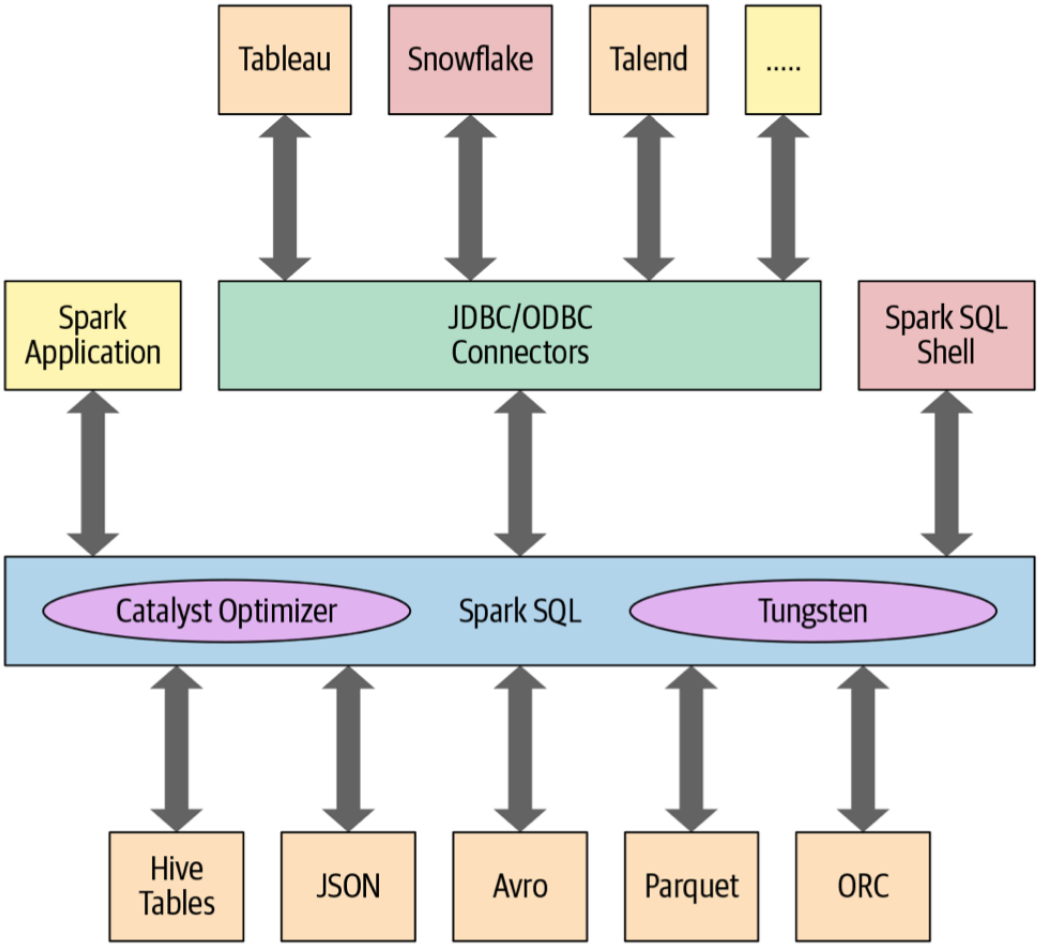
```
Lines with a: 41, Lines with b: 12
```

就说明成功了。这个任务的目的是，统计/etc/sysctl.conf文件，其中包含字母a的有多少行，包含字母b的有多少行。

第十节. Spark SQL和流简介

Spark SQL是SQL引擎上的应用，它跟dataframe背后的机制一样，只是API的表现形式不同。大多数数据开发人员都十分熟悉SQL，所以Spark SQL应运而生。它的表现形式就是SQL语句，基本兼容Hive SQL.

如下是Spark SQL的生态架构图：



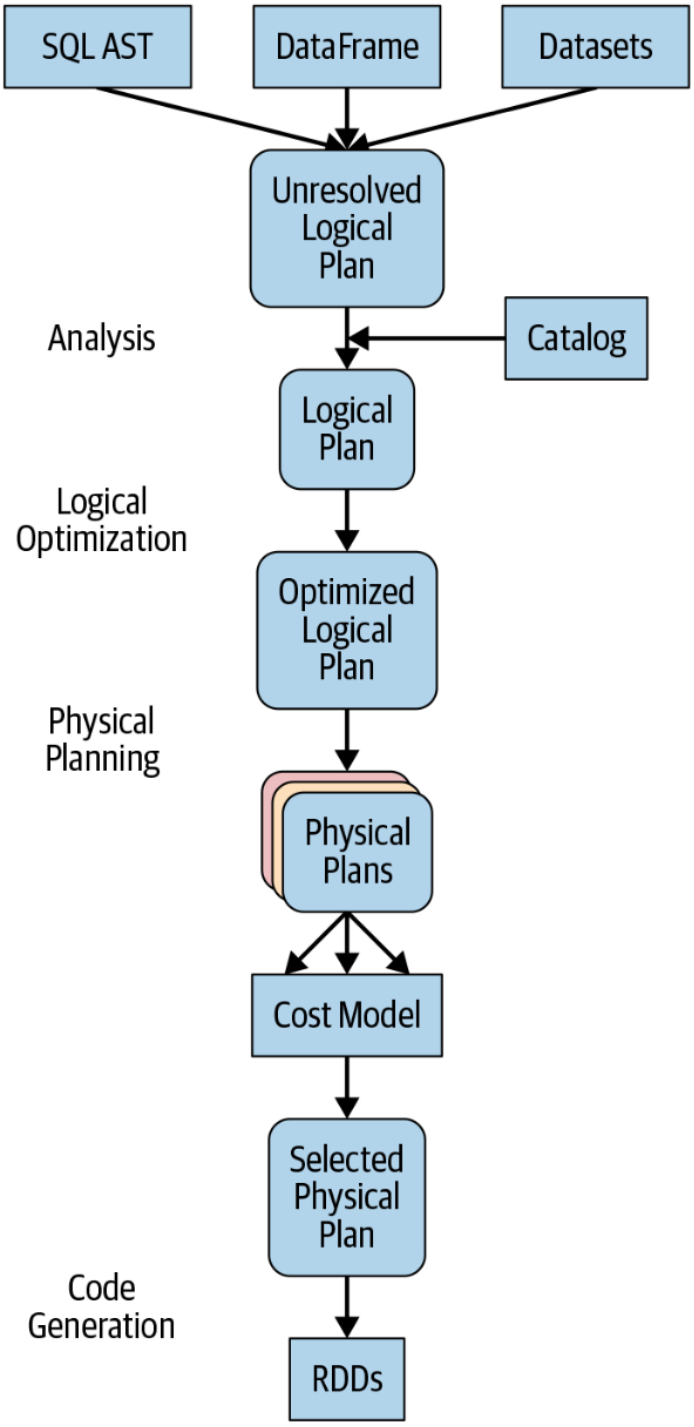
上述架构简单说明如下：

1. Spark SQL引擎底层支持多种数据源，包括Hive, JSON, Avro等。
2. Spark SQL是一套高度优化过的解析和计算框架，spark应用和spark-sql shell可以直接访问它。
3. 通过JDBC/ODBC连接器，第三方的应用比如Snowflake也可以访问Spark SQL。

Spark SQL为什么很快？主要是它的优化工作做得好。有一个专门的优化器叫做Catalyst。这个优化器在SQL解析和执行的层面做了大量优化工作。包括：

1. 语法分析
2. 逻辑优化
3. 物理规划
4. 代码产生

它的流程图如下：



对上述流程的简单解释如下：

1. 语法分析：分析用户的SQL或者dataframe查询，产生未解析的逻辑规划（此时可能列不存在或者类型不正确）。接着在catalog对象（存放元数据）的参与下，未解析逻辑规划转换成可执行的逻辑规划。
2. 逻辑规划优化：基于标准规则的优化，包括常量折叠、谓词下推、投影修剪、空传播、布尔表达式简化和其他规则。打个比如，谓词下推将用户的SQL查询推到对应的数据存储的节点上，减少不必要的网络传输，使查询效率最高。
3. 物理规划：有个成本模型来对每个逻辑规划，衡量其物理成本。最后从多个物理规划中，选择一个最优的应用给逻辑规划。

4. 代码产生：生成Java字节码以在集群中的每台机器上运行，它用到了scala的一个特殊功能叫做Quasi quotes。

正是因为catalyst优化器的参与，Spark SQL才大放异彩，执行效率很高。并且这套引擎与语言无关，除非你要编写自定义的UDF，那么用python或者用scala来编写应用层代码，效率都是差不多的。

Spark SQL用起来很简单，请参考如下示例。

```
scala> val schema = """id INT, name STRING, year INT, mon INT, day INT,
     |                 hour INT, lat FLOAT, lon FLOAT, status STRING,
     |                 cate INT, wind INT, pressure INT"""
schema: String =
id INT, name STRING, year INT, mon INT, day INT,
                hour INT, lat FLOAT, lon FLOAT, status STRING,
                cate INT, wind INT, pressure INT

scala> val storms = spark.read.schema(schema).csv("data/storm-data.csv")
storms: org.apache.spark.sql.DataFrame = [id: int, name: string ... 10 more fields]

scala> storms.createOrReplaceTempView("storm_table")
```

首先我们加载外部数据（csv文件）到dataframe，这个细节前文已介绍。

接着我们把这个dataframe另存为一个临时表，表名字为”storm_table”.

然后，针对这个临时表，我们就可以执行一系列SQL查询操作了。比如：

```
scala> spark.sql("with t1 as (select name,year,wind,pressure from storm_table) select * from t1 where wind > 50
order by wind desc").show
+--------+----+----+--------+
|   name|year|wind|pressure|
+--------+----+----+--------+
|Gilbert|1988| 160|     888|
|  Wilma|2005| 160|     882|
|   Rita|2005| 155|     895|
|   Rita|2005| 155|     897|
|  Mitch|1998| 155|     910|
|Gilbert|1988| 155|     889|
|  Mitch|1998| 155|     905|
|  Mitch|1998| 150|     917|
| Andrew|1992| 150|     922|
|  Mitch|1998| 150|     922|
|  David|1979| 150|     926|
|  Felix|2007| 150|     935|
|  Anita|1977| 150|     926|
|Katrina|2005| 150|     902|
|  Wilma|2005| 150|     892|
|   Dean|2007| 150|     907|
|  Felix|2007| 150|     930|
|  David|1979| 150|     924|
|   Rita|2005| 150|     897|
|   Dean|2007| 150|     905|
+--------+----+----+--------+
only showing top 20 rows
```

SQL语句我想每个人都很熟悉，就不在此赘述。

最后，谈一谈spark的流（streaming）。spark是通过微批次（batch）来实现流处理的，可以自定义微批次的size，比如一秒执行一次。

它不算真正意义上的流处理框架，至少不是scala的fs2这样的流框架。不过大多数场景下，这种基于batch的方式也够用了。而且，spark是流批一体化进行，无疑简化了API，让用户用起来更简单。

spark有两种流处理方式，一种是基于传统RDD的DStream ,文档在此。一种是基于新的SQL引擎的结构化流 ,文档在此。后者用起来更简单，但对输入数据的要求更高，必须是高度结构化的数据。如果是非结构化或者半结构化的数据，还得使用基于RDD的DStream.

我两者都用过，并且把部署和执行写了相关博客，博客内容在如下链接 DStream的使用，结构化流的使用。

spark还支持与Apache Kafka的直接集成，可以从kafka上读取流来进行处理。文档在此。

如果你对spark的技术深度有兴趣，在阅读了databricks官方的Learning Spark后，可以接着阅读High Performance Spark这本书，里面讲了不少技术细节，相信可以加深你的理解。