中国电信 CHINA TELECOM

函数式与RDD

邢森@上海理想

2016-9-10

什么函数式(一个问题)

中国电信 CHINA TELECOM

统计文本文件中单词出现的次数

什么函数式(常规解法)

- 读取文件为一个字符串
- 把字符串按“非字符”分割（得到没有任何标点符号的“单词列表”）
- 定义一个map计数器，key是单词，value是次数
- 循环“单词列表”修改map计数器

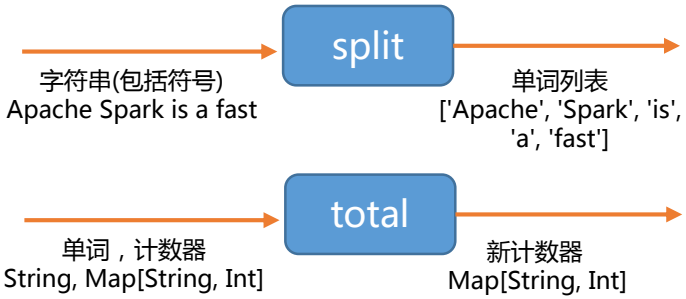
```
object Main {
  def main(args: Array[String]): Unit = {
    val s = Source.fromFile("/Users/fireflyc/1.txt").mkString

    var map = Map.empty[String, Int]

    s.split("\\W+").foreach(word => {
      val count = map.getOrElse(word, 0) + 1

      map += (word -> count)
    })

    println(map)
  }
}
```

定义“计数器”

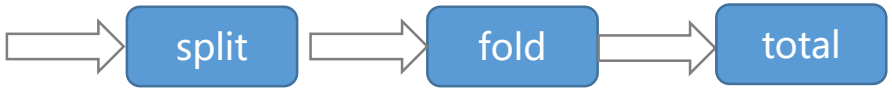
此处就是“副作用”

什么函数式(函数式解法)

- 需要一个List数据结构，里面存放“单词”
- 循环“单词列表”重新计算计数器，然后返回




```
object FPMain {
  def total(word: String, map: Map[String, Int]) = {

    var newMap = Map.empty[String, Int] ++ map
    val count = newMap.getOrElse(word, 0) + 1

    newMap += (word -> count)
    newMap
  }

  def main(args: Array[String]): Unit = {
    val s = Source.fromFile("/Users/fireflyc/1.txt").mkString

    val out = s.split("\\W+")
      .foldRight(Map.empty[String, Int])(total)
    println(out)
  }
}
```



什么函数式(函数式解法)

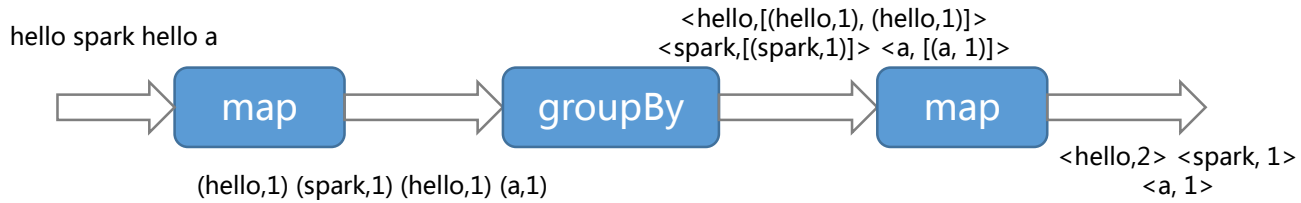
- 需要一个List数据结构，里面存放“单词”
- 一个List，里面存放(Word, count)
- 通过分组，重新得到一个List。(Word，[count1, count2])
- 得到一个新的List，是通过上一个List的value.reduce累加出来的

```
def main(args: Array[String]): Unit = {
  val s = Source.fromFile("/Users/fireflyc/1.txt").mkString

  val out = s.split("\\W+").map(word => (word, 1))
    .groupBy(_._1)
    .map { tuple =>
      tuple._2.reduce { (a, b) => (a._1, a._2 + b._2) }
    }

  println(out)
}
```

(hello,1) (spark, 1)...

<hello,[(hello,1), (hello,1)]>....

<hello,2> <spark, 1> <a, 1>

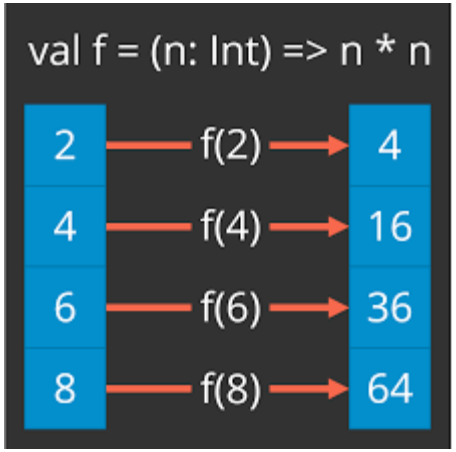


什么函数式(函数式思维)

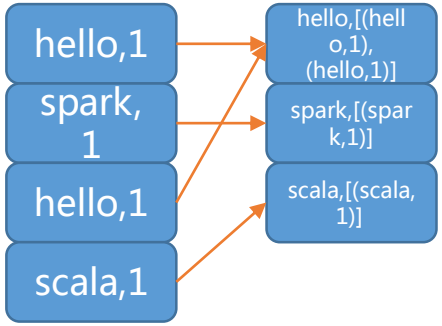
- **数据+算法（操作）=函数式**

- **特点**
  - 定义数据结构
  - 从上之下分解问题
  - 确定每个步骤的输入，输出
  - 函数式重用的是“操作”

比如上面的例子中，函数式语言提供给我们了map、groupBy这两个“操作”
它们的定义是这样的



Map
1:1的函数，输入多少数据输出多少数据，参数用于“转换”



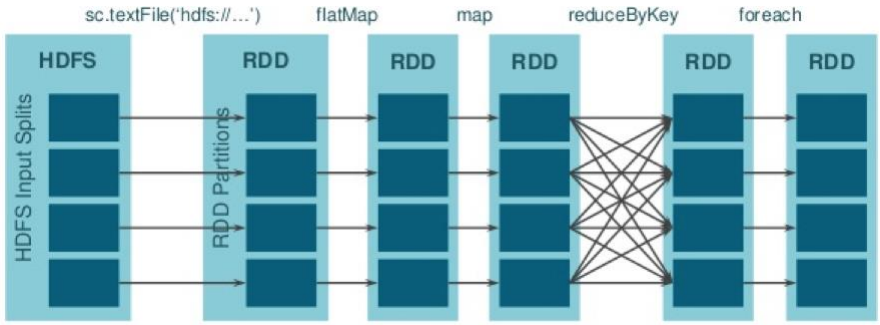
groupby
通过某个条件对数据进行分组(输出小于等于输入)，参数定义“分组条件”

函数式与RDD

RDD

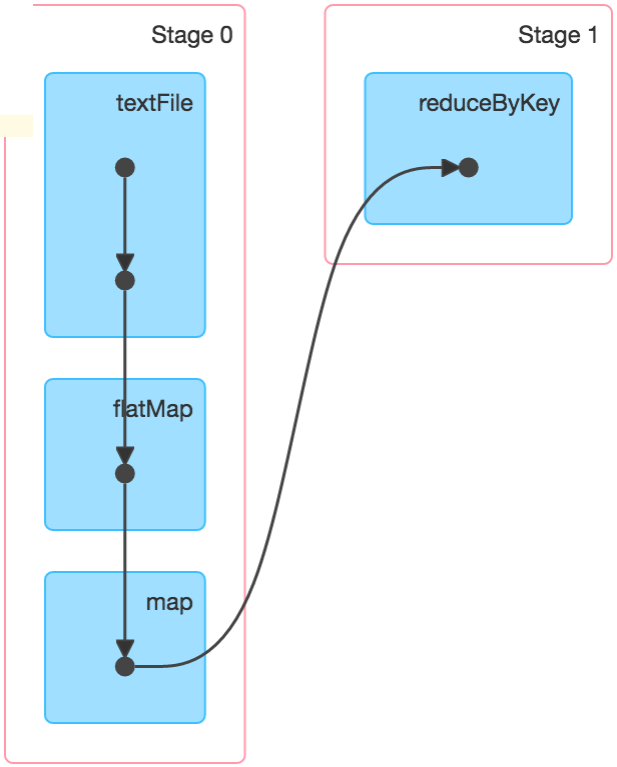
- RDD就是一组数据+一组“操作”
- RDD的操作细分为“转换”、“动作”
- “动作”会产生一个Job
- 一个Job会产生多个“stage”，划分依据是“宽依赖”“窄依赖”。由宽变窄或者由窄变宽都会产生新的stage

- 如果仔细观察RDD的架构（比如右侧单词统计的RDD变化图）不难发现它本质和“函数式”的图很像，二者都是 数据 + “操作”



函数式与RDD

```
val counts = lines.flatMap(line => line.split("\\W+"))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
```



Spark版本的单词统计

- 程序的思路和我们的版本一样
- 最后会生成一个Job
- 一个Job会生成两个stage，map都是宽依赖，reduce开始变窄，所以有两个stage
- 每个stage由一组“线程”负责执行，每个线程叫一个Task

函数式与Spark

Spark提供了一种DSL

- Spark其实提供了一种DSL机制
- RDD是对数据和操作的抽象，对于我们来说我们执行执行指定的转换或者动作就可以完成计算
- 转换和动作本质上是函数式的“操作”，所以如果没有函数式思维是很难编写Spark程序的
- 举个例子，我们按照传统的思路编写“单词统计”几乎是不可能被移植到Spark上的。如果没有函数式思维，不熟悉提供的“操作”有哪些是很难想到通过map，reduce之类的“抽象操作”来实现的

Spark是函数式的再次应用

- Hadoop的M/R其实是函数式操作在大数据中一次很成功的应用，我们几乎用最简单的两个操作解决了80%的问题
- Spark提供了更加丰富的“函数式操作”，这几乎可以解决我们100%的问题
- 无论是写Hadoop还是Spark，如果不熟悉函数式思维是很难用这二者去解决问题的

中国电信
CHINA TELECOM

谢谢