

## 译序

嗯，姑且腆着脸叫译序吧。

说下缘起，因为之前工作忙也没顾上看病，后来严重了，被迫割了一刀。所以现在闲赋在家，看看书，顺便就写下来吧，也算没白看。因为医生不建议久坐，所以我是站在电脑面前打字的，敲的有点慢，站久了腿疼，你懂的。计划是翻译第三章到第九章，实际上完成到第八章。

下面说点正经的。请注意：翻译的 *Learning Spark* 是 2015 年出版的第一版，不是之前的预览版，详见封面和第二页的出版信息，对照原文的时候不要弄错。翻译的时候，我尽量保持内容和原作所在的页码一致，方便各位对照原文帮我纠错。

没译的章节是第一章 *Introduction to Data Analysis with Spark*，第二章 *Downloading Spark and Getting Started*，第九章 *Spark SQL*，第十章 *Spark Streaming*，第十一章 *Machine Learning*。其中第一章和第二章好像有网友翻译过了，可以放狗找找。第九，十，十一章是 *Spark* 的上层应用。因为没有讲我最喜欢的 *GraphX*，所以这几章我也不想再继续翻了，哼！有兴趣的可以看看原书吧。

另外，本书算是 *Spark* 的入门内容，看本书的应该也是初学者。推荐 Databricks 提供的一个压缩包：<http://training.databricks.com/workshop/usb.zip>。这是个 *Spark* 的练习环境，有代码样例，测试数据。只需要你有个单机的 Linux 环境，然后下载这个压缩包，解压后就能用了。

很明显英语不是我的母语，所以翻译的不正确请原谅，我只能说我尽力了，谢谢！当然，如果你愿意告诉我哪里错了，那真是太感谢了，好人一生平安~！

如果有啥要指教我的，可以 email: [coding\\_hello@126.com](mailto:coding_hello@126.com)  
有更新的话，会放在我的 CSDN 博客: [http://blog.csdn.net/coding\\_hello/article/](http://blog.csdn.net/coding_hello/article/)  
或者直接在 CSDN 资源里搜书名吧，那里也会放一份方便查找下载的。

祝各位阅读愉快，工作之余注意身体！

对了，看别人的译著好像一般也都有个感谢的部分会开个单章。现在内容也不全，我就不开单章感谢了，但是仍然要感谢下我的爸妈，辛苦把我拉扯大了现在还要照顾我这病号。确实应该去相亲找个媳妇了，一把年纪了还要父母来照顾。

唉，loser！不多说了，我得去各大婚恋网站看看收件箱里有咩妹纸对我有兴趣~

咦，那个妹纸，不要走啊，了解一下嘛~~ 都能谈啊~~ 喂~ 喂喂~~~

2015-10-07

# 目 录

第三章 RDD 编程.....	23
RDD 基础.....	23
创建 RDD.....	25
RDD 操作.....	26
传递函数到 Spark.....	30
常见的变换和动作.....	34
持久化（缓存）.....	44
总结.....	46
第四章 处理键值对(Key/Value Pairs).....	47
动机.....	47
创建 Pair RDD.....	48
对 Pair RDD 的变换.....	49
Pair RDD 上可用的动作.....	60
数据分区（高级）.....	61
总结.....	70
第五章 加载和保存你的数据.....	71
动机.....	71
文件格式.....	72
文件系统.....	89
结构化数据和 Spark SQL.....	91
数据库.....	93
总结.....	98
第六章 Spark 高级编程.....	99
介绍.....	99
累加器.....	100
广播变量.....	104
基于分区工作.....	107
管道输出到外部程序.....	109
数值 RDD 操作.....	113
总结.....	115
第七章 在集群上运行.....	117
介绍.....	117
Spark 运行架构.....	117
用 spark-submit 分发应用.....	121
打包你的代码和依赖.....	123
Spark 应用内部和 Spark 应用之间的调度.....	128
集群管理器.....	129
该使用哪个集群管理器？.....	138
总结.....	139
第八章 Spark 的调优和调试.....	141
用 SparkConf 配置 Spark.....	141

运行组件: Jobs, Tasks, Stages.....	145
寻找信息.....	150
关键性能考量.....	155
总结.....	160

## 第三章

---

# RDD 编程

本章介绍 Spark 处理数据的核心抽象：弹性分布式数据集（RDD）。RDD 简单来说就是元素的分布式集合。在 Spark 中，所有的工作都被表达为创建新 RDD，对已存在的 RDD 做变换，或者对 RDD 调用某些操作来计算得到一个结果。在底层，Spark 将包含在 RDD 中的数据自动分布到你的整个集群，并将你对其执行的操作并行化。

数据科学家和工程师都应该阅读本章，因为 RDD 是 Spark 的核心概念。我们强烈建议你在交互式 Shell 中尝试一些示例（见本书第 11 页的“Spark 的 Python 和 Scala Shell 简介”）。另外，本章所有的代码都在本书的 [github](#) 库有下载。

### RDD 基础

Spark 中的 RDD，简单来说就是所有对象的一个不可变的分布式集合。每个 RDD 都被分割为多个分区，这就可以在集群的不同节点上进行计算。RDD 可以包含任何 Python, Java, Scala 对象类型，包括用户自定义类型。

用户可以用两种方式创建 RDD：通过加载一个外部数据集，或者在驱动程序中分发一个对象集合（如 list 或 set）。如同示例 3-1 展示的，我们知道了使用 `SparkContext.textFile()` 函数加载一个文本文件作为一个字符串 RDD。

*示例 3-1：在 Python 中用 `textFile()` 函数创建一个字符串 RDD*

```
>>> lines = sc.textFile("README.md")
```

RDD 一旦创建好了，可以提供两种不同类型的操作：变换(transformation)和动作(action)。变换是从前一个 RDD 构造出一个新的 RDD。例如，有一个常见的变换是用谓词匹配来过滤数据。在我们之前的文本文件的示例中，我们可以用这个变换来创建一个新的 RDD，这个 RDD 容纳的数据是只包含了单词“Python”的字符串。如示例 3-2 所示：

*示例 3-2：调用 filter() 变换*

```
>>> pythonlines = lines.filter(lambda line: "Python" in line)
```

动作，另一方面来看，是基于 RDD 来计算某个结果，并将结果返回给驱动程序或者保存结果到一个外部的存储系统（比如 HDFS）。更早之前我们调用过一个动作的例子是 first()。它返回 RDD 中的第一个元素，示例 3-3 展示了这点：

*示例 3-3：调用 first() 动作*

```
>>> pythonlines.first()
```

```
u'## Interactive Python Shell'
```

变换和动作不相同是因为 Spark 计算 RDD 的方式。虽然，在任何时候你都可以定义一个新的 RDD，但是 Spark 总是以一种 lazy 的方式计算它们，也就是它们被第一次用于动作的时候。这种方式最初可能觉得不太寻常，但是当你开始处理大数据时，你就会有同感了。举例来说，考虑下前面的示例 3-2 和示例 3-3，我们定义了一个文本文件 RDD 然后过滤出了包含“Python”的行。如果当我们一写完 lines = sc.textFile(...)语句，Spark 就立刻加载和保存整个文件的所有行的话，考虑到我们马上就要过滤掉很多的行，这会导致浪费很多存储空间。反过来说，一旦 Spark 知道了整个变换链，它就能只计算结果需要的数据。实际上，对于 first() 动作来说，Spark 只需要扫描文件直到它找到第一个符合条件的行就可以了，这甚至不需要读整个文件。

最后，每次你执行个动作，Spark 的 RDD 默认会被重新计算。如果你想在多个动作中重用 RDD，你可以用 RDD.persist()要求 Spark 对 RDD 持久化。我们可以用一些不同的方式要求 Spark 对我们的数据持久化，详见表 3-6。在初次计算之

后，Spark 可以保存 RDD 的内容到内存中（在你的集群中跨机器分区），并在未来的动作中重用。持久化 RDD 到磁盘上，而不是内存中，也是可能的。默认不持久化的行为看起来也有点奇怪，但是对大数据集来说就该这样：如果你不会重用这个 RDD，那就没有理由浪费存储空间。相反的，一旦 Spark 流过数据，只是计算结果就好了。<sup>1</sup>

实际上，你会经常使用 `persist()` 来加载你的数据子集到内存并反复查询。比如，如果我们知道我们想要计算关于 README 文件中包含“Python”的行的多个结果，我们会写示例 3-4 那样的脚本。

示例 3-4: 持久化 RDD 到内存

```
>>> pythonLines.persist
>>> pythonLines.count()
2
>>> pythonLines.first()
u'## Interactive Python Shell'
```

总之，每个 Spark 程序或者 shell 会话都是像这样工作：

1. 从外部数据创建一些作为输入的 RDD
2. 使用类似 `filter()` 之类的变换来定义出新的 RDD
3. 要求 Spark 对需要重用的任何中间 RDD 进行 `persist()`
4. 启动类似 `count()` 和 `first()` 的动作开始并行计算，然后 Spark 会优化并执行。



`cache()` 和在默认存储级别上和调用 `persist()` 的效果一样。

在本章接下来的部分，我们将从头到尾的详细讨论每个步骤，包括 Spark 中一些最常用的操作。

## 创建 RDD

Spark 提供两种方式创建 RDD：通过加载一个外部数据集，或者在你的驱动程序中并行化一个集合。

<sup>1</sup> 总是重新计算一个 RDD 的能力事实上就是为什么 RDD 被称为“弹性”的原因。当拥有 RDD 数据的机器发生故障，Spark 就利用这个能力重新计算丢失的分区，这对用户来说是透明的。

最简单的创建 RDD 的方式就是将你程序中已存在的集合传递给 SparkContext 的 `parallelize()` 方法，见示例 3-5 到 3-7。当你在学习 Spark 的时候，这种方法非常有用。你可以在 Shell 中快速创建你自己的 RDD 并对其进行操作。然而，请记住在原型和测试之外，这种方式并不常用。因为这要求你所有的数据都在一台机器上的内存中。

示例 3-5: Python 的 `parallelize()` 方法

```
lines = sc.parallelize(["pandas", "i like pandas"])
```

示例 3-6: Scala 的 `parallelize()` 方法

```
val lines = sc.parallelize(List("pandas", "i like pandas"))
```

示例 3-7: Java 的 `parallelize()` 方法

```
JavaRDD<String> lines = sc.parallelize(Arrays.asList("pandas", "i like pandas"))
```

更常见的方式是从外部存储加载数据，详见第 5 章。然而我们之前已经见过加载文本文件为字符串 RDD 的方法: `SparkContext.textFile()`，见示例 3-8 到 3-10。

示例 3-8: Python 的 `textFile()` 方法

```
lines = sc.textFile("/path/to/README.md")
```

示例 3-9: Scala 的 `textFile()` 方法

```
val lines = sc.textFile("/path/to/README.md")
```

示例 3-10: Java 的 `textFile()` 方法

```
JavaRDD<String> lines = sc.textFile("/path/to/README.md");
```

## RDD 操作

我们已经提到过，RDD 支持两种类型的操作：变换和动作。变换是对一个 RDD 进行操作得到一个新的 RDD，如 `map()` 和 `filter()`。动作是返回一个结果到驱动程序或者写入到存储并开始计算的操作，如 `count()` 和 `first()`。Spark 对待变换和动作很不一样，所以理解你要执行的是何种操作十分重要。如果你对一个给定的函数是变换还是动作还有些混淆，那就看它的返回类型。返回 RDD 的就是变换，反之，动作是返回其他类型。

## 变换

变换是对一个 RDD 进行操作得到一个新的 RDD。如 29 页的“延迟计算”中讨论的一样，RDD 的变换计算是会延迟的，直到你在一个动作中用到。大多数变换都是元素级的，也就是说，每次处理一个元素，但不是所有变换都这样。

看个例子。假设我们有个日志文件 log.txt，里面有些日志消息。我们想仅选择出错误消息。可以用之前见过的 filter() 变换。这次我们用 Spark 的所有三种语言 API 来展示 filter 的用法（见示例 3-11 到 3-13）。

示例 3-11: Python 的 filter() 方法

```
inputRDD = sc.textFile("log.txt")
errsRDD = inputRDD.filter(lambda x: "error" in x)
```

示例 3-12: Scala 的 filter() 方法

```
val inputRDD = sc.textFile("log.txt")
val errsRDD = inputRDD.filter(line => line.contains("error"))
```

示例 3-13: Java 的 filter() 方法

```
JavaRDD<String> lines = sc.textFile("log.txt");
JavaRDD<String> errsRDD = inputRDD.filter({
    New Function<String, Boolean>(){
        Public Boolean call(string x) { return x.contains("error"); }
    }
});
```

注意到 filter 操作并没改变已存在的 RDD，相反，它返回了整个新 RDD 的指针。InputRDD 仍然可以在后面的程序中重用——比如要查找别的词。实际上，让我们再来用这个 RDD 找包含词语“warning”的行。然后用另一个变换 union 来打印这些包含了“error”或者“warning”的行。在示例 3-14 中我们是用 Python 展示，但是 union 在所有的三种语言中都支持。



示例 3-14: Python 中的 union() 变换

```
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```

union 和 filter 变换有一点点不同，它是对两个 RDD 操作，而不是一个。实际上变换可以对任意个输入 RDD 进行操作。



达到示例 3-14 同样结果的更好的方式是只过滤 inputRDD 一次，查找”error”或者”warning”。

最终，从 RDD 的相互变换形成了新的 RDD，Spark 跟踪了这些不同 RDD 之间的依赖关系的集合，称之为血统图(lineage graph)。Spark 用这个信息来根据需要计算每个 RDD，并恢复丢失的数据，如果持久化的 RDD 的一部分丢失了的话。图 3-1 展示了示例 3-14 的血统图。

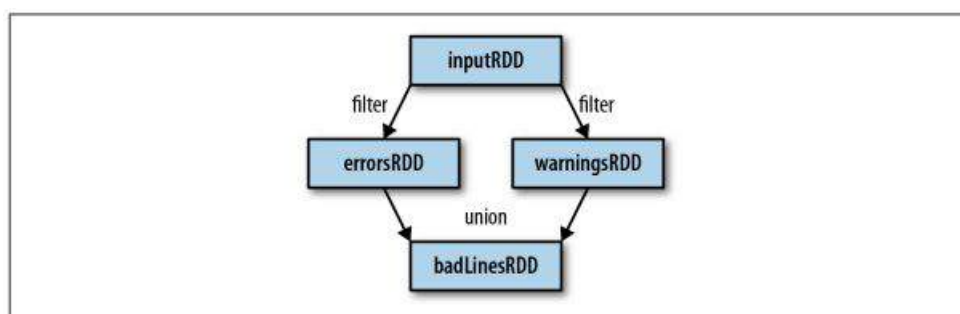


图 3-1 日志分析中创建的 RDD 血统图

## 动作

我们已经知道如何从各种变化创建 RDD，但有时候我们想实际对数据做点什么。动作是第二种操作。它们是返回一个最终值给驱动程序或者写入外部存储系统的操作。动作迫使对调用的 RDD 的变换请求进行求值，因为需要实际产生输出。

继续前一章的日志的例子，我们想打印一些关于 badlinesRDD 的信息。为此，我们将使用 count()和 take()两个动作。count()将返回其中的记录数，而 take()返回 RDD 中的一些元素，见例子 3-15 至 3-17。

示例 3-15: Python 中用 count 进行错误计数

```
print "Input had" + badLinesRDD.count() + " concerning lines"
print "Here are 10 examples:"
for line in badLinesRDD.take(10):
    print line
```

示例 3-16: Scala 中用 count 进行错误计数

```
println("Input had" + badLinesRDD.count() + " concerning lines")
println("Here are 10 examples:")
badLinesRDD.take(10).foreach(println)
```

示例 3-17: Java 中用 count 进行错误计数

```
System.out.println("Input had" + badLinesRDD.count() + " concerning lines");
System.out.println("Here are 10 examples:");
for (String line: badLinesRDD.take(10)){
    System.out.println(line);
}
```

本例中，我们在驱动程序中用 `take()` 来获取 RDD 中少量的元素，然后在本地遍历这些元素并打印出来。RDD 也有 `collect()` 函数可以获取所有的元素。如果你的程序将 RDD 过滤到一个比较小的数据集并且想要在本地处理时，`collect()` 会有用。记住，你使用 `collect()` 的话整个数据集必须适合单机的内存大小，所以 `collect()` 不适合对大数据集使用。

大多数情况下，RDD 都不能 `collect()` 到驱动程序，因为 RDD 一般都太大。这样的话，通常是将所有数据输出到分布式存储系统中，如 HDFS 或 S3 等。你可以用 `saveAsTextFile()`、`saveAsSequenceFile()` 或者任何其他各种内置格式的动作来保存 RDD 的内容。我们会在第五章讨论数据输出的不同选择。

值得注意的是，每次我们调用一个新动作，整个 RDD 都必须“从头开始”计算。要避免低效，用户可以像 44 页提到的“持久化（缓存）”那样持久化中间结果。

## 延迟求值

如前所述，RDD 的变换是延迟求值，这意味着 Spark 直到看到一个动作才会进行求值。这对新用户来说可能有点反直觉，而使用过函数语言如 Haskell 或 LINQ-like 之类数据处理框架的用户会熟悉一些。

延迟求值表示当我们对 RDD 调用变换时（比如 `map()`），该操作不会立即执行。相反的，Spark 内部记录元数据来指明该操作被请求。与其认为 RDD 包含了特殊的数据，不如认识 RDD 是由累积的数据变换如何计算的指令组成。加载数据到 RDD 也是一样的。所以我们调用 `sc.textFile()` 的时候数据直到有必要时才实际加载。如同变换一样，该操作（在这是指读取数据）可以发生多次。



尽管变换是延迟的，你可以在任何时候通过调用类似 `count()` 的动作来强制 Spark 执行。这是测试你的部分程序的一个简单方式。

Spark 使用延迟求值机制，通过对一组操作一起执行来减少其不得不接管数据的次数。像 Hadoop MapReduce 这样的系统，开发人员经常不得不花大量的时间思考如何将操作分组一起执行来最小化 MapReduce 的次数。而在 Spark 中，编写单个复杂的 `map` 操作来替代有许多单个操作组成的操作链并没有明显的好处。因此，用户可以自由的组织他们的程序为更小的，更可控的操作。

## 传递函数到 Spark

大多数的 Spark 的变换和一些动作都依赖于向 Spark 传入函数，这些函数被 Spark 用于计算数据。各个语言对于传递参数到 Spark 的机制有些细微的差异。

### Python

在 Python 中，传递函数到 Spark 有三种方式。对于较短的函数，可以通过 `lambda` 表达式来传递，见示例 3-2 和 3-18 所示。或者，也可以用顶级函数或者局部定义的函数。

示例 3-18: Python 中传入函数

```
word = rdd.filter(lambda s: "error" in s)
def containsError(s):
    return "errors" in s
word = rdd.filter(containsError)
```

当传递函数时，该函数包含要序列化的对象，那么有个问题要注意。当你传递的

函数是一个对象的成员，或者包含了一个对象的字段的引用（比如 `self.field`），Spark 是发送整个对象到 `worker` 节点。这可能会比你需要的信息多得多（见示例 3-19）。有时候这会导致你的程序出错，当你的类中包含了 `python` 不知道如何 `pickle` 的对象的话。

示例 3-19：传入带字段引用的函数（别这么干！）

```
class SearchFunctions(object):
    def __init__(self, query):
        self.query = query
    def isMatch(self, s):
        return self.query in s
    def getMatchesFunctionReference(self, rdd):
        # Problem: references all of "self" in "self.isMatch"
        return rdd.filter(self.isMatch)
    def getMatchesMemberReference(self, rdd):
        # Problem: references all of "self" in "self.query"
        return rdd.filter(lambda x: self.query in x)
```

替代的做法是仅取出需要的字段保存到局部变量并传入，如示例 3-20。

示例 3-20：Python 传入函数，没有字段引用

```
class WordFunctions(object):
    ...
    def getMatchesNoReference(self, rdd):
        # Safe: extract only the field we need into a local variable
        query = self.query
        return rdd.filter(lambda x: query in x)
```

## Scala

在 Scala 中，我们可以通过定义内联函数，引用方法，或者像我们在 Scala 的其他功能的 API 中的静态函数等方式来传递函数。随之而来的其他问题，也就是我们传递的函数引用的数据需要序列化（通过 Java 的序列化接口）。此外，如同 Python 中传递方法或者对象的字段会包括整个对象，虽然这个不明显，因为我们没有强制引用 `self`。就像示例 3-20 那么处理，我们抽取需要的字段到局部变量来避免传递整个对象，见示例 3-21。

示例 3-21: Scala 传入函数

```
class SearchFunctions(val query: String) {
  def isMatch(s: String): Boolean = {
    s.contains(query)
  }
  def getMatchesFunctionReference(rdd: RDD[String]): RDD[String] = {
    // Problem: "isMatch" means "this.isMatch", so we pass all of "this"
    rdd.map(isMatch)
  }
  def getMatchesFieldReference(rdd: RDD[String]): RDD[String] = {
    // Problem: "query" means "this.query", so we pass all of "this"
    rdd.map(x => x.split(query))
  }
  def getMatchesNoReference(rdd: RDD[String]): RDD[String] = {
    // Safe: extract just the field we need into a local variable
    val query_ = this.query
    rdd.map(x => x.split(query_))
  }
}
```

如果发生了 `NotSerializableException` 异常，通常是引用了不可序列化的类中的方法或字段。注意，传递顶级对象的局部可序列化的变量或函数总是安全的。

## Java

在 Java 中，函数是实现了 `org.apache.spark.api.java` 包中的 Spark 函数接口的对象。基于函数的返回类型有些不同的接口。表 3-1 中列出了最基本的函数接口以及一些我们需要的返回类似 key/value 的特定数据类型的函数接口。

表格 3-1 标准 Java 函数接口

函数名	实现的方法	用法
<code>Function&lt;T, R&gt;</code>	<code>R call(T)</code>	一个输入一个输出，用于 <code>map()</code> , <code>filter()</code> 之类的操作
<code>Function2&lt;T1, T2, R&gt;</code>	<code>R call(T1, T2)</code>	两个输入一个输出，用于 <code>aggregate()</code> , <code>fold()</code> 之类的操作
<code>FlatMapFunction&lt;T, R&gt;</code>	<code>Iterable&lt;R&gt; call(T)</code>	一个输入零个或多个输出，用于 <code>flatMap()</code> 之类的操作

我们可以在类内部定义匿名的内联函数类，见示例 3-22，或者定义命名类，见示例 3-23。

示例 3-22: Java 通过内部匿名类传递函数

```
RDD<String> errors = lines.filter(new Function<String, Boolean>() {  
    public Boolean call(String x) { return x.contains("error"); }  
});
```

示例 3-23: Java 通过命名类传递函数

```
class ContainsError implements Function<String, Boolean>() {  
    public Boolean call(String x) { return x.contains("error"); }  
}  
RDD<String> errors = lines.filter(new ContainsError());
```

选择哪种风格是个人习惯。但是我们发现在组织大型程序的时候，顶级命名类通常更清晰。使用顶级命名类的另一个好处是你定义构造参数，如示例 3-24。

示例 3-24: Java 带参数的函数类

```
class Contains implements Function<String, Boolean>() {  
    private String query;  
    public Contains(String query) { this.query = query; }  
    public Boolean call(String x) { return x.contains(query); }  
}  
RDD<String> errors = lines.filter(new Contains("error"));
```

在 Java8 中，你也可以用 lambda 来简洁的实现函数接口。由于在本书写作时，Java8 还相对较新，我们的例子使用的前一版本更冗长的语法来定义函数。然而，用 lambda 表达式，我们的搜索例子可以像 3-25 这样写。

示例 3-25: Java8 的 lambda 表达式传递函数

```
RDD<String> errors = lines.filter(s -> s.contains("error"));
```

如果你对使用 Java8 的 lambda 表达式有兴趣，可以看看 Oracle 的文档和 Databricks 的关于 Spark 如何使用 lambda 表达式的博客。



用匿名内部类或者 `lambda` 表达式都可以引用方法内部的 `final` 变量。所以，你可以传递这些变量，就像 `Python` 和 `Scala` 中的一样。

## 常见的变换和动作

在本章中，我们巡视一遍 `Spark` 中最常见的变换和动作。对包含某种类型数据的 `RDD` 还有些另外的操作可用，比如 `RDD` 的数量的统计函数，对 `RDD` 的 `key/value` 对按照 `key` 进行聚合的 `key/value` 操作。在后面的章节中我们会讲到 `RDD` 类型之间的转换和其他操作。

### 基本 `RDD`

我们先从对于不管是什么数据的 `RDD` 都适用的变换和动作说起。

#### 元素级的变换

两个最常见的你可能用到的变换是 `map()` 和 `filter()`（见图 3-2）。`map()` 变换传入一个函数，并将该函数应用到 `RDD` 中的每一个元素。函数的返回结果就是变换后的每个元素构成的新 `RDD`。`filter()` 变换也是传入一个函数，返回的是该 `RDD` 中仅能通过该函数的元素构成的新 `RDD`。

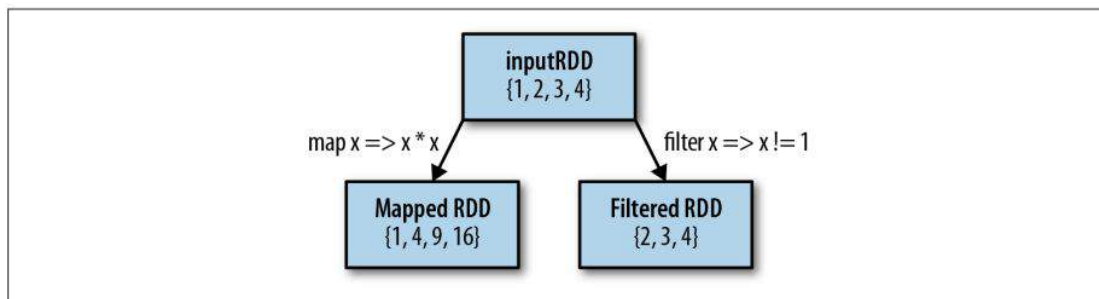


图 3-2 从输入 `RDD` 中 `map` 和 `filter` 后的 `RDD`

我们可以用 `map()` 做任何的事情，从我们的集合中取出网站关联的每个 `url` 到计算平方数。`map()` 的返回类型不必和输入类型相同，这很有用。如果我们有一个 `String` 类型的 `RDD`，通过 `map()` 将字符串解析后返回 `double`，那么我们的输入类型就是 `RDD[String]`，而结果类型就是 `RDD[Double]`。

让我们看一个 `map()` 的简单例子, 计算 RDD 中所有数的平方(示例 3-36 到 3-28)。

示例 3-26: Python 计算 RDD 中的平方值

```
nums = sc.parallelize([1, 2, 3, 4])
squared = nums.map(lambda x: x * x).collect()
for num in squared:
    print "%i " % (num)
```

示例 3-27: Scala 计算 RDD 中的平方值

```
val input = sc.parallelize(List(1, 2, 3, 4))
val result = input.map(x => x * x)
println(result.collect().mkString(", "))
```

示例 3-28: Java 计算 RDD 中的平方值

```
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3, 4));
JavaRDD<Integer> result = rdd.map(new Function<Integer, Integer>() {
    public Integer call(Integer x) { return x*x; }
});
System.out.println(StringUtils.join(result.collect(), ", "));
```

有时我们想为每个输入元素产生多个输出元素。这个操作叫做 `flatMap()`。和 `map()` 一样, 我们提供给 `flatMap()` 的函数被输入 RDD 中的每个元素单独调用, 但不是返回单个元素, 而是返回的一个返回值的迭代器。与其说是产生了一个迭代器的 RDD, 不如说是得到了一个来自所有迭代器的元素组成的 RDD。`flatMap()` 的一个简单用法是分割输入字符串到单词, 见示例 3-29 到 3-31。

示例 3-29 Python 中使用 `flatMap()` 分割文本行到单词

```
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
words.first() # returns "hello"
```

示例 3-30 Scala 中使用 `flatMap()` 分割文本行到单词

```
val lines = sc.parallelize(List("hello world", "hi"))
val words = lines.flatMap(line => line.split(" "))
words.first() // returns "hello"
```



示例 3-31 Java 中使用 flatMap() 分割文本行到单词

```
JavaRDD<String> lines = sc.parallelize(Arrays.asList("hello world", "hi"));
JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {
    public Iterable<String> call(String line) {
        return Arrays.asList(line.split(" "));
    }
});
words.first(); // returns "hello"
```

在图 3-3 中我们说明了 flatMap() 和 map() 之间的不同。你可以认为 flatMap() 是“压扁”了返回给它的迭代器。所以最终得到的不是一个 list 的 RDD，而是这些 list 中的元素的 RDD。

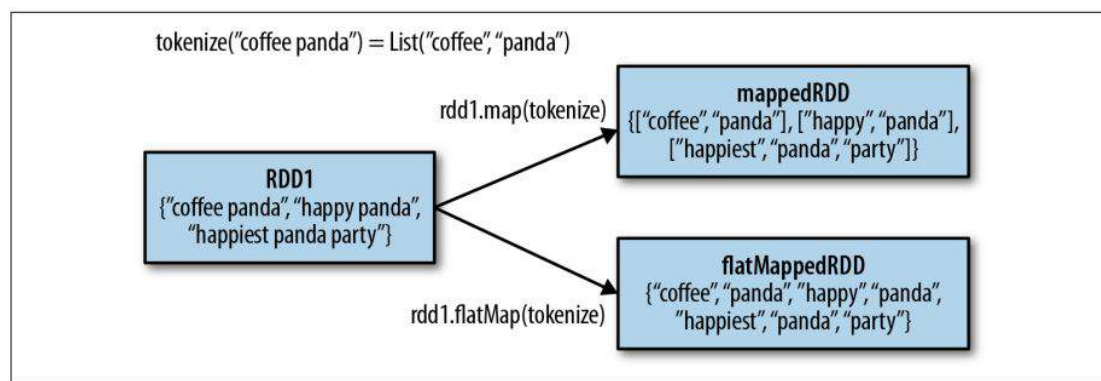


图 3-3 flatMap() 和 map() 之间的区别

### 伪集合操作

RDD 支持许多数学集合操作，比如并集合交集。甚至即使 RDD 本身不是严格的集合。图 3-4 列出了 4 个操作。这里的重点是所有这些操作要求被操作的 RDD 是同一个类型。

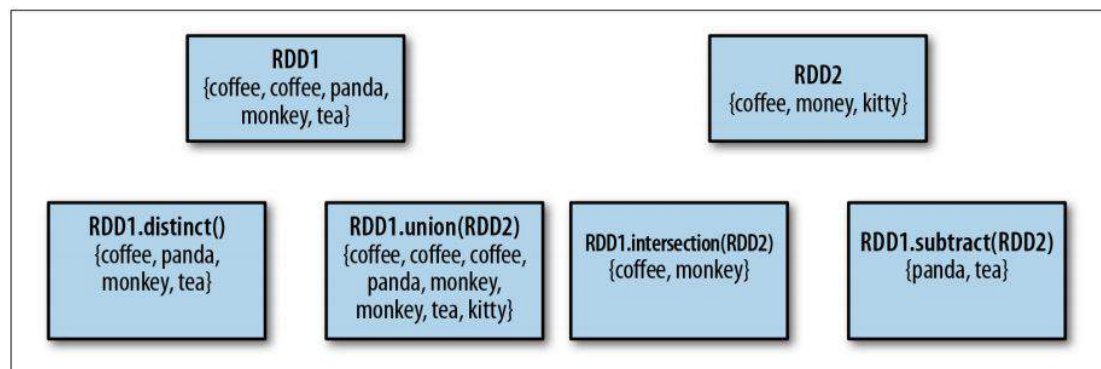


图 3-4 一些简单的集合操作

我们的 RDD 中最常丢失的集合属性是唯一性，经常会有重复元素。如果需要元

素唯一可以用 `RDD.distinct()` 变换来生成一个新的无重复元素的 RDD。然而，请注意 `distinct()` 很昂贵，因为它需要所有的数据通过网络进行 `Shuffling` 以确保唯一性。我们将在第四章详细讨论关于 `Shuffling` 以及如何避免它。

最简单的集合运算是 `union(other)`，它返回一个由两个源的数据一起组成的 RDD。在有些情况下会有用，比如处理来自多个源的日志文件。和数学意义上的并集不同，如果在输入的多个 RDD 中有重复元素，则 Spark 的 `union()` 的结果也有重复元素（可以通过 `dictinct()` 修复）。

Spark 也提供了 `intersection(other)` 方法，它返回两个 RDD 中都存在的元素。`Intersection()` 也会去除所有的重复元素（包括在单个 RDD 中存在的重复元素）。虽然 `union()` 和 `intersection()` 是相似的概念，但是 `intersection()` 的性能要差得多。因为它需要通过网络 `shuffle` 数据来识别公共元素。

有时我们需要根据想法去掉一些数据。`subtract(other)` 函数传入一个 RDD，返回的是只在第一个 RDD 中存在并且不在第二个 RDD 中存在的值的 RDD。就像 `intersection()` 一样，也需要 `shuffle` 数据。

我们还可以计算两个 RDD 的笛卡尔积，如图 3-5。`cartesian(other)` 变换返回 (a,b) 的所有可能的组合，其中 a 在源 RDD 中，b 在另一个 RDD 中。笛卡儿积在我们考虑所有可能的组合的相似性的时候会有用，比如计算用户对每个机会感兴趣的预期。也可以对 RDD 自己做笛卡儿积，对于用户相似度类似的任务会有用。然而请注意，笛卡儿积操作对于太大的 RDD 来说非常昂贵。

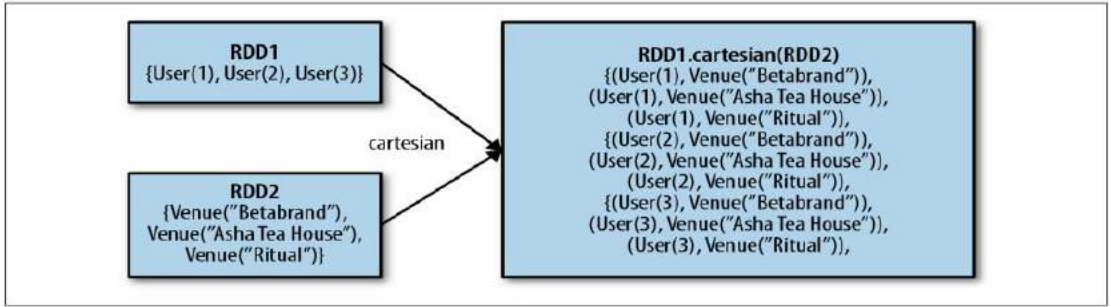


图 3-5 计算两个 RDD 的笛卡儿积

表 3-2 和 3-3 汇总了常见的 RDD 变换。

表格 3-2 对一个包含{1, 2, 3, 3}的 RDD 进行基本的变换

函数名	目的	示例	结果
map()	应用函数到 RDD 中的每个元素，并返回一个结果 RDD	<code>rdd.map(x =&gt; x + 1)</code>	{2, 3, 4, 4}
flatMap()	应用函数到 RDD 中的每个元素，并返回一个返回的迭代器内容的 RDD，通常用于提取单词	<code>rdd.flatMap(x =&gt; x.to(3))</code>	{1, 2, 3, 2, 3, 3, 3}
filter()	返回由仅通过传入 filter() 的条件元素组成的 RDD	<code>rdd.filter(x =&gt; x != 1)</code>	{2, 3, 3}
distinct()	去重	<code>rdd.distinct()</code>	{1, 2, 3}
Sample(withReplacement, fraction, [seed])	RDD 采样数据，可替换	<code>rdd.sample(false, 0.5)</code>	Nondeterministic

表格 3-3 对包含{1, 2, 3}和{3, 4, 5}的两个 RDD 进行变换

函数名	目的	示例	结果
union()	生成一个包含两个 RDD 中所有元素的 RDD	<code>rdd.union(other)</code>	{1, 2, 3, 3, 4, 5}
intersection()	生成两个 RDD 中都有的元素组成的 RDD	<code>rdd.intersection(other)</code>	{3}
subtract()	从一个 RDD 中去掉另一个 RDD 中存在的元素 (比如去掉训练数据)	<code>rdd.subtract(other)</code>	{1, 2}
cartesian()	生成两个 RDD 的笛卡尔积的 RDD	<code>rdd.cartesian(other)</code>	{(1, 3), (1, 4), ..., (3, 5)}

## 动作

对于基本 RDD，你最常用到的动作是 `reduce()`。它传入一个函数，该函数对 RDD 中两个元素进行处理，并返回一个同类型的元素。这类函数的一个简单例子是`+`，用于计算 RDD 中元素的和。有了 `reduce()`，我们可以轻松的计算 RDD 中元素的和，元素的个数，以及其他类型的聚合（见示例 3-32 到 3-34）。

示例 3-32 Python 中使用 `reduce()`  
`sum = rdd.reduce(lambda x, y: x + y)`

示例 3-33 Scala 中使用 `reduce()`  
`val sum = rdd.reduce((x, y) => x + y)`

示例 3-32 Java 中使用 `reduce()`  
`Integer sum = rdd.reduce(new Function2<Integer, Integer, Integer>() {  
public Integer call(Integer x, Integer y) { return x + y; }  
});`

`fold()`函数和 `reduce()`函数类似，也是带了一个和 `reduce()`相同的函数参数，但是多了一个“零值”用于在每个分区调用时初始化。你提供的初值对你的操作来说是恒等值，也就是说，你的函数对其应用多次都不会改变该值（例如，0 对于加法操作，1 对于乘法操作，或者空列表对于连接操作）。



你可以在 `fold()`中通过修改和返回两个参数中的第一个参数来最小化对象创建。但是你不能修改第二个参数。

`fold()`和 `reduce()`都要求返回结果的类型和处理的 RDD 的类型相同。对于求和来说很好，但是有时候我们想返回不同的类型。比如，当计算一个运行时的平均值，我们需要同时记录总量和元素个数，这就要求我们返回一个对值(pair)。我们可以先用 `map()`对每个元素做变换形成元素和数字 1 的对值，也就是我们要返回的类型，然后就能用 `reduce()`进行处理。

`aggregate()`函数将我们从被约束只能返回处理的 RDD 的相同类型 RDD 中解脱了。`aggregate()`和 `fold()`一样有一个初始的零值，但是可以是我们想要返回的类型。然后我们提供一个函数合并所有元素到累加器。最后，我们需要提供第二个函数来合并这些累加器，每个累加器都是它们本地结果数据的累积。

我们用 `aggregate()`来计算 RDD 的平均值，避免用 `fold()`前还要先 `map()`，见示例 3-35 到 3-37。

示例 3-35 Python 中使用 `aggregate()`

```
sumCount = nums.aggregate((0, 0),
                           (lambda acc, value: (acc[0] + value, acc[1] + 1),
                            (lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1]))))
return sumCount[0] / float(sumCount[1])
```

示例 3-36 Scala 中使用 `aggregate()`

```
val result = input.aggregate((0, 0))(
    (acc, value) => (acc._1 + value, acc._2 + 1),
    (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
val avg = result._1 / result._2.toDouble
```

示例 3-37 Java 中使用 `aggregate()`

```
class AvgCount implements Serializable {
    public AvgCount(int total, int num) {
        this.total = total;
        this.num = num;
    }
    public int total;
    public int num;
    public double avg() {
        return total / (double) num;
    }
}

Function2<AvgCount, Integer, AvgCount> addAndCount =
    new Function2<AvgCount, Integer, AvgCount>() {
        public AvgCount call(AvgCount a, Integer x) {
            a.total += x;
            a.num += 1;
            return a;
        }
    };

Function2<AvgCount, AvgCount, AvgCount> combine =
    new Function2<AvgCount, AvgCount, AvgCount>() {
        public AvgCount call(AvgCount a, AvgCount b) {
            a.total += b.total;
            a.num += b.num;
            return a;
        }
    };

AvgCount initial = new AvgCount(0, 0);
AvgCount result = rdd.aggregate(initial, addAndCount, combine);
System.out.println(result.avg());
```

有些 RDD 的动作会以常规的集合或值的形式返回部分或所有数据到驱动程序。

最简单最常用的返回数据到驱动程序的操作是 `collect()`，返回整个 RDD 的数据。`collect()`通常用于单元测试，整个 RDD 的内容能放到内存中，这样就能轻易的比较 RDD 是否是我们期待的结果。`collect()`受限于所有的数据必须适合单机，因为所有的数据要复制到驱动程序所在机器上。

`take(n)`返回 RDD 中的 `n` 个元素，试图最小化访问的分区数目。所以它返回的是有偏差的集合。重要的是知道这操作不会以你期待的顺序返回数据。

这些操作对于单元测试或者快速调试时很有用，但是处理大量数据时会有瓶颈。如果是已经有序的数据集，我们可以用 `top()`函数从 RDD 中提取前面的若干元素。`top()`使用数据的默认顺序，但是你可以提供一个比较函数来提取前面的元素。

有时在驱动程序中需要数据的样本。`takeSample(withReplacement, num, seed)`函数允许我们对数据采用，可以同时用随机数替换值或者不替换。

有时对 RDD 中所有元素都执行一个动作，但是不返回任何结果到驱动程序，也是有用的。一个不错的例子是发送 JSON 到 `webserver` 或者插入记录到数据库，这两种情况都能用 `foreach()`这个动作对每个元素执行计算，但是不返回到本地。

对基本 RDD 的更多的标准操作的准确的行为你都能从它们的名字上想象的到。`Count()`返回元素的个数，`countByValue()`返回每个唯一值对应的个数的 `map`。表 3-4 汇总了这些动作。

表格 3-4 对包含{1, 2, 3, 3}的 RDD 执行动作

函数名	目的	示例	结果
<code>collect()</code>	返回 RDD 中的所有元素	<code>rdd.collect()</code>	{1, 2, 3, 3, 4, 5}
<code>count()</code>	返回 RDD 中元素个数	<code>rdd.count()</code>	4
<code>countByValue()</code>	RDD 中每个元素出现的次数	<code>rdd.countByValue()</code>	{(1, 1), (2, 1), (3, 2)}

函数名	目的	示例	结果
take(num)	返回 RDD 中的 num 个元素	rdd.take(2)	{1, 2}
top(num)	返回 RDD 中前 num 个元素	rdd.top(2)	{3, 3}
takeOrdered(num)(ordering)	返回 RDD 中基于给定顺序的 num 个元素	rdd.takeOrder(2)(myOrdering)	{3, 3}
takeSample(withReplacement, num, [seed])	随机返回 RDD 中的 num 个元素	rdd.takeSample(false, 1)	Nondeterministic
reduce(func)	并行合并 RDD 中的元素(比如求和)	rdd.reduce((x,y) => x+y)	9
fold(func)	和 reduce()一样,但是提供了一个初值	rdd.fold(0)((x,y)=>x+y)	9
aggregate(zeroValue)(seqOp, combOp)	类似 reduce(),但是用于返回不同的类型	rdd.aggregate((0,0))((x, y) => (x._1 + y, x._2 + 1), (x, y) => (x._1 + y._1, x._2 + y._2))	(9, 4)
foreach(func)	对 RDD 中的每个元素应用函数 func	rdd.foreach(func)	Nothing

## RDD 类型之间的转换

有些函数只对某种类型的 RDD 可用,比如 mean()和 variance()对数值类型的 RDD 可用,而 join()对键值对类型的 RDD 可用。我们会在第六章涉及到数值 RDD,第四章涉及键值对的 RDD。在 Scala 和 Java 中,标准 RDD 没有定义这些方法。所以,要访问这些附加的方法,我们必须确保我们得到了正确的类型。

## Scala

在 Scala 中转换有特定功能的 RDD（比如对 `RDD[Double]` 暴露数值功能）是通过隐式转换自动处理的。在 17 页提到的“初始化 `SparkContext`”中，我们需要添加 `import org.apache.spark.SparkContext._` 以便这些转换能工作。你可以看看 `SparkContext` 对象的 Scala 文档中列出的隐式转换。RDD 被隐式的转换成各种封装类，比如 `DoubleRDDFunctions`（数值数据的 RDD）和 `PairRDDFunctions`（键值对的 RDD），以便暴露出类似 `mean()` 或者 `variance()` 等附加的功能。

隐式转换虽然很强大，但有时会让人混淆。如果你对 RDD 调用 `mean()` 类似的函数，可能你看到 Scala 的文档中的 RDD 类并没有 `mean()` 函数。这个调用能成功是因为从 `RDD[Double]` 到 `DoubleRDDFunctions` 之间的隐式转换。在 Scala 文档中查找 RDD 的这些函数时，确保看看这些封装类中可用的函数。

## Java

在 Java 中，特定类型的 RDD 之间的转换要明显一些。特别是 `JavaDoubleRDD` 和 `JavaPairRDD` 这些对数据类型有额外的方法的类。好处是让你更好的理解转换时如何进行的，但是有一点点麻烦。

要构造这些特殊类型的 RDD，而不是总使用函数类，我们需要使用特定的版本。如果我们想从一个类型为 `T` 的 RDD 创建 `DoubleRDD`，我们使用 `DoubleFunction<T>` 而不是 `Function<T, Double>`。这些特殊函数及用法见表 3-5。

我们同样需要对 RDD 调用不同的函数（我们不能只是创建一个 `Double` 函数传递给 `map()`）。当我们想要一个 `DoubleRDD` 时，和下面的其他函数的模式一样，我们需要调用 `mapToDouble()` 而不是 `map()`。

表格 3-5 特定类型函数的 Java 接口

函数名	等价函数* <code>&lt;A,B,...&gt;</code>	用法
<code>DoubleFlatMapFunction&lt;T&gt;</code>	<code>Function&lt;T, Iterable&lt;Double&gt;&gt;</code>	从 <code>flatMapToDouble()</code> 得到 <code>DoubleRDD</code>
<code>DoubleFunction&lt;T&gt;</code>	<code>Function&lt;T, double&gt;</code>	从 <code>mapToDouble()</code> 得到 <code>DoubleRDD</code>



函数名	等价函数*<A,B,...>	用法
PairFlatMapFunction<T, K, V>	Function<T, Iterable<Tuple2<K,V>>>	从 flatMapToPair()得到 PairRDD<K,V>
PairFunction<T, K, V>	Function<T, Tuple2<K,V>>	从 mapToPair()得到 PairRDD<K,V>

我们修改一下示例 3-28，在那里我们计算 RDD 中的数值的平方来生成一个新的 JavaDoubleRDD，见示例 3-38。这使得我们可以访问 JavaDoubleRDD 的额外的特殊函数，如 mean()和 variance()等。

示例 3-38 Java 中创建 DoubleRDD

```
JavaDoubleRDD result = rdd.mapToDouble(
    new DoubleFunction<Integer>() {
        public double call(Integer x) {
            return (double) x * x;
        }
    });
System.out.println(result.mean());
```

Python

Python API 的结构跟 Java 和 Scala 不同。在 Python 中，所有的函数都实现在基本 RDD 中，但是如果运行时 RDD 中的数据类型不正确会失败。

## 持久化（缓存）

之前说过，Spark RDD 是延迟求值的，有时候我们会想多次使用同一个 RDD。如果我们这么天真的做了，那么每次对这个 RDD 执行动作时，Spark 都会重新计算这个 RDD 和所有依赖的 RDD。这对于迭代计算时尤其昂贵，它会查找这些数据很多次。另一个浅显的例子是对同一个 RDD 先计数然后输出，如示例 3-39。

示例 3-39 Scala 中两次执行

```
val result = input.map(x => x*x)
println(result.count())
println(result.collect().mkString(","))
```

为避免多次计算同一个 RDD,我们可以要求 Spark 缓存该数据。当我们要求 Spark 缓存该 RDD 时，计算该 RDD 的节点都会保存它们的分区。如果缓存了该数据

的节点出错了，Spark 会在需要的时候重新计算丢失的分区。如果我们想在节点失败是处理不会变慢，那么我们可以复制数据到多个节点。

基于我们的目的，Spark 有多个级别的持久策略可选择，见表 3-6。在 Scala(示例 3-40)和 Java 中，默认的 `persist()` 是存储数据在 JVM 中作为非序列化对象。Python 中我们总是序列化数据持久保存，所以默认是在 JVM 中保存为序列化对象。当我们输出数据到磁盘或者堆外存储时，数据总是序列化的。

表格 3-6 `org.apache.spark.storage.StorageLevel` 和 `pyspark.StorageLevel` 中的存储级别；如果需要的话还可以通过添加 `_2` 到存储级别末尾来复制数据到 2 台机器

级别	空间占用	CPU	在内存	在磁盘	注释
MEMORY_ONLY	高	低	是	否	
MEMORY_ONLY_SE	低	高	是	否	
MEMORY_AND_DISK	高	中	有时	有时	如果数据太多不能放在内存里，则溢出到磁盘
MEMORY_AND_DISK_SER	低	高	有时	有时	如果数据太多不能放在内存里，则溢出到磁盘。内存中的数据表现为序列化。
DISK_ONLY	低	高	否	是	



堆外缓存正在测试，用的是 Tachyon。如果你对 Spark 的堆外缓存有兴趣，可以看看 [Running Spark On Tachyon guide](#)。

```
示例 3-40 Scala 中使用 persist()
val result = input.map(x => x * x)
result.persist(StorageLevel.DISK_ONLY)
println(result.count())
println(result.collect().mkString(","))
```

注意，我们是在第一个动作之前对 RDD 调用的 `persist()`。`persist()` 对其自身调用不会导致求值。

如果你试图缓存太多的数据，一致超出了内存，Spark 会使用 LRU 缓存策略丢弃旧的分区。对于 memory-only 存储级别，Spark 会在需要访问数据时重新计算；而对于 memory-and-disk 级别，会将数据写到磁盘。无论哪种方式，你都不用担心是否缓存态度数据会使任务停止。然而，不必要的缓存数据会导致有用的数据被丢弃而进行过多的计算。

最后，RDD 还提供了 unpersist() 函数给你手动释放缓存。

## 总结

在本章中，我们讲到了 RDD 的执行模型和大量常见的 RDD 操作。如果你都掌握了，恭喜——你已经学到了所有 Spark 的核心概念。在下一章中，我们会讲一组针对键值对的 RDD 的特殊操作，这在聚合或分组并行计算时很常用。之后会讨论各种数据源的输入和输出，以及关于 SparkContext 的更进一步的主题。

# 处理键值对(Key/Value Pairs)

本章介绍如何处理键值对，这是 Spark 中常见的一种数据类型。键值对 RDD 通常用于聚合操作，也经常会将一些初始 ETL(提取，转换，加载)获取的数据保存为键值对的格式。键值对的 RDD 也暴露了一些新的操作（比如每个产品的评价计数，按相同的键对数据进行分组，对两个不同的 RDD 分组）。

我们也会讨论一个键值对 RDD 的高级特征：分区(partitioning)，使用户可以跨节点控制 RDD 的布局。通过可控的分区，应用程序有时可确保数据在同一个机器上，可以集中访问，就可以大量的减少通信的开销，以此获得显著的提速。我们会用一个 PageRank 算法的例子来阐述分区。选择正确的分区对于分布式系统来说就和本地程序选择正确的数据结构类似，这两者都说明数据的布局对性能的影响非常大。

## 动机

Spark 为包含键值对的 RDD 提供了一些特殊的操作。这种 RDD 被称之为 pair RDD。Pair RDD 在许多程序中都是很有用的组件，因为它们对外的操作可以让你并行的处理每个键，或者跨网络重组数据。例如，Pair RDD 有一个 `reduceByKey()` 的方法，它可以对每个键的数据分别进行聚合；`join()` 方法可以通过对两个 RDD 中相同的元素进行分组合并。从 RDD 中抽取字段（例如事件的事件，客户 ID 或者其他标识）并用这些字段作为 pair RDD 的键进行处理是很常见的。

## 创建 Pair RDD

Spark 中有多种方式能得到 pair RDD。在第五章中我们要探索的很多格式加载时都可以直接的返回其键值数据为 pair RDD。另外，我们有一个普通 RDD 想要转换为 pair RDD，可以通过 map()操作来返回键值对。通过代码来看个例子，从一个包含文本行的 RDD 开始，用每一行的第一个单词作为 key。

这种方式构造键值 RDD 会根据编程语言有些不同。在 Python 中，为了处理有 key 的数据，我们需要返回 tuple 组成的 RDD（见示例 4-1）。

*示例 4-1 Python 中使用第一个单词做 key 来创建 pair RDD*

```
pairs = lines.map(lambda x: (x.split(" ")[0], x))
```

在 Scala 中，为了处理有 key 的数据，我们同样需要返回 tuple（见示例 4-2）。tuple 类型的 RDD 存在隐式转换，可以提供附加的键值函数。

*示例 4-2 Scala 中使用第一个单词做 key 来创建 pair RDD*

```
val pairs = lines.map(x => (x.split(" ")[0], x))
```

Java 没有内置的 tuple 类型，所以 Spark 的 Java API 有一个用户创建的 scala.Tuple2 类。该类很简单：Java 用户可以编写 new Tuple2(elem1, elem2)来创建一个新 tuple，然后用.\_1()和.\_2()方法来访问 tuple 中的元素。

Java 用户在创建 pair RDD 时同样需要调用特殊版本的 Spark 函数。比如用 mapToPair()替换基本函数 map()，在 43 页的“JAVA”部分有更多讨论。不过可以看一个简单的示例 4-3。

*示例 4-3 Java 中使用第一个单词做 key 来创建 pair RDD*

```
PairFunction<String, String, String> keyData =  
    new PairFunction<String, String, String>() {  
        public Tuple2<String, String> call(String x) {  
            return new Tuple2(x.split(" ")[0], x);  
        }  
    };  
JavaPairRDD<String, String> pairs = lines.mapToPair(keyData);
```

在 Scala 和 Python 中，当从内存中的集合创建 pair RDD，我们只需要对集合

调用 `SparkContext.parallelize()` 即可。而在 Java 中，要从内存中的集合创建 pair RDD，需要调用 `SparkContext.parallelizePairs()` 函数。

## 对 Pair RDD 的变换

Pair RDD 允许使用标准 RDD 的所有转换。30 页中“传入函数到 Spark”的规则同样适用。由于 pair RDD 包含 tuple，我们需要传入操作 tuple 而不是单个元素的函数。表 4-1 和 4-2 汇总了对 pair RDD 的变换，在后面的章节中会深入这些变换的细节。

表格 4-1 对单个 pair RDD 的变换（例子：{(1,2), (3,4), (3,6)}）

函数名	目的	示例	结果
<code>reduceByKey(func)</code>	按相同的键合并	<code>rdd.reduceByKey( (x, y) =&gt; x + y )</code>	<code>{(1,2), (3,10)}</code>
<code>groupByKey()</code>	按相同的键分组	<code>rdd.groupByKey()</code>	<code>{(1,[2]),(3,[4,6]) }</code>
<code>combineByKey( createCombiner, mergeValue, mergeCombiners, partitioner)</code>	按相同的键合并，返回不同的结果类型	见示例 4-12 到 4-14	
<code>mapValues(func)</code>	应用函数到 pair RDD 的每个值，但是不改变键	<code>rdd.mapValues( x =&gt; x+1)</code>	<code>{(1,3), (3, 5), (3, 7)}</code>
<code>flatMapValues(func)</code>	应用一个返回 pair RDD 中每个值的迭代器的函数，并对每个返回的元素以原来的键生成键值对，通常用于分词	<code>rdd.flatMapValues( x =&gt; (x to 5) )</code>	<code>{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)}</code>
<code>keys()</code>	只返回 RDD 中的所有键	<code>rdd.keys()</code>	<code>{1, 3, 3}</code>

函数名	目的	示例	结果
values()	只返回 RDD 中的所有值	rdd.values()	{2, 4, 6}
sortByKey()	返回按键排序的 RDD	rdd.sortByKey()	{{(1, 2), (3, 4), (3, 6)}

表格 4-2 对两个 pair RDD 的变换 (rdd={{(1,2), (3,4), (3,6)}}, other={{(3,9)}})

函数名	目的	示例	结果
subtractByKey	去 除 另 一 个 RDD 中 存 在 键 的 元 素	rdd.subtractByKey(other)	{{(1, 2)}
join	两个 RDD 执行内连接	rdd.join(other)	{{(3, (4, 9)), (3, (6, 9))}
rightOutJoin	两个 RDD 执行连接操作, 但是 other RDD 中的 key 必须存在	rdd.rightOutJoin(other)	{{(3, (Some(4), 9)), (3, (Some(6), 9))}
leftOutJoin	两个 RDD 执行连接操作, 但是第一个 RDD 中的 key 必须存在	rdd.leftOutJoin(other)	{{(1,(2,None)), (3,(4,Some(9))), (3,(6,Some(9)))}
cogroup	对两个 RDD 的数据共享相同的键分组	rdd.cogroup(other)	{{(1, ([2], [])), (3, ([4, 6], [9]))}

我们将在接下来的章节讨论每个 pair RDD 函数簇的更多细节。

Pair RDD 也仍然是 RDD（在 Java/Scala 中是 Tuple2 类型的 RDD，在 Python 中是 Tuple 类型的 RDD），所以也支持 RDD 的相同功能。例如，我们可以用前一节中的 pair RDD 来过滤掉大于 20 字节的行。见示例 4-4 到 4-6 和图 4-1。

示例 4-4 Python 中对第二个元素简单过滤

```
result = pairs.filter(lambda keyValue: len(keyValue[1]) < 20)
```

示例 4-5 Scala 中对第二个元素简单过滤

```
pairs.filter{case (key, value) => value.length < 20}
```

示例 4-6 Java 中对第二个元素简单过滤

```
Function<Tuple2<String, String>, Boolean> longWordFilter =  
    new Function<Tuple2<String, String>, Boolean>() {  
        public Boolean call(Tuple2<String, String> keyValue) {  
            return (keyValue._2().length() < 20);  
        }  
    };  
JavaPairRDD<String, String> result = pairs.filter(longWordFilter);
```

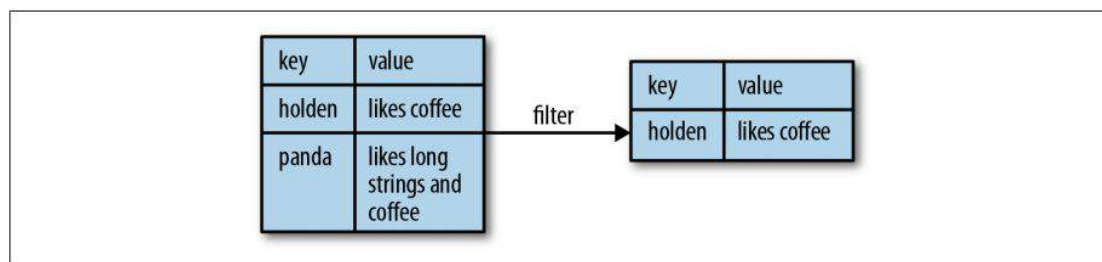


图 4-1 按值过滤

有时我们只是想访问 pair RDD 的值那部分，处理 pair 就会有些不太灵活。由于这是很常见的模式，所以 Spark 提供了 `mapValues(func)` 函数，等同于 `map{case (x, y) => (x, func(y))}`。在我们的示例中会大量使用该函数。

## 聚合

当数据集被表述成键值对，通常是想要对所有元素按相同的键进行聚合统计。我们已经看到了 `fold()`，`combin()` 和 `reduce()` 这些动作作用于基本 RDD。类似的也存在对 pair RDD 的每个键的变换。Spark 提供了按相同的键合并的一组类似的操作。这些返回 RDD 的操作是变换，而不是动作。

`reduceByKey()` 和 `reduce()` 非常相似，都带了一个函数参数用于合并。`reduceByKey()` 运行多个 `reduce()` 操作，数据集中的每个键对应一个，以便对相同的键进行合并。因为数据集可以有非常多的键，所以 `reduceByKey()` 没有实现成返回一个值到用户程序的动作，而是返回一个由每个键及对应合并后的值组成的新 RDD。

`foldByKey()` 和 `fold()` 十分相似，都有一个和 RDD 中数据同类型的零值和合并函数。和 `fold()` 一样，为 `foldByKey()` 提供的零值应用到合并函数与其他元素相加时应该没有副作用。



如示例 4-7 和 4-8 所示，我们可以把 `reduceByKey()` 和 `mapValues()` 一起使用来计算每个键的平均值，这跟用 `fold()` 和 `map()` 一起计算整个 RDD 的均值（见图 4-2）十分相似。对于平均值，我们还可以用一个更特殊函数来得到相同的结果，后面会讲到。

示例 4-7. Python 中用 `reduceByKey()` 和 `mapValues` 计算每个键的均值

```
rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
```

示例 4-7. Scala 中用 `reduceByKey()` 和 `mapValues` 计算每个键的均值

```
rdd.mapValues(x => (x, 1)).reduceByKey((x, y) => (x._1 + y._1, x._2 + y._2))
```

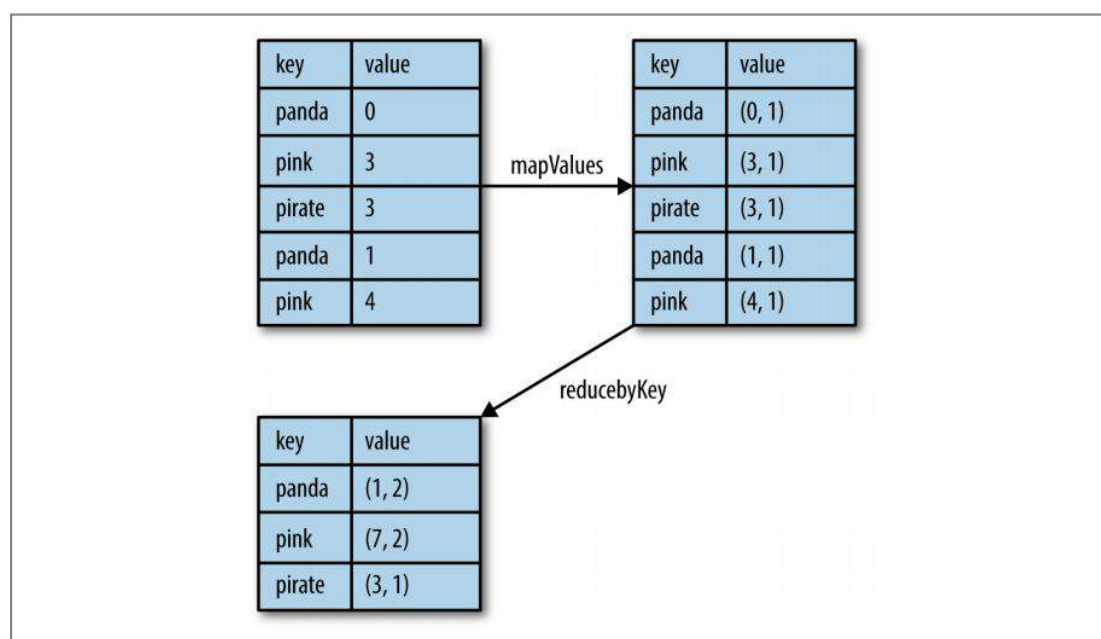


图 4-2 每个键的平均值的数据流



这些来自 MapReduce 的熟悉的合并概念需要指出的是调用 `reduceByKey()` 和 `foldByKey()` 会在每个机器上本地自动合并，然后再对每个键进行全局计算。用户不用再指定合并函数。而更一般的 `combineByKey()` 接口则允许你可以自定义合并行为。

我们在示例 4-9 到 4-11 中用类似方式来实现经典的分布式单词计数问题。使用前一章中的 flatMap()来生成一个单词和数值 1 的 pair RDD，然后像示例 4-7 和 4-8 那样使用 reduceByKey()来计算这些单词的总数。

示例 4-9. Python 中的 Word count

```
rdd = sc.textFile("s3://...")
words = rdd.flatMap(lambda x: x.split(" "))
result = words.map(lambda x: (x, 1)).reduceByKey(lambda x, y: x + y)
```

示例 4-10. Scala 中的 Word count

```
val input = sc.textFile("s3://...")
val words = input.flatMap(x => x.split(" "))
val result = words.map(x => (x, 1)).reduceByKey((x, y) => x + y)
```

示例 4-11. Java 中的 Word count

```
JavaRDD<String> input = sc.textFile("s3://...")
JavaRDD<String> words = rdd.flatMap(new FlatMapFunction<String, String>() {
    public Iterable<String> call(String x) { return Arrays.asList(x.split(" ")); }
});
JavaPairRDD<String, Integer> result = words.mapToPair(
    new PairFunction<String, String, Integer>() {
        public Tuple2<String, Integer> call(String x) { return new Tuple2(x, 1); }
    }).reduceByKey(
    new Function2<Integer, Integer, Integer>() {
        public Integer call(Integer a, Integer b) { return a + b; }
    });
```



实际上，我们可以对第一个 RDD 使用 countByValue()更快的实现单词计数：

```
input.flatMap(x => x.split(" ")).countByValue()
```

combineByKey()是针对每个键的更一般的聚合函数。大多数其他的对每个键合并的函数都是用它实现的。和 aggregate()一样，combinByKey()允许用户返回和输入数据类型不同的值。

要理解 combineByKey()，想想它是如何处理每个元素会有些帮助。combineByKey()会遍历一个分区中的所有元素。每个元素都有一个键，要么之间

没出息过，要么之前已经出现过。如果是新元素，`combineByKey()` 会创建 `combiner()`，也就是我们提供给它的函数来为这个新的键创建初值用于累积计算。这里有一点很重要，这个第一次是每个分区中新的键的第一次出现，而不是整个 RDD 第一次出现。

如果在这个分区这个键之前出现过，就对这个键进行多次累计就行了。当我们合并各个分区的数据时，如果两个以上的分区对于同一个键都有累积值，可以用用户提供的 `mergeCombiner()` 合并即可。



我们可以在 `combineByKey()` 中禁用 `map-side` 聚合，如果我们知道这对我们没有好处的话。比如 `groupByKey()` 禁用了 `map-side` 聚合，因为聚合函数（追加到 `list` 中）并不会节省空间。如果要禁用，需要指定 `partitioner`。目前你可以通过传入 `rdd.partitioner` 给源 RDD 来指定。

由于 `combineByKey()` 有很多不一样的参数，很适合一个解释性的例子。要更好的理解 `combineByKey()` 如何工作，我们看看为每个键计算平均值，见示例 4-12 到 4-14 和图 4-3。

示例 4-12. Python 中使用 `combineByKey()` 计算每个键的平均值

```
sumCount = nums.combineByKey((lambda x: (x,1)),
                              (lambda x, y: (x[0] + y, x[1] + 1)),
                              (lambda x, y: (x[0] + y[0], x[1] + y[1])))
sumCount.map(lambda key, xy: (key, xy[0]/xy[1])).collectAsMap()
```

示例 4-13. Scala 中使用 `combineByKey()` 计算每个键的平均值

```
val result = input.combineByKey(
  (v) => (v, 1),
  (acc: (Int, Int), v) => (acc._1 + v, acc._2 + 1),
  (acc1: (Int, Int), acc2: (Int, Int)) => (acc1._1 + acc2._1, acc1._2 + acc2._2)
).map{ case (key, value) => (key, value._1 / value._2.toFloat) }
result.collectAsMap().map(println(_))
```

示例 4-12. Java 中使用 `combineByKey()` 计算每个键的平均值

```
public static class AvgCount implements Serializable {
    public AvgCount(int total, int num) { total_ = total; num_ = num; }
```

```

    public int total_;

    public int num_;
    public float avg() { return total_ / (float) num_; }
}

Function<Integer, AvgCount> createAcc = new Function<Integer, AvgCount>() {
    public AvgCount call(Integer x) {
        return new AvgCount(x, 1);
    }
};

Function2<AvgCount, Integer, AvgCount> addAndCount =
    new Function2<AvgCount, Integer, AvgCount>() {
        public AvgCount call(AvgCount a, Integer x) {
            a.total_ += x;
            a.num_ += 1;
            return a;
        }
};

Function2<AvgCount, AvgCount, AvgCount> combine =
    new Function2<AvgCount, AvgCount, AvgCount>() {
        public AvgCount call(AvgCount a, AvgCount b) {
            a.total_ += b.total_;
            a.num_ += b.num_;
            return a;
        }
};

AvgCount initial = new AvgCount(0,0);
JavaPairRDD<String, AvgCount> avgCounts =
    nums.combineByKey(createAcc, addAndCount, combine);
Map<String, AvgCount> countMap = avgCounts.collectAsMap();
for (Entry<String, AvgCount> entry : countMap.entrySet()) {
    System.out.println(entry.getKey() + ":" + entry.getValue().avg());
}

```

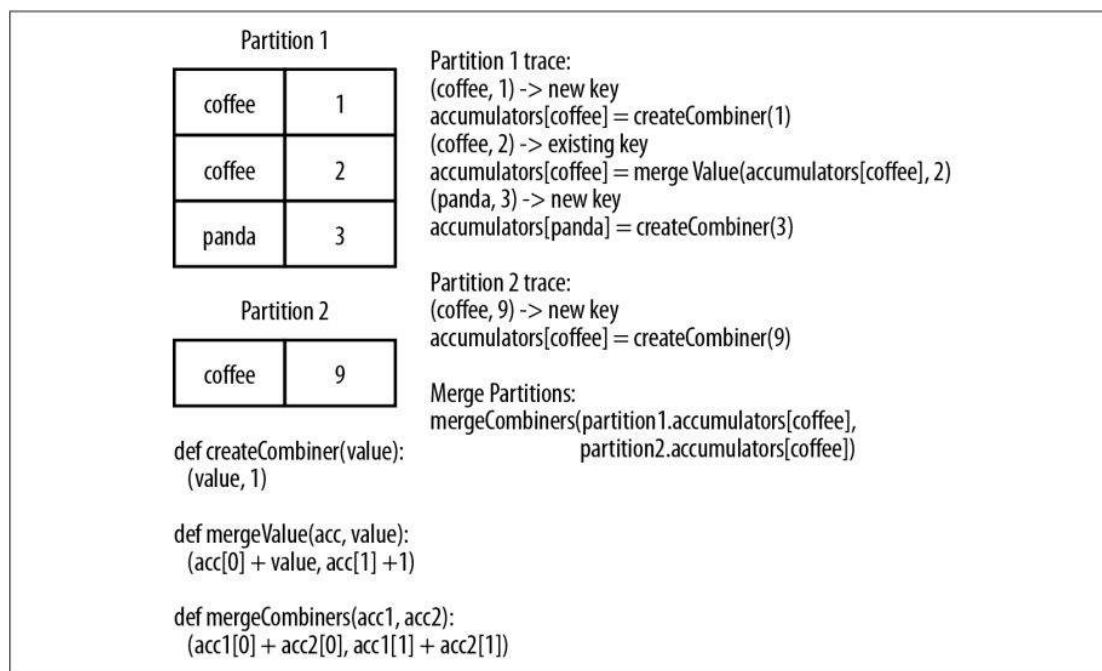


图 4-3 combineByKey()示例的数据流

Spark 中有很多根据键来合并数据的操作。大多数都是用 `combineByKey()` 来实现的，只是提供了一个更简单的接口。无论如何，用 Spark 中的这些特定的聚合函数都会比单纯的对数据分组后再归约要快得多。

## 调整并行级别

到目前为止，我们谈到的所有的变换都是分布式的。但是没有真正的了解过 Spark 是如何决定来怎样对处理进行分割的。每个 RDD 都有一个固定分分区数，这就决定了当对 RDD 进行操作时的并行度。

当进行聚合或者分组操作，我们可以要求 Spark 指定一个分区数。Spark 总是会根据集群的大小来试图推断出一个合适的默认值，但有时你会想要调整并行级别以获得更好的性能。

当创建分组后的或者聚合后的 RDD 时，本章讨论的大多数操作都接受给定一个分区数作为第二个参数来使用。见示例 4-15 和 4-16。

示例 4-15. Python 中自定义并行度执行 `reduceByKey()`

```
data = [("a", 3), ("b", 4), ("a", 1)]
sc.parallelize(data).reduceByKey(lambda x, y: x + y) # Default parallelism
sc.parallelize(data).reduceByKey(lambda x, y: x + y, 10) # Custom parallelism
```

示例 4-16. Scala 自定义并行度执行 `reduceByKey()`

```
val data = Seq(("a", 3), ("b", 4), ("a", 1))
sc.parallelize(data).reduceByKey((x, y) => x + y) // Default parallelism
sc.parallelize(data).reduceByKey((x, y) => x + y) // Custom parallelism
```

有时我们需要对 RDD 的分组或聚合操作的上下文之外来改变分区。对这些情况，Spark 提供了 `repartition()` 函数。它可以跨网络对数据进行 shuffle，并创建一组新的分区。记住，对数据进行重新分区是一个相当昂贵的操作。Spark 也有一个 `repartition()` 的优化版本叫做 `coalesce()`。它可以避免数据移动，但是仅限于减少 RDD 的分区数的情况。要知道你是否能安全的调用 `coalesce()` 函数，可以在 Java/Scala 中调用 `rdd.partitions.size()` 或者在 Python 中调用 `rdd.getPartitions()` 来检查 RDD 的分区数，以确保你正在合并到比当期更少的分区。

## 数据分组

有主键的数据的惯用法是根据键来对数据分组。比如查看一个客户的所有订单。

如果数据已经以我们想要的方式有了主键，`groupByKey()` 就可以用 RDD 中的主键对我们的数据进行分组。对于一个由类型为 K 的主键和类型为 V 的值组成的 RDD，`groupByKey()` 后我们得到的 RDD 类型为 `[K, Iterable[V]]`。

`groupByKey()` 用于非键值对的数据或者是使用当前主键之外的不同条件分组的情况。它传入一个函数应用到源 RDD 的每个元素，并将结果作为主键。



如果你发现你写的代码先 `groupByKey()` 然后对返回值执行 `reduce()` 或 `fold()`，那么你大概可以通过调用某个基于主键的聚合函数更高效的得到相同的结果。应该对数据按每个主键进行归约得到每个主键对应的归约后的值的 RDD，而不是归约数据到一个内存值中。例如，`rdd.reduceByKey(func)` 生成了和 `rdd.groupByKey().mapValues(value => value.reduce(func))` 相同的 RDD，但是更高效。因为避免了为每个主键创建列表的步骤。

除了从单个 RDD 对数据分组外，我们还可以用 `cogroup()` 对多个 RDD 有共同的主键的数据进行分组。`cogroup()` 涉及两个共享相同类型主键的 RDD，对应的值类型分别是 V 和 W，返回 `RDD[(K, (Iterable[V], Iterable[W]))]`。如果某个主键在一个 RDD 中的没有而在另一个 RDD 中存在，则对应的迭代器就仅是个空。`cogroup()` 给我们对多个 RDD 进行分组的能力。

`cogroup()` 是我们下一节要讲到的连接操作的基本组件。



`cogroup()` 可不止是用于实现连接。我们还可以用它来根据主键实现交集。另外，`cogroup()` 还可以一次性对 3 个或更多的 RDD 进行处理。

## 连接

我们得到了有主键的数据后，一个最有用的操作就是可以和其他有主键的数据一起使用。对 pair RDD 来说，连接数据到一起恐怕是最常见的操作。Spark 支持全系列的连接操作，包括右外连接，左外连接，交叉连接和内连接。

最简单的连接是内连接<sup>1</sup>。只有在两个 pair RDD 中都存在的主键才会输出。当某个输入的数据的同一个主键有多个值时，结果 RDD 会包含两个 RDD 按主键配对的所有可能的条目。看看示例 4-17 来简单理解一下。

示例 4-17. Scala shell 中进行内连接

```
storeAddress = {  
  (Store("Ritual"), "1026 Valencia St"), (Store("Philz"), "748 Van Ness Ave"),  
  (Store("Philz"), "3101 24th St"), (Store("Starbucks"), "Seattle")}  
storeRating = {  
  (Store("Ritual"), 4.9), (Store("Philz"), 4.8)}  
storeAddress.join(storeRating) == {  
  (Store("Ritual"), ("1026 Valencia St", 4.9)),  
  (Store("Philz"), ("748 Van Ness Ave", 4.8)),  
  (Store("Philz"), ("3101 24th St", 4.8))}
```

<sup>1</sup> “连接”是一个数据库术语，表示从两个表中使用共同的值来合并字段。

有时我们不需要两个 RDD 的连接的结果中都有同一个主键。例如，我们想连接用户信息和推荐信息。即使没有任何推荐信息，我们也不希望丢掉客户信息。`leftOuterJoin(other)`和 `rightOuterJoin(other)`都能按主键对两个 pair RDD 进行连接，并且其中一个可以没有对应另一个的主键。

对于 `leftOuterJoin()`，结果 RDD 中的每个主键都在源 RDD 中存在。而每个主键关联的值是源 RDD 中的值和另一个 pair RDD 中的 `Option`(Java 中是 `Optional`)类型值的二元组。在 Python 中如果值不存在就使用 `None`，如果存在，则就是用该值，不必封装。和 `join()`一样，对于每个主键可以有多个记录，此时我们得到的是两个列表的笛卡尔积。



`Optional` 是 Google 的 Guava 库的一部分，表示一个可能丢失的值。我们可以通过 `isPresent()`检查是否存在，用 `get()`来获取提供了值的数据。

`rightOuterJoin()`几乎和 `leftOuterJoin()`一样，除了主键必须存在于另一个 RDD 中，而可选值是来自源 RDD 而不是另一个 RDD。

重新看一下示例 4-17，对我们用来做 `join()`的两个 pair RDD 进行 `leftOuterJoin()`和 `rightOuterJoin()`看看，见示例 4-18。

示例 4-18. `leftOuterJoin()` 和 `rightOuterJoin()`

```
storeAddress.leftOuterJoin(storeRating) ==  
  {(Store("Ritual"),("1026 Valencia St",Some(4.9))),  
   (Store("Starbucks"),("Seattle",None)),  
   (Store("Philz"),("748 Van Ness Ave",Some(4.8))),  
   (Store("Philz"),("3101 24th St",Some(4.8)))}  
storeAddress.rightOuterJoin(storeRating) ==  
  {(Store("Ritual"),(Some("1026 Valencia St"),4.9)),  
   (Store("Philz"),(Some("748 Van Ness Ave"),4.8)),  
   (Store("Philz"), (Some("3101 24th St"),4.8))}
```

## 数据排序

有序的数据在许多情况下很有用，特别是当你要生成下行输出时。我们可以对提供了键值对并且定义了主键顺序的 RDD 进行排序。一旦排序好了，对有序数据接下来调用 `collect()`或者 `save()`的结果都是有序的。



由于我们经常会想对 RDD 逆序，所以 `sortByKey()` 函数有个 `ascending` 参数来指示是否升序（默认是 `true`）。有时我们会想以一种完全不同的顺序排序，为此我们可以提供我们自己的比较函数。示例 4-19 到 4-21 中，我们将 RDD 中的数字转换成字符串，使用字符串函数进行比较。

示例 4-19. 在 Python 中使用自定义排序，将整数当成字符串排序

```
rdd.sortByKey(ascending=True, numPartitions=None, keyfunc = lambda x: str(x))
```

示例 4-20. 在 Scala 中使用自定义排序，将整数当成字符串排序

```
val input: RDD[(Int, Venue)] = ...
implicit val sortIntegersByString = new Ordering[Int] {
  override def compare(a: Int, b: Int) = a.toString.compare(b.toString)
}
rdd.sortByKey()
```

示例 4-21. 在 Java 中使用自定义排序，将整数当成字符串排序

```
class IntegerComparator implements Comparator<Integer> {
  public int compare(Integer a, Integer b) {
    return String.valueOf(a).compareTo(String.valueOf(b))
  }
}
rdd.sortByKey(comp)
```

### Pair RDD 上可用的动作

和变换一样，基本 RDD 的所有传统动作都适用于 pair RDD。而 pair RDD 有些额外的动作，用到了数据的键值对的特性，见表 4-3。

表格 4-3 Pair RDD 的动作（示例：{(1,2), (3,4), (3,6)}）

函数名	目的	示例	结果
<code>countByKey()</code>	计算每个主键的元素个数	<code>rdd.countByKey()</code>	{(1, 1), (3, 2)}
<code>collectAsMap()</code>	收集数据为 map 结构方便查找	<code>rdd.collectAsMap()</code>	Map{(1, 2), (3, 4), (3, 6)}
<code>lookup(key)</code>	返回关联 key 的所有值	<code>rdd.lookup(3)</code>	[4, 6]

保存 pair RDD 的动作也有多个，在第五章中会描述。

## 数据分区（高级）

本章讨论的最后一个 Spark 的特性是如何控制数据跨节点分区。分布式程序中，通信十分昂贵，所以对数据布局来最小化网络传输可以大幅提高性能。跟单机程序为数据集合选择一个正确的数据结构很类似，Spark 程序可以选择控制分区来减少网络通信。分区不是对所有的应用都有好处，比如，对给定的 RDD 仅扫描一次，那么预先分区就没有意义。仅当数据集需要多次重用来面向主键比如连接这样的操作时才会有用。随后我们会给出示例。

Spark 的分区对所有的键值对 RDD 都可用，使系统基于函数对每个主键进行数据分组。尽管 Spark 没有显式的控制每个主键的数据到哪个 worker 节点（部分是因为系统被设计为即使某个节点出错也能工作），但是也会让程序确保同一个主键的数据集中出现在某些节点上。比如你可以选择用哈希分区将 RDD 分成 100 个分区，主键模 100 有相同结果的数据在同一个节点上。或者按范围分区，使得主键在同一个范围的元素在同一个节点上。

一个简单例子，假设程序在内存中持有一个非常大的用户信息表，或者说一个 (UserID, UserInfo) 键值对 RDD。其中 UserInfo 包含用户订阅的主题列表。应用程序周期性的将该表和另一个小文件合并。这个小文件以 (UserID, LinkInfo) 键值对的形式记录过去 5 分钟里在网页上点击过链接的事件表。比如我们现在想计算有多少用户访问了不是他们订阅的主题的链接。这可以通过 Spark 的 join() 操作来完成这个合并，按照主键将 UserInfo 和 LinkInfo 配对。程序看起来如示例 4-22。

*示例 4-22. Scala 的一个简单应用*

```
// Initialization code; we load the user info from a Hadoop SequenceFile on HDFS.  
// This distributes elements of userData by the HDFS block where they are found,  
// and doesn't provide Spark with any way of knowing in which partition a  
// particular UserID is located.  
val sc = new SparkContext(...)  
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()  
  
// Function called periodically to process a logfile of events in the past 5 minutes;  
// we assume that this is a SequenceFile containing (UserID, LinkInfo) pairs.
```

```
def processNewLogs(logFileName: String) {
  val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
  val joined = userData.join(events) // RDD of (UserID, (UserInfo, LinkInfo)) pairs
  val offTopicVisits = joined.filter {
    case (userId, (userInfo, linkInfo)) => // Expand the tuple into its components
      !userInfo.topics.contains(linkInfo.topic)
  }.count()
  println("Number of visits to non-subscribed topics: " + offTopicVisits)
}
```

代码可以执行，但是有些低效。这是因为每次 `processNewLogs()` 被调用时都会调用的 `join()` 操作不知道数据集中的主键是如何分区的。默认情况下，`join()` 操作会对两个 RDD 的主键都做哈希，通过网络发送相同主键的元素到同一个机器上，然后在这个机器上按主键进行连接（见图 4-4）。因为我们能预料到 `userData` 表远远大于每五分钟一个的较小的事件日志，太浪费了：即使根本没有变化，也要每次都把 `userData` 表跨网络的哈希和 shuffle 一遍。

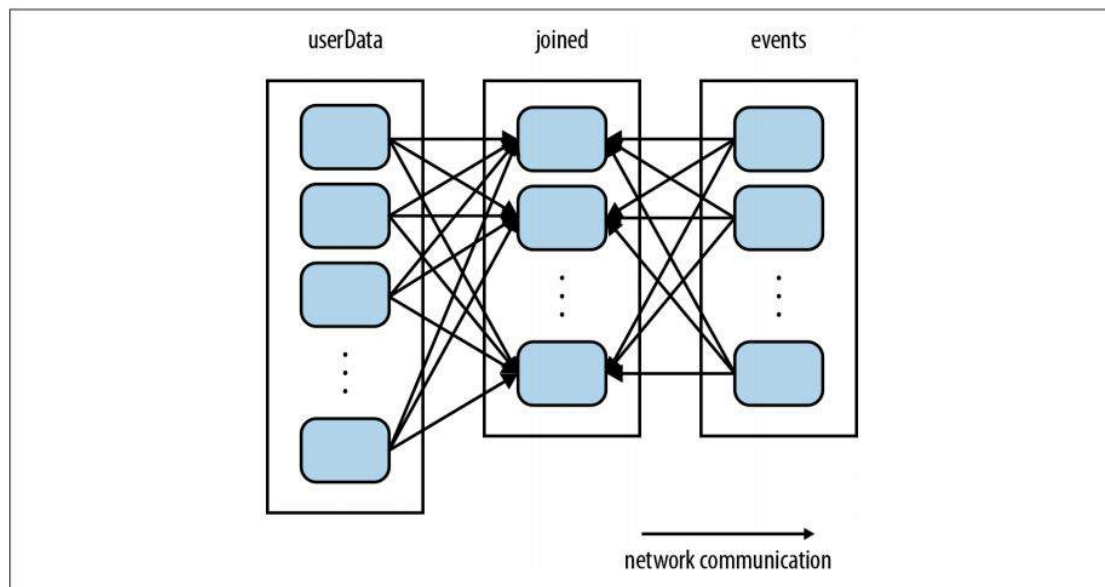


图 4-4 未使用 `partitionBy()` 的 `userData` 和 `events` 的每一次 `join`

修改很简单：在程序启动时的对 `userData` 使用 `partitionBy()` 变换进行哈希分区。将 `spark.HashPartitionner` 对象传入 `partitionBy()` 函数，见示例 4-23。

示例 4-23. Scala 中使用自定义分区

```
val sc = new SparkContext(...)
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")
.partitionBy(new HashPartitioner(100)) // Create 100 partitions
.persist()
```

processNewLogs()方法保持不变：对于 processNewLogs()来说 events RDD 是本地的，并且只在这个方法里用到，所以对 event RDD 指定分区不会有改善。因为在构建 userData 时我们调用了 partitionBy()函数，这样 Spark 就知道它是哈希分区的，在执行 join()时会利用这个信息。特别是在调用 userData.join(events)的时候，Spark 仅会对 events RDD 进行 shuffle，并根据 UserID 将 events 发送到 userData 的哈希分区对应的机器上（见图 4-5）。结果就是网络上传输的数据大量减少，程序运行的明显快了。

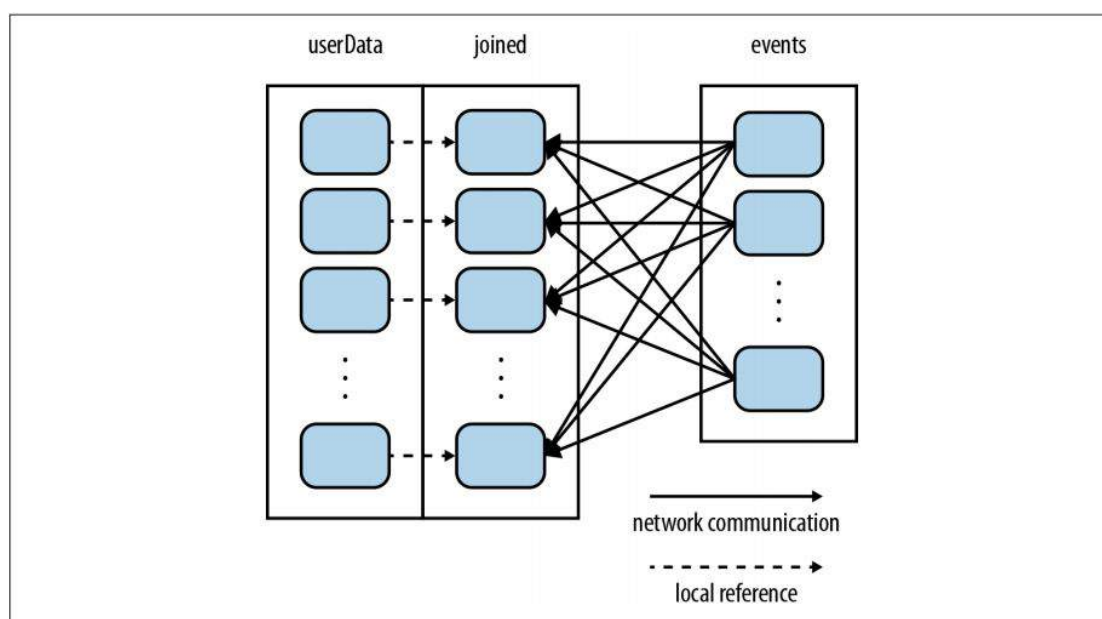


图 4-5 使用了 partitionBy()的 userData 和 events 的每一次 join

注意，partitionBy()是个变换，它总是返回一个新的 RDD，不会改变原始的 RDD。RDD 一旦创建就不会被修改。因此，重要的是持久化和保存 userData 这个 partitionBy()的结果，而不是原始的 sequenceFile()。传递给 partitionBy()的 100 是表达的分区个数，它控制对 RDD 执行更多操作（比如 join）的并发任务数。一般来说，至少让它和集群的核数一样大。



RDD 在 `partitionBy()` 变换后如果持久化失败会导致随后对 RDD 操作会对数据重复分区。没有持久化，使用分区后的 RDD 会导致整个 RDD 血统的重新计算。这会使 `partitionBy()` 起负作用，结果就是跨网络重复的分区和 `shuffle`，就跟没指定分区类似了。

事实上，Spark 的很多其他操作自动地导致 RDD 中带有已知的分区信息，除了 `join()` 之外的很多操作都会利用这些信息。例如 `sortByKey()` 和 `groupByKey()` 分别会产生范围分区和哈希分区的 RDD。另一方面，像 `map()` 这样的操作会导致新 RDD 忘记父 RDD 的分区信息，因为这些操作理论上能修改每个记录的主键。下一节讲述如何确定 RDD 是已分区的，以及分区如何精确的影响各种 Spark 操作。



### Java 和 Python 中的分区

Spark 的 Java 和 Python API 跟 Scala API 一样会从分区中获益。然而在 Python 中不能传入 `HashPartition` 对象来分区，你只能指定分区的数目（比如 `rdd.partitionBy(100)`）。

## 确定 RDD 的分区

在 Scala 和 Java 中可以通过分区属性（或者 Java 中用 `partitioner()` 方法）来确定 RDD 是如何分区的<sup>2</sup>。它会返回一个 `scala.Option` 对象，这是个 Scala 的容器类，可能包含或者没包含一个对象。你可以对 `Option` 对象调用 `isDefined()` 来检查是否有值，并用 `get()` 函数得到该值。如果有的话，这个值将是一个 `spark.Partitioner` 对象。本质上，这是个函数，告诉 RDD 每个主键都到了哪个分区里。后面会更详细说这个。

在 Spark Shell 中，分区属性是测试 Spark 操作对分区影响有多么不同的很好的方式，以及检查你程序中要做的操作会否得到正确的结果（见示例 4-24）。

<sup>2</sup> Python API 没有提供方法来查询分区，尽管在内部仍然使用分区。

示例 4-24. 确定 RDD 的 *partitioner*

```
scala> val pairs = sc.parallelize(List((1, 1), (2, 2), (3, 3)))
pairs: spark.RDD[(Int, Int)] = ParallelCollectionRDD[0] at parallelize at <console>:12
scala> pairs.partitioner
res0: Option[spark.Partitioner] = None
scala> val partitioned = pairs.partitionBy(new spark.HashPartitioner(2))
partitioned: spark.RDD[(Int, Int)] = ShuffledRDD[1] at partitionBy at <console>:14
scala> partitioned.partitioner
res1: Option[spark.Partitioner] = Some(spark.HashPartitioner@5147788d)
```

在这个短会话中，我们创建了一个 `(Int, Int)` 类型的 pair RDD，最初是没有分区信息的（`Option` 的值是 `None`）。然后对第一个 RDD 进行哈希分区创建了第二个 RDD。实际上如果我们要在以后的操作中用到这个分区后的 RDD，应该在输入的第三行的结尾增加 `persist()` 持久化一下，这时分区就定义好了。原因和之前的 `userData` 要持久化的例子相同：没有持久化，后续的 RDD 动作会对整个分区的 RDD 的血统重新求值，这会导致不断地哈希分区动作。

## 受益于分区操作

Spark 的许多操作都牵扯到根据主键跨网络 shuffle 数据的问题。所有这些都能从分区受益。Spark 1.0 中，受益于分区的操作包括 `cogroup()`, `groupWith()`, `join()`, `leftOuterJoin()`, `rightOuterJoin()`, `groupByKey()`, `reduceByKey()`, `combineByKey()` 和 `lookup()`。

作用于单个 RDD 的操作，比如 `reduceByKey()`，运行在预分区的 RDD 上会导致每个主键的所有数据在单机本地计算，在本地归约之后，最后才需要从各个 worker 节点发送到主节点。对于二元操作，比如 `cogroup()` 和 `join()`，预分区至少能使其其中一个 RDD（已知 `partitioner` 的那个）不用被 shuffle。如果两个 RDD 有同样的 `partitioner`，并且被缓存在同一台机器（比如其中一个是用另一个进行 `mapValues()` 创建的，会保留主键和分区信息），或者其中一个还没有被计算，那么就不会有跨网络的 shuffle 发生。

## 影响分区操作

Spark 知道内部每个操作是如何影响分区的，所以会自动对这些分区操作创建的 RDD 设置 `partitioner`。比如说，假设你调用 `join()` 来连接两个 RDD，由于有相同主键的元素已经被哈希到同一个机器，Spark 知道结果是哈希分区，那么像

`reduceByKey()`这个函数来操作连接的结果就会更加快速。

然而，另一面是，对于变换不能保证产生已知的分区，输出的 RDD 将不会有 `partitioner`。比如，你对一个哈希分区的键值对 RDD 调用 `map()`，传入 `map()` 的函数理论上可以改变每个元素的主键，所以结果 RDD 不会有 `partitioner`。Spark 不会分析你的函数是否保持着主键，相反，它提供了 `mapValues()` 和 `flatMapValues()` 这两个操作，它们都能保证每个二元组的主键保持相同。

如上所述，这些就是所有导致输出的 RDD 被设置 `partitioner` 的操作：`cogroup()`, `groupWith()`, `join()`, `leftOuterJoin()`, `rightOuterJoin()`, `groupByKey()`, `reduceByKey()`, `combineByKey()`, `partitionBy()`, `sort()`, `mapValues()` (如果父 RDD 有 `partitioner`), `flatMapValues()` (如果父 RDD 有 `partitioner`) 和 `filter()` (如果父 RDD 有 `partitioner`)。所有其他操作产生的结果 RDD 都没有 `partitioner`。

最终，对于二元操作，哪一个 `partitioner` 会被设置到输出的 RDD 依赖于父 RDD 的 `partitioner`。默认情况下，是带分区数设置了操作并行度的哈希 `partitioner`。然而，如果其中一个 RDD 有 `partitioner`，则输出 RDD 的 `partitioner` 就是它；如果两个父 RDD 都有 `partitioner`，那就会是第一个父 RDD 的那个。

## 示例：PageRank

作为一个能从 RDD 分区受益的涉及更多算法的例子，我们想到了 PageRank。PageRank 算法，是以 Google 的 Larry Page 命名，目标是对于一组文档中的每一个，依据有多少个文档连接到它来作为其重要性(rank)的度量。这当然可以用于网页重要度，也能够用于学术论文，或者社交网络中的用户影响。

PageRank 是个迭代算法，要执行很多的连接，所以这是个 RDD 分区的好用例。该算法包括两个数据集：一个是 `(pageID, linkList)`，其元素包含了每个页的邻居列表；另一个是 `(pageID, rank)`，其元素是每个页的当前 rank。处理流程如下：

1. 初始化每页的 rank 为 1.0
2. 对每次迭代，页面 p 发送  $\text{rank}(p)/\text{numNeighbors}(p)$  的贡献给它的邻居（该页连接出去的）
3. 设置每页的 rank 为  $0.15 + 0.85 * \text{contributionsReceived}$ 。



最后两步重复的多次迭代，算法会收敛到每个页面都有正确的 PageRank 值。实际上，一般要运行大约 10 次迭代。

示例 4-25 给出了 Spark 中 PageRank 的实现代码。

*示例 4-25. Scala PageRank*

```
// Assume that our neighbor list was saved as a Spark objectFile
val links = sc.objectFile[(String, Seq[String])]("links")
    .partitionBy(new HashPartitioner(100))
    .persist()
// Initialize each page's rank to 1.0; since we use mapValues, the resulting RDD
// will have the same partitioner as links
var ranks = links.mapValues(v => 1.0)
// Run 10 iterations of PageRank
for (i <- 0 until 10) {
    val contributions = links.join(ranks).flatMap {
        case (pageId, (links, rank)) =>
            links.map(dest => (dest, rank / links.size))
    }
    ranks = contributions.reduceByKey((x, y) => x + y).mapValues(v => 0.15 + 0.85*v)
}
// Write out the final ranks
ranks.saveAsTextFile("ranks")
```

就是这！算法最开始为 ranks RDD 的每个元素初始化成 1.0，然后每轮迭代保持更新 ranks 变量。PageRank 的主体在 Spark 中表达的很简单：首先，为了按 pageID 将链接列表和 rank 值关联到一起，使用当前的 ranks RDD 和静态的 links 做了个 join()，然后用 flatMap 创建“贡献”值发送到页面的每个邻居。然后按照 pageID 将这些值加起来（比如按页接收贡献值），并设置页面的 rank 值为  $0.15 + 0.85 * \text{contributionsReceived}$ 。

尽管代码本身很简单，本例也做了些事确保 RDD 被高效的分区，最小化通信。

1. 注意 links RDD 每次迭代都被 join() 到 ranks。由于 links 是静态数据集，我们在一开始就用 partitionBy() 对其分区，所以不需要跨网络 shuffle 数据。实际上 links RDD 很可能比 ranks 在字节数上要大得多。因为它包含了每个页的所有邻居列表，而不只是个 Double。所以这个优化对于这个



- PageRank 的简单实现（例如，相对于 MapReduce）节省了可能的网络传输。
2. 同样的原因，我们对 links 调用 `persist()` 使其在迭代过程中一直在内存里。
  3. 当我们第一次创建 ranks，使用的是 `mapValues()` 而不是 `map()`，保留了父 RDD(links) 的分区信息。所以我们的第一次 `join()` 开销不大。
  4. 在循环体中，我们在 `reduceByKey()` 后面跟了一个 `mapValues()` 调用。因为 `reduceByKey()` 的结果是哈希分区的，这会使得在下一轮迭代中对 map 的结果和 links 做连接时更高效。



要最大化潜在的分区相关的优化，你应该使用 `mapValues()` 或者 `flatMapValues()`。无论什么时候都不要改变元素的主键。

## 自定义分区

虽然 Spark 的 `HashPartitioner` 和 `RangePartitioner` 对大多数情况下都适用，Spark 也支持你用自定义的 `partitioner` 对象来调整 RDD 分区。这可以帮助你利用领域相关的知识进一步降低通信。

例如，假设我们想对一组网页运行前一节的 PageRank 算法。这里每个页面的 ID（RDD 的主键）将是它的 URL。用简单的哈希函数来分区，有着近似 URL 的网页（比如 `http://www.cnn.com/WORLD` 和 `http://www.cnn.com/US`）可能会被分区到不同机器上。然而，我们知道相同域名的页面容易产生很多的相互链接。由于 PageRank 每轮迭代都要从一个页面发送消息到它所有邻居，所以将这些页面分组到同一个分区会有帮助。我们可以通过自定义 `partitioner` 查找域名而不是 URL 来处理它。

要实现自定义 `partitioner`，你需要子类化 `org.apache.spark.Partitioner` 并实现三个方法：

- `numPartitions: Int`，返回你要创建的分区数
- `getPartition(key: Any): Int`，对给定主键返回分区 ID(从 0 到 `numPartitions-1`)
- `equals()`：标准的 Java 等值方法。实现它很重要，因为当 Spark 判决两个 RDD 对象的分区方式相同否，需要比较你的 `Partitioner` 对象和它自身的其他实例。

有一个问题，如果你的算法中依靠 Java 的 `hashCode()` 方法，它可以返回负数。你要小心确保 `getPartition()` 总是返回非负数。

示例 4-26 显示了我们如何写之前概述的基于域名的 `partitioner`，仅哈希每个 URL 的域名部分。

示例 4-26. *Scala custom partitioner*

```
class DomainNamePartitioner(numParts: Int) extends Partitioner {  
  override def numPartitions: Int = numParts  
  override def getPartition(key: Any): Int = {  
    val domain = new Java.net.URL(key.toString).getHost()  
    val code = (domain.hashCode % numPartitions)  
    if (code < 0) {  
      code + numPartitions // Make it non-negative  
    } else {  
      code  
    }  
  }  
  // Java equals method to let Spark compare our Partitioner objects  
  override def equals(other: Any): Boolean = other match {  
    case dnp: DomainNamePartitioner =>  
      dnp.numPartitions == numPartitions  
    case _ =>  
      false  
  }  
}
```

注意 `equals()` 方法，我们用的是 Scala 的模式匹配操作符(`match`)来测试另一个是不是 `DomainNamePartitioner`，如果是就转换。就跟 Java 中的 `instanceOf()` 一样。

使用用户自定义的 `Partitioner` 很容易：将它传到 `partitionBy()` 方法。在 Spark 中有许多基于 Shuffle 的方法，比如 `join()` 和 `groupByKey()`，都能带一个可选的 `Partitioner` 参数来控制输出的分区。

在 Java 中创建自定义 `Partitioner` 和 Scala 中很类似：继承 `spark.Partitioner` 类并实现要求的方法。

在 Python 中不用继承 `Partitioner` 类，而是传入一个哈希函数作为 `RDD.partitionBy()` 的额外参数，见示例 4-27 所示。

*示例 4-27. Python custom partitioner*

```
import urlparse
def hash_domain(url):
    return hash(urlparse.urlparse(url).netloc)
rdd.partitionBy(20, hash_domain) # Create 20 partitions
```

注意，你传入的哈希函数会和其他 RDD 做一致性的比较。如果你想多个 RDD 有相同的 `partitioner`，传入同一个函数（比如全局函数）而不是为每一个都创建新的 `lambda` 函数。

## 总结

本章我们学习了在 Spark 中用特殊的函数处理键值对的数据。第三章中的技术仍然适用于 `pair RDD`。下一章我们看看如何加载和保存数据。

# 加载和保存你的数据

工程师和科学家在本章都能找到有用的部分。工程师会希望探索更多的输出格式，看有什么能适合他们的目标用户。而数据科学家则关注已有的数据格式。

## 动机

我已经看到一旦数据在 Spark 中分发，就可以对数据执行许多操作。迄今为止我们的示例中的加载和保存的数据都来自原生集合或普通文件，还有一些数据不适合单机，是时候来探索加载和保存数据的选项了。

Spark 支持各种不同的输入输出源，在一定程度上是因为其构建于 Hadoop 生态系统。特别是，Spark 可以直接通过 Hadoop MapReduce 的 InputFormat 和 OutputFormat 接口访问数据，这对于很多常见的文件格式和存储系统（比如 S3,HDFS,Cassandra,HBase 等）都有用<sup>1</sup>。84 页的“Hadoop 的输入输出格式”一节直接展示了如何使用这些格式。

不过，更常见的情况是你想使用构建于这些原始接口之上的高级 API。还好，Spark 及其生态系统也提供了许多操作。本章我们会覆盖三种数据源：

## 文件格式和文件系统

对于保存在本地或者分布式系统中的数据，比如 NFS，HDFS，Amazon S3，Spark 能访问包括文本文件，JSON 文件，SequenceFile 和 protocol buffer 等各种文件格式。我们会展示如何使用这些格式，以及 Spark 如何处理不同的文件系统和压缩配置。

<sup>1</sup> InputFormat 和 OutputFormat 是用 MapReduce 连接到数据源的 Java API

## 通过 SparkSQL 访问结构化数据源

第九章中讲到的 Spark SQL 模块提供了访问结构化数据源的更友好也更高效的 API，包括 JSON 和 Apache Hive。本章中会大概了解下如何使用 Spark SQL，更多细节见第九章。

## 数据库和键值存储

我们会大概了解下通过内置库和第三方库连接 Cassandra，HBase，ElasticSearch 和 JDBC 数据库。

我们选取的大多数方法对 Spark 的所有语言都适用，但是有些库只能用于 Java 和 Scala。遇到这种情况会指出来的。

## 文件格式

Spark 大大简化了对大量文件格式的加载和保存。文件格式的范围从文本文件这样的非结构化文件，到 JSON 这样的半结构化文件，再到 SequenceFile 这样的结构化文件（见表 5-1）。Spark 封装了输入格式，基于文件扩展完全透明的处理压缩格式。

表格 5-1 支持的常见文件格式

格式名	结构化	说明
文本文件	否	普通文本文件，一行一条记录
JSON	半	普通文本格式，半结构化，大多数库要求一条记录一行
CSV	是	很常见的文本格式，通常用于电子表格应用
SequenceFile	是	普通的 Hadoop 文件格式，用于 key/value 的数据
Protocol buffers	是	一种快速的，空间高效的多语言格式
ObjectFile	是	从 Spark 任务保存数据并共享代码给客户时很有用。如果你修改了你的类会出错，因为它依赖于 Java 序列化。

除了 Spark 直接支持的输出格式之外，我们还可以使用 Hadoop 的新旧两种 API 来处理 key/value 数据，甚至有些格式忽略主键。万一有格式忽略了主键，通常会有个假主键（比如 null）。

## 文本文件

从 Spark 中加载和保存文本文件很简单。当我们加载单个文本文件到 RDD，每一行就是 RDD 中的一个元素。我们也可以一次加载多个文件到 pair RDD，主键就是文件名，值就是每个文件的内容。

### 加载文本文件

加载文本文件就是简单的对 SparkContext 调用 `textFile()` 函数并带上文件路径为参数，见示例 5-1 到 5-3。如果要控制分区数，指定 `minPartitions` 即可。

示例 5-1. Python 中加载一个文本文件

```
input = sc.textFile("file:///home/holden/repos/spark/README.md")
```

示例 5-2. Scala 中加载一个文本文件

```
val input = sc.textFile("file:///home/holden/repos/spark/README.md")
```

示例 5-3. Java 中加载一个文本文件

```
JavaRDD<String> input = sc.textFile("file:///home/holden/repos/spark/README.md")
```

以目录形式包含多部分输入的所有部分的处理有两种方式。我们可以就用 `textFile()` 将目录传给它，它就会加载所有的部分到 RDD。有时候知道某部分数据（比如文件中带主键的时间数据）来自哪个文件比较重要，或者我们想一次处理一个文件。如果文件足够小，可以用 `SparkContext.wholeTextFile()` 方法，返回的是 pair RDD，主键是输入文件的文件名。

`wholeTextFile()` 对于每个文件表示一个时间周期的数据时非常有用。如果我们有不同时期的销售数据文件，就能很轻松的计算每个时期的平均值，见示例 5-4。

示例 5-3. Scala 中计算每个文件的平均值

```
val input = sc.wholeTextFiles("file:///home/holden/salesFiles")
val result = input.mapValues{y =>
    val nums = y.split(" ").map(x => x.toDouble)
    nums.sum / nums.size.toDouble
}
```



Spark 支持读取给定目录中的所有文件，以及对输入的通配符扩展（比如 `part-*.txt`）。这个很实用，因为通常大数据集是散布在多个文件，尤其是还有其他文件(比如成功标志文件)也在同一个目录里。

## 保存文本文件

输出文本文件也十分简单。在示例 5-5 中，用 `saveAsTextFile()` 方法带一个路径参数，就会输出 RDD 的内容到那。路径是一个目录，Spark 会输出多个文件到那个目录下。这容许 Spark 从多个节点输出文件。用这个方法我们控制不了哪个文件终止于我们数据的哪一部分，不过有其他输出格式允许我们这个干。

*示例 5-4. Python 中保存为文本文件*  
`result.saveAsTextFile(outputFile)`

## JSON 文件

JSON 是一种流行的半结构化数据格式。加载 JSON 数据最简单的方式是当成文本文件加载 JSON 数据后再用 JSON 解析器来映射到值。同样的，我们可以用我们首选的 JSON 序列化库将值输出成字符串。在 Java 和 Scala 中我们也可以用自定义的 Hadoop 格式来处理 JSON 数据。172 页的“JSON”一节也展示了如何用 Spark SQL 加载 JSON 数据。

## 加载 JSON

以文本方式加载 JSON 数据然后解析 JSON 数据是所有支持的语言都能用的一种方式。这个工作是假设每行一个 JSON 记录，如果你有多行的 JSON 文件，你必须加载整个文件然后解析每一行。如果你用的语言构造 JSON 解析器开销很大，可以用 `mapPartitions()` 来重用该解析器，细节见 107 页的“基于分区来工作”。

对我们使用的三种语言，有大量的各种 JSON 库可用。不过，简单起见，每个语言我们只考虑一个库。在 Python 中内置的库（示例 5-6），在 Java 和 Scala 中用 Jackson(示例 5-7 和 5-8)。选这些库是因为它们都运行的很好，也相对简单。如果你花了大量时间在解析环节，看看 Scala 和 Java 的其他库吧。

示例 5-6. Python 中加载非结构化的 JSON 数据

```
import json
data = input.map(lambda x: json.loads(x))
```

在 Scala 和 Java 中，通常做法是加载记录到表达这些数据的模式的类中。在这个环节我们也会想跳过无效记录。来看一个加载记录到 Person 类实例的例子。

示例 5-7. Scala 中加载 JSON 数据

```
import com.fasterxml.jackson.module.scala.DefaultScalaModule
import com.fasterxml.jackson.module.scala.experimental.ScalaObjectMapper
import com.fasterxml.jackson.databind.ObjectMapper
import com.fasterxml.jackson.databind.DeserializationFeature
...
case class Person(name: String, lovesPandas: Boolean) // Must be a top-level class
...
// Parse it into a specific case class. We use flatMap to handle errors
// by returning an empty list (None) if we encounter an issue and a
// list with one element if everything is ok (Some(_)).
val result = input.flatMap(record => {
  try {
    Some(mapper.readValue(record, classOf[Person]))
  } catch {
    case e: Exception => None
  })
})
```

示例 5-7. Java 中加载 JSON 数据

```
class ParseJson implements FlatMapFunction<Iterator<String>, Person> {
  public Iterable<Person> call(Iterator<String> lines) throws Exception {
    ArrayList<Person> people = new ArrayList<Person>();
    ObjectMapper mapper = new ObjectMapper();
    while (lines.hasNext()) {
      String line = lines.next();
      try {
        people.add(mapper.readValue(line, Person.class));
      } catch (Exception e) {
        // skip records on failure
      }
    }
    return people;
  }
}

JavaRDD<String> input = sc.textFile("file.json");
JavaRDD<Person> result = input.mapPartitions(new ParseJson());
```





处理格式错误的记录是个大问题，尤其是 JSON 这样的半结构化数据。对于小数据集，遇到错误的输入格式后停止一切处理（不如程序出错）尚可接受。但是通常情况下，输入不正确格式的大数据集简直就是生活的一部分。如果你选择跳过格式错误的数据，你也许该考虑用个累加器跟踪错误的个数。

## 保存 JSON

输出 JSON 文件比加载要简单，因为不必担心数据的格式错误，而且我们知道正要输出的数据的类型。我们可以用转换 RDD 中的字符串到解析后的 JSON 数据的相同的库来处理 RDD 结构化数据转换回字符串，然后就可以用 Spark 的文本文件的 API 输出了。

比如说我们正在为喜欢熊猫的人做个促销。我们可以从第一步得到的输入数据中过滤出喜爱熊猫的人，见示例 5-9 到 5-11。

示例 5-9. Python 中保存为 JSON

```
(data.filter(lambda x: x['lovesPandas']).map(lambda x: json.dumps(x))
.saveAsTextFile(outputFile))
```

示例 5-10. Scala 中保存为 JSON

```
result.filter(p => P.lovesPandas).map(mapper.writeValueAsString(_))
.saveAsTextFile(outputFile)
```

示例 5-10. Java 中保存为 JSON

```
class WriteJson implements FlatMapFunction<Iterator<Person>, String> {
    public Iterable<String> call(Iterator<Person> people) throws Exception {
        ArrayList<String> text = new ArrayList<String>();
        ObjectMapper mapper = new ObjectMapper();
        while (people.hasNext()) {
            Person person = people.next();
            text.add(mapper.writeValueAsString(person));
        }
        return text;
    }
}

JavaRDD<Person> result = input.mapPartitions(new ParseJson()).filter(new LikesPandas());
JavaRDD<String> formatted = result.mapPartitions(new WriteJson());
formatted.saveAsTextFile(outfile);
```

就这样，我们就能用 Spark 通过处理文本和增加 JSON 库这些已存在的机制轻松的加载和保存 JSON 数据了。

## CSV 和 TSV

Comma-separated value(CSV)文件是每行包含固定字段，并用逗号分隔开（或用 tab 分隔开，也就是 tab-separated value，即 TSV 文件）的文件。通常是每行一条记录，但也不一定，有时候一条记录跨多行。CSV 和 TSV 文件有时候不一致，常见于处理换行，转义以及非 ASCII 字符和非整型数字的表达上。CSV 文件本身不能嵌套字段类型，所以我们不得不手工打包和解包特殊的字段。

不像 JSON 的字段，CSV 的每条记录都没有关联的字段名，不过我们可以取回行号。通常会使 CSV 文件的第一行是每个字段的字段名。

### 加载 CSV

加载 CSV/TSV 数据和加载 JSON 数据类似，先作为文本文件载入后再处理。格式缺乏标准化导致对同一个库的不同版本有时要用不同的方式处理输入。

跟 JSON 一样，这里有许多处理 CSV 的库。不过我们每个语言只用一种。Python 又是用的内置的库，而 Java 和 Scala 使用 opencsv。



Hadoop 中也有个 CSVInputFormat 的类可以在 Java 和 Scala 中用来加载 CSV 数据。但是它不支持包含换行的记录。

如果你的 CSV 数据中任意字段都没有包含换行的情况，你就可以用 `textFile()` 加载数据并解析，见示例 5-12 到 5-14。

*示例 5-12. 在 Python 中用 `textFile()` 加载 Loading CSV*

```
import csv
import StringIO
...
def loadRecord(line):
    """Parse a CSV line"""
    input = StringIO.StringIO(line)
    reader = csv.DictReader(input, fieldnames=["name", "favouriteAnimal"])
    return reader.next()
input = sc.textFile(inputFile).map(loadRecord)
```

示例 5-13. 在 Scala 中用 `textFile()` 加载 Loading CSV

```
import Java.io.StringReader
import au.com.bytecode.opencsv.CSVReader
...
val input = sc.textFile(inputFile)
val result = input.map{ line =>
    val reader = new CSVReader(new StringReader(line));
    reader.readNext();
}
```

示例 5-14. 在 Java 中用 `textFile()` 加载 Loading CSV

```
import au.com.bytecode.opencsv.CSVReader;
import Java.io.StringReader;
...
public static class ParseLine implements Function<String, String[]> {
    public String[] call(String line) throws Exception {
        CSVReader reader = new CSVReader(new StringReader(line));
        return reader.readNext();
    }
}
JavaRDD<String> csvFile1 = sc.textFile(inputFile);
JavaPairRDD<String[]> csvData = csvFile1.map(new ParseLine());
```

如果字段中有嵌入换行，我们就需要完整加载每个文件并解析整段数据，如示例 5-15 到 5-17 所示。很不幸，如果文件很大会，在加载和解析时会导致瓶颈。不同的文本加载方法见 73 页的“加载文本文件”。

示例 5-15. Python 中加载完整的 CSV

```
def loadRecords(fileNameContents):
    """Load all the records in a given file"""
    input = StringIO.StringIO(fileNameContents[1])
    reader = csv.DictReader(input, fieldnames=["name", "favoriteAnimal"])
    return reader
fullFileData = sc.wholeTextFiles(inputFile).flatMap(loadRecords)
```

示例 5-16. Scala 中加载完整的 CSV

```
case class Person(name: String, favoriteAnimal: String)
val input = sc.wholeTextFiles(inputFile)
val result = input.flatMap{ case (_, txt) =>
    val reader = new CSVReader(new StringReader(txt));
    reader.readAll().map(x => Person(x(0), x(1)))
}
```

示例 5-17. Java 中加载完整的 CSV

```
public static class ParseLine
implements FlatMapFunction<Tuple2<String, String>, String[]> {
    public Iterable<String[]> call(Tuple2<String, String> file) throws Exception {
        CSVReader reader = new CSVReader(new StringReader(file._2()));
        return reader.readAll();
    }
}

JavaPairRDD<String, String> csvData = sc.wholeTextFiles(inputFile);
JavaRDD<String[]> keyedRDD = csvData.flatMap(new ParseLine());
```



如果输入文件不多，并且你需要用 `wholeFile()` 方法，你应该对你的输入数据重新分区，以便 Spark 能高效的并行处理随后的操作。

## 保存 CSV

跟 JSON 一样，输出 CSV 也很简单，并且我们还能从重用输出编码对象中获益。因为在 CSV 中我们不需要输出每个记录的字段名，为了输出一致，我们需要创建一种映射。其中一种简单的方式是写个函数转换字段到一个数组中的给定位。Python 里面如果你要输出字典类型数据，CSV writer 可以帮我们。当用我们提供的字段名构造好 CSV writer 后，CSV writer 会基于这个顺序来输出。

我们使用的 CSV 库输出到文件或 CSV writer 后，就可以用 `StringWriter` 或者 `StringIO` 将结果放到 RDD 中，见示例 5-18 和 5-19。

示例 5-18. Python 输出 CSV

```
def writeRecords(records):
    """Write out CSV lines"""
    output = StringIO.StringIO()
    writer = csv.DictWriter(output, fieldnames=["name", "favoriteAnimal"])
    for record in records:
        writer.writerow(record)
    return [output.getvalue()]
pandaLovers.mapPartitions(writeRecords).saveAsTextFile(outputFile)
```

示例 5-19. Scala 输出 CSV

```
pandaLovers.map(person => List(person.name, person.favoriteAnimal).toArray)
.mapPartitions{people =>
    val stringWriter = new StringWriter();
```

```
val csvWriter = new CSVWriter(stringWriter);  
csvWriter.writeAll(people.toList)  
Iterator(stringWriter.toString)  
}.saveAsTextFile(outFile)
```

你也许注意到了，前面的例子仅能处理我们已知正要输出的所有字段的情况。然而，如果有些字段名是需要运行时用户输入的，我们就得换一种方法。最简单的办法是遍历所有的数据，提取出那些不重复的主键，然后以另一种途径输出。

## SequenceFile

SequenceFile 是一种流行的 Hadoop 文件格式，由一些带 key/value 元组的无格式文件组成。它具有同步标志，允许 Spark 定位到文件中的某处后再次同步到记录边界。这使得 Spark 可以从多个节点高效的并行读取 SequenceFile。对于 Hadoop 的 MapReduce 任务，SequenceFile 也是一种常见的输入输出格式。如果你正工作在已存在的 Hadoop 系统，这是个使用 SequenceFile 的好机会，你的数据可用。

SequenceFile 由一些实现了 Hadoop 的 Writable 接口的元素组成，Hadoop 用的是自定义的系列化框架。表 5-2 列出了一些常见的类型及其对应的 Writable 类。经验规则是试着将单词 Writable 加在你的类名的后面看看是否是一个 org.apache.hadoop.io.Writable 的已知子类。如果你没找到你要输出的数据（比如自定义类）的 Writable 子类，那就实现一个自己的 Writable 类，重写 org.apache.hadoop.io.Writable 类中的 readFields 和 write 方法。



Hadoop 的 RecordReader 对每一个记录重用同一个对象，所以像这样对你读入的 RDD 直接调用 cache() 会失败。应该是增加一个简单的 map() 操作并缓存结果。此外，很多 Hadoop 的 Writable 类没实现 java.io.Serializable 接口，要使这些在 RDD 中工作，无论如何都要用 map() 将它们转换一下。

表格 5-2 对应于 Hadoop Writable 类型

Scala 类型	Java 类型	Hadoop Writable
Int	Integer	IntWritable 或 VIntWritable <sup>2</sup>
Long	Long	LongWritable 或 VLongWritable <sup>2</sup>
Float	Float	FloatWritable
Double	Double	DoubleWritable
Boolean	Boolean	BooleanWritable
Array[Byte]	byte[]	BytesWritable
String	String	Text
Array[T]	T[]	ArrayWritable<TW> <sup>3</sup>
List[T]	List<T>	ArrayWritable<TW> <sup>3</sup>
Map[A, B]	Map<A, B>	MapWritable<AW, TW> <sup>3</sup>

在 Spark 1.0 及之前的版本，SequenceFile 仅对 Java 和 Scala 可用。不过 Spark 1.1 为 Python 增加了加载和保存 SequenceFile 的能力。然而你必须用 Java 和 Scala 定义定制的 Writable 类。Python 的 Spark API 只知道如何转换在 Hadoop 里可用的基本 Writable 类到 Python，并对其他类会基于它们可用的 getter 方法努力。

### 加载 SequenceFile

Spark 有特定的读取 SequenceFile 的 API，在 SparkContext 中调用 sequenceFile(path, keyClass, valueClass, minPartitions)。之前提到过 SequenceFile 用 Writable 类处理，所以 keyClass 和 valueClass 都必须是正确的 Writable 类。思考一下从 SequenceFile 中加载人和他们看到的熊猫数量，这里 keyClass 应该是 Text，而 valueClass 应该是 IntWritable 或者 VIntWritable。简单起见，在示例 5-20 到 5-22 中使用 IntWritable。

<sup>2</sup> int 和 long 通常保存为固定长度。保存数字 12 和保存 2 的 30 次方占用相同数量的空间。如果你有大量的较小的数字，使用变长类型的 VIntWritable 或 VLongWritable，它们用更少的位数来保存小数字

<sup>3</sup> 模板类型必须是 Writable 类型。

示例 5-20. Python 中加载 SequenceFile

```
val data = sc.sequenceFile(inFile,  
"org.apache.hadoop.io.Text", "org.apache.hadoop.io.IntWritable")
```

示例 5-21. Scala 中加载 SequenceFile

```
val data = sc.sequenceFile(inFile, classOf[Text], classOf[IntWritable]).  
map{case (x, y) => (x.toString, y.get())}
```

示例 5-21. Java 中加载 SequenceFile

```
public static class ConvertToNativeTypes implements  
PairFunction<Tuple2<Text, IntWritable>, String, Integer> {  
    public Tuple2<String, Integer> call(Tuple2<Text, IntWritable> record) {  
        return new Tuple2(record._1.toString(), record._2.get());  
    }  
}  
  
JavaPairRDD<Text, IntWritable> input = sc.sequenceFile(fileName, Text.class,  
    IntWritable.class);  
JavaPairRDD<String, Integer> result = input.mapToPair(  
    new ConvertToNativeTypes());
```



在 Scala 中有个便利函数能自动将 Writable 类转换成对应的 Scala 类型。不用指定 keyClass 和 valueClass，我们可以调用 sequenceFile[Key, Value](path, minPartitions)就能得到原生 Scala 类型的 RDD。

## 保存 SequenceFile

在 Scala 中写数据到 SequenceFile 相当类似。首先，因为 SequenceFile 是键值对，我们需要一个 SequenceFile 可以输出的类型的 Pair RDD。许多原生 Scala 类型都存在 Scala 类型和 Hadoop 的 Writable 类之间的隐式转换，所以如果要输出原生类型，你就调用 saveAsSequenceFile(path)来保存 PairRDD，就会为你输出数据了。如果不存在键值类型到 Writable 的自动转换，或者想使用变长类型（比如 VIntWritable），你可以在保存之前对数据做映射并转换。考虑下要输出前一个例子中加载的数据（人以及人看到的熊猫数量），看看示例 5-23。

示例 5-23. Scala 中保存 SequenceFile

```
val data = sc.parallelize(List(("Panda", 3), ("Kay", 6), ("Snail", 2)))  
data.saveAsSequenceFile(outputFile)
```

在 Java 中保存 SequenceFile 要稍复杂点, 因为 JavaPairRDD 缺 saveSequenceFile() 方法。不过我们可以用 Spark 的能力来保存自定义 Hadoop 格式, 并且在 84 页的“Hadoop 输入输出格式”中会展示如何用 Java 保存 SequenceFile。

## ObjectFile

ObjectFile 只是简单的虚假的包装了一下的 SequenceFile, 允许我们保存仅含有值的 RDD。和 SequenceFile 不同, 在 ObjectFile 中值是通过 Java 的序列化输出的。



如果你修改你的类——比如增加或删除字段——旧的 Object File 就不再可读了。Object File 使用 Java 序列化, 对跨版本的兼容性管理能有些支持, 但是需要程序员全力以赴。

Object File 使用 Java 的序列化会带来些影响。不像普通的 SequenceFile, 其输出会和 Hadoop 输出同样的对象不相同。也不像其他格式, Object File 主要是用于 Spark 任务和其他的 Spark 任务通信。Java 的序列化也比较慢。

保存 Object File 就是简单的对 RDD 调用 saveAsObjectFile() 即可。读取 Object File 也十分简单: 在 SparkContext 中调用 objectFile() 传入路径参数就能返回 RDD。

有了以上这么多警告, 你可能会想为什么有人会用它。主要的原因是用 Object File 的话, 几乎不需要做任何工作就能保存任意对象。

ObjectFile 在 Python 中不可用。不过 Python 的 RDD 和 SparkContext 支持 saveAsPickleFile() 和 pickleFile() 方法, 用的是 Python 的 pickle 序列化库。然而, 和 Object File 有同样的警告: pickle 库很慢, 并且如果修改了类, 旧文件不可读。



## Hadoop 的输入输出格式

除了 Spark 封装的格式之外，我们还可以跟任何 Hadoop 支持的格式交互。Spark 支持新版本和旧版本的 Hadoop 文件 API，提供了很好地弹性<sup>4</sup>。

### 用其他的 Hadoop 输入格式加载

要使用新版 Hadoop API 读取文件，我们需要告诉 Spark 一点东西。函数 `newAPIHadoopFile` 带一个路径参数和三个类参数。第一个是用来表达输入格式的“格式”类。类似的函数 `hadoopFile()` 是为了处理旧 API 实现的 Hadoop 输入格式而存在。下一个类是主键的类型，最后一个是值的类型。如果要指定额外的 Hadoop 属性配置，也可以传入一个 `conf` 对象。

最简单的一个 Hadoop 输入格式是 `KeyValueTextInputFormat`，用于从文本中读取 `key/value` 元组数据（见示例 5-24）。每一行单独处理，主键和值用 `tab` 字符分隔。该格式源自 Hadoop，所以我们使用它不需要处理任何额外的依赖。

*示例 5-24. Scala 中用旧版 API 加载 `KeyValueTextInputFormat`()*

```
val input = sc.hadoopFile[Text, Text, KeyValueTextInputFormat](inputFile).map{  
  case (x, y) => (x.toString, y.toString)  
}
```

我们知道加载 JSON 数据是通过以文本文件加载后再解析，但是我们也可以用自定义 Hadoop 输入格式来加载。这个例子需要为压缩设置一些额外的信息，所以暂时跳过。Twitter 的 `Elephant Bird` 包支持大量的数据格式，包括 JSON，Lucene，Protocol buffers 相关格式等。这个包也可以工作于新旧版本的 Hadoop 文件 API。为了阐述从 Spark 中调用新版 API，看看用 `Lzo JsonInputFormat` 类加载 LZO 压缩过的 JSON 数据的例子，见示例 5-25。

*示例 5-25. Scala 中用 `Elephant Bird` 包加载 LZO 压缩过的 JSON 数据*

```
val input = sc.newAPIHadoopFile(inputFile, classOf[Lzo.JsonInputFormat],  
  classOf[LongWritable], classOf[MapWritable], conf)  
// Each MapWritable in "input" represents a JSON object
```

<sup>4</sup> Hadoop 在其生命周期的早期增加了新的 MapReduce API，但有些库还再用旧的。



要支持 LZO 需要安装 `hadoop-lzo` 包，并将 Spark 指向原生库。如果你安装 Debian 的包，添加参数 `--driver-library-path /usr/lib/hadoop/lib/native/` `--driver-class-path /usr/lib/hadoop/lib/` 到你的提交命令也行。

使用旧版 Hadoop API 读文件从使用的角度看是一样的，除了要提供一个旧版的 `InputClass`。Spark 的许多便利函数（如 `sequenceFile()`）都是用旧版 API 实现的。

### 用 Hadoop 输出格式保存

从某种程度上我们已经研究过了 `SequenceFile`，不过在 Java 里没有保存 pair RDD 的便利函数。我们将以此来举例说明如何使用旧版的 Hadoop 格式 API(见示例 5-26)，对于新版 API(`saveAsNewAPIHadoopFile`)也类似。

示例 5-26. Java 中保存 `SequenceFile`

```
public static class ConvertToWritableTypes implements
PairFunction<Tuple2<String, Integer>, Text, IntWritable> {
    public Tuple2<Text, IntWritable> call(Tuple2<String, Integer> record) {
        return new Tuple2(new Text(record._1), new IntWritable(record._2));
    }
}

JavaPairRDD<String, Integer> rdd = sc.parallelizePairs(input);
JavaPairRDD<Text, IntWritable> result = rdd.mapToPair(new ConvertToWritableTypes());
result.saveAsHadoopFile(fileName, Text.class, IntWritable.class,
    SequenceFileOutputFormat.class);
```

### 非文件系统数据源

除 `hadoopFile()/saveAsHadoopFile()` 函数系列之外，你还可以使用 `hadoopDataset / saveAsHadoopDataset` 和 `newAPIHadoopDataset / saveAsNewAPIHadoopDataset` 来访问 Hadoop 支持的非文件系统的存储格式。比如说类似 HBase 和 MongoDB 这样的一些 key/value 存储，提供了直接从 key/value 存储中读取的 Hadoop 输入格式。你可以在 Spark 中轻松的使用这些输入格式。

`HadoopDataset()` 函数系列只带了一个 `Configuration` 对象做参数，你可以设置需要访问你的数据源的 `hadoop` 属性。配置和 Hadoop 的 MapReduce 任务的配置一样，所以你可以按照 MapReduce 中访问数据源的指令并将对象传给 Spark。例如 96 页的“HBase”部分展示的如何用 `newAPIHadoopDataset` 从 HBase 加载数据。

## 示例：Protocol buffers

Protocol buffers<sup>5</sup>最初是 Google 为了内部的远程过程调用(RPC)开发的，现在已经开源了。Protocol buffers(PBs)是结构化数据，字段和字段类型都有清楚的定义。被优化为快速编解码，同时占用最少的空间。和 XML 对比，PBs 比其小 3 到 10 倍，编解码快 20 到 100 倍。由于 PB 具有编码一致性，我们可以通过多种方式创建由许多 PB 消息组成的文件。

Protocol buffer 使用特定领域的语言来定义，然后可以用 Protocol buffer 编译器来生成各种语言（包括所有 Spark 支持的语言）的存取器方法。因为 PBs 的目标是最小化占用空间，所以编码后的数据不是自描述的，因为编码这些描述信息会占用额外的空间。这表示要解析 PB 格式的数据，必须了解 protocol buffer 的定义。

PBs 由字段组成，字段分为可选或必须或者可重复三类。当你解析数据时，丢失可选字段结果不会出错，但是丢失必选字段会导致解析失败。因此，当你添加新字段到已存在的 protocol buffers 时，最好是将新字段作为可选字段添加。毕竟不是每个人都会同时升级（即使同时升级，你也许会想要读老的数据）。

PB 的字段可以使预定义类型或者是其他 PB 消息。这些类型包括 string, int32, enums 等等。当然这并不是 protocol buffers 的完整介绍，如果你有兴趣的话，可以查阅 Protocol Buffers 的网站。

在示例 5-27 中我们将看到从单个 protocol buffers 格式中提取很多 VenueResponse 对象。VenueResponse 示例很简单，只有一个可重复字段，包含了另一个有必选字段，可选字段和枚举字段的消息。

示例 5-27. protocol buffer 定义的样例

```
message Venue {  
    required int32 id = 1;  
    required string name = 2;  
    required VenueType type = 3;  
    optional string address = 4;
```

<sup>5</sup> 有时候也叫作 pbs 或者 protocolbufs

```

enum VenueType {
    COFFEESHOP = 0;
    WORKPLACE = 1;
    CLUB = 2;
    OMNOMNOM = 3;
    OTHER = 4;
}

message VenueResponse {
    repeated Venue results = 1;
}

```

前一节用来加载 JSON 数据的 Twitter 的 Elephant Bird 库也支持加载和保存 protocol buffers。通过示例 5-28 看看输出些 Venues。

示例 5-28. Scala 中用 Elephant Bird protocol buffer 输出

```

val job = new Job()
val conf = job.getConfiguration
LzoProtobufBlockOutputFormat.setClassConf(classOf[Places.Venue], conf);
val dnaLounge = Places.Venue.newBuilder()
dnaLounge.setId(1);
dnaLounge.setName("DNA Lounge")
dnaLounge.setType(Places.Venue.VenueType.CLUB)
val data = sc.parallelize(List(dnaLounge.build()))
val outputData = data.map{ pb =>
    val protoWritable = ProtobufWritable.newInstance(classOf[Places.Venue]);
    protoWritable.set(pb)
    (null, protoWritable)
}
outputData.saveAsNewAPIHadoopFile(outputFile, classOf[Text],
classOf[ProtobufWritable[Places.Venue]],
classOf[LzoProtobufBlockOutputFormat[ProtobufWritable[Places.Venue]]], conf)

```

完整版本的示例见本书源码。



当构建你自己的项目时，确保使用和 Spark 同一个版本的 protocol buffer 库。

本书写作时是 2.5 版本。

## 文件压缩

在处理大数据时我们发现我们经常需要使用压缩数据来节约存储空间和网络消

耗。对于大多数的 Hadoop 输出格式可以指定压缩编码器进行数据压缩。我们已经看到 Spark 的原生输入格式(textFile 和 sequenceFile)可以自动为我们处理一些压缩类型。当我们读压缩数据时，有些压缩编码器可用于自动猜测压缩类型。

这些压缩选项仅能应用于支持压缩的 Hadoop 格式，或者说那些被输出到文件系统的格式。数据库的 Hadoop 格式通常没有实现对压缩的支持，或者数据库本身被配置了压缩记录。

选择输出的压缩编码器会对以后使用数据的用户有较大的影响。在 Spark 这样的分布式系统中，一般我们会从多个不同的机器尝试读取数据。为此，每个 worker 节点需要能找到一条新记录的起始点。有些压缩格式没法支持，它们需要单个节点读取所有数据，因此很容易导致瓶颈。可以简单的从多个机器读取的格式被称为“可分割的”。表 5-3 列出了可用的压缩选项。

表格 5-3 压缩选项

格式	可分割	平均压缩速度	对文本的效率	Hadoop 压缩编码器	纯 JAVA	原生	说明
gzip	N	快	高	org.apache.hadoop.io.compress.GzipCodec	是	是	
lzo	Y <sup>6</sup>	非常快	中	com.hadoop.compression.lzo.LzoCodec	是	是	LZO 要在每个 worker 节点安装
bzip2	Y	慢	非常高	org.apache.hadoop.io.compress.BZip2Codec	是	是	可分割的版本要使用纯 JAVA 实现
zlib	N	慢	中	org.apache.hadoop.io.compress.DefaultCodec	是	是	Hadoop 默认的压缩编码器

<sup>6</sup> 依赖于使用的库

格式	可分割	平均压缩速度	对文本的效率	Hadoop 压缩编码器	纯 JAVA	原生	说明
snappy	N	非常快	低	org.apache.hadoop.io.compress.SnappyCodec	否	是	有纯 JAVA 的移植版本，但是对 Hadoop/Spark 不可用



虽然 Spark 的 `textFile()` 方法能处理压缩的输入，但是它会自动禁用可分割性，即使数据被压缩成可分割的压缩格式。如果你要处理一个巨大的压缩文件，考虑跳过 Spark 的封装，使用 `hadoopFile` 或 `newAPIHadoopFile()` 并指定正确的编码器来处理吧。

有些输入格式（如 `SequenceFile`）仅允许对 `key/value` 数据中的 `value` 部分压缩，对于查找来说很有用。其他的输入格式都有自己的压缩控制：比如 Twitter 的 Elephant Bird 库中的很多格式使用 LZO 压缩数据。

## 文件系统

Spark 支持许多文件系统的读写，我们可以使用任何我们想要的格式。

### 本地/“普通”文件系统

虽然 Spark 支持从本地文件系统加载文件，但是要求该文件在每个节点上的同一路径下都可用。

有些网络文件系统，比如 NFS，AFS 和 MapR 的 NFS 层，是以普通的文件系统被暴露给用户的。如果你的数据已经在这些文件系统中，你也可以用其作为输入。只是需要在路径前指定 `file://` 前缀。Spark 也能处理他们，只要每个节点上这些文件系统都挂在到同一个路径下（见示例 5-29）。

示例 5-29. Scala 中从本地文件系统加载压缩的文本文件

```
val rdd = sc.textFile("file:///home/holden/happypandas.gz")
```

如果你的文件不是在所有节点上都有，你可以不通过 Spark 而是从驱动程序本地加载数据，然后调用 `parallelize()` 分发内容到各 worker 节点。但是这种方式比较慢，建议将文件放到像 HDFS，NFS 或 S3 这样的共享文件系统中。

## Amazon S3

Amazon S3 是存储大数据的一个日益流行的选择。当你的计算节点位于 Amazon EC2 内部时，S3 特别快。但是如果你不得经过公共互联网，性能就特别糟糕。

要从 Spark 来访问 S3，你需要先设置好环境变量 `AWS_ACCESS_KEY_ID` 和 `AWS_SECRET_ACCESS_KEY` 为 S3 证书。可以从 Amazon Web 服务控制台创建这些证书。然后向 Spark 的文件输入方法传入一个以 `s3n://` 开头的路径，形如 `s3n://bucket/path-within-bucket`。和所有其他文件系统一样，Spark 对 S3 也支持路径通配符，比如 `s3n://bucket/my-files/*.txt`。

如果你从 Amazon 得到一个访问许可错误，请确保你的账号已经指定了对 bucket 的“读取”和“列示”两个权限。Spark 需要列出 bucket 中的对象以确定你想要读的是哪一个。

## HDFS

Hadoop 分布式文件系统(HDFS)是一种流行的分布式文件系统，Spark 在上面运行良好。HDFS 被设计为工作在商用硬件上，并且提供高吞吐处理时节点失败的容错。Spark 和 Hadoop 可以在同一个机器上部署，这样 Spark 就能利用数据的局部性避免网络开销。

Spark 使用 HDFS 很简单，为你的输入输出指定 `hdfs://master:port/path` 即可。



对 Hadoop 的不同版本，HDFS 协议有变化。如果你用的是和 Spark 当前运行的 Hadoop 版本不同的版本编译的 Spark，会出现错误。默认情况下 Spark 是用 Hadoop 1.0.4 编译。如果你用源码编译，可以指定环境变量 `SPARK_HADOOP_VERSION`=不同的版本来编译。或者根据你运行的 Hadoop 的版本下载预编译好的不同版本的 Spark。



## 结构化数据和 Spark SQL

Spark SQL 是 Spark 1.0 新增的组件，并且迅速成为了 Spark 处理结构化和半结构化数据最受偏爱的方式。对于结构化数据，意味着数据有一个模式，也就是说所有的数据都有一组一致的字段。Spark 支持多种结构化数据，因为它理解数据的模式，可以高效的从数据中仅读取你需要的字段。在第九章会详细介绍 Spark SQL，目前的话，我们会展示如何用它从几种常见的源来加载数据。

一概而论，我们给 Spark SQL 一个 sql 语句对数据源执行（选择某些字段或字段的函数），然后得到数据行对象的 RDD，每个记录一个对象。在 Java 和 Scala 中，行对象允许基于列号来访问。每个行对象都有个 `get()` 操作，返回可转换的一种通用类型，并且为常用基本类型特化了 `get()` 方法(如 `getFloat()`, `getInt()`, `getLong()`, `getString()`, `getShort()`和 `getBoolean()`)。在 Python 中用 `row.column_name` 和 `row[column_number]`来访问元素即可。

## Apache Hive

Hadoop 中常见的结构化数据源是 Apache Hive。Hive 能以各种格式，从文本格式到列格式，存储数据表到 HDFS 或其他文件系统中。Spark SQL 可以加载任何 Hive 支持的表。

要链接 Spark SQL 到 Hive，需要提供 Hive 的配置。复制你的 `hive-site.xml` 到 Spark 的 `./conf/` 目录。一旦弄好后，你就可以创建 `HiveContext` 对象，Spark SQL 的入口点，然后写 Hive 查询语言(HQL)来查询你的表获得行对象的 RDD。见示例 5-30 到 5-32。

*示例 5-30. Python 中创建 HiveContext 并选择数据*

```
from pyspark.sql import HiveContext
hiveCtx = HiveContext(sc)
rows = hiveCtx.sql("SELECT name, age FROM users")
firstRow = rows.first()
print firstRow.name
```

*示例 5-31. Scala 中创建 HiveContext 并选择数据*

```
import org.apache.spark.sql.hive.HiveContext
val hiveCtx = new org.apache.spark.sql.hive.HiveContext(sc)
val rows = hiveCtx.sql("SELECT name, age FROM users")
```



```
val firstRow = rows.first()
println(firstRow.getString(0)) // Field 0 is the name
```

示例 5-32. Java 中创建 HiveContext 并选择数据

```
import org.apache.spark.sql.hive.HiveContext;

import org.apache.spark.sql.Row;
import org.apache.spark.sql.SchemaRDD;
HiveContext hiveCtx = new HiveContext(sc);
SchemaRDD rows = hiveCtx.sql("SELECT name, age FROM users");
Row firstRow = rows.first();
System.out.println(firstRow.getString(0)); // Field 0 is the name
```

在 170 页的”Apache Hive”中我们会谈到更多从 Hive 加载数据的细节。

## JSON 文件

如果你有所有记录的模式一致的 JSON 数据，Spark SQL 可以推断出模式并加载数据为行对象，这使得提取你需要的字段变得十分简单。要加载 JSON 数据，首先像使用 Hive 一样创建 HiveContext 对象(本例不需要安装 Hive，因为不需要 hive-site.xml 文件)，然后使用 HiveContext.jsonFile 方法得到整个文件的行对象的 RDD。除了使用行对象，你还可以将 RDD 注册为表并从中选择特定的列。比如，假设我们有包含如示例 5-33 所示格式的微博的 JSON 文件，每个一行。

示例 5-33. 微博的 JSON 样例

```
{"user": {"name": "Holden", "location": "San Francisco"}, "text": "Nice day out today"}
{"user": {"name": "Matei", "location": "Berkeley"}, "text": "Even nicer here :)}"
```

我们可以只选择用户名和文本字段，见示例 5-34 到 5-36。

示例 5-34. Python 中用 Spark SQL 加载 JSON

```
tweets = hiveCtx.jsonFile("tweets.json")
tweets.registerTempTable("tweets")
results = hiveCtx.sql("SELECT user.name, text FROM tweets")
```

示例 5-35. Scala 中用 Spark SQL 加载 JSON

```
val tweets = hiveCtx.jsonFile("tweets.json")
tweets.registerTempTable("tweets")
val results = hiveCtx.sql("SELECT user.name, text FROM tweets")
```

示例 5-37. Java 中用 Spark SQL 加载 JSON

```
SchemaRDD tweets = hiveCtx.jsonFile(jsonFile);
tweets.registerTempTable("tweets");
SchemaRDD results = hiveCtx.sql("SELECT user.name, text FROM tweets");
```

在 172 页的“JSON”中我们会讨论更多关于如何用 Spark SQL 加载 JSON 以及访问它的模式。此外，Spark SQL 可不止支持加载数据，还包括查询数据，通过更复杂的方式和 RDD 合并数据，对其运行自定义函数等，这些在第九章会讲。

## 数据库

Spark 可以通过 Hadoop 连接器和定制的 Spark 连接器访问多种数据库。本节会讨论四种常见的连接器。

### JDBC

Spark 能从支持 JDBC 的关系数据库加载数据，包括 MySQL，Postgres 及其它系统。要访问这些数据，我们构造 `org.apache.spark.rdd.JdbcRDD`，并为其提供 `SparkContext` 和其他参数。示例 5-37 会帮你度过 `JdbcRDD` 访问 MySQL 的难关。

示例 5-37. Scala 使用 `JdbcRDD`

```
def createConnection() = {
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    DriverManager.getConnection("jdbc:mysql://localhost/test?user=holden");
}
def extractValues(r: ResultSet) = {
    (r.getInt(1), r.getString(2))
}
val data = new JdbcRDD(sc,
    createConnection, "SELECT * FROM panda WHERE ? <= id AND id <= ?",
    lowerBound = 1, upperBound = 3, numPartitions = 2, mapRow = extractValues)
println(data.collect().toList)
```

`Jdbc` 的几个参数：

- 第一个参数，我们提供了一个函数和数据库建立连接。这使得每个节点在完成所有连接要求的配置以后都创建自己的连接来加载数据。

- 第二个参数，我们提供了一个查询读取一定范围内的数据，就是为查询参数指定上下的边界值。这些参数允许 Spark 在不同的机器上查询不同范围的数据，这就不会有从单个机器试图加载所有数据的瓶颈问题<sup>7</sup>。
- 最后的参数也是一个函数，用于将输出的每一行从 `java.sql.ResultSet` 转换为对维护数据有利的格式。示例 5-37 中，我们得到的是 `(Int, String)` 元组。如果忽略该参数，Spark 会自动转换每一行到一个对象数组中。

和使用别的数据源一样，当使用 Jdbc 时，确保你的数据库能承载从 Spark 并行读取的负载。如果你想查询离线数据而不是活动数据库，你总是可以用数据库的导出功能将数据导出成文本文件。

## Cassandra

Spark 对 Cassandra 的支持自从引入 DataStax 的开源 Spark Cassandra 连接器以后性能大幅提升。由于该连接器目前还不是 Spark 的一部分，你需要在构建文件中自己添加依赖。Cassandra 还没有使用 Spark SQL，但是它能返回一个 `CassandraRow` 类型的 RDD 对象，与 Spark SQL 的 `Row` 对象有一些相同的方法，见示例 3-28 和 3-29。当前 Cassandra 连接器在 Java 和 Scala 中可用。

示例 5-38. *sbt requirements for Cassandra connector*

```
"com.datastax.spark" %% "spark-cassandra-connector" % "1.0.0-rc5",
"com.datastax.spark" %% "spark-cassandra-connector-java" % "1.0.0-rc5"
```

示例 5-39. *Maven requirements for Cassandra connector*

```
<dependency> <!-- Cassandra -->
  <groupId>com.datastax.spark</groupId>
  <artifactId>spark-cassandra-connector</artifactId>
  <version>1.0.0-rc5</version>
</dependency>
<dependency> <!-- Cassandra -->
  <groupId>com.datastax.spark</groupId>
  <artifactId>spark-cassandra-connector-java</artifactId>
  <version>1.0.0-rc5</version>
</dependency>
```

<sup>7</sup> 如果你不知道有多少条记录，可以先手动执行一个 `count` 查询，靠它来确定上下限

Cassandra 连接器读取任务属性来确定连接哪个集群，这和 ElasticSearch 更类似。我们设置 `spark.cassandra.host` 来指出连接到哪个 Cassandra 集群。并且，如果你有用户名和密码信息要设置，也可以通过 `spark.cassandra.auth.username` 和 `spark.cassandra.auth.password` 来指定。假设我们只有一个 Cassandra 集群能连接，当我们创建 `SparkContext` 时可以设定它，见示例 5-40 和 5-41。

示例 5-40. Scala 中设置 Cassandra 属性

```
val conf = new SparkConf(true)
    .set("spark.cassandra.connection.host", "hostname")
val sc = new SparkContext(conf)
```

示例 5-41. Java 中设置 Cassandra 属性

```
SparkConf conf = new SparkConf(true)
    .set("spark.cassandra.connection.host", cassandraHost);
JavaSparkContext sc = new JavaSparkContext(
    sparkMaster, "basicquerycassandra", conf);
```

在 Scala 中，DataStax 的 Cassandra 连接器在 `SparkContext` 和 `RDD` 上还默默的提供了一些附加功能。让我们来导入隐式转换试着加载别的数据吧(示例 5-42)。

示例 5-42. Scala 中加载整个表为 key/value 数据的 RDD

```
// Implicits that add functions to the SparkContext & RDDs.
import com.datastax.spark.connector._
// Read entire table as an RDD. Assumes your table test was created as
// CREATE TABLE test.kv(key text PRIMARY KEY, value int);
val data = sc.cassandraTable("test", "kv")
// Print some basic stats on the value field.
data.map(row => row.getInt("value")).stats()
```

在 Java 中没有隐式转换，所以需要为该功能显式的转换 `SparkContext` 和 `RDD`(示例 5-43)。

示例 5-43. Java 中加载整个表为 key/value 数据的 RDD

```
import com.datastax.spark.connector.CassandraRow;
import static com.datastax.spark.connector.CassandraJavaUtil.javaFunctions;
// Read entire table as an RDD. Assumes your table test was created as
// CREATE TABLE test.kv(key text PRIMARY KEY, value int);
JavaRDD<CassandraRow> data = javaFunctions(sc).cassandraTable("test", "kv");
// Print some basic stats.
```

```
System.out.println(data.mapToDouble(new DoubleFunction<CassandraRow>() {
    public double call(CassandraRow row) { return row.getInt("value"); }
}).stats());
```

除了加载整个表之外，还可以查询数据子集。我们可以通过 `cassandraTable()` 调用中添加 `where` 字句来约束数据。例如，`sc.cassandraTable(...).where("key=?", "panda")`。

Cassandra 连接器也支持从各种类型的 RDD 将数据保存到 Cassandra 中。我们可以直接保存 `CassandraRow` 对象的 RDD，这对于表间复制数据很有用。我们还可以保存不是以行形式，而是指定了列映射的元组和列表，见示例 5-44。

*Example 5-44. Saving to Cassandra in Scala*

```
val rdd = sc.parallelize(List(Seq("moremagic", 1)))
rdd.saveToCassandra("test", "kv", SomeColumns("key", "value"))
```

本节只是简单的介绍 Cassandra 连接器。更多的信息请参考连接器的 Github。

## HBase

Spark 可以通过它的 Hadoop 输入格式来访问 HBase，具体的实现类是 `org.apache.hadoop.hbase.mapreduce.TableInputFormat`。该输入格式返回 `key/value` 对，其中 `key` 类型是 `org.apache.hadoop.hbase.io.ImmutableBytesWritable`，而 `value` 类型是 `org.apache.hadoop.hbase.client.Result`。`Result` 类提供了各种基于它们的列簇返回值的方法，在其 API 文档中有讲到。

要用 Spark 来访问 HBase 数据库，你可以用正确的 Hadoop 输入格式类来调用 `SparkContext.newAPIHadoopRDD`，见 Scala 示例 5-45。

*示例 5-45. Scala 中从 HBase 读取数据的示例*

```
import org.apache.hadoop.hbase.HBaseConfiguration
import org.apache.hadoop.hbase.client.Result
import org.apache.hadoop.hbase.io.ImmutableBytesWritable
import org.apache.hadoop.hbase.mapreduce.TableInputFormat
val conf = HBaseConfiguration.create()
conf.set(TableInputFormat.INPUT_TABLE, "tablename") // which table to scan
val rdd = sc.newAPIHadoopRDD(
    conf, classOf[TableInputFormat], classOf[ImmutableBytesWritable], classOf[Result])
```

为了优化从 HBase 中读取数据，TableInputFormat 类包括了多个设置项，比如限制只扫描一组列，限制扫描时间段。你可以从 TableInputFormat 的 API 文档中找到这些选项，在传入 Spark 之前设置到你的 HBaseConfiguration 即可。

## Elasticsearch

Spark 可以使用 Elasticsearch-Hadoop 从 Elasticsearch 读写数据。Elasticsearch 是基于 Lucene 的一个新的开源检索系统。

Elasticsearch 的连接器和我们已经研究过的其他连接器有些不一样。它忽略了路径信息，而是依赖于在 SparkContext 中设置的信息。Elasticsearch 的 OutputFormat 连接器也没有使用 Spark 封装的类型，而是用的 saveAsHadoopSet。这意味着我们需要手动设置更多的属性。看看示例 5-46 和 5-47 是如何读写简单数据到 Elasticsearch 的。



最新的 Elasticsearch 的 Spark 连接器更好用，支持返回 Spark SQL 行。但这个连接器还有用，因为行转换还不支持 Elasticsearch 中的所有原生类型。

示例 5-46. Elasticsearch output in Scala

```
val jobConf = new JobConf(sc.hadoopConfiguration)
jobConf.set("mapred.output.format.class", "org.elasticsearch.hadoop.mr.EsOutputFormat")
jobConf.setOutputCommitter(classOf[FileOutputCommitter])
jobConf.set(ConfigurationOptions.ES_RESOURCE_WRITE, "twitter/tweets")
jobConf.set(ConfigurationOptions.ES_NODES, "localhost")
FileOutputFormat.setOutputPath(jobConf, new Path("-"))
output.saveAsHadoopDataset(jobConf)
```

示例 5-47. Elasticsearch input in Scala

```
def mapWritableToInput(in: MapWritable): Map[String, String] = {
  in.map{case (k, v) => (k.toString, v.toString)}.toMap
}
val jobConf = new JobConf(sc.hadoopConfiguration)
jobConf.set(ConfigurationOptions.ES_RESOURCE_READ, args(1))
jobConf.set(ConfigurationOptions.ES_NODES, args(2))
val currentTweets = sc.hadoopRDD(jobConf,
  classOf[EsInputFormat[Object, MapWritable]], classOf[Object],
  classOf[MapWritable])
// Extract only the map
```

```
// Convert the MapWritable[Text, Text] to Map[String, String]
val tweets = currentTweets.map{ case (key, value) => mapWritableToInput(value) }
```

和我们研究过的其他一些连接器比较，这个复杂一点。但是对了解这类型的连接器如何工作是有参考价值的。



在输出方面，Elasticsearch 可以做映射推理，但偶尔会推断类型不正确。所以如果你要保存非字符串的数据类型，最好是显式的设置好映射。

## 总结

随着本章的结束，你现在应该可以用 Spark 加载数据并将计算结果以对你有用的格式保存。我们研究了用于我们的数据的一些不同的格式，压缩选项以及数据如何被处理的影响。现在我们可以加载和保存大数据集了，随后的章节将研究写出更高效，更强大的 Spark 程序的方法。

## 第六章

# Spark 高级编程

### 介绍

本章介绍各种之前没提到的 Spark 高级编程特性。引入两种共享变量：用于聚合信息的累加器和高效分发大型数据的广播变量。依赖于对 RDD 的已存在的变换，引入对设置开销大的任务进行批处理操作，比如查询数据库。为拓展我们可访问的工具的范围，我们还将介绍 Spark 与外部语言的交互，比如 R 语言脚本。

我们构造了一个以业务无线电台操作员的呼叫日志作为输入的例子贯穿本章。这些日志至少包括联系的站点的呼号(Call signs)。呼号是由国家分配，每个国家有它自己的呼号范围，这样我们就能查找涉及的国家。有些日志中有操作员的物理位置，我们可以用它来确定距离。示例 6-1 是日志记录的例子。本书的示例库包括用来查找呼叫日志的呼号列表并处理结果。

示例 6-1. JSON 格式的呼叫日志记录示例，移除了一些字段

```
{"address":"address here", "band":"40m", "callsign":"KK6JLK", "city":"SUNNYVALE",  
"contactlat":"37.384733", "contactlong":"-122.032164",  
"county":"Santa Clara", "dxcc":"291", "fullname":"MATTHEW McPherrin",  
"id":57779, "mode":"FM", "mylat":"37.751952821", "mylong":"-122.4208688735", ...}
```

要看的第一组 Spark 特性是共享变量，这是用于 Spark 任务中的一种特定类型的变量。在我们的示例中用 Spark 的共享变量来计算非致命错误的条件的数量以及分发一个大型查找表的数据。



当我们的任务涉及过大的设置时间，比如创建数据库连接或者随机数生成器，跨多个数据项来共享这些设置工作会很有用。使用对端呼号查找数据库，我们来研究下如何通过基于每个分区基础上的操作来重用设置。

除了 Spark 直接支持的语言外，系统也可以调用到其他语言写的程序中。本章将介绍如何用 Spark 的语言无关的 `pipe()` 函数通过标准输入输出与其他程序交互。我们将使用 `pipe()` 方法访问 R 语言的库来计算无线电台操作员联系的距离。

最后，类似处理 `key/value` 元组的工具，Spark 也有方法可以处理数值数据。通过用无线电台呼叫日志数据计算的距离来去除异常值的例子阐述这些方法。

## 累加器

当我们普通的传递函数给 Spark，比如 `map()` 函数或者 `filter()` 函数的条件，它们可以用在驱动程序中外部定义的变量。但每个任务在集群上运行时会得到每个变量的副本，并且对这些变量更新也不会传送回驱动程序。Spark 的共享变量，也就是累加器和广播变量，为了两种通信模式放宽了这个限制：聚合结果和广播。

第一类共享变量，累加器，提供了简单的方法将从各 `worker` 节点聚合后的值传回驱动程序。累加器最常见的用法是调试时计算任务执行期间发生的事件数量。比方说我们想加载所有呼号的列表，可以从文件中获取日志。但我们也对输入文件中有多少空行感兴趣（也许我们不希望在一个有效文件中见到如此多的空行）。示例 6-2 到 6-4 演示了该场景。

示例 6-2. Python 中用累加器计算空行数

```
file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)
def extractCallSigns(line):
    global blankLines # Make the global variable accessible
    if (line == ""):
        blankLines += 1
    return line.split(" ")

callSigns = file.flatMap(extractCallSigns)
```

```
callSigns.saveAsTextFile(outputDir + "/callsigns")
print "Blank lines: %d" % blankLines.value
```

示例 6-3. Scala 中用累加器计算空行数

```
val sc = new SparkContext(...)
val file = sc.textFile("file.txt")
val blankLines = sc.accumulator(0) // Create an Accumulator[Int] initialized to 0
val callSigns = file.flatMap(line => {
    if (line == "") {
        blankLines += 1 // Add to the accumulator
    }
    line.split(" ")
})
callSigns.saveAsTextFile("output.txt")
println("Blank lines: " + blankLines.value)
```

示例 6-4. Java 中用累加器计算空行数

```
JavaRDD<String> rdd = sc.textFile(args[1]);
final Accumulator<Integer> blankLines = sc.accumulator(0);
JavaRDD<String> callSigns = rdd.flatMap(
    new FlatMapFunction<String, String>() { public Iterable<String> call(String line) {
        if (line.equals("")) {
            blankLines.add(1);
        }
        return Arrays.asList(line.split(" "));
    }
});
callSigns.saveAsTextFile("output.txt")
System.out.println("Blank lines: " + blankLines.value());
```

在这些例子中，我们创建了名为 `blankLines` 的 `Accumulator[Int]` 变量，然后无论何时在输入中看到空行时为其加 1。当变换结束时，打印这个变量的值。注意只有在执行完 `saveAsTextFile()` 动作时才能看到正确的值。因为 `map()` 变换是滞后的，因此累加器的递增效应仅在 `saveAsTextFile()` 动作发生后迫使 `map()` 变换执行。

当然，使用像 `reduce()` 这样的动作使整个 RDD 返回驱动程序也可以聚合出值，但有时候我们需要用简单的方式进行聚合，在处理 RDD 变换时可以对不同的比例或粒度来生成，而不是整个 RDD。前面的例子用累加器对加载的数据进行了错误计数，没有做单独的 `filter()` 或 `reduce()` 处理。

总而言之，累加器是下面这么工作的：

- 在驱动程序中通过调用 `SparkContext.accumulator(initialValue)` 创建，生成了一个有初值的累加器。返回类型是 `org.apache.spark.Accumulator[T]`，`T` 是初值的类型。
- Spark 中，Worker 的代码闭包可以用 `+=` 的方法添加到累加器（或 Java 中相加）。
- 驱动程序可以访问累加器的 `value` 属性访问值（或 Java 中调用 `value()` 和 `setValue()` 方法）。

注意，Worker 节点中的任务执行时不能访问累加器的 `value()` 方法。从这些任务的角度看，累加器就是只可写类型的变量。这使得累加器的实现很高效，不用每次更新都通信。

当有多个值要记录或者在并行程序中同一个值在多个地方要递增时（比如你想贯穿整个程序计算调用 JSON 解析库的次数），这里演示的计数类型会特别好用。例如，我们经常想知道数据损坏的百分比，或者允许后台失败几次。为了阻止产生有太多错误的垃圾数据，可以使用一个有效记录计数器和一个无效记录计数器。因为累加器只在驱动程序中可用，所以这就是我们进行检查的位置。

继续上一个例子，现在我们可以验证呼号仅当大多数输入有效时才输出。业余无线电台呼号是由国际通信联盟(ITU)的条款 19 规范的，据此我们构造了一个正则表达式来校验一致性，见示例 6-5。

*示例 6-5. Accumulator error count in Python*

*# Create Accumulators for validating call signs*

```
validSignCount = sc.accumulator(0)
invalidSignCount = sc.accumulator(0)
def validateSign(sign):
    global validSignCount, invalidSignCount
    if re.match(r"\A\d?[a-zA-Z]{1,2}\d{1,4}[a-zA-Z]{1,3}\Z", sign):
        validSignCount += 1
        return True
    else:
        invalidSignCount += 1
        return False
```

```

# Count the number of times we contacted each call sign
validSigns = callSigns.filter(validateSign)
contactCount = validSigns.map(lambda sign: (sign, 1)).reduceByKey(lambda (x, y): x + y)
# Force evaluation so the counters are populated
contactCount.count()
if invalidSignCount.value < 0.1 * validSignCount.value:
    contactCount.saveAsTextFile(outputDir + "/contactCount")
else:
    print "Too many errors: %d in %d" % (invalidSignCount.value, validSignCount.value)

```

## 累加器和容错

Spark 可以通过重新执行来自动处理失败的或者太慢的机器。比如，对一个分区进行 `map()` 的节点崩溃了，Spark 会从另一个节点重新运行。即使该节点没有崩溃，只是相比其他节点慢太多，Spark 也会在别的节点上抢先运行一个“推测性”的副本任务。如果完成了就用这个结果。即使没有节点出错，Spark 也可能会不得不重新运行任务来构造发生了内存溢出的缓存。因此，最终效果就是依赖于集群上发生的情况对相同数据的相同功能可能会运行多次。

和累加器要如何交互？最终结果是在动作中使用累加器，Spark 仅申请每个任务对每个累加器更新一次。所以如果我们想要一个绝对可靠地计数值，不管失败还是多次求值，必须将其放在类似 `foreach()` 的动作里。

对于在 RDD 变换中而不是动作中使用累加器，没有保证。变换中的累加器可能发发生多次更新。一个这样的可能没意识到的多次更新的情况是，当不常用的缓存被 LRU 算法淘汰，而随后又需要。这迫使 RDD 从其血统被重新计算，带来了计划外的副作用，随着血统的变换发送到驱动程序，又调用了对累加器的更新。因此，对于变换，累加器仅能用于调试目的。

以后的 Spark 版本可能会改变计数只能更新一次的行为，当前版本(1.2.0)没有 (译注：原文是 *does have*，笔误吧) 多次更新行为，所以累加器在变换中仅建议用于调试目的。

## 自定义累加器

目前我们已经知道了如何使用 Spark 内置的累计器类型：整数 (`Accumulator[Int]`)。除此之外，Spark 还支持 `Double`，`Long` 和 `Float`。另外，Spark 也有个 API 可自定义累加器类型和自定义聚合操作(例如找聚合后的最大值，而不是累加)。自定

义累加器需要继承 `AccumulatorParam`，见 Spark API 文档。除了加一个数值以外，我们可以做任何可交换和可结合的增加操作。例如我们可以记录目前为止的最大值，而不是累积的总数。



说一个操作 `op` 是可交换的，如果对所有的值 `a, b` 都有  $a \text{ op } b = b \text{ op } a$ 。

说一个操作是可结合的，如果对所有的值 `a, b, c` 都有  $(a \text{ op } b) \text{ op } c = a \text{ op } (b \text{ op } c)$ 。

例如 `sum` 和 `max` 都是可交换可结合的操作，经常被用于 Spark 累加器。

## 广播变量

Spark 的第二类共享变量是广播变量，允许程序高效的发送巨大的只读的值到所有的 `worker` 节点用于一个或多个 Spark 操作。当你的程序需要发送一个巨大的只读查找表到所有节点或者是机器学习算法中巨大的特征向量，广播变量可以派上用场。

回想一下，Spark 自动发送你的闭包中引用的所有变量到各 `worker` 节点。这很方便，但是也很低效。因为（1）默认的任务运行机制是为小任务优化，（2）实际上，你可能是在多个并行操作中用同一个变量，而 Spark 会为每个操作单独发送这些变量。例如，你想写个 Spark 程序通过在数组中对呼号的前缀匹配来查找国家。由于每个国家都有自己的前缀，虽然前缀长度不统一。这对业余无线电台呼号查找会有用。如果我们没经验，在 Spark 里写的代码可能会像示例 6-6 那样。

示例 6-6. Country lookup in Python

```
# Look up the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup.
signPrefixes = loadCallSignTable()
def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes)
    count = sign_count[1]
    return (country, count)
countryContactCounts = (contactCounts
    .map(processSignCount)
    .reduceByKey((lambda x, y: x+ y)))
```

这程序能跑。但是如果我们有一个巨大的表（比如 ip 地址，而不是呼号），`signPrefixes` 很容易就有好几兆字节大小，使得从驱动程序发送这个大数组到每个任务开销过大。尤其是如果我们之后还要用同一个 `signPrefixes` 对象，会要再发一遍。

我们可以通过把 `signPrefixes` 设为广播变量来修改这个问题。广播变量就是个简单的 `org.apache.spark.broadcast.Broadcast[T]` 类型的对象，封装了类型 `T`。可以在任务中通过调用广播变量对象的 `value` 方法来访问值。该值仅被发送到每个节点一次，使用高效的类似 BT 的通信机制。

使用广播变量，之前的例子看起来如同示例 6-7 到 6-9。

示例 6-7. Python 中用广播变量查找国家

```
# Look up the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup.
signPrefixes = sc.broadcast(loadCallSignTable())
def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes.value)
    count = sign_count[1]
    return (country, count)
countryContactCounts = (contactCounts
    .map(processSignCount)
    .reduceByKey((lambda x, y: x + y)))
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt")
```

示例 6-8. Scala 中用广播变量查找国家

```
// Look up the countries for each call sign for the
// contactCounts RDD. We load an array of call sign
// prefixes to country code to support this lookup.
val signPrefixes = sc.broadcast(loadCallSignTable())
val countryContactCounts = contactCounts.map{case (sign, count) =>
    val country = lookupInArray(sign, signPrefixes.value)
    (country, count)
}.reduceByKey((x, y) => x + y)
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt")
```

示例 6-9. Java 中用广播变量查找国家

```
// Read in the call sign table
// Look up the countries for each call sign in the
// contactCounts RDD
final Broadcast<String[]> signPrefixes = sc.broadcast(loadCallSignTable());
JavaPairRDD<String, Integer> countryContactCounts = contactCounts.mapToPair(
    new PairFunction<Tuple2<String, Integer>, String, Integer> () {
        public Tuple2<String, Integer> call(Tuple2<String, Integer> callSignCount) {
            String sign = callSignCount._1();
            String country = lookupCountry(sign, callSignInfo.value());
            return new Tuple2(country, callSignCount._2());
        }
    }).reduceByKey(new SumInts());
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt");
```

如上面的例子所示，使用广播变量的过程很简单：

- 对对象类型 T 调用 `SparkContext.broadcast` 创建 `Broadcast[T]`。只要是能序列化的类型。
- 用 `value` 属性来访问值（或者，Java 中是用 `value()` 方法）。
- 变量将发送到每个节点仅一次，并且是只读的（更新不会被传到别的节点）。

满足只读需求最简单的方法就是广播一个简单值或者是一个不可变对象的引用。这样的话，你就不能改变广播变量的值，除了在驱动程序里。然而，有时候广播一个可变对象会更方便或者更高效。如果是这样，就要有你自己来维护只读条件。就如我们对呼号前缀表 `Array[String]` 所做的，我们必须确保我们运行在 `worker` 节点上的代码不会尝试做像 `theArray=broadcastArray.value; theArray(0)=newValue` 这样的操作。因为当运行在 `worker` 节点时，这仅会将 `newValue` 赋给运行这个代码的 `worker` 节点本地的数组副本的第一个元素，不会改变任何其他 `worker` 节点上的 `broadcastArray.value`。

## 广播优化

当我们要广播一个巨大的值，选择一个快速，轻巧的数据序列化格式很重要。因为如果花太长时间去序列化或者通过网络发送序列化值会迅速成为瓶颈。特别是 Scala 和 Java API 中默认使用的 Java 序列化库除了对简单类型数组外非常低效。你可以通过用 `spark.serializer` 属性来指定一个不同的序列化库进行优化（第八章将描述如何使用 Kryo，一个快速的序列化库），或者为你的数据类型自定义序



列化函数（比如对 Java 序列化使用 `java.io.Externalizable` 接口，或者用 `reduce()` 为 Python 的 `pickle` 库自定义序列化）。

## 基于分区工作

基于分区来处理数据使得我们可以避免为每条数据做重复设置。像打开数据库连接或者创建随机数发生器这样的操作就是我们希望避免为每个元素都要设置步骤的例子。Spark 有分区版本的 `map` 和 `foreach` 函数可让你对 RDD 的每个分区仅运行这些代码一次来帮助减少这些操作的开销。

回到呼号的例子，有一个业余无线电台呼号的在线数据库，我们可以查询已记录的联系人公共列表。通过使用基于分区的操作，我们可以共享数据库连接池避免多个连接，也可以重用 JSON 解析器。如示例 6-10 到 6-12 所示，我们用 `mapPartitions()` 函数。该函数给我们输入 RDD 的每个分区的元素迭代器，期待我们返回结果的迭代器。

示例 6-10. *Shared connection pool in Python*

```
def processCallSigns(signs):
    """Lookup call signs using a connection pool"""
    # Create a connection pool
    http = urllib3.PoolManager()
    # the URL associated with each call sign record
    urls = map(lambda x: "http://73s.com/qsos/%s.json" % x, signs)
    # create the requests (non-blocking)
    requests = map(lambda x: (x, http.request('GET', x)), urls)
    # fetch the results
    result = map(lambda x: (x[0], json.loads(x[1].data)), requests)
    # remove any empty results and return
    return filter(lambda x: x[1] is not None, result)

def fetchCallSigns(input):
    """Fetch call signs"""
    return input.mapPartitions(lambda callSigns : processCallSigns(callSigns))

contactsContactList = fetchCallSigns(validSigns)
```

示例 6-11. *Shared connection pool and JSON parser in Scala*

```
val contactsContactLists = validSigns.distinct().mapPartitions{
signs =>
    val mapper = createMapper()
    val client = new HttpClient()
    client.start()
    // create http request
```



```

signs.map {sign =>
  createExchangeForSign(sign)
  // fetch responses
}.map { case (sign, exchange) =>
  (sign, readExchangeCallLog(mapper, exchange))
}.filter(x => x._2 != null) // Remove empty CallLogs
}

```

示例 6-12. Shared connection pool and JSON parser in Java

*// Use mapPartitions to reuse setup work.*

```

JavaPairRDD<String, CallLog[]> contactsContactLists =
  validCallSigns.mapPartitionsToPair(
    new PairFlatMapFunction<Iterator<String>, String, CallLog[]>() {
      public Iterable<Tuple2<String, CallLog[]>> call(Iterator<String> input) {
        // List for our results.
        ArrayList<Tuple2<String, CallLog[]>> callsignLogs = new ArrayList<>();
        ArrayList<Tuple2<String, ContentExchange>> requests = new ArrayList<>();
        ObjectMapper mapper = createMapper();
        HttpClient client = new HttpClient();
        try {
          client.start();
          while (input.hasNext()) {
            requests.add(createRequestForSign(input.next(), client));
          }
          for (Tuple2<String, ContentExchange> signExchange : requests) {
            callsignLogs.add(fetchResultFromRequest(mapper, signExchange));
          }
        } catch (Exception e) {
        }
        return callsignLogs;
      }
    });
System.out.println(StringUtils.join(contactsContactLists.collect(), ","));

```

基于分区进行操作时，Spark 给我们的函数分区元素的迭代器。要返回值也是返回迭代器。除了 `mapPartitions()`，Spark 还有其他一些基于分区操作，见表 6-1。

表格 6-1 基于分区操作

函数名	被调用的参数	返回值	基于 RDD[T]的函数签名
<code>mapPartitions()</code>	分区的元素迭代器	我们返回元素的迭代器	<code>f: (Iterator[T]) → Iterator[U]</code>
<code>mapPartitionsWithIndex()</code>	分区数和分区的元素迭代器	我们返回元素的迭代器	<code>f: (Int, Iterator[T]) → Iterator[U]</code>

函数名	被调用的参数	返回值	基于 RDD[T]的函数签名
foreachPartition()	分区的元素迭代器	无	f: (Iterator[T]) → Unit

除了避免设置工作外，有时我们能用 `mapPartitions()` 避免额外的对象创建开销。有时我们需要构造一个对象来聚合不同类型的结果。回忆下第三章我们计算平均值，我们用的一个方法是将数值 RDD 转换成元组 RDD，然后就能在 `reduce` 的步骤中追踪元素的个数。我们还可以每个分区创建元组一次，而不是每个元素创建，见示例 6-13 和 6-14。

示例 6-13. Average without `mapPartitions()` in Python

```
def combineCtrs(c1, c2):
    return (c1[0] + c2[0], c1[1] + c2[1])
def basicAvg(nums):
    """Compute the average"""
    nums.map(lambda num: (num, 1)).reduce(combineCtrs)
```

示例 6-14. Average with `mapPartitions()` in Python

```
def partitionCtr(nums):
    """Compute sumCounter for partition"""
    sumCount = [0, 0]
    for num in nums:
        sumCount[0] += num
        sumCount[1] += 1
    return [sumCount]
def fastAvg(nums):
    """Compute the avg"""
    sumCount = nums.mapPartitions(partitionCtr).reduce(combineCtrs)
    return sumCount[0] / float(sumCount[1])
```

## 管道输出到外部程序

绑定了三种语言可选择使用，你可能就有了编写 Spark 程序需要的所有选择。然而，如果 Python、Java 或 Scala 都不是你要的，那么 Spark 也提供了一种通用机制来输送数据到其他语言写的程序，如 R 语言脚本。

Spark 对 RDD 提供了 `pipe()` 方法。该方法让我们可以用我们想要的任何语言编写部分作业，只要该语言可以读写 Unix 的标准流。有了 `pipe()`，你可以写一个 RDD

的变换从标准输入以字符串来读取每个 RDD 元素，随意处理该字符串，然后将结果以字符串写入到标准输出。接口和编程模型是有约束和限制，但有时候这就是你需要的，类似随着 `map` 和 `filter` 处理利用原生代码功能。

最有可能是，你想从外部程序或脚本输送 RDD 的内容。因为你已经构建了一个复杂的软件，且已测试，你想在 Spark 里重用它。很多数据科学家都有 R 语言<sup>1</sup>的代码。我们可以用 `pipe()`和 R 语言交互。

示例 6-15 中，我们用 R 语言库来计算所有联系人的距离。我们的 RDD 中的每个元素被我们的程序输出，以换行为分隔符，并且程序中输出的每一行是结果 RDD 中的字符串元素。为简化 R 程序来解析这些输入，我们会重新格式化我们的数据为 `mylat, mylon, theirlat, theirlon` 这四个字段，并以逗号分隔。

示例 6-15. R distance program

```
#!/usr/bin/env Rscript
library("Imap")
f <- file("stdin")
open(f)
while(length(line <- readLines(f,n=1)) > 0) {
  # process line
  contents <- Map(as.numeric, strsplit(line, ","))
  mydist <- gdist(contents[[1]][1], contents[[1]][2],
    contents[[1]][3], contents[[1]][4],
    units="m", a=6378137.0, b=6356752.3142, verbose = FALSE)
  write(mydist, stdout())
}
```

如果上面代码被写入到名为 `./src/R/finddistance.R` 的可执行文件，那么是像下面这么使用：

```
$ ./src/R/finddistance.R
37.75889318222431,-122.42683635321838,37.7614213,-122.4240097
349.2602
coffee
NA
ctrl-d
```

目前为止，很好。我们已经有了方法来变换从 `stdin` 的每一行输出到 `stdout`。现在我们需要使得 `finddistance.R` 对每个 worker 节点可用，并实际用我们的 shell

<sup>1</sup> SparkR 项目也提供了轻量级的前端可以从 R 语言里使用 Spark。

脚本来转换 RDD。在 Spark 里这很容易做到，见示例 6-16 到 6-18。

示例 6-16. Driver program using pipe() to call finddistance.R in Python

```
# Compute the distance of each call using an external R program
distScript = "./src/R/finddistance.R"
distScriptName = "finddistance.R"
sc.addFile(distScript)
def hasDistInfo(call):
    """Verify that a call has the fields required to compute the distance"""
    requiredFields = ["mylat", "mylong", "contactlat", "contactlong"]
    return all(map(lambda f: call[f], requiredFields))
def formatCall(call):
    """Format a call so that it can be parsed by our R program"""
    return "{0},{1},{2},{3}".format(
call["mylat"], call["mylong"],
call["contactlat"], call["contactlong"])
pipeInputs = contactsContactList.values().flatMap(
    lambda calls: map(formatCall, filter(hasDistInfo, calls)))
distances = pipeInputs.pipe(SparkFiles.get(distScriptName))
print distances.collect()
```

示例 6-17. Driver program using pipe() to call finddistance.R in Scala

```
// Compute the distance of each call using an external R program
// adds our script to a list of files for each node to download with this job
val distScript = "./src/R/finddistance.R"
val distScriptName = "finddistance.R"
sc.addFile(distScript)
val distances = contactsContactLists.values.flatMap(x => x.map(y =>
    s"$y.contactlay,$y.contactlong,$y.mylat,$y.mylong")).pipe(Seq(
    SparkFiles.get(distScriptName)))
println(distances.collect().toList)
```

示例 6-18. Driver program using pipe() to call finddistance.R in Java

```
// Compute the distance of each call using an external R program
// adds our script to a list of files for each node to download with this job
String distScript = "./src/R/finddistance.R";
String distScriptName = "finddistance.R";
sc.addFile(distScript);
JavaRDD<String> pipeInputs = contactsContactLists.values()
    .map(new VerifyCallLogs()).flatMap(
    new FlatMapFunction<CallLog[], String>() {
        public Iterable<String> call(CallLog[] calls) {
            ArrayList<String> latLons = new ArrayList<String>();
            for (CallLog call: calls) {
```

```

        latLons.add(call.mylat + "," + call.mylong +
            "," + call.contactlat + "," + call.contactlong);
    }
    return latLons;
}

});
JavaRDD<String> distances = pipeInputs.pipe(SparkFiles.get(distScriptName));
System.out.println(StringUtils.join(distances.collect(), ","));

```

使用 `SparkContext.addFile(path)`，我们可以为每个 `worker` 节点构建个文件列表在 `Spark` 任务里下载。这些文件可以是来自驱动所在本地系统（就像这些示例一样），来自 `HDFS` 或者其他 `Hadoop` 支持的文件系统，或者来自 `HTTP`，`HTTPS`，或者 `FTP URI`。当作业中有动作执行，这些文件就会被每个节点下载。这些文件可以在 `worker` 节点的 `SparkFiles.getRootDirectory` 找到或者用 `SparkFiles.get(filename)` 来定位。当然，这只是确认每个 `worker` 节点的 `pipe()` 能找到脚本的一种方式。你可以用别的远程复制工具将脚本文件放到每个节点都知道的位置。



用 `SparkContext.addFile(path)` 添加的所有文件都保存在同一个目录，所以命名唯一很重要。

一旦脚本可用，`RDD` 的 `pipe()` 方法就能很容易的通过脚本来输送 `RDD` 的元素。也许一个智能的 `finddistance()` 版本可以从命令行接受一个分隔符参数。假如那样的话，下面两行都可以干这个，然而首选第一个~

- `rdd.pipe(Seq(SparkFiles.get("finddistance.R"), ","))`
- `rdd.pipe(SparkFiles.get("finddistance.R") + ",")`

第一种方式中，我们传入位置参数序列（命令本身在零偏移的位置）作为命令行。第二种方式，我们传入单个命令行字符串，`Spark` 会分解为位置参数。

如果我们要求，我们也可以用 `pipe()` 指定 `shell` 环境变量。只要传入环境变量到值得映射作为 `pipe()` 的第二个参数，`Spark` 就会设置这些环境变量。

你现在至少理解了如何用 `pipe()` 通过外部命令处理 `RDD` 元素，以及如何分发这样的脚本到集群，这样一来，`worker` 节点总能找到它们。

## 数值 RDD 操作

Spark 提供了一些对包含数值的 RDD 的描述性统计操作。不包括复杂的统计和机器学习方法，那些在第十一章细说。

Spark 的数值操作实现为流式算法，允许一次一个元素来构建模型。通过调用 `stat()` 函数，在数据遍历一次后，所有描述性统计都计算好了，并返回一个 `StatsCounter` 对象。表 6-2 列出了 `StatsCounter` 对象的可用方法。

表格 6-2 `StatsCounter` 的可用统计汇总

方法	含义
<code>count()</code>	RDD 中的元素个数
<code>mean()</code>	元素平均值
<code>sum()</code>	元素的和
<code>max()</code>	最大值
<code>min()</code>	最小值
<code>variance()</code>	元素的方差
<code>sampleVariance()</code>	元素的样本方差
<code>stdev()</code>	标准误差
<code>sampleStdev()</code>	样本标准误差

如果你只想计算其中之一，你可以对 RDD 直接调用对应的方法。比如 `rdd.mean()`，或者 `rdd.sum()`。

在示例 6-19 到 6-22 中，我们将使用汇总统计去除数据中的异常值。由于我们要使用同一个 RDD 两次（一次计算汇总统计，一次移除异常值），我们会希望缓存 RDD。回到我们的呼叫日志的例子，我们可以从日志中移除掉太遥远的联系点。

示例 6-19. Removing outliers in Python

```
# Convert our RDD of strings to numeric data so we can compute stats and  
# remove the outliers.  
distanceNumerics = distances.map(lambda string: float(string))  
stats = distanceNumerics.stats()  
stddev = std.stdev()  
mean = stats.mean()  
reasonableDistances = distanceNumerics.filter(  
lambda x: math.fabs(x - mean) < 3 * stddev)  
print reasonableDistances.collect()
```

示例 6-20. Removing outliers in Scala

```
// Now we can go ahead and remove outliers since those may have misreported locations  
// first we need to take our RDD of strings and turn it into doubles.  
val distanceDouble = distance.map(string => string.toDouble)  
val stats = distanceDoubles.stats()  
val stddev = stats.stdev  
val mean = stats.mean  
val reasonableDistances = distanceDoubles.filter(x => math.abs(x-mean) < 3 * stddev)  
println(reasonableDistance.collect().toList)
```

示例 6-21. Removing outliers in Java

```
// First we need to convert our RDD of String to a DoubleRDD so we can  
// access the stats function  
JavaDoubleRDD distanceDoubles = distances.mapToDouble(new DoubleFunction<String>() {  
    public double call(String value) {  
        return Double.parseDouble(value);  
    }  
});  
final StatCounter stats = distanceDoubles.stats();  
final Double stddev = stats.stdev();  
final Double mean = stats.mean();  
JavaDoubleRDD reasonableDistances =  
distanceDoubles.filter(new Function<Double, Boolean>() {  
    public Boolean call(Double x) {  
        return (Math.abs(x-mean) < 3 * stddev);  
    }  
});  
System.out.println(StringUtils.join(reasonableDistance.collect(), " "));
```

随着上面最后一块，我们完成了这个示例程序，使用了累加器和广播变量，分区化处理，外部程序接口以及汇总统计。例子程序的完整源代码分别在 `src/python/ChapterSixExample.py`, `src/main/scala/com/oreilly/learningsparkexamples/scala/ChapterSixExample.scala`, 和 `src/main/java/com/oreilly/learningsparkexamples/java/ChapterSixExample.java`。

## 总结

在本章我们引入了一些更高级的 **Spark** 编程特性，我们可以用来使我们的程序更高效，更富于表达。后面的章节涉及了 **Spark** 程序的部署和调优，此外还有内置的 SQL 库和流计算以及机器学习。我们也将开始看看大量利用目前为止讲述过的功能来构造更复杂和更完整的示例程序，这将引导并启发你自己的 **Spark** 用法。





# 在集群上运行

### 介绍

到目前为止,我们关注的是用 Spark Shell 学习 Spark 以及在本地模式下运行示例。而编写 Spark 应用的好处是 Spark 程序可以跑在集群模式,并通过添加机器进行大规模计算。好消息是,编写集群并行执行的应用是使用的和你在本书中已学到的相同的 API。之前写过的示例和应用可以在集群模式“开箱即用”。这就是使用 Spark 的高级别 API 的好处之一:用户可以在本地小数据集上快速开发应用原型,然后不用修改代码就能在非常大的集群上运行。

本章首先解释下 Spark 分布式应用的运行架构,然后讨论运行在分布式集群上的可选参数。Spark 能以本地部署或云部署的方式运行于多种集群管理器(Hadoop YARN, Apache Mesos 以及 Spark 内置的 Standalone 集群管理器)。我们将讨论每种运行情况必要的权衡和配置。期间也会讲到计划,部署和配置 Spark 应用的具体细节。本章读完后,你就了解了运行分布式 Spark 应用需要的一切,后续章节是应用的调优和调测。

### Spark 运行架构

在我们对集群运行 Spark 的细节深入之前,理解 Spark 的分布式模式架构会有些帮助(见图 7-1)。

分布式模式中,Spark 使用主从架构,有一个中心协调器和许多分布式 worker。中心协调器被称为 driver。Driver 和被称为 executor 的大量分布式 worker 通信。

Driver 运行在它自己的 Java 进程，而每个 executor 是单独的 Java 进程。Driver 和它的所有 executor 一起被称为 Spark 应用。

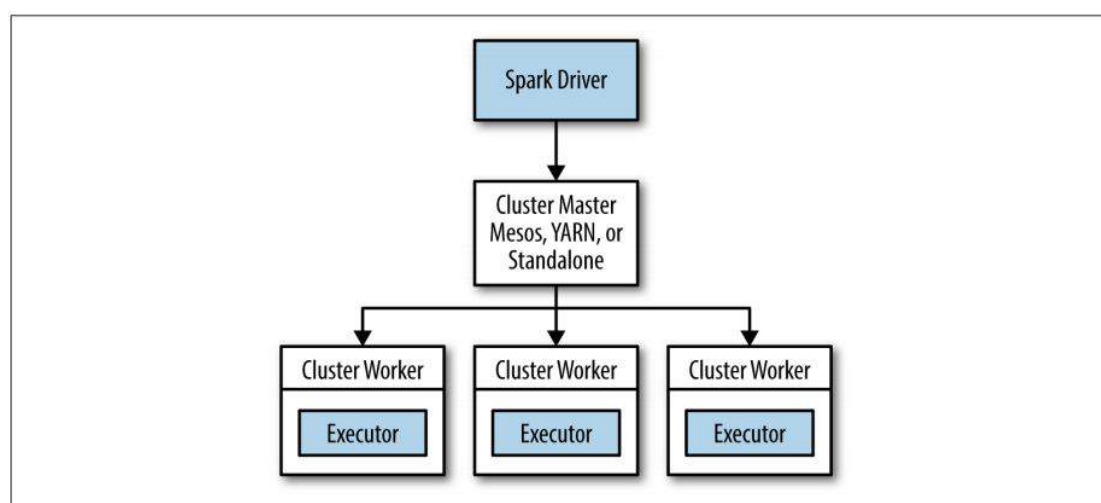


图 7-1 分布式 Spark 应用的组件

Spark 应用运行在一组使用被称为集群管理器的外部服务的机器上。注意，Spark 打包了一个内置的集群管理器，叫做 Standalone 集群管理器。Spark 也可以工作于 Hadoop YARN 和 Apache Mesos 这两个开源的集群管理器。

## Driver

Driver 是你的程序的 `main()` 方法所在的进程。该进程运行用户代码创建 `SparkContext`，创建 RDD，执行变换和动作。当你这行一个 Spark Shell，你就创建了一个 driver 程序（如果你记得的话，Spark Shell 带了一个预加载的叫做 `sc` 的 `SparkContext`）。一旦 driver 终止，整个应用就结束了。

当 driver 运行时，它有两个职责：

### 转换用户程序到任务

Spark 的 driver 有责任转换用户程序到被称为任务的物理执行单元。从上层看，所有的 Spark 程序都遵循同样的结构：它们从输入创建 RDD，通变换从这些 RDD 得到新的 RDD，然后执行动作来采集数据或保存数据。Spark 程序隐含创建了操作的逻辑合理的有向无环图(DAG)。当 driver 运行时，它转换该图到物理执行计划。

Spark 执行多种优化，比如“流水线”映射转换合并，并转换执行图到一组 stage。每个 stage 又由一组 task 组成。Task 则被捆绑在一起准备被发送到集群。Task 是 Spark 处理中的最小单元。典型的用户程序要执行成百上千个单独的任务。

### *调度 task 到 executor*

有了物理执行计划，driver 必须协调各独立任务到 executor 中。当 executor 启动后，它们会将自己注册到 driver，所以 driver 随时都能看到完整的 executor 视图。每个 executor 表现为能执行任务和保存 RDD 数据的进程。

Spark Driver 会寻找当前的 executor 组，然后基于数据分布尝试调度每个 task 到合适的位置。当任务执行时，可能会对正缓存的数据有副作用。Driver 也要记录缓存数据的位置并用来调度将来访问这些数据的任务。

Driver 从 web 接口暴露出了这些 Spark 应用的运行信息，默认端口是 4040。例如，在本地模式，可用的 UI 是 <http://localhost:4040>。我们将在第八章覆盖 Spark 的 UI 和调度机制的更多信息。

## **Executors**

Spark Executor 是 worker 进程，其职责是运行给定的 Spark 作业中的单个任务。Executor 在 Spark 应用开始的时候被启动一次，一般会在应用的整个生命周期都运行。虽然 executor 出错了 Spark 也可以继续。Executor 有两个任务。一个是运行构成应用的任务并返回结果到 driver。第二个是通过每个 executor 中都存在的被称为块管理器(Block Manager)的服务为用户程序中缓存的 RDD 提供内存存储。因为 RDD 被直接缓存在 executor 中，任务可以和数据在一起运行。



### **本地模式的 Driver 和 executor**

本书中大部分示例都是运行在本地模式。该模式中，driver 和 executor 运行在同一个 Java 进程。这是特例，通常每个 executor 都运行在一个专有的进程。

## **集群管理器**

目前为止我们讨论的 driver 和 executor 稍微有点抽象。但是，driver 和 executor

进程最初是如何被启动的？Spark 依赖于集群管理器来启动 `executor`，并且有时候也启动 `driver`。集群管理器在 Spark 中是个可拔插的组件。这允许 Spark 运行在不同的外部管理器上面，比如 YARN 和 Mesos 以及内置的 Standalone 集群管理器。



Spark 的文档在描述执行每个 Spark 应用的进程时一贯使用术语 `driver` 和 `executor`。术语 `master` 和 `worker` 用于描述集群管理器的中心和分布的部分。这些术语很容易混淆，要特别注意。例如，Hadoop YARN 运行一个 `master daemon`(称为资源管理器)和多个 `worker daemon`（称为节点管理器）。Spark 可以在 YARN 的 `worker` 节点上运行 `driver` 和 `executor`。

## 启动一个程序

不论你使用哪个集群管理器，Spark 提供了名为 `spark-submit` 的单一脚本，你可以用于提交程序。通过各种选项，`spark-submit` 能连接到不同的集群管理器并且控制你的应用获得多少资源。对于有些集群管理器，`spark-submit` 可以在集群上运行 `driver`(比如在 YARN 的 `worker` 节点)，有一些只能运行在本地。下一节我们会更详细的探讨 `spark-submit`。

## 汇总

为汇总本节这些概念，我们来过一遍发生在 Spark 应用在集群运行时的准确步骤：

1. 用户用 `spark-submit` 提交了一个应用。
2. `spark-submit` 启动 `driver` 程序，并调用用户指定的 `main()` 方法。
3. `driver` 程序联系集群管理器请求资源来启动各 `executor`。
4. 集群管理器代表 `driver` 程序启动各 `executor`。
5. `Driver` 进程运行整个用户应用。程序中基于 RDD 的变换和动作，`driver` 程序以 `task` 的形式发送到各 `executor`。
6. `Task` 在 `executor` 进程运行来计算和保存结果。
7. 如果 `driver` 的 `main()` 方法退出或者调用了 `SparkContext.stop()`，就会终止 `executor` 的运行并释放从集群管理器分配的资源。

## 用 spark-submit 分发应用

正如你已经学过的，Spark 提供了一个名为 spark-submit 的单一工具来跨集群管理器的提交作业。在第二章中，我们看到了一个用 spark-submit 提交 Python 程序的简单例子。示例 7-1 中重看一下。

*示例 7-1. 提交一个 Python 应用*

```
bin/spark-submit my_script.py
```

当 spark-submit 被调用时除了一个脚本或 JAR 包的名字其他什么都没有，就会在本地运行所提供的 Spark 程序。比如说我们想提交这个程序到 Spark 的 Standalone 集群，我们可以提供 Spark 集群的地址作为额外标志并指定我们要启动的每个 executor 进程的内存大小，见示例 7-2。

*示例 7-2. 用额外的参数提交应用*

```
bin/spark-submit --master spark://host:7077 --executor-memory 10g my_script.py
```

--master 标志指定要连接的集群的 URL。本例中，spark://URL 表示一个用 Spark 的 Standalone 模式的集群（见表 7-1）。后面会讨论其他 URL 类型。

表格 7-1 spark-submit 中--master 标志的可能值

值	解释
spark://host:port	连接到 Spark 的 Standalone 集群的指定端口。默认的 Spark Standalone 的 master 使用 7077 端口。
mesos://host:port	连接到 mesos 集群的 master 的指定端口。默认的 mesos master 使用 5050 端口。
yarn	连接到 YARN 集群。当运行在 YARN 时你需要设置 HADOOP_CONF_DIR 环境变量，指向你的 Hadoop 配置目录的位置，其中包含了集群信息。
local	运行于本地模式，使用单核
local[N]	运行于本地模式，使用 N 个核
local[*]	运行于本地模式，有多少个核就用多少个

除了集群 URL 参数外，spark-submit 提供了各种选项让你控制关于你的一个特定

程序运行的指定细节。这些选项大致上分两类。第一类是调度信息，比如为你的作业请求的资源数（见示例 7-2）。第二类是你的应用运行时的依赖信息，比如需要分发到所有 worker 机器上的库或文件。

spark-submit 的一般格式见示例 7-3。

示例 7-3. spark-submit 的一般格式

```
bin/spark-submit [options] <app jar | python file> [app options]
```

[options]是 spark-submit 的标志列表。你可以运行 spark-submit --help 来枚举所有可能的标志。表 7-2 枚举了常用的标志列表。

<app jar | python file>指向包含你程序入口点的 JAR 包或者 python 脚本。

[app options]是要传入你的程序的参数。如果你程序中的 main()方法解析参数，看到的就只是[app options]，看不到指定给 spark-submit 的标志。

表格 7-2 spark-submit 的常用标志

标志	解释
--master	指示要连接的集群管理器。可选值描述于表 7-1。
--deploy-mode	是否启动 driver 程序于本地（“client”）或者是集群上某个 worker 机器（“cluster”）。在 client 模式，spark-submit 将在其本身被执行的同一个机器上运行 driver 程序。在 cluster 模式中，driver 会被转移到集群上某个 worker 节点运行。默认是 client 模式。
--class	提交的应用的主类，如果是 Java 或者 Scala 程序
--name	你的应用的一个可理解的名字。会在 Spark 的 UI 上显示。
--jars	要上传的 JAR 文件列表，并放置在你程序的 classpath 上。如果你的程序依赖于少量的第三方 JAR 包，可以加在这。
--files	要放到你程序的工作目录的文件列表。用于你要分发到每个节点的数据文件。
--py-files	要加到你程序的 PYTHONPATH 去的文件列表。可以包含 .py，.egg，或者 .zip 文件。

标志	解释
<code>--executor-memory</code>	用于每个 <code>executor</code> 的内存数量，单位：字节。后缀可以用来指定一个较大的数量，比如 “512m”（512 MB)或者”15g”(15 GB)。
<code>--driver-memory</code>	用于 <code>driver</code> 进程的内存数量，单位：字节。后缀可以用来指定一个较大的数量，比如 “512m”（512 MB)或者”15g”(15 GB)。

`spark-submit` 也允许设置任意的 `SparkConf` 配置项，可以用 `--conf prop=value` 标志或者是通过 `--properties-file` 提供一个包含 `key/value` 元组的属性文件。第八章将讨论 `Spark` 的配置系统。

示例 7-4 演示了 `spark-submit` 使用各种选项的较长形式的调用。

```

示例 7-4. Using spark-submit with various options
# Submitting a Java application to Standalone cluster mode
$ ./bin/spark-submit \
--master spark://hostname:7077 \
--deploy-mode cluster \
--class com.databricks.examples.SparkExample \
--name "Example Program" \
--jars dep1.jar,dep2.jar,dep3.jar \
--total-executor-cores 300 \
--executor-memory 10g \
myApp.jar "options" "to your application" "go here"
# Submitting a Python application in YARN client mode
$ export HADOOP_CONF_DIR=/opt/hadoop/conf
$ ./bin/spark-submit \
--master yarn \
--py-files somelib-1.2.egg,otherlib-4.4.zip,other-file.py \
--deploy-mode client \
--name "Example Program" \
--queue exampleQueue \
--num-executors 40 \
--executor-memory 10g \
my_script.py "options" "to your application" "go here"

```

## 打包你的代码和依赖

尽管大部分本书提供的示例程序都是自包含的且没有 `Spark` 外的库依赖，更常见的是用户程序依赖于第三方库。如果你的程序导入了不是 `org.apache.spark` 的包或



语言库的部分，你需要确保你的所有依赖在你的 `spark` 应用运行时存在。

对 Python 用户，有几种方式来安装第三方库。由于 PySpark 使用 worker 机器上已存在的 Python 安装，你可以在集群机器上直接用标准的 Python 包管理器（比如 `pip` 或 `easy_install`）来安装依赖库，或者到你的 Python 安装的 `site-packages/` 目录手动安装。你也可以用 `--py-files` 参数提交单独的库到 `spark-submit`，这些库就会被加到 Python 解释器的路径中。如果你没有权限在集群上安装包，手动添加库可能会更方便。但是要考虑和机器上已经安装的已存在的包的潜在冲突。



### Spark 本身如何？

当你捆绑你的应用时，你永远不用包含 Spark 自身到提交的依赖列表。

`Spark-submit` 会自动确保 Spark 在程序中存在。

对于 Java 和 Scala 用户，也可以用 `--jars` 标志提交单独的 JAR 文件到 `spark-submit`。如果你只有很简单的对一两个库的依赖而且它们本身不依赖其他库，这是没问题的。然而更常见的是，对于使用 Java 和 Scala 项目的用户都会依赖多个库。当你提交应用到 Spark，必须附带整个传递依赖图到集群。不仅包括直接依赖的库，还有这些库的依赖等等。手动的记录和提交这么一组 JAR 文件是极其麻烦的。相反的，常见做法是依靠构建工具生成一个大的 JAR 文件，包含了应用的整个传递依赖图，通常称之为 uber JAR 或者 assembly JAR。大部分 Java 和 Scala 的构建工具都能生成这种 JAR 文件。

Java 和 Scala 最流行的构建工具是 Maven 和 `sbt`(Scala build tool)。这俩工具两种语言都能用，不过通常 Maven 用于 Java 项目，`sbt` 用于 Scala 项目。这里我们给出用两种工具构建 Spark 应用的示例。你可以将这些作为自己的 Spark 项目模板。

## 用 Maven 构建 Java 的 Spark 应用

我们来看一个有多个依赖的生成 uber JAR 的例子。示例 7-5 提供了一个 Maven 的 `pom.xml` 文件，里面包含了编译的信息。这个例子没有展示实际的 Java 代码或项目目录结构，但是 Maven 期待用户代码相对于项目根目录的路径为

src/main/java（根目录中应包含 pom.xml 文件）。

示例 7-5. pom.xml file for a Spark application built with Maven

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <!-- Information about your project -->
  <groupId>com.databricks</groupId>
  <artifactId>example-build</artifactId>
  <name>Simple Project</name>
  <packaging>jar</packaging>
  <version>1.0</version>
  <dependencies>
    <!-- Spark dependency -->
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
      <version>1.2.0</version>
      <scope>provided</scope>
    </dependency>
    <!-- Third-party library -->
    <dependency>
      <groupId>net.sf.jopt-simple</groupId>
      <artifactId>jopt-simple</artifactId>
      <version>4.3</version>
    </dependency>
    <!-- Third-party library -->
    <dependency>
      <groupId>joda-time</groupId>
      <artifactId>joda-time</artifactId>
      <version>2.0</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <!-- Maven shade plug-in that creates uber JARs -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>2.3</version>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

```

        </execution>
    </executions>
</plugin>
</plugins>
</build>
</project>

```

该项目申明了两个传递依赖：jopt-simple，一个 Java 库用于执行参数解析；joda-time，一个用于日期时间转换的工具库。项目同样依赖于 Spark，不过 Spark 是被标记的确定永远不用打包到应用里的。构建动作包含了 maven-shade-plugin 来创建包含所有依赖的 uber JAR 包。每次出现打包过程时你可以要求 Maven 执行插件的 shade-goal 来使能该功能。有了这个构建配置，当 mvn package 执行时 uber JAR 就会自动被创建（见示例 7-6）。

*示例 7-6. Packaging a Spark application built with Maven*

```

$ mvn package
# In the target directory, we'll see an uber JAR and the original package JAR
$ ls target/
example-build-1.0.jar
original-example-build-1.0.jar
# Listing the uber JAR will reveal classes from dependency libraries
$ jar tf target/example-build-1.0.jar
...
joptsimple/HelpFormatter.class
...
org/joda/time/tz/UTCProvider.class
...
# An uber JAR can be passed directly to spark-submit
$ /path/to/spark/bin/spark-submit --master local ... target/example-build-1.0.jar

```

## 用 sbt 构建 Scala 的 Spark 应用

sbt 是一种新的构建工具，多用于 Scala 项目。Sbt 采用一种类似 Maven 的项目布局。在项目的根目录创建一个名为 build.stb 的构建文件，源代码不出意料是在 src/main/scala 目录。Sbt 用一种配置语言编写，为了定义项目的构建，为特定的键赋值。比如有个键名为 name，它包含了项目名字；而有个键名为 libraryDependencies，包含了项目的依赖列表。示例 7-7 给出了一个依赖 Spark 和几个其他第三方库的简单应用的完整 sbt 构建文件。该构建文件可用于 sbt 0.13。因为 sbt 发展迅速，你可能要读一下最新的文档了解关于构建文件格式的变化。

示例 7-7. *build.sbt* file for a Spark application built with sbt 0.13

```
import AssemblyKeys._
name := "Simple Project"
version := "1.0"
organization := "com.databricks"
scalaVersion := "2.10.3"
libraryDependencies ++= Seq(
  // Spark dependency
  "org.apache.spark" % "spark-core_2.10" % "1.2.0" % "provided",
  // Third-party libraries
  "net.sf.jopt-simple" % "jopt-simple" % "4.3",
  "joda-time" % "joda-time" % "2.0"
)
// This statement includes the assembly plug-in capabilities
assemblySettings
// Configure JAR used with the assembly plug-in
jarName in assembly := "my-project-assembly.jar"
// A special option to exclude Scala itself from our assembly JAR, since Spark
// already bundles Scala.
assemblyOption in assembly :=
  (assemblyOption in assembly).value.copy(includeScala = false)
```

构建文件第一行从 sbt 构建插件导入了一些功能，支持创建项目的 assembly JAR 文件。要启用该插件，我们必须在 project/ 目录包含一个小文件，列出插件的包依赖。简单的创建一个 project/assembly.sbt 文件，添加如下内容：addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.11.2")。如果你用的是更新的 sbt 版本，你用的 sbt-assembly 的准确版本可能会不一样。示例 7-8 用于 sbt 0.13。

示例 7-8. *Adding the assembly plug-in to an sbt project build*

```
# Display contents of project/assembly.sbt
$ cat project/assembly.sbt
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.11.2")
```

现在我们已经有了定义好的构建文件，可以创建完整合并的 Spark 应用 JAR 包了（示例 7-9）。

*Example 7-9. Packaging a Spark application built with sbt*

```
$ sbt assembly
# In the target directory, we'll see an assembly JAR
$ ls target/scala-2.10/
my-project-assembly.jar
# Listing the assembly JAR will reveal classes from dependency libraries
$ jar tf target/scala-2.10/my-project-assembly.jar
...
joptsimple/HelpFormatter.class
...
org.joda.time.tz.UTCProvider.class
...
# An assembly JAR can be passed directly to spark-submit
$ /path/to/spark/bin/spark-submit --master local ...
    target/scala-2.10/my-project-assembly.jar
```

## 依赖冲突

当用户应用和 Spark 都依赖于同一个库时，间或会有颠覆性的问题要处理，依赖冲突。这种情况相对较少发生，但一旦发生，用户会十分困扰。在 Spark 作业执行期间，当出现 `NoSuchMethodError` 或者 `ClassNotFoundException` 或者其他一些关于类加载的其他 JVM 异常被抛出，通常就是依赖冲突。对此有两个解决方案。一个是修改你的应用，使用和 Spark 使用的相同版本的第三方库。第二种是用一种常被称为 `shading` 的过程来修改你的应用的包。在示例 7-5 中 Maven 构建工具通过 `plug-in` 的高级配置支持 `shading`(事实上，`shading` 的能力就是为啥插件被命名为 `maven-shade-plugin` 的原因)。`Shading` 运行你在不同的名字空间创建冲突的包的第二个副本，并使用重命名的版本来重写你的代码。这种稍微暴力的技术在处理运行时依赖冲突时十分有效。对于如何隐蔽依赖的特殊指定，详见你的构建工具的文档。

## Spark 应用内部和 Spark 应用之间的调度

之前的示例都只是参考单用户提交作业到集群。现实中，许多集群被多个用户共享。共享的环境就有调度的挑战：如果两个用户都启动 Spark 应用，每个都要用整个集群资源，会发生什么？调度策略帮助确保资源不会被压垮，并且允许对工作量有优先级。

对于多用户集群的调度，Spark 主要依赖于集群管理器来共享 Spark 应用之间的资源。当 Spark 应用向集群管理器要求 `executor`，可能会接收到或多或少的

executor，依赖于集群当时的可用性和竞争情况。许多集群管理器提供定义不同优先级队列的能力或者容量限制的能力，那么 Spark 就是提交作业到队列即可。更多的细节看你的集群管理器的文档。

长时间运行的 Spark 应用是个特例，意味着它们从来不会要求退出。长时间运行的 Spark 应用的例子是捆绑了 Spark SQL 的 JDBC 服务。当 JDBC 服务启动，它从集群管理器申请了一组 executor，然后扮演为用户提交的 SQL 查询永久代理的角色。由于是单个应用为多用户的工作进行调度，它需要更细粒度的机制来加强共享策略。Spark 提供了这样的机制，一种可配置的应用内调度策略，Spark 内部称为公平调度器(Fair Scheduler)，使得长时间运行的应用定义优先级队列来调度任务。更详细的评论超出本书范围，官方文档对公平调度器提供了不错的参考。

## 集群管理器

Spark 可以运行在多种集群管理器上来访问集群中的机器。如果你只想在一组机器上运行 Spark 自己，Spark 内置的 Standalone 模式是最简单的部署方式。然而，如果你已经有了一个集群，想和其他分布式应用共享（例如 Spark 作业和 Hadoop 的 MapReduce 作业），Spark 也可以运行于两种流行的集群管理器：Hadoop YARN 和 Apache Mesos。最后，为了部署到 Amazon EC2 上，和 Spark 一起提供了启动 Standalone 集群的内置脚本和各种支撑服务。本节将讨论如何在各种环境运行 Spark。

## Standalone 集群管理器

Spark 的 Standalone 管理器提供了在集群运行应用的一种简单方法。它由一个 master 和多个 worker 组成，每个都配置了一定数量的内存和 CPU。当你提交应用，你可以选择 executor 用多少内存，以及所有的 executor 使用的总核数。

### 启动 Standalone 集群管理器

你可以通过两种方式启动 Standalone 集群管理器：通过手动启动 master 和所有 worker，或者通过 Spark 的 sbin/目录下的启动脚本。启动脚本是最简单的选择，但是需要机器间进行 SSH 访问，并且当前(Spark 1.1)仅在 MacOS 和 Linux 可用。我们先说这些，然后讨论其他平台如何手动启动集群。

要使用集群启动脚本，可按照下列步骤：

1. 复制编译好的 Spark 版本到所有机器的同一位置, 比如/home/yourname/spark。
2. 设置从 master 到各 worker 的无密码 SSH 访问。这要求所有机器有同名的用户, 通过 ssh-keygen 为其生成私有的 SSH 密钥, 然后添加该密钥到所有 worker 机器的.ssh/authorized\_keys 文件中。如果你以前没设置过, 可用下面的命令:

```
# On master: run ssh-keygen accepting default options
$ ssh-keygen -t dsa
Enter file in which to save the key (/home/you/.ssh/id_dsa): [ENTER]
Enter passphrase (empty for no passphrase): [EMPTY]
Enter same passphrase again: [EMPTY]
# On workers:
# copy ~/.ssh/id_dsa.pub from your master to the worker, then use:
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
$ chmod 644 ~/.ssh/authorized_keys
```

3. 编辑 master 机器的 conf/slaves 文件, 填入所有 worker 的机器名。
4. 启动集群, 在 master 机器上运行 sbin/start-all.sh (重点是在 master 运行, 不是其他 worker 上)。如果一切正常, 你不会得到密码提示, 并且集群管理器的 Web 界面会出现在 http://masternode:8080, 显示所有的 worker 节点。
5. 要停止集群, 在 master 机器上运行 sbin/stop-all.sh (译注: 原文是 bin/stop-all.sh, 笔误)。

如果你不是在 UNIX 系统上, 或者是想手动启动集群。可以用 Spark 的 bin/目录下的 spark-class 脚本来手动启动 master 和各 worker。在你的 master 机器上输入:

```
bin/spark-class org.apache.spark.deploy.master.Master
```

然后在 worker 上输入:

```
bin/spark-class org.apache.spark.deploy.worker.Worker spark://masternode:7077 (masternode 是 master 的主机名), 在 Windows 上, 用\`替换`/。
```

默认情况下, 集群管理器会为每个 worker 自动分配 CPU 数量和内存, 选择个合适的值给 Spark 使用。配置 Standalone 集群管理器的更多细节见 Spark 官方文档。

## 提交应用

要提交应用到 Standalone 集群管理器, 传入 spark://masternode:7077 为 master 参



数给 `spark-submit`，例如：

```
bin/spark-class org.apache.spark.deploy.worker.Worker spark://masternode:7077
```

这个集群 URL 也被显示在 Standalone 集群管理器的 web 界面，地址是 `http://masternode:8080`。注意，提交期间的主机名和端口必须严格匹配界面上的显示。这可能会让用户犯错，比如用 `ip` 作为代替机器名。即使 `ip` 和机器名是关联的，提交也会失败，名字没有精确匹配。有些管理员可能会配置 Spark 使用不同于 7077 的端口，要确保主机名和端口的一致性，安全的做法是从 master 节点的 web 界面直接复制粘贴 URL。

你也可以通过传入 `--master` 参数同样的方式在集群中启动 `spark-shell` 或 `PySpark`。

```
spark-shell --master spark://masternode:7077
```

```
pyspark --master spark://masternode:7077
```

要检查你的应用或 shell 是否正在运行，可以查看集群管理器的 web 界面 `http://masternode:8080`。确认(1)你的程序已连接（出现在 Running Application 下面）和（2）列出的 CPU 和内存数大于 0。



组织你的应用运行的一个常见问题是为每个 `executor` 请求了超过集群可用的更多的内存(对 `spark-submit` 使用 `--executor-memory` 参数)。这样的话，Standalone 集群管理器永远不会为该应用分配 `executor`。要确保你申请的值能被集群满足。

最后，Standalone 集群管理器对你的应用的 `driver` 程序的运行提供了两种部署方式。在 `client` 模式（默认的），`driver` 程序运行在你执行 `spark-submit` 的机器上，作为 `spark-submit` 命令的一部分。这意味着你可以直接看到 `driver` 程序的输出或者发送输入给它（例如，对交互式 shell）。但是这要求你提交应用的机器有到各 `worker` 的快速连接，并在你的应用期间保持可用。与此相反，在 `cluster` 模式，`driver` 程序在集群上启动，作为其中一个 `worker` 节点上的另一个进程，然后连接回去请求 `executor`。这种模式下，`spark-submit` 是提交后就不管了，你可以在应用运行起来后关闭笔记本。你仍然可以通过集群管理器的 web 界面访问日志。你可以通过传入 `--deploy-mode cluster` 参数到 `spark-submit` 来切换集群模式。



## 配置资源使用

当在多个用户间共享 Spark 集群时，你需要决定在各 executor 之间如何分配资源。Standalone 集群管理器有基本调度策略，可以限制每个应用的资源占用，所以多个应用可以并发执行。Apache Mesos 支持应用运行时更动态的共享。而 YARN 有队列的概念，允许你限制不同应用的资源占用。

Standalone 集群管理器中，资源分配通过两个设置控制：

### *Executor 内存*

可以对 spark-submit 使用 `--executor-memory` 参数来配置。每个应用的每个 worker 至少要有 1 个 executor，所以该设置控制应用要求的 worker 的内存数量。默认情况下设置为 1GB。对大多数服务器，你很可能会想增加该值。

### *最大核数*

这是应用中所有 executor 使用的总核数。默认情况下是不限制。也就是说应用会启动集群上每个可用节点的 executor。对于多用户负载的集群，最好是要用用户限制其使用。可以对 spark-submit 使用 `--total-executor-cores` 参数来配置，或者在你的 Spark 配置文件中配置 `spark.cores.max` 参数。

要校验设置，可以在 Standalone 集群管理器的 web 界面 <http://masternode:8080> 来查看当前资源分配。

最后，Standalone 集群管理器默认按最大 executor 数分摊到每个应用。比如说，假设你有 20 个节点的集群，每个节点 4 核。使用 `--executor-memory 1G` 和 `--total-executor-cores 8` 来提交应用。那么 Spark 就会在不同机器上启动 8 个 executor，每个分配 1G 内存。Spark 的默认处理是给每个应用机会满足分布式系统的数据局部性，即和数据运行在同一个机器上（如 HDFS）。因为这些系统中数据是散布到所有节点的。如果你需要的话，可以要求 Spark 尽可能集中分配 executor 到少数几个机器上，通过 `conf/spark-default.conf` 中设置配置属性 `spark.deploy.spreadOut` 为 `false`。这时候，之前的例子可能只分配到 2 个 executor，每个 1GB 内存，4 个核。该设置影响所有 Standalone 集群管理器上的所有应用，必须在集群管理器启动前设置。

## 高可用性

当运行在生产环境，你会希望 Standalone 集群即使在集群中单个节点挂掉的时候也能接受任务。没问题，Standalone 模式完美支持 worker 节点出错。如果你还希望 master 节点也是高可用，Spark 支持用 Apache ZooKeeper(一种分布式协调系统)来保持多个备用 master 节点，当其中任意一个出错时会切换到一个新的上。配置 Standalone 集群使用 ZooKeeper 超出了本书范围，但 Spark 官方文档有描述。

## Hadoop YARN

YARN 是 Hadoop 2.0 引入的集群管理器，允许各种数据处理框架运行在共享资源池上。并且通常是安装在和 Hadoop Filesystem(HDFS)的相同节点上。运行 Spark 在 YARN 环境很有用，使得 Spark 可以快速访问 HDFS 数据，就在数据所在的节点。

YARN 上运行 Spark 很直接：设置一个环境变量指向指向你的 Hadoop 配置目录，然后用 spark-submit 提交作业到特定的 master URL。

第一步是找到 Hadoop 配置目录，并设置到环境变量 HADOOP\_CONF\_DIR。也就是包含 yarn-site.xml 和其他配置文件的目录。通常是在 HADOOP\_HOME/conf，如果你安装 Hadoop 在 HADOOP\_HOME；或者是类似/etc/hadoop/conf 的系统路径。然后按如下提交应用：

```
export HADOOP_CONF_DIR="..."  
spark-submit --master yarn yourapp
```

和 Standalone 模式一样，有两种方式连接你的应用到集群：client 模式，应用的 driver 程序运行在你提交应用的同一台机器上（比如你的笔记本）；cluster 模式，driver 程序运行在 YARN 的容器里。你可以通过--deploy-mode 参数设置模式到 spark-submit。

Spark 的交互式 shell 和 PySpark 都可以在 YARN 上运行。只要简单设置 HADOOP\_CONF\_DIR 并传入--master yarn 到应用即可。注意，这俩只能运行在 client 模式，因为需要从用户获得输入。

## 配置资源使用

当运行在 YARN 上，Spark 应用使用固定的 executor 数目，通过设置标志 --num-executors 到 spark-submit, spark-shell 等。默认值是 2，你可以自己按需要增加。也可以通过--executor-memory 来设置每个 executor 使用的内存，以及通过

`--executor-cores` 向 YARN 请求的申请的核数。对于一组给定的硬件资源，Spark 通常使用大量 `executor` 中的少数（有多个核以及更多内存）会运行的更好，因为可以优化每个 `executor` 之间的通信。然而注意有些集群对单个 `executor` 的最大内存数有限制（默认 8GB），不会让你启动更多。

有些 YARN 集群为了资源管理的目的，被配置为调度应用到多个队列中。用 `--queue` 选项指定队列名。

最后，更多的配置参数信息详见 Spark 官方文档。

## Apache Mesos

Apache Mesos 是个通用目的的集群管理器，可以在集群上运行分析工作和长时间运行的服务（例如 web 应用或者 key/value 存储）。要在 Mesos 使用 Spark，传入 `mesos://URL` 到 `spark-submit`:

```
spark-submit --master mesos://masternode:5050 yourapp
```

当运行在多用户模式时，你也可以配置 Mesos 集群使用 ZooKeeper 来推举 master。这时候用 `mesos://zk:// URL` 指向一组 ZooKeeper 节点列表。比如你有 3 个 ZooKeeper 节点 (`nod1,node2,node3`)，ZooKeeper 运行在 2181 端口，用如下 URI:

```
mesos://zk://node1:2181/mesos,node2:2181/mesos,node3:2181/mesos
```

## Mesos 调度模式

不像其他的集群管理器，Mesos 提供 2 种模式在同一个集群的 `executor` 之间共享资源。在细粒度模式中，这是默认的，`executor` 在运行任务时会按比例增减它们从 Mesos 申请的 CPU 数目，所以运行多个 `executor` 的机器可以在它们中动态共享 CPU 资源。在粗粒度模式中，Spark 预先分配固定的 CPU 数到每个 `executor`，直到应用退出才释放，即使某个 `executor` 没有在执行任务。你可以启用粗粒度模式，通过传入 `--conf spark.mesos.coarse=true` 到 `spark-submit`。

当多用户共享集群运行类似 shell 的交互式任务，细粒度模式更有吸引力。因为当应用没有工作时可以减少它们的核数给运行集群中其他用户程序使用。但是，通过细粒度模式调度任务的缺点是延迟增大（所以像 Spark Streaming 这样的低延迟应用会遭殃），当用户敲入一个新命令时，应用需要等待一些时间使 CPU

核从空闲到再次“升温”。不过，注意你可以在同一个集群使用混合调度模式（比如一些 Spark 应用设置 `spark.mesos.coarse` 为 `true`，另一些不设置）。

## Client 和 cluster 模式

到 Spark 1.2 为止，Spark 在 Mesos 上只支持应用运行在 `client` 部署模式，也就是 `driver` 运行在提交应用的那台机器上。如果你想运行 `driver` 到 Mesos 集群上，类似 Aurora 和 Chronos 这样的框架允许你提交任意的脚本到 Mesos 上运行，并监视。你可以使用其中之一来启动你的 `driver` 程序。

## 配置资源使用

你可以通过到 `spark-submit` 的两个参数控制资源使用：`--executor-memory` 设置每个 `executor` 的内存，`--total-executor-cores` 设置应用申请的最大核数（对所有 `executor`）。默认情况下，Spark 启动 `executor` 使用尽可能多的核数，合并应用到少量 `executor` 上，给其需要的核数。如果你没有设置 `--total-executor-cores` 参数，它会尝试使用集群中所有的核。

## Amazon EC2

Spark 的内置的脚本可以在 Amazon EC2 上启动集群。该脚本启动一组节点然后上面安装 Standalone 集群管理器。所以一旦集群启动，就可以使用之前章节的 Standalone 模式的指令。另外，EC2 脚本还设置了支撑服务，如 HDFS, Tachyon 和 Ganglia 来监视你的集群。

Spark 的 EC2 脚本名为 `spark-ec2`，位于 Spark 安装的 `ec2` 目录下。它需要 Python 2.6 以上版本。你可以下载 Spark 并执行 EC2 脚本，不用事先编译 Spark。

EC2 脚本可以管理多个命名集群，使用 EC2 安全组来标志。对每个集群，脚本将创建一个安全组，对 `master` 节点是叫 `clustername-master`，对 `worker` 节点是叫 `clustername-slaves`。

## 启动集群

要启动集群，你先要创建一个 Amazon Web Service(AWS)账号，获得 Key ID 和 Key。然后导出这些环境变量：

```
export AWS_ACCESS_KEY_ID="..."
export AWS_SECRET_ACCESS_KEY="..."
```

另外，再创建 EC2 的 SSH 的密钥对，并下载私钥文件（通常名为 `keypair.pem`）以便你能 SSH 到机器上。

接下来，运行 `spark-ec2` 脚本的启动命令，传入密钥对的名字，私钥文件，以及集群名。默认情况下，这会启动带一个 master 和一个 slave 的集群，使用 `m1.xlarge` EC 实例。

```
cd /path/to/spark/ec2
./spark-ec2 -k mykeypair -i mykeypair.pem launch mycluster
```

你也可以配置实例类型，slave 个数，EC2 分区数及其他参数到 `spark-ec2`。例如：

```
# Launch a cluster with 5 slaves of type m3.xlarge
./spark-ec2 -k mykeypair -i mykeypair.pem -s 5 -t m3.xlarge launch mycluster
```

完整的选项列表，运行 `spark-ec2 --help` 查看。最常用的一些见表 7-3。

表格 7-3 `spark-ec2` 的常用选项

选项	含义
-k KEYPAIR	密钥对的名字
-i IDENTITY_FILE	私钥文件（以 .pem 结尾）
-s NUM_SLAVES	slave 节点数
-t INSTANCE_TYPE	要使用的 Amazon 实例类型
-r REGION	要用的 Amazon 分区（例如 us-west-1）
-z ZONE	Amazon 区域（例如 us-west-1b）
--spot-price=PRICE	以给定的 spot 加个使用 spot Instance(单位：美元)

一旦你执行了脚本，它通常花 5 分钟启动机器，登陆，安装 Spark。

## 登陆到集群

你可以通过 SSH 用你的密钥对的 `.pem` 文件登录到 master 节点。为了方便，`spark-ec2` 为此提供了登陆命令：

```
./spark-ec2 -k mykeypair -i mykeypair.pem login mycluster
```

或者，你可以先找到 **master** 节点的主机名，通过执行下面命令：

```
./spark-ec2 get-master mycluster
```

然后使用 `ssh -i keypair.pem root@masternode` 登陆进去。

一旦你进入了集群，你就可以使用安装在 `/root/spark` 目录下的 **Spark** 来运行程序。这是安装的 **Standalone** 集群，**master** 的 URL 是 `spark://masternode:7077`。如果你用 `spark-submit` 提交应用，就会自动用正确的配置提交你的应用到集群。你可以查看集群的 web 界面 `http://masternode:8080`。

注意，只有从集群启动的程序能用 `spark-submit` 提交作业。防火墙规则会因为安全原因阻止外部主机提交。要在集群上运行预打包的应用，先使用 **SCP** 复制到集群上。

```
scp -i mykeypair.pem app.jar root@masternode:~
```

## 销毁集群

要销毁 `spark-ec2` 启动的集群，执行：

```
./spark-ec2 destroy mycluster
```

这会终止集群关联的所有实例（如两个安全组的所有实例，`mycluster-master` 和 `mycluster-slaves`）。

## 暂停和重启集群

除了彻底终止集群外，`spark-ec2` 可以让你停止集群上运行的 **Amazon** 实例再重新启动。停止实例会关闭所有的示例，并丢失为 `spark-ec2` 配置的安装了 **HDFS** 的“临时的”磁盘上的数据（见 138 页的“集群存储”）。不过，停止的实例保留了所有的数据在根目录（如你上传的任何文件），所以你可以快速恢复工作。

要停止集群，使用：

```
./spark-ec2 stop mycluster
```

之后再次启动：

```
./spark-ec2 -k mykeypair -i mykeypair.pem start mycluster
```



尽管 Spark 的 EC2 脚本没有提供命令调整集群大小，你可以通过添加或删除 mycluster-slaves 安全组的机器来调整大小。要添加机器，首先停止集群，然后使用 AWS 管理台右键某个 slave 节点并选择“Launch more like this”。这会创建更多的实例到同一个安全组。然后用 spark-ec2 启动集群。要移除节点，只需要从 AWS 控制台终止掉即可（请小心提防 HDFS 上的数据）。

## 集群存储

Spark 的 EC2 集群随之配置了两个 Hadoop Filesystem 的安装，可以用于暂存空间。这会有利于保存数据到比访问 Amazon S3 更快的中间媒介。两个安装是：

- “临时的” HDFS 安装使用节点上的临时驱动器。多数 Amazon 实例类型带有大量的本地空间附着在“临时”驱动器上，当实例停止时消失。这个临时的 Hadoop 安装使用的这个空间，给你大量的临时空间可用。但是当你停止实例或者重新启动 EC2 集群式，会丢失所有数据。这是安装在节点的 /root/ephemeral-hdfs 目录，你可以用 bin/hdfs 访问文件。你也可以查看 web 界面和 HDFS，URL 是 <http://masternode:50070>。
- “持久的” HDFS 安装是在节点的 root 卷。这个实例会持久化数据，哪怕集群重启。但是一般比“临时的” HDFS 访问起来更小更慢。这对于中等大小的数据集很好，你不会希望要去下载多次。这是安装在 /root/persistent-hdfs 目录，你也可以查看 web 界面和 HDFS，URL 是 <http://masternode:60070>。

除此之外，你最想的是从 Amazon S3 直接访问数据，可以在 Spark 中使用 s3n://URI 模式来处理。细节参考 90 页的“Amazon S3”。

## 该使用哪个集群管理器？

Spark 支持的集群管理器为部署应用提供了各种不同的选项。如果你要进行一次新的部署，正在寻找一个集群管理器，我们建议跟随下列指导：

- 如果是新部署，从 Standalone 集群开始吧。Standalone 模式是安装最简单的，并且提供了其他集群管理器的几乎所有相同功能，如果你只是运行 Spark 的话。

- 如果你想和其他应用一起运行 Spark，或者使用更丰富的资源调度能力（比如队列），YARN 和 Mesos 都提供了这些特性。说到这里，YARN 在许多 Hadoop 分发包中都是预先安装的。
- Mesos 领先 YARN 和 Standalone 模式的优势是其细粒度共享选项。这使得像 Spark shell 这样的交互式应用在空闲时减少了 CPU 占用。对于多用户运行交互式 shell 的环境很有吸引力。
- 不管哪种情况，最好是在 HDFS 同节点上运行 Spark 以便快速访问存储。你可以手动安装 Mesos 或者 Standalone 集群管理器在同一个节点。或者，大多数 Hadoop 分发包都已经安装了 YARN 和 HDFS 在一起。

最后记住，集群管理器正在快速发展。随着本书出版，也许 Spark 对当前的集群管理器有了更多的特性，也许有了新的集群管理器。这里描述的提交应用的方法将不会改变，不过你可以检查你使用的 Spark 发行版的官方文档看最新的选项。

## 总结

本章描述了 Spark 应用的运行架构，由 driver 进程和一组分布式 executor 进程组成。然后我们讨论了如何构建，打包，和提交 Spark 应用。最后我们概览了 Spark 的常用部署环境，包括内置的集群管理器，运行 Spark 到 YARN 或 Mesos，运行 Spark 到 Amazon 的 EC2。在下一章，我们要探讨更多高级的操作问题，关注 Spark 应用产品的调优和调试。





# Spark 的调优和调试

本章描述如何配置 Spark 应用以及给出如何对 Spark 产品进行调优和调试的概述。Spark 被设计为对许多情况都可以直接工作，然而还是有些配置用户可能会想要修改。本章概述 Spark 的配置机制，强调下用户会想微调的一些选项。配置对于调整应用的性能也有用。在本章第二部分，会谈一些理解 Spark 应用的性能的必要基础，相关的配置设定，以及编写高性能应用的设计模式。还会讲到 Spark 的用户接口，instrumentation，以及日志机制的信息。当你进行调优或排错时，这些都很有用。

## 用 SparkConf 配置 Spark

Spark 调优简单来说就是改变 Spark 应用的运行时配置。Spark 的主要的配置机制是 SparkConf 类。当你创建新的 SparkContext 时需要一个 SparkConf 实例。见示例 8-1 到 8-3。

*示例 8-1. Creating an application using a SparkConf in Python*

```
# Construct a conf
conf = new SparkConf()
conf.set("spark.app.name", "My Spark App")
conf.set("spark.master", "local[4]")
conf.set("spark.ui.port", "36000") # Override the default port
# Create a SparkContext with this configuration
sc = SparkContext(conf)
```

示例 8-2. Creating an application using a SparkConf in Scala

```
// Construct a conf
val conf = new SparkConf()
conf.set("spark.app.name", "My Spark App")
conf.set("spark.master", "local[4]")
conf.set("spark.ui.port", "36000") // Override the default port
// Create a SparkContext with this configuration
val sc = new SparkContext(conf)
```

示例 8-3. Creating an application using a SparkConf in Java

```
// Construct a conf
SparkConf conf = new SparkConf();
conf.set("spark.app.name", "My Spark App");
conf.set("spark.master", "local[4]");
conf.set("spark.ui.port", "36000"); // Override the default port
// Create a SparkContext with this configuration
JavaSparkContext sc = JavaSparkContext(conf);
```

SparkConf 类很简单：一个 SparkConf 实例包含了用户想要覆盖的配置项的 key/value 值。在 Spark 中的每个配置项是基于一个字符串 key 和值 value。要使用你创建的 SparkConf 对象，调用 set() 添加配置值，然后将该对象提供给 SparkContext 做构造用即可。除了 set()，SparkConf 类还包括了少量的工具方法用于常用参数设置。在前面的例子中，你也可以分别调用 setAppName() 和 setMaster() 来设置 spark.app.name 和 spark.master 的配置。

这些例子中，SparkConf 值是在应用的代码里编程设置的。在许多情况下，对给定的应用动态的填入配置会更方便。Spark 运行通过 spark-submit 工具动态设置参数。当应用被 spark-submit 启动时，会将配置值注入到环境里。当新的 SparkConf 被构造时，这些值会被检测到并自动填入。因此，用户的应用也可以简单的构造一个“空的” SparkConf 对象，直接传入 SparkContext 的构造函数，如果你是用 spark-submit 提交的话。

Spark-submit 工具提供了内置的标志用于常见的 Spark 配置参数，记忆一个通用的 --conf 标志，可以接受任何 Spark 配置参数值。详见示例 8-4。

示例 8-4. Setting configuration values at runtime using flags

```
$ bin/spark-submit \  
  --class com.example.MyApp \  
  --master local[4] \  
  --name "My Spark App" \  
  --conf spark.ui.port=36000 \  
  myApp.jar
```

spark-submit 也支持从文件加载配置。这对设置环境配置会有用，环境配置是对多个用户共享的，比如默认的 master。默认情况下，spark-submit 会在 Spark 目录中寻找名为 conf/spark-default.conf 的文件并试图从文件中读取以空格分隔的 key/value 值。你也可以用 --properties-file 标志自定义该文件的准确位置给 spark-submit。如示例 8-5 所示。

示例 8-5. Setting configuration values at runtime using a defaults file

```
$ bin/spark-submit \  
  --class com.example.MyApp \  
  --properties-file my-config.conf \  
  myApp.jar
```

```
## Contents of my-config.conf ##  
spark.master local[4]  
spark.app.name "My Spark App"  
spark.ui.port 36000
```



关联于给定应用的 SparkConf 一旦传入了 SparkContext 构造器就不可变了。就是说所有的配置必须在 SparkContext 初始化前确定好。

有些情况下，同样的配置属性可能要在多个地方设置。比如，用户可能直接对 SparkConf 对象调用 setAppName()，同时也会传入 --name 标志到 spark-submit。这时候就有个优先级的问题。优先级最高的是在用户代码的 SparkConf 对象中用 set() 函数显式配置的值，接下来是传入到 spark-submit 的配置值，然后是属性文件中的值，最后是默认值。如果你想知道对于给定的应用，哪个配置在生效，可以通过本章后面讨论的应用的 web 界面的显示来检查活动配置列表。

几个常用配置列示在表 7-2。表 8-1 提到了几个额外的选项，可能有人感兴趣。完整的配置项列表请查看 Spark 文档。

表格 8-1 常见 Spark 配置

配置	默认值	解释
spark.executor.memory (--executor-memory)	512m	每个 executor 进程使用的内存数量，和 JVM 内存字符串同样格式（例如 512m, 2g）。本选项的更多细节见 158 页的“硬件配置”。
spark.executor.cores(-- executor-cores)	1	配置应用使用的核数的上限。YARN 模式中 spark.executor.cores 给每个 executor 分配指定的核数。Standalone 模式和 Mesos 模式可以用 spark.cores.max 指定所有 executor 使用的总核数的上限。更多细节参考 158 页的“硬件配置”。
spark.cores.max(--total executor-cores)	(none)	
spark.speculation	false	设置为 true 将激活任务的预测执行。意思是运行慢的任务将有其他节点启动的第二个副本。激活本项有助于大集群减少掉队的任务。
spark.storage.blockManagerTimeoutInterval Ms	45000	用于跟踪活跃 executor 的内部超时值。对于长时间 GC 暂停的任务，调整该值到 100 秒（值是 100000）或者更高可以防止丢弃。在未来的 Spark 版本中，本项可能会被通用的超时设置替换，所以，请检查当前的文档。
spark.executor.extraJavaOptions spark.executor.extraClasspath spark.executor.extraLibraryPath	(empty)	这三个配置允许你自定义 executor 的 JVM 的启动行为。这三个配置添加额外的 Java 参数，classpath 和 JVM 库的路径。这些参数用字符串指定（如 spark.executor.extraJavaOptions="-XX:+PrintGCDetails -XX:PrintGCTimeStamps"）。要注意尽管这允许你增加 executor 的 classpath，推荐的方式还是通过 --jars 标志添加依赖到 spark-submit（不要用该配置）。

配置	默认值	解释
Spark.serializer	org.apache.spark.serializer.JavaSerializer	用于需要发送到网络的序列化对象或者需要以序列化形式被缓存的对象的类。默认的 Java 序列化类可以对任何对象序列化，但是太慢了。所以我们推荐 org.apache.spark.serializer.KryoSerializer，并且当速度是必要的情况下也配置 Kryo 序列化类。还可以是任何 org.apache.spark.serializer 的子类。
Spark.[X].port	(random)	允许设置整型端口值用于运行 Spark 应用。对于集群所在网络访问是被安全保卫的情况下会有用。X 的可能值包括：driver，fileserver，broadcast，replClassServer，blockManager，以及 executor。
spark.eventLog.enabled	false	设置成 true 激活事件日志，允许使用 history server 来查看完整的 Spark 作业。更多关于 Spark 的 history server 的信息，参见官方文档。
spark.eventLog.dir	file:///tmp/spark-events	事件日志的存储位置，如果激活了的话。需要配置到全局可见的文件系统，例如 HDFS。

几乎所有的 Spark 配置都可以使用 SparkConf，但有一个重要配置不可以。要设置用于 Spark 处理 shuffle 数据的本地存储目录（对 Standalone 和 Mesos 模式是必须的），要导出 conf/spark-env.sh 中的 SPARK\_LOCAL\_DIRS 环境变量到逗号分隔的存储位置列表。SPARK\_LOCAL\_DIRS 在 158 页的“硬件配置”中详述。该参数和其他参数都不一样，因为它的值对不同的物理主机可能不一样。

## 运行组件：Jobs, Tasks, Stages

调优和调试 Spark 应用的第一步是深入理解 Spark 的内部设计。在前一章中你已经看到了 RDD 及其分区的逻辑表示。当执行时，Spark 通过合并多个操作到 task 来将逻辑表示转换为物理执行计划。理解 Spark 的各个方面超出了本书的范围，但是在调优和调试 job 时，对步骤中涉及的相关术语的理解会很有帮助。

为展示 Spark 的执行阶段，我们将走读一个应用示例，看看用户代码是如何编译为底层执行计划的。我们考虑的应用是在 Spark shell 里进行简单的日志分析。对输入数据，我们用由各种严重程度的日志消息组成的文本文件，还散布了一些空行（示例 8-6）。

*示例 8-6. input.txt, the source file for our example*

```
## input.txt ##
INFO This is a message with content
INFO This is some other content
(empty line)
INFO Here are more messages
WARN This is a warning
(empty line)
ERROR Something bad happened
WARN More details on the bad thing
INFO back to normal messages
```

我们想在 Spark shell 中打开该文件并计算每个严重级别的日志消息出现多少条。首先创建 RDD 来帮我们回答这个问题，见示例 8-7。

*示例 8-7. Processing text data in the Scala Spark shell*

```
// Read input file
scala> val input = sc.textFile("input.txt")
// Split into words and remove empty lines
scala> val tokenized = input.
  | map(line => line.split(" ")).
  | filter(words => words.size > 0)
// Extract the first word from each line (the log level) and do a count
scala> val counts = tokenized.
  | map(words => (words(0), 1)).
  | reduceByKey{ (a, b) => a + b }
```

一系列的 RDD 命令导致有了一个 RDD：counts，包含了每个严重级别的日志条数。在 shell 中执行完这些行后，程序没有执行任何动作。而是暗中定义了 RDD 的一个有向无环图(DAG)，一旦之后有动作发生就能用上。每个 RDD 都维护一个指针指向一个或多个上层对象，连同其为关于何种关系类型的元数据。例如，当你对 RDD 调用 val b=a.map()，RDD b 就维护了到上层节点 a 的引用。这些指针允许 RDD 被追溯到它的所有有祖先。

要显示 RDD 的血统，Spark 提供了 toDebugString()方法。示例 8-8 中，我们将看到前一个例子中创建的一些 RDD 的血统。

示例 8-8. Visualizing RDDs with `toDebugString()` in Scala

```
scala> input.toDebugString
res85: String =
(2) input.text MappedRDD[292] at textFile at <console>:13
  | input.text HadoopRDD[291] at textFile at <console>:13
scala> counts.toDebugString
res84: String =
(2) ShuffledRDD[296] at reduceByKey at <console>:17
  +- (2) MappedRDD[295] at map at <console>:17
    | FilteredRDD[294] at filter at <console>:15
    | MappedRDD[293] at map at <console>:15
    | input.text MappedRDD[292] at textFile at <console>:13
    | input.text HadoopRDD[291] at textFile at <console>:13
```

首先显示的是 `input` RDD。我们通过调用 `sc.textFile()` 创建了它。血统给了我们一些线索关于 `sc.textFile()` 做了些什么。它显示了在 `textFile()` 中哪些 RDD 被创建了。我们能看到它先创建了一个 `HadoopRDD`，然后对其执行了一个 `map` 创建了返回的 RDD。`counts` 的血统更复杂一些。该 RDD 有多个祖先，因为对 `input` RDD 还做了其他操作，比如另外的映射，过滤，以及归约。这里展示的 `counts` 的血统也图形化的显示在了图 8-1 的左侧。

在我们执行动作之前，这些 RDD 只是简单保存了帮助我们以后计算的元数据。为了触发计算，让我们对 `counts` RDD 调用个 `collect()` 返回到 `driver`，见示例 8-9。

示例 8-9. Collecting an RDD

```
scala> counts.collect()
res86: Array[(String, Int)] = Array((ERROR,1), (INFO,4), (WARN,2))
```

Spark 的调度器创建了物理执行计划来计算需要执行动作的 RDD。当对 RDD 调用 `collect()`，RDD 的每个分区必须被兑现并传输到 `driver` 程序。调度器从最终被计算的 RDD 开始回溯找到要计算的 RDD。访问那个 RDD 的上级节点，上级的上级，等等。递归的形成必要的物理执行计划来计算所有的祖先。最简单的情况下，调度器为图中的每个 RDD 输出计算 stage。而 stage 包含 RDD 的每个分区的 task。这些 stage 最后反序执行，计算最终需要的 RDD。

对更复杂的情况，物理的 stage 集合不会精确的 1:1 对应到 RDD 图。这可能会发生在调度器执行流水化或者是压缩多个 RDD 到一个 stage。流水化发生在当 RDD



被从上层节点开始计算而没有数据移动的时候。示例 8-8 中的血统输出显示使用缩进级别来说明哪些 RDD 被一起流水化到一个物理 stage。和上层节点在同一个缩进级别存在的 RDD 在物理执行期间会被流水化。例如，当我们计算 counts，即使有很多上层 RDD，但只有两级缩进显示。这表示物理执行计划只需要两个 stage。这种情况下流水化是因为有多个连续的过滤和映射操作。图 8-1 的右边展示了需要计算 counts RDD 的 2 个 stage。

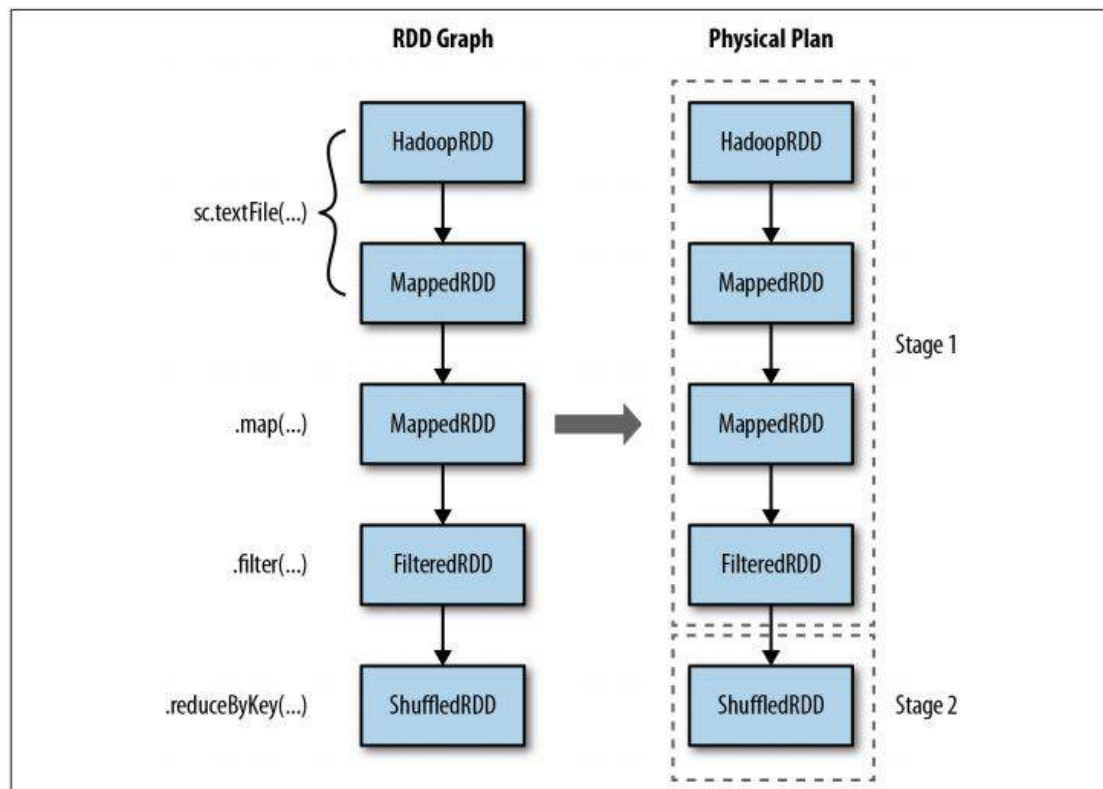


图 8-1 RDD 变换流水化到物理 stage

如果你访问应用的 web 界面，你会看到为了实现 `collect()` 动作有两个 stage。如果你在自己机器上运行本例，Spark 界面可以在 <http://localhost:4040> 找到。这个界面会在本章后面讨论，不过你可以在这里用来快速查看下当前正在执行哪个 stage。

除了流水化，Spark 内部的调度器也可以截断 RDD 的血统图，前提是某个存在的 RDD 已经被持久化到内存或磁盘中。这种情况下 Spark 可以缩短周期，只基于已持久化的 RDD 开始计算。第二种发生截断的情况是 RDD 作为更早的 shuffle

过程的副作用已经实现了，甚至没有被显式 `persist()` 过。这是个底层优化，利用了 `shuffle` 输出到磁盘的事实。并且，利用这个事实，RDD 血统图中的很多部分被重新计算。

为了看到在物理执行中缓存的效果，让我们缓存 `counts` RDD 来看看是如何截断对将来的动作的执行图的（示例 8-10）。如果你重新访问界面，你会看到当执行后来的计算时，缓存减少了需要的 `stage` 的个数。调用 `collect()` 多次将只显示一个 `stage` 来执行动作。

示例 8-10. Computing an already cached RDD

```
// Cache the RDD
scala> counts.cache()
// The first subsequent execution will again require 2 stages
scala> counts.collect()
res87: Array[(String, Int)] = Array((ERROR,1), (INFO,4), (WARN,2), (##,1),
((empty,2))
// This execution will only require a single stage
scala> counts.collect()
res88: Array[(String, Int)] = Array((ERROR,1), (INFO,4), (WARN,2), (##,1),
((empty,2))
```

为一个特别的动作产生的这一组 `stage` 术语叫 `job`。当我们每次调用类似 `count()` 的动作，我们就创建了由一个或多个 `stage` 构成的一个 `job`。

一旦 `stage` 图被产生，`task` 就被创建，并分发到内部调度器。这十分依赖于所使用的部署模式。`Stage` 在物理计划里会基于血统图互相依赖，所以会被以特定顺序执行。例如，输出 `shuffle` 数据的 `stage` 必须在依赖于该输出数据的 `stage` 发生之前。

物理 `stage` 将启动 `task`，每个 `task` 都做同样的事情，除了数据分区不同。每个 `task` 内部执行相同的步骤：

1. 拿到输入。可以从数据存储（如果该 RDD 是输入 RDD），已存在的 RDD（如果 `stage` 基于已缓存的数据），或者 `shuffle` 输出。
2. 执行计算 RDD 的必要操作。比如对输入数据执行 `filter()` 或者 `map()` 函数，或者执行分组或归约。
3. 输出数据 `shuffle` 到外部存储，或者返回到 `driver`（如果是像 `count()` 这样的动作的最终 RDD）。

在 Spark 中,大多数 logging 和 instrumentation 被表示为术语 stage, task 和 shuffle。理解如何从用户代码编译到底层物理计划是个高级概念,但是能极大地帮助你调优和调测应用。

简而言之,在 Spark 执行期间出现下面的阶段:

#### *用户代码定义 RDD 的 DAG(有向无环图)*

对 RDD 的操作创建新的 RDD,新 RDD 重新提及父节点,从而创建了图。

#### *动作迫使 DAG 转换到执行计划*

当你对 RDD 调用动作,就必须计算。也要计算其上层 RDD。Spark 的调度器提交 job 来计算所有需要的 RDD。Job 就会有一个或多个 stage, stage 由一波又一波的 task 并行计算组成。每个 stage 对应到 DAG 中的一个或多个 RDD。因为流水化的原因,单个的 stage 能对应到多个 RDD。

#### *Task 被调度在集群执行*

Stage 按顺序处理,每个单独的任务计算 RDD 的一段。一旦 job 的最后 stage 完成,动作就结束了。

在给定的 Spark 应用中,这一系列步骤可能发生多次,连续的改变会创建新的 RDD。

## 寻找信息


Spark 在应用执行时记录了详细的进度信息和性能指标。这些信息在两个地方展现给用户: Spark 的 web 界面以及 driver 和 executor 进程的产生的日志文件。

## Spark Web 界面

学习 Spark 的行为和性能的第一站是 Spark 内置的 web 界面。默认是在 driver 运行的机器的 4040 端口。警告,在 YARN 的 cluster 模式下,应用的 driver 是运行在集群上,你应该访问 YARN 的资源管理器,它会直接代理请求到 driver。

Spark 的界面包含多个不同页面,不同 Spark 版本格式也可能不一样。对于 Spark 1.2,界面由四个不同部分组成,下面会看到。

job 页面，如图 8-2 所示，包含了活动的和最近完成的 Spark job 的详细执行信息。本页一个非常有用的信息是正在运行的 job、stage 和 task 的进度。对于每个 stage，本页提供了多个指标，你可以用来更好的理解物理执行。

 job 页面只是在 Spark 1.2 中才加入的，所以在更早版本的 Spark 中你可能不会看到。

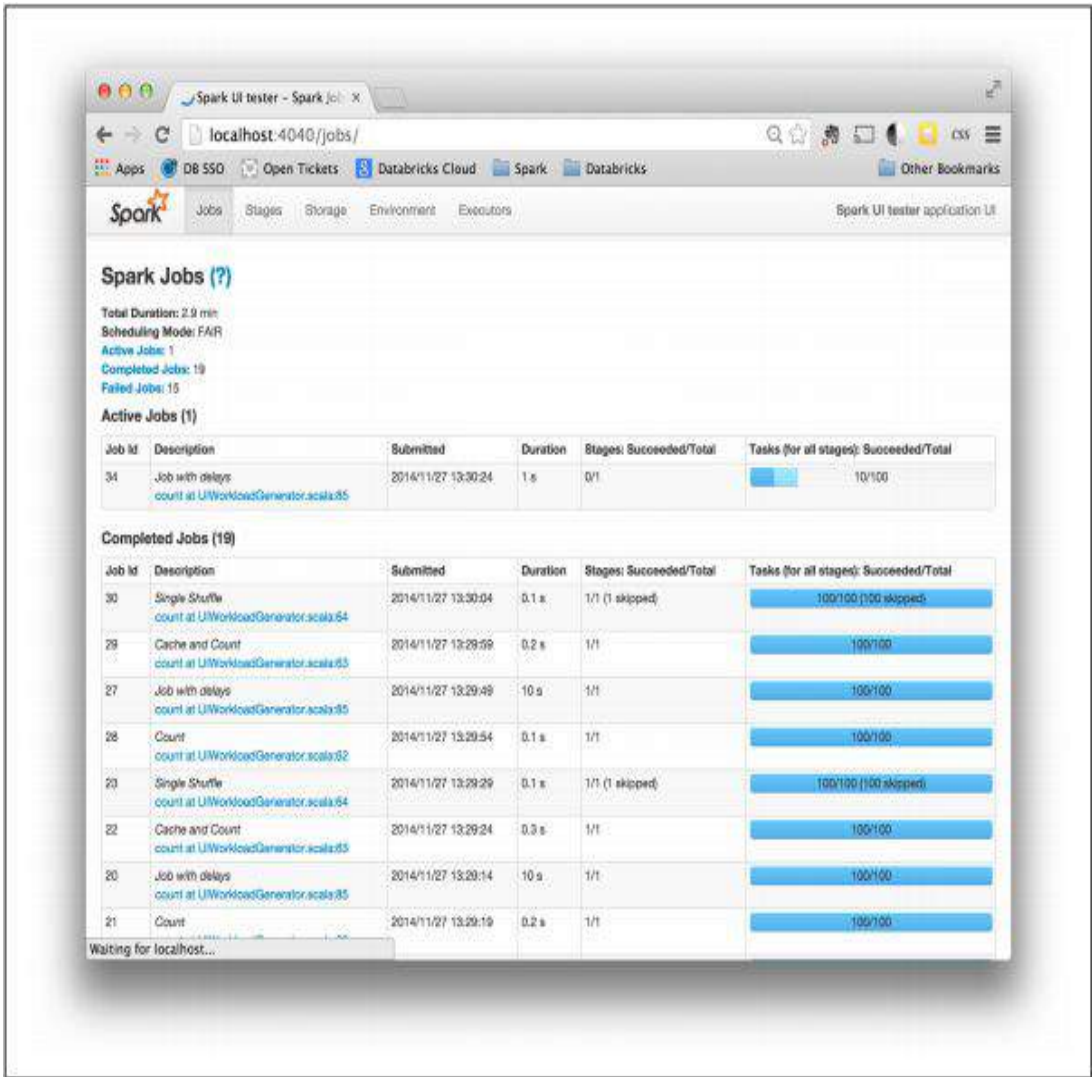


图 8-2 Spark 应用界面的 job 页面

本页面常用于评估 job 的性能。一个良好的开端是浏览组成 job 的这些 stage，看看是否有些特别慢或者相同 job 运行多次的响应时间明显不同。如果你有个代价

高昂的 stage，你可以点击链接来更好的理解该 stage 关联的用户代码是什么。

一旦你缩小范围到一个感兴趣的 stage，stage 的页面见图 8-3，它可以帮助你隔离性能问题。类似 Spark 的数据并行的系统中，性能问题的常见来源是数据倾斜。这发生在少数 task 相比其他占用了非常多的时间。Stage 页面可以帮助你通过查看所有 task 的不同指标的分布来识别倾斜的情况。最好一开始先看运行中的 task，是少数 task 比其他的占用了更多时间吗？如果是这种情况，你可以深度挖掘看是什么导致 task 过慢。是少数 task 比其他读写了更多的数据吗？是运行在某个节点上的 task 非常慢吗？当你调试 job 时，首先确认这些会有帮助。

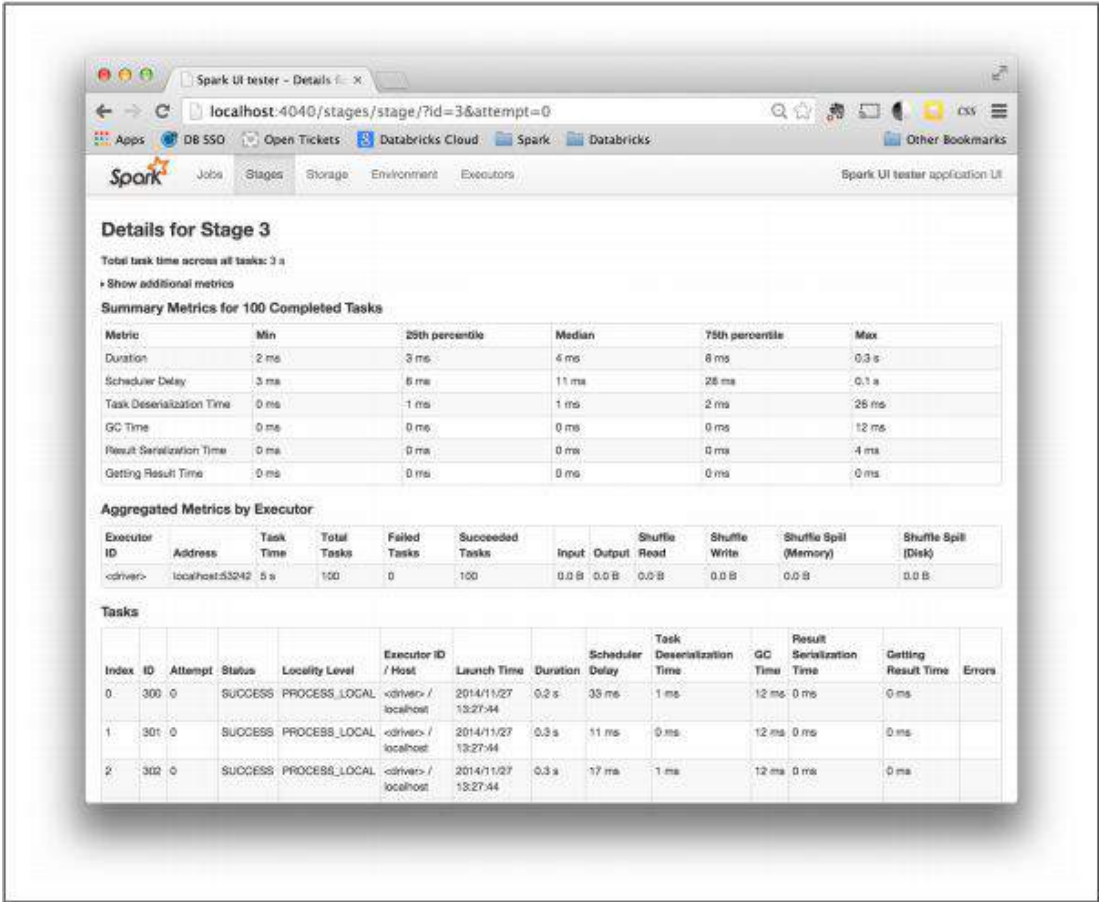


图 8-3 Spark 应用界面的 stage 详情页面

除了查看 task 倾斜，还能帮助确认在 task 生命周期中每个阶段所花的时间是多少：读数据阶段，计算节点，以及输出阶段。如果读写数据花很少的时间，但整个过程耗时长，可能是用户代码本身太耗时（用户代码优化的示例见 107 页的“基

于分区工作”）。有些 task 可能要花掉几乎所有的时间于从外部存储系统加载数据，这无法从 Spark 的优化中受益太多，因为瓶颈是在数据输入。

### **Storage: 持久化 RDD 的信息**

Storage 页面包含了已持久化的 RDD 的信息。如果对 RDD 调用了 `persist()`，RDD 就会被持久化，并且之后在某些 job 中被计算。有些情况下，如果缓存了太多的 RDD，老的数据就会被从内存丢弃，给新数据腾出空间。本页会准确的告诉你每个 RDD 被缓存的份数以及数据缓存在各种媒介的大小（磁盘或者内存等）。细看本页有助于了解重要的数据是否保存在内存中。

### **Executors: 应用在存在的 executor 列表**

本页列出了应用中所有活动的 executor，以及处理中的相关指标和每个 executor 的存储。本页的价值之一是确认你的应用拥有期待的资源数。当调试问题是，最好首先细看本页，错误的配置会导致分配的 executor 比期待的少，很明显，这是影响性能的。还可以帮助查看 executor 的反常行为，比如非常大比例的失败的 task。Executor 有较高的错误率可能是说明配置不正确，或者是物理主机出错的问题。只要去掉这个主机就能提高集群的性能。

本页的另一个特征是可以从 executor 搜集堆栈跟踪信息，点下 Thread Dump 按钮（Spark 1.2 才引入该特征）。executor 线程的调用堆栈可视化可以准确及时显示代码当前运行到了哪里。如果用这个特征，在较短的时间周期内多次采样，你可以识别出代码中的“热点”或者开销大的部分。这种非正式的 profile 能检测出低效的用户代码中。

### **Environment: 调试 Spark 的配置**

本页枚举了你 Spark 应用的环境中激活的属性集。这里的配置代表了你的应用配置的“落地值”。这有助于你调测哪个标准被使能了，特别是当你使用了多种配置机制时。本页还会罗列出你添加到应用中的 JAR 包和文件，这有助于你追踪类似依赖缺失的问题。

## Driver 和 Executor 的日志

某些情况下，用户通过检查 driver 和 executor 直接生成的日志可以从 Spark 得到更多信息。日志包含了更多非正常事件的痕迹，比如内部警告，或者来自用户代码的详细的异常信息。这些数据有助于你定位错误和不可预期的行为。

Spark 日志文件的准确位置依赖于部署模式：

- Standalone 模式，应用的日志直接显示在 master 的 web 界面。默认是保存在每个 worker 节点的 work/目录。
- Mesos 模式，日志保存在 Mesos 的 slave 节点的 work/目录，可以从 Mesos 的 master 界面访问。
- YARN 模式，最简单的搜集日志的方法是使用 YARN 的日志搜集工具（运行 `yarn logs -applicationId <app ID>`）来生成包含你的应用的日志报告。这只有在应用完全执行完才有用，因为 YARN 必须先将日志搜集到一起。为了在 YARN 中查看正在运行的应用的日志，你可以从资源管理器的页面点击到节点页面，然后浏览某个特定的节点，并从这里到某个特定容器。YARN 会给你在那个容器中被 Spark 输出产生的关联日志。在未来的 Spark 版本中，这个过程可能会变得没那么绕，而是直接链接到相关日志。

默认 Spark 输出正常数量的日志信息。定制日志行为来改变日志级别或者日志输出到非标准输出也是可以的。Spark 的日志子系统基于 log4j，一个广泛使用的 Java 库，并且使用的 log4j 的配置格式。Log4j 的配置文件的示例捆绑在 Spark 的 `conf/log4j.properties.template` 文件。要定制 Spark 的日志，先复制示例文件为 `log4j.properties`，然后修改其行为，比如根日志级别（日志输出的门限级别）。默认级别是 INFO。对减少日志输出，可以设为 WARN 或 ERROR。一旦你调整好日志到你想要的级别或格式，你可以用 `spark-submit` 的 `--files` 标志来添加 `log4j.properties` 文件。如果你用这种方法设置日志级别还有问题，确保你的应用中没有包括本省也含有 `log4j.properties` 文件的 JAR 包。Log4j 使用扫描 classpath 找到的第一个属性文件，所以如果在别的地方先找到，就会忽略你自定义的那个。

## 关键性能考量

至此，你已经了解了一些 Spark 内部如何工作，如何追踪正在运行的 Spark 应用的进度，以及从哪里去查看指标和日志信息。本节再前进一步，讨论在 Spark 应用中你可能遇到的常见的性能问题以及相关的调整应用获得最佳性能的小窍门。前三节涉及到为提高性能你可以进行的代码级的修改，最后一节讨论集群调优和 Spark 的运行环境。

## 并行度

RDD 的逻辑表达是一个集合对象。在物理执行期间，如本书以讨论过几次的，RDD 被分割为一组分区，每个分区包含了所有数据的一部分。当 Spark 调度和执行 task，它会对每个分区的数据创建单个的 task。默认情况下，这个 task 会请求一个单独的核在集群中执行。Spark 会推断它认为对 RDD 的一个较好的并行度，大多数情况下很有效。输入 RDD 通常根据底层存储选择并行度。例如，HDFS 上的输入 RDD 是 HDFS 文件底层的每个块一个分区。来自其它 RDD shuffle 后的 RDD 基于上级 RDD 的大小来确定并行集合。

并行度从两方面影响性能。首先，如果并行度太低，Spark 的资源可能会闲置。比如，如果你的应用分配了 1000 个核，而你正在运行只有 30 个 task 的 stage，那么你就可以增加并行级别以利用更多的核。如果并行度太高，每个分区就增加了更多的开销，会导致低效。标志就是能让你的 task 都几毫秒内迅速完成，或者 task 不用读写任何数据。

Spark 提供两种方式调整操作并行度。第一种是对 shuffle 数据的操作，总是可以对生成的 RDD 通过参数来设置并行度。第二种是对于任何存在的 RDD，重新分布其到更多或更少的分区。Repartition() 操作可以随机 shuffle RDD 数据到你希望的分区个数。如果你知道你正在收缩 RDD，可以用 coalesce() 操作；这比 repartition() 操作更高效，因为避免了 shuffle 数据。如果你认为有太多或太少的并行度，用这些操作可以帮助你重新分区。

看个例子，比如说我们正从 S3 读取大量数据，然后立即执行 filter() 操作，看起来会排除掉所有的数据，仅剩一个很小的数据集。默认 filter() 返回的 RDD 和父 RDD 的大小相同，这可能会有很多小的空的分区。这种情况下，你可以通过收



缩到一个更小的 RDD 来提高应用的性能。见示例 8-11。

*示例 8-11. Coalescing a large RDD in the PySpark shell*

*# Wildcard input that may match thousands of files*

```
>>> input = sc.textFile("s3n://log-files/2014/*.log")
```

```
>>> input.getNumPartitions()
```

```
35154
```

*# A filter that excludes almost all data*

```
>>> lines = input.filter(lambda line: line.startswith("2014-10-17"))
```

```
>>> lines.getNumPartitions()
```

```
35154
```

*# We coalesce the lines RDD before caching*

```
>>> lines = lines.coalesce(5).cache()
```

```
>>> lines.getNumPartitions()
```

```
4
```

*# Subsequent analysis can operate on the coalesced RDD...*

```
>>> lines.count()
```

## 序列化格式

当 Spark 通过网络传输数据或数据溢出到磁盘时，需要序列化数据到二进制格式。在 shuffle 操作时能发挥作用，可能有大量的数据要传输。默认情况下 Spark 使用 Java 的内置序列化库。Spark 也支持使用 Kyro，一种第三方序列化库，比 Java 序列化库提升了更快的速度和更紧凑的二进制表示。不过，不能序列化所有类型的对象。几乎所有的应用都会从改用 Kyro 序列化库受益。

要使用 Kyro 库序列化，你可以设置 `org.apache.spark.serializer.KyroSerializer` 到 `spark.serializer` 参数。为了最佳性能，你会想将要序列化的类注册到 Kyro，见示例 8-12。注册类可以使 Kyro 避免为每个独立对象写完整的类名。节省的空间累加起来超过成千上万条序列化记录。如果你想强迫这种类型注册，可以设置参数 `spark.kyro.registrationRequired` 为 `true`，这时如果遇到未注册的类，Kyro 将会抛出异常。

示例 8-12. Using the Kryo serializer and registering classes

```
val conf = new SparkConf()
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
// Be strict about class registration
conf.set("spark.kryo.registrationRequired", "true")
conf.registerKryoClasses(Array(classOf[MyClass], classOf[MyOtherClass]))
```

不管是用 Java 序列化还是用 Kryo，如果你的代码引用了没有继承 Java 序列化接口的类，你都可能遇到 `NotSerializableException` 异常。这种情况下，很难追踪是哪个类导致的问题，因为用户代码中引用了太多不同的类。许多 JVM 支持一个特殊参数来帮助调试这种情况："`-Dsun.io.serialization.extended DebugInfo=true`"。你可以对 `spark-submit` 用 `--driver-java-options` 和 `--executor-java-options` 来使能该选项。一旦你找到了有问题的类，最简单的解决方案就是修改它实现序列化接口。如果你不能修改该类，你将需要高级点的变通方案，比如创建有问题的类的子类并实现 Java 的序列化接口，或者用 Kryo 定制序列化行为。

## 内存管理

Spark 以不同的方式使用内存，所以理解并调整 Spark 的内存使用可以优化你的应用。在每个 `executor` 内部，内存被用于这几个目的：

### *RDD 存储*

当你对 RDD 调用 `persist()` 或 `cache()` 时，分区被保存到内存缓冲里。当缓存使用达到 JVM 的整个堆的一定比例时，Spark 会限制内存使用的数量，可通过 `spark.storage.memoryFraction` 来设置。如果超限，旧的数据会从内存被丢弃。

### *Shuffle 和聚合缓存*

当执行 `shuffle` 操作，Spark 会创建临时缓存来保存 `shuffle` 的输出数据。这些缓存用来保存聚合的中间结果，除了 `shuffle` 直接输出的部分。Spark 会通过 `spark.shuffle.memoryFraction` 尝试限制 `shuffle` 相关缓存占用的内存总量。

### *用户代码*

Spark 执行任意的用户代码，所以用户函数本身会请求大量的内存。例如用

户的应用分配大数组或者对象，这些都会竞争内存的使用。用户代码可以访问 JVM 堆中 RDD 存储和 shuffle 操作之后剩余的所有内存。

默认情况下，Spark 保留 60%的空间给 RDD 存储，20%给 shuffle，剩下的 20%给用户程序。有时用户可以调整这些参数来获得更好的性能。如果你的用户代码分配了非常大的对象，可以减少 RDD 存储和 shuffle 的空间以避免内存溢出。

除了调整内存区域，你还可以提升 Spark 默认缓存行为的某些元素。Spark 默认的 `cache()`操作使用 `MEMORY_ONLY` 存储级别的内存持久化。这意味着如果没有足够的空间缓存新的 RDD 分区，旧的就会被丢弃，并在需要的时候重新计算。有时使用 `MEMORY_AND_DISK` 存储级别调用 `persist()`操作更好，它会移除 RDD 分区到磁盘，并在需要的时候从磁盘读取加载回内存里。这比重新计算更廉价，也导致更可预期的性能。如果 RDD 分区重新计算非常昂贵（比如从数据库读数据），这会特别有用。完整的存储级别列表给出在了表 3-6。

对默认缓存策略的第二个提升是缓存序列化对象而不是原始 Java 对象，可以用 `MEMORY_ONLY_SER` 或者 `MEMORY_AND_DISK_SER` 存储级别来实现。缓存序列化后的对象会稍微减慢缓存操作，因为序列化花时间。但充分减少了花在 JVM 的垃圾回收上的时间，因为许多单独的对象可以被存储为一个单个的序列化缓存。这是因为垃圾回收花的时间和堆中对象的数量成比例，而不是数据的字节大小，而这种缓存方法会将许多对象序列化到一个大缓存里。如果你要缓存大量的数据为对象并且/或者看到较长的垃圾回收暂停，可以使用该选项。这样的暂停可以在应用界面的每个 task 的 GC Time 列看到。

## 硬件配置

你给 Spark 的硬件资源将会显著的影响你的应用完成时间。影响集群大小的主要参数是给每个 `executor` 的内存数量，每个 `executor` 的核数，总的 `executor` 个数，以及用于临时数据的本地磁盘数。

在所有的部署模式中，`executor` 的内存是用 `spark.executor.memory` 参数设置或给 `spark-submit` 的 `--executor-memory` 标志设置的。而 `executor` 数和 `executor` 的核数

选项则依赖于部署模式而不同。在 YARN 中你可以设置 `spark.executor.cores` 或者是 `--executor-cores` 标志和 `--num-executors` 标志来决定总数。在 Mesos 和 Standalone 模式中，Spark 会贪婪的请求调度器能提供的尽可能多的核数和 executor 数。不过 Mesos 和 Standalone 模式都支持设置 `spark.cores.max` 来限制应用中所有 executor 使用的最大核数。本地磁盘用于 shuffle 操作时临时存储。

明显的是，Spark 受益于更多的内存和更多的核数。Spark 的架构允许线性缩放，添加两倍的资源经常能使你的应用快两倍。当规划一个应用时，要额外注意的是你是否计划缓存中间数据集作为你的负载的一部分。如果你计划使用缓存，越多的数据能缓存的做内存里，性能越好。Spark 的 storage 界面给出了关于你缓存在内存的数据比例的详细信息。一种方法是一开始就缓存你的数据的子集在一个较小的集群上，以此推测你的更大的数据需要占用的内存量。

除了内存和 CPU 之外，Spark 需要本地磁盘卷来保存 shuffle 操作需要保存的中间数据以及相关 RDD 分区被溢出到磁盘的数据。用大量的本地磁盘可以帮助加速 Spark 应用的性能。在 YARN 模式中，本地磁盘的配置是直接从 YARN 读取，YARN 提供了自己的机制来指定临时存储目录。在 Standalone 模式中，可以在部署 Standalone 集群时设置 `SPARK_LOCAL_DIR` 环境变量到 `spark-env.sh` 中，Spark 应用在启动时会继承该配置。在 Mesos 模式，或者你正运行在另一个模式，想要覆盖集群的默认存储位置，可以设置 `spark.local.dir` 参数。无论哪种情况，你指定本地目录都是使用一个逗号分隔的目录列表。通常是每个磁盘卷有一个目录给 Spark。写操作会均匀的分布到提供的所有本地目录。所以大量的磁盘将提供更高的吞吐率。

对“越多越好”原则的一个警告是在改变 executor 的内存的时候。用巨大的堆会导致垃圾回收暂停，这会损害 Spark job 的产出率。有时更小的 executor（比如 64GB 或更少）会有利于减轻该问题。Mesos 和 YARN 直接支持将多个更小的 executor 打包进一个物理机器，所以请求更小的 executor 不表示你的应用只有更少的资源。在 Standalone 模式中，对于单个的应用你需要启动多个 worker(取决于 `SPARK_WORKER_INSTANCE`)在同一个机器上超过 1 个 executor。这个限制在以后的 Spark 版本中可能会被去掉。除了使用更小的 executor，保存数据为序列化的形式（见 157 页的“内存管理”）也可以帮助减轻垃圾回收。

## 总结

如果你已经理解了本章，你就已经准备好了解决生产的 Spark 使用情况。我们谈到了配置管理，Spark 界面中的仪表和指标，常用的生产调优技术。要更深入的研究 Spark 调优，可以访问官方文档的调优指南。