从实用角度系统讲解Spark数据处理的工具及使用方法

手把手教你充分利用Spark提供的各种功能，快速编写高效分布式程序

华章科技

[PACKT] PUBLISHING

大数据 技术丛书

Fast Data Processing with Spark

Spark快速数据处理

（美）Holden Karau 著

余璜 张磊 译

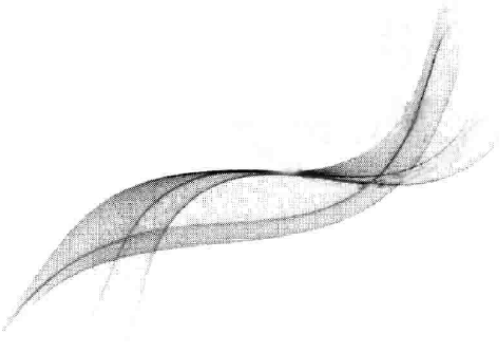
机械工业出版社
China Machine Press

技术丛书

Fast Data Processing with Spark

Spark快速数据处理

（美）Holden Karau 著
余璜 张磊 译



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Spark 快速数据处理 /（美）凯洛（Karau, H.）著；余璜，张磊译．—北京：机械工业出版社，2014.7
（大数据技术丛书）
书名原文：Fast Data Processing with Spark

ISBN 978-7-111-46311-5

Ⅰ. S…　Ⅱ. ①凯…　②余…　③张…　Ⅲ. 数据处理软件－程序设计　Ⅳ. TP274

中国版本图书馆 CIP 数据核字（2014）第 063495 号

本书版权登记号：图字：01-2014-1848

Spark 快速数据处理

[美] Holden Karau 著

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：吴　怡
印　　刷：三河市宏图印务有限公司
版　　次：2014 年 4 月第 1 版第 1 次印刷
开　　本：147mm×210mm　1/32
印　　张：4.125
书　　号：ISBN 978-7-111-46311-5
定　　价：29.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzjsj@hzbook.com

译者序

Spark 起源于 2009 年，是美国加州大学伯克利分校 AMP 实验室的一个研究性项目，于 2010 年开源。随着 Spark 社区的不断成熟，它已被广泛应用于阿里巴巴、百度、网易、英特尔等各大公司的生产环境中。

Spark 是用 Scala 语言写成的一套分布式内存计算系统，它的核心抽象模型是 RDD（Resilient Distributed Dataset，弹性分布式数据集），围绕 RDD 构建了一系列分布式 API，可以直接对数据集进行分布式处理。相对于 MapReduce 上的批量计算、迭代型计算，以及基于 Hive 的 SQL 查询，Spark 可以带来一到两个数量级的效率提升。在 Spark 之上还有若干工具，它们一起构成了一个软件栈——BDAS（Berkeley Data Analytics Stack，伯克利数据分析栈），包括支持用 SQL 查询 Spark 的工具 Shark、支持流式计算的 Spark Streaming、专门针对机器学习的 MLlib，以及专门针对图数据处理的 GraphX。

Spark 对于大多数人来说还是一个新生事物，中文资料十分匮乏，社区内开发者们主要的学习方式还限于阅读有限的官方文档、源码、AMPLab 发表的论文，以及社区讨论。这也是译者翻译本书的初衷，希望借助这本小书让更多的国内读者有机会了解 Spark，使用 Spark，

为 Spark 社区贡献力量。

Spark 的发展日新月异，本书英文版撰写时，Spark 版本为 0.7，时隔半年，Spark 0.9.0 已经发布。原书中的部分内容或链接已经失效，中文版都尽力进行了更新，但难免有所疏漏，欢迎指正。

本书第 1 章 ~ 第 5 章由张磊翻译，第 6 章 ~ 第 9 章由余璜翻译。在本书的翻译过程中还得到了连城、夏帆、杨雪、杨松鹤等人的帮助，在此一并致谢。

余璜

2014.4.1 北京

作者简介

Holden Karau是一名来自加拿大的软件工程师，现居美国旧金山。Holden于2009年毕业于滑铁卢大学，获计算机科学专业数学学士学位，曾是Google软件开发工程师[⊖]，曾在Foursquare工作，在那里结识了Scala语言，也曾在亚马逊从事搜索分类工作。Holden很早就开始对开源情有独钟，Slashdot曾经报道过。编程之外，Holden对火、焊接和舞蹈很着迷。更多信息见网站（http://www.holdenkarau.com）、博客（http://blog.holdenkarau.com）以及（Http://github.com/holdenk）。

感谢每一位参与审阅本书早期版本的同行，尤其是Syed Albiz、March Burns、Peter J.J. MacDonald、Norbert Hu和Noah Fiedel。

关于审阅者

Andrea Mostosi是一名充满激情的软件开发者。2003年还在高中的时候，他从一个单节点LAMP栈开始了软件开发之路，并通过添

⊖ 本书翻译完成时Holden已从Google离职，现就职于Databricks。——译者注

加更多语言、组件和节点与这个栈一起成长。他毕业于米兰，从事过一些网络相关的项目。Andrea 现在与数据打交道，试图发现隐藏在海量数据后的信息。

Andrea 表示：

谢谢我的女友 Khadija，她总是对我所做的一切给予爱和支持。谢谢同我合作过的每一个人，不论是为了快乐还是工作，谢谢他们教我的一切。我还要感谢 Packt 出版社及其员工给予了我参与这本书写作的机会。

Reynold Xin 致力于 Apache Spark 研究，是 Shark 和 GraphX 的主要开发人员，这两个软件是建立在 Spark 基础之上的计算框架。他还是 Databricks 的合伙人，这家公司主要致力于通过 Apache Spark 平台提供大规模数据分析的能力。在加入 Databricks 之前，他曾在加州大学伯克利分校 AMP 实验室攻读博士学位，那里正是 Spark 的发源地。

除了开发开放源码项目，他还经常在大数据学术和行业大会上发表演讲，演讲主题涉及数据库、分布式计算机系统和数据分析。他还曾在业余时间为巴勒斯坦和以色列的高中学生讲授 Android 程序开发。

关于 www.PacktPub.com

支持文件、电子书、打折优惠等

你可以通过访问 www.PacktPub.com 来获取支持文件或英文图书的相关下载。

你可否知道 Packt 为每一本已经出版的英文图书提供电子书版本，同时支持 PDF 和 ePub 格式？

在 www.PacktPub.com 你可以将英文图书升级为电子书，同时作为纸质书消费者，你还可以享受电子书版本的折扣优惠。如果你想了解更多详细内容，请通过 service@packtpub.com 联系我们。

在 www.PacktPub.com，你可以阅读到一系列免费的科普文章。注册即可获得免费时事通讯、专享折扣及在 Packt 英文图书和电子书的各项服务。

http://PacktLib.PacktPub.com

想要即刻得到IT问题的解决方案？ PacktLib是Packt的网上图书馆。在这里，你可以获取、阅读以及搜索Packt图书馆里的所有图书。

为什么订阅我们?

- 实现对Packt出版图书的全面搜索。
- 实现对图书内容的复制粘贴、打印、网摘。
- 实现网页应需。

Packt账户免费资源

如果你在www.PacktPub.com拥有Packt账户，你可以利用该账户访问PacktLib，阅读9本完整的免费英文图书。仅仅使用你的登录账号便可访问。

前　言

程序员经常需要解决一些单机无法完成的任务。网络应用开发领域有很多方便开发的框架，但很少有框架能让编写分布式程序变得简单。本书要讨论的 Spark 能让你更容易地编写分布式应用，并且能够根据自己的喜好使用 Scala、Java 或者 Python 作为开发语言。

本书内容

第 1 章介绍如何安装配置 Spark 集群，该章介绍如何在多种机器上安装 Spark，以及如何配置一个 Spark 集群，包括从在本地机器上部署开发调试环境，到在 EC2 集群上部署用 Chef 管理的大规模集群。

第 2 章介绍如何使用 Spark shell，在交互模式下运行你的第一个 Spark 作业（job）。Spark shell 是一个有用的调试和快速开发工具，特别适合初学者。

第 3 章介绍如何构建和运行 Spark 应用，该章介绍了如何在 Spark 集群上构建一个生产级的脱机 / 独立作业。用 Spark shell 快速开发出应用原型后，剩下的大部分工作就是在 Spark 上构建独立作业了。

第 4 章介绍如何创建 sparkContext，该章介绍如何与 Spark 集群建立连接。SparkContext 是你的程序连接 Spark 集群的入口点。

第 5 章介绍如何加载和保存数据，介绍如何创建和保存 RDD（弹性分布式数据集）。Spark 支持从 Hadoop 加载数据。

第 6 章介绍如何操作 RDD，介绍如何用 Spark 进行分布式数据处理，这一章会非常有趣。

第 7 章介绍如何 Shark-Hive 与 Spark 的综合运用，介绍如何设置 Shark（一种基于 Spark 的 HiveQL 兼容系统），将 Hive 查询集成到你的 Spark 作业中来。

第 8 章介绍如何测试，介绍如何测试你的 Spark 作业。分布式任务调试困难，测试就显得更尤为重要。

第 9 章介绍技巧和窍门，介绍如何提升 Spark 任务的性能。

预备知识

熟悉 Linux/UNIX 以及 C++、Java 或 Python 语言对理解本书内容大有裨益。如果有一台以上机器或 EC2 来做实验，则有利于更深入地了解 Spark 的分布式特性，不过这并非必要，因为 Spark 也有非常出色的单机模式。

目标读者

任何希望学习用 Spark 编写高效分布式程序的开发者都适合阅读本书。

凡例

本书使用多种风格的文本来表示不同信息，下面举例说明这些风格及其含义。

代码、数据库表名、文件夹名、文件名、文件扩展名、路径名、示例 URL、用户输入、Twitter 内容如下所示：“打包文件中包含一个二进制文件目录，该目录需要添加到搜索路径中，SCALA_HOME 必

须指向包解压位置”。

命令行输入或输出风格如下：

```
./run spark.examples.GroupByTest local[4]
```

新词汇或重要词汇加粗。屏幕、菜单、对话框等上面的文字风格如下：

“在 Network & Security 菜单下选择 Key Pairs”。

包括警告或重要提示内容。

技巧和窍门等内容。

读者反馈

欢迎读者反馈。请让我们知道你对本书的看法，喜欢哪些部分，不喜欢哪些部分。读者反馈能够帮助我们开发出更优质的内容，以更好的内容回馈读者。

一般反馈，请发送邮件至 feedback@packtpub.com，请在邮件标题中注明书名。

如果有你擅长并感兴趣的内容，希望写书或参与写书，请参考作者指引：www.packtpub.com/authors。

客户支持

我们准备了大量内容回馈 Packt 出版物的读者，让你觉得物有所值。

源码下载

本书所有源码可以在 github 中下载：

- https://github.com/holdenk/fastdataprocessingwithsparksharkexamples
- https://github.com/holdenk/fastdataprocessingwithsparkexamples
- https://github.com/holdenk/chef-cookbook-spark

免责声明

本书所有观点均出自原书作者，并不代表本出版社观点。本书作者已尽力保证示例源码的使用安全，但是，读者将本书代码用于处理重要数据之前请自行验证。本书作者对内容的完整性、正确性或及时性不作任何保证。作者对由本书内容直接或间接引起的损失、行为、索赔、诉讼等不负任何责任。

勘误

尽管本书经过了严密的审校，但还是难免有纰漏存在。如果你发现我们的书籍中有任何错误，包括文字或代码，请向我们反馈，非常感谢。你的帮助能让其他读者免受疑惑，也有助于我们在后续发行版中有所改进。如果发现任何错误，请访问 http://www.packtpub.com/submit-errata，选择书名，点击 errata submission form 链接，输入错误的详细信息。一旦你提交的勘误被确认，该勘误将会上传到网站，或加入到现有勘误栏下的勘误表中。在 http://www.packtpub.com/support 中选中书名后可以查看已有勘误内容。

严禁盗版

互联网版权内容的盗版问题是由来已久的问题。Packt 出版社严肃对待版权和授权保护，如果读者在互联网发现针对本出版社的任何形式的非法盗版物，请立即与我们联系，提供网址或网站名称，我们

将追究到底。

请通过 copyright@packtpub.com 与我们联系，并提供涉嫌盗版内容的链接地址。

感谢你保全了作者利益，保全了我们提供优质内容的能力。

提问

对于本书任何方面的疑问都可以发送邮件至 questions@packtpub.com，我们将竭力回复。

目　录

第 1 章
安装 Spark 以及构建 Spark 集群

本章将详细介绍搭建Spark的常用方法。Spark的单机版便于测试，同时本章也会提到通过SSH用Spark的内置部署脚本搭建Spark集群，使用Mesos、Yarn或者Chef来部署Spark。对于Spark在云环境中的部署，本章将介绍在EC2（基本环境和EC2MR）上的部署。如果你的机器或者集群中已经部署了Spark，可以跳过本章直接开始使用Spark编程。

不管如何部署Spark，首先得从`http://spark-project.org/download`获得Spark的一个版本，截止到写本书时，Spark的最新版本为0.7版。对于熟悉github的程序员，则可以从`git://github.com/mesos/spark.git`直接复制Spark项目。Spark提供基本源码压缩包，同时也提供已经编译好的压缩包。为了和Hadoop分布式文件系统(HDFS)交互，需要在编译源码前设定相应的集群中所使用的Hadoop版本。对于0.7版本的Spark，已经编译好的压缩包依赖的是1.0.4版本的Hadoop。如果想更深入地学习Spark，推荐自己编译基本源码，因为这样可以灵活地选择HDFS的版本，如果想对Spark源码有所贡献，比如提交补丁，自己编译源码是必须的。你需要安装合适版本的Scala和与之对应的JDK版本。对于Spark的0.7.1版本，需要Scala 2.9.2或者更高的Scala 2.9版本（如2.9.3版）。在写本书时，Linux发行版Ubuntu的LTS版本已经有Scala 2.9.1版，除此之外，最近的稳定版本已经有2.9.2版。Fedora 18已经有2.9.2版。软件包的更新信息可以在`http://packages.ubuntu.com/search?keywords=scala`查看到。Scala官网上的最新版在`http://scala-lang.org/download`。选择Spark支持的Scala版本十分重要，Spark对Scala的版本很敏感。

下载Scala的压缩包，解压后将`SCALA_HOME`设置为Scala

的根目录，然后将 Scala 根目录下的 bin 目录路径加到 PATH 环境变量中。Scala 设置如下：

```
wget http://www.scala-lang.org/files/archive/scala-2.9.3.tgz && tar -xvf
scala-2.9.3.tgz && cd scala-2.9.3 && export PATH=`pwd`/bin:$PATH &&
export SCALA_HOME=`pwd`
```

也可以在 .bashrc 文件中加入：

```
export PATH=`pwd`/bin:\$PATH
export SCALA_HOME=`pwd`
```

Spark 用 sbt（即 simple build tool，现在已经不是个简单工具了）构建，编译源码的时候会花点时间，如果没有安装 sbt，Spark 构建脚本将会为你下载正确的 sbt 版本。

一台双核且安装有 SSD 的笔记本性能不算高，在它之上安装 Spark 的最新版本花了大概 7 分钟。如果从源码开始构建 0.7 版的 Spark，而不是直接下载编译好的压缩包。可以执行：

```
wget http://www.spark-project.org/download-spark-0.7.0-sources-tgz &&
tar -xvf download-spark-0.7.0-sources-tgz && cd spark-0.7.0 && sbt/sbt
package
```

如果底层存储采用 HDFS，而其版本又和 Spark 中的默认 HDFS 版本不一致，则需要修改 Spark 根目录下 project/SparkBuild.scala 文件中的 HADOOP_VERSION，然后重新编译：

```
sbt/sbt clean compile
```

虽然 sbt 工具在依赖性解析方面已经做得非常好了，但是还是强烈推荐开发者去做一次 clean，而不是增量编译。因为增量编译仍然不能保证每次完全正确。

从源码构建 Spark 将花费一些时间，当编译过程停止在 "Resolving [XYZ]...." 很长时间（大概五分钟）之后，停止

然后重新执行 sbt/sbt package 安装。

如果对 HDFS 版本没有特殊要求，只需要下载 Spark 已经编译好的压缩包，解压就能使用：

```
wget http://www.spark-project.org/download-spark-0.7.0-prebuilt-tgz &&
tar -xvf download-spark-0.7.0-prebuilt-tgz && cd spark-0.7.0
```

Spark 最近已经成为 Apache 孵化器项目，作为一个 Spark 应用开发者，所关心的最明显的变化可能就是 org.apache 命名空间下包名的变化。

一些有用的链接和参考如下：

http://spark-project.org/docs/latest
http://spark-project.org/download/
http://www.scala-lang.org

1.1 单机运行 Spark

单机运行是使用 Spark 最简单的方式，同时也是检查 Spark 构建是否有误的明智方法。在 Spark 的根目录下，有个名为 run 的 shell 脚本，能够用来提交一个 Spark 作业。run 脚本的输入是一个代表 Spark 作业的类名和一些参数。./examples/src/main/scala/spark/examples/ 目录下含有大量 Spark 样例作业。

所有的样例程序都有一个输入参数 master，master 参数是分布式集群中 master 节点的 URL，在本地模式下则是 local[N]（N 是线程的个数）。本地模式下用四线程运行 GroupByTest 样例的命令如下：

```
./run spark.examples.GroupByTest local[4]
```

如果出现错误，可能是因为 SCALA_HOME 没有设置。在 bash

中，能通过 `export SCALA_HOME=[pathyouextractscalato]` 设置。

如果出现如下错误，可能是你在使用 Scala 2.10 版，然而 Spark 0.7 版本还不支持 2.10 版的 Scala：

```
[literal]"Exception in thread "main" java.lang.NoClassDefFoundError:
scala/reflect/ClassManifest"[/literal]
```

Scala 的开发者决定重新组织介于 2.9 版和 2.10 版之间的一些类。要解决上面的错误，你可以降低 Scala 版本，也可以等待新版本的 Spark 构建支持 Scala 2.10 版。

1.2 在 EC2 上运行 Spark

Spark 提供了很多在 EC2 环境下运行的脚本，脚本文件都存储在根目录下的 ec2 目录中。这些脚本可以用来同时运行多个 Spark 集群，甚至是运行 `on-the-spot` 实例。Spark 也可以运行在 EMR(Elastic MapReduce) 上，EMR 是 Amazon 关于 MapReduce 集群管理的解决方案，它将会给你扩展实例更大的灵活度。

在 EC2 上用脚本运行 Spark

开始之前，你应该确保有 EC2 账号，如果没有，请访问 `https://portal.aws.amazon.com/gp/aws/manageYourAccount` 注册，对于 Spark 集群，最好访问 `https://portal.aws.amazon.com/gp/aws/securityCredentials` R 生成一个单独访问密钥对。同时也需要生成一个 EC2 的密钥对，这样 Spark 的脚本能够 ssh 到其他已经启动的机器上，这通过访问

https://console.aws.amazon.com/ec2/home 并且选择“Network & Security”下的“Key Pairs”选项来实现。注意密钥对是以区域来创建的。所以你需要确保创建的密钥对和将要使用的 Spark 实例在同一个区域。别忘了给密钥对命名（我们将在本章接下来的内容中使用 spark-keypair 来命名示例中的密钥对），因为将要在脚本中使用它。也可以将你 SSH 公钥上传，从而就不需要重新生成密钥对。这些公钥属于安全敏感信息，所以确保它不被泄露，对于 Amazon 的 EC2 的脚本，也需要设置环境变量 AWS_ACCESS_KEY 和 AWS_SECRET_KEY：

```
chmod 400 spark-keypair.pem
export AWS_ACCESS_KEY="..."
export AWS_SECRET_KEY="..."
```

从 http://aws.amazon.com/developertools/Amazon-EC2/351 下载 Amazon 提供的 EC2 脚本非常有用，解压 zip 文件后，将 bin 目录直接加到 PATH 环境变量中，就像对 Spark 中的 bin 目录所做的一样：

```
wget http://s3.amazonaws.com/ec2-downloads/ec2-api-tools.zip
unzip ec2-api-tools.zip
cd ec2-api-tools-*
export EC2_HOME=`pwd`
export PATH=$PATH:`pwd`:/bin
```

Spark 的 EC2 脚本自动地创建一个用来运行 Spark 集群的隔离安全组和防火墙规则。默认情况下 Spark 的外部通用端口为 8080，这种方式并不好，不幸的是 spark_ec2.py 脚本暂时还不提供限制访问你机器的简单方法。如果你有一个静态 IP 地址，强烈建议在 spark_ec2.py 中限制访问，简单地用 [yourip]/32 替换所有的 0.0.0.0/0。这将不会影响集群内的通信，因为在一个安全组内的所有机器都默认能够和其他机器通信。

然后，启动一个 EC2 上的集群：

```
./ec2/spark-ec2 -k spark-keypair -i pk-[....].pem -s 1 launch
myfirstcluster
```

如果遇到错误："The requested Availability Zone is currently constrained and....", 你可以通过传递 --zone 标记指向另一个 zone。

如果不能 SSH 到集群 master 上的话，请确保只有你有权限去读取私钥，否则 SSH 将会拒绝使用私钥。

由于当节点报告它们自己状态的时候，由于竞争条件你还有可能遇到上面的错误，但是 Spark-ec2 脚本还不能 SSH 过去。关于这个问题，在 https://github.com/mesos/spark/pull/555 上有个修复办法。在 Spark 下一个版本出来之前，有一个临时解决问题的简单方法，就是让 setup_cluster 在启动的时候睡眠 120 秒。

如果启动集群的时候遇到一个短暂的错误，可以用下面命令提供的恢复功能完成启动：

```
./ec2/spark-ec2 -i ~/spark-keypair.pem launch myfirstsparkcluster
--resume
```

万事俱备，你将看到屏幕截图的内容，如图 1-1 所示。

这将分配给你最基本的一个 master 实例和一个 worker 实例的集群，两个实例都是默认配置。接下来，确认 master 节点的 8080 端口没有防火墙规则并确认其已经启动。能看见上面屏幕最后输出 master 的名字。

图 1-1

尝试运行一个 Spark 样例作业来确保配置没有问题：

```
sparkuser@h-d-n:~/repos/spark$ ssh -i ~/spark-keypair.pem root@ec2-107-22-48-231.compute-1.amazonaws.com
Last login: Sun Apr  7 03:00:20 2013 from 50-197-136-90-static.hfc.comcastbusiness.net
       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-ami/2012.03-release-notes/
There are 32 security update(s) out of 272 total update(s) available
Run "sudo yum update" to apply all updates.
Amazon Linux version 2013.03 is available.
[root@domU-12-31-39-16-B6-08 ~]# ls
ephemeral-hdfs  hive-0.9.0-bin  mesos  mesos-ec2  persistent-hdfs
scala-2.9.2  shark-0.2  spark  spark-ec2
[root@domU-12-31-39-16-B6-08 ~]# cd spark
[root@domU-12-31-39-16-B6-08 spark]# ./run spark.examples.GroupByTest
spark://`hostname`:7077
13/04/07 03:11:38 INFO slf4j.Slf4jEventHandler: Slf4jEventHandler started
13/04/07 03:11:39 INFO storage.BlockManagerMaster: Registered BlockManagerMaster Actor
....
```

```
13/04/07 03:11:50 INFO spark.SparkContext: Job finished: count at
GroupByTest.scala:35, took 1.100294766 s
2000
```

既然已经能够在 EC2 的集群上运行一个简单的 Spark 作业，现在就可以针对具体 Spark 作业配置一下你的 EC2 集群了。下面是用 `Spark-ec2` 脚本配置集群的多种配置选项。

首先，考虑所需要的实例类型，EC2 提供一个不断丰富的实例集合，能针对 master 和 worker 选择不同的实例类型。实例类型的选择对 Spark 集群性能有较大影响。如果一个 Spark 作业需要很大内存，最好选择一个含有更大内存的实例。可以通过特指 `--instance-type=(name of instance type)` 来指定实例类型。默认情况下，master 和 worker 的实例类型是一样的。当作业是计算密集型的话，将会出现浪费，因为 master 的资源不能充分被利用。你能通过 `--master-instance-type=(name of instance)` 来指定不同的 master 实例类型。

EC2 也有对于 worker 十分有用的 GPU 实例类型。但对 master 来说完全是浪费。注意，由于虚拟机管理程序的高 IO 负载，EC2 的 GPU 性能将比你本地测试的低。

下载实例代码

本书中所有的实例程序都在三个独立的 github repo 中：

- `https://github.com/holdenk/fastdataprocessingwithspark-sharkexamples`
- `https://github.com/holdenk/fastdataprocessingwithsparkexamples`
- `https://github.com/holdenk/chef-cookbook-spark`

Spark 的 EC2 脚本使用 Spark 组提供的 AMI(Amazon Machine Images)。这些 AMI 往往跟不上 Spark 版本的更新速度，如果有对

于 Spark 的自定义补丁（比如要用不同版本的 HDFS），将不会被包含在机器镜像中。目前，AMI 也只能在美国东部地区使用，如果想在其他地区运行它，需要复制 AMI，或者自己在不同的地区制作你的 AMI。

为了使用 Spark 的 EC2 脚本，你所在的地区要有 AMI。为了将默认的 Spark 的 EC2 AMI 复制到一个新的地区，通过查看 spark_ec2.py 脚本中最新 Spark AMI 是什么，这又是通过查看 `LATEST_AMI_URL` 指向的 URL。对于 Spark 0.7，运行下面的命令获得最新 AMI：

```
curl https://s3.amazonaws.com/mesos-images/ids/latest-spark-0.7
```

`ec2-copy-image` 脚本包含你希望获得的复制镜像的功能，但是不能复制不属于自己的镜像。所以必须登录一个前面 AMI 的实例，然后记录它的快照，通过运行以下命令可以获得当前正在运行的镜像的描述信息：

```
ec2-describe-images ami-a60193cf
```

这将显示一个基于 EBS(Elastic Block Store) 的镜像，需要参照 EC2 的命令创建基于 EBS 的实例。通过已获得的启动实例的脚本，可以启动一个 EC2 集群上的实例，然后将其快照记录下来，通过以下命令找到你运行的实例：

```
ec2-describe-instances -H
```

可以复制 `i-[string]` 实例名字，然后保存留有它用

如果想用 spark 的一个普通版本，或者安装工具和依赖，并让它们成为你的 AMI 的一部分，需要在快照复制之前这样做：

```
ssh -i ~/spark-keypair.pem root@[hostname] "yum update"
```

一旦安装了更新，并且安装好你需要的其他软件，你能够继续然后复制快照你的实例：

```
ec2-create-image -n "My customized Spark Instance" i-[instancename]
```

用前面代码中的 AMI 名字，可以通过指定 [cmd]--ami[/cmd] 命令行参数启动你自定义的 Spark 版本，也可以将它复制到另一个地区使用：

```
ec2-copy-image -r [source-region] -s [ami] --region [target region]
```

你将获得新的 AMI 名字，可以用它提交你的 EC2 任务，如果想用不同的 AMI 名字，只要指定 --ami [aminame] 即可。

当写本书时，有个关于默认 AMI 和 HDFS 的问题，如果 Hadoop 版本和 Spark 编译指定的 Hadoop 版本不一致，需要更新 AMI 上 Hadoop 的版本，参照 https://spark-project.atlassian.net/browse/SPARK-683 获得详细信息。

1.3 在 ElasticMapReduce 上部署 Spark

除了 EC2 基本环境之外，Amazon 还提供一个名为 Elastic MapReduce 的 MapReduce 托管解决方案，伴随地也提供一个 Spark 启动脚本，该脚本可以简化在 EMR 上 Spark 的初始使用流程。可以在 Amazon 上通过下面的命令安装 EMR 工具：

```
mkdir emr && cd emr && wget http://elasticmapreduce.s3.amazonaws.com/
elastic-mapreduce-ruby.zip && unzip *.zip
```

为了使 EMR 脚本能够访问你的 AWS(Amazon Web Service) 账

户，你要创建这样一个 credentials.json 文件：

```
{
    "access-id": "<Your AWS access id here>",
    "private-key": "<Your AWS secret access key here>",
    "key-pair": "<The name of your ec2 key-pair here>",
    "key-pair-file": "<path to the .pem file for your ec2 key pair
here>",
    "region": "<The region where you wish to launch your job flows
(e.g us-east-1)>"
}
```

一旦安装了 EMR 工具，就可以通过运行下面的命令启动一个 Spark 集群：

```
elastic-mapreduce --create --alive --name "Spark/Shark Cluster" \
--bootstrap-action s3://elasticmapreduce/samples/spark/install-spark-
shark.sh \
--bootstrap-name "install Mesos/Spark/Shark" \
--ami-version 2.0  \
--instance-type m1.large --instance-count 2
```

5 到 10 分钟之后 EC2MR 实例将会启动。通过执行 `elastic-mapreduce -list` 可以列出集群的状态。一旦其输出 `j-[jobid]`，准备完毕。

有些参考链接如下：

- http://aws.amazon.com/articles/4926593393724923
- http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-cli-install.html

1.4 用 Chef(opscode) 部署 Spark

Chef 是一个渐渐流行的部署大、小集群的自动化管理平台。Chef 可以用来管理一个传统的静态集群，也可以和 EC2 或者其他的云计算提供商一起使用。Chef 用 cookbook 作为最基本的配置单元，可以被泛化或者特指。如果你以前没有用过 Chef，在 `https://learnchef.opscode.com/` 能找到 Chef 的入门教

程。你可以使用一个泛化的 Spark cookbook 作为最基本的启动集群的配置。

为了让 Spark 运行，需要为 master 和 worker 都创建一个角色，同时完成 worker 连接到 master 的配置。首先从 `https://github.com/holdenk/chef-cookbook-spark` 下载 cookbook。最小化的配置是直接将 master 的 hostname 作为 master（便于 worker 节点连接）和 username，以至于 Chef 可以在正确的位置安装。你也需要接受 Sun Java 的证书，或者切换到可选的 JDK 上。spark-env.sh 中的绝大多数设置都可以通过 cookbook 的配置完成。在 1.7 节“通过 SSH 部署集群”，你能了解这些设置的解释。这些设置可以在每个节点个性化地设定，同时你也可以改变全局的默认值。

为 master 创建一个角色，执行 `spark_master_role -e[editor]`。你将得到一个可编辑的模板文件，比如对于 master，设置如下：

```
{
  "name": "spark_master_role",

  "description": "",
  "json_class": "Chef::Role",

  "default_attributes": {
    },
  "override_attributes": {
   "username":"spark",
   "group":"spark",
   "home":"/home/spark/sparkhome",
   "master_ip":"10.0.2.15",
  },
  "chef_type": "role",
  "run_list": [
    "recipe[spark::server]",
    "recipe[chef-client]",
  ],
  "env_run_lists": {
    },
}
```

除了将 spark::server 改为 spark::client，用相同的方式创建一个对应于 client 端的角色，然后部署到不同的节点：

```
knife node run_list add master role[spark_master_role]
knife node run_list add worker role[spark_worker_role]
```

最后在节点上运行 chef-client 更新。至此就可以运行一个 Spark 集群了。

1.5 在 Mesos 上部署 Spark

Mesos 是一个能够让多个分布式应用和框架运行在同一集群上的集群管理平台。Mesos 可以在同一个集群上智能地、并发地调度和运行 Spark、Hadoop 以及其他框架（这里 Hadoop 和 Spark 都是一个框架）。Spark 可以将它的任务作为 Mesos 任务提交，或者将 Spark 整体作为一个 Mesos 任务提交。Mesos 支持快速扩展以管理一个更大的集群。Mesos 最初是加州大学伯克利分校的一个研究项目，现在已经成为 Apache 孵化器项目，并已被 Twitter 采用。

在使用 Mesos 之前，先从 http://mesos.apache.org/downloads/ 下载最新版本解压。Mesos 提供各种配置脚本，对于 Ubuntu 下的安装使用 configure.ubuntu-lucid-64 脚本，对于其他的 linux 发行版，请参照 Mesos 的 README 文件指明的对应配置脚本。除了安装 Spark 的要求，检查一下你是否安装了 Python C 头文件（Debian 系统下的 python-dev 包），或者传递 --disable-python 参数给配置脚本。由于 Mesos 需要在每台机器上安装，所以最好不要将 Mesos 安装目录配置到 root 用户的目录下，将 Mesos 安装配置到 Spark 用户的目录下会更简便，如下：

```
./configure --prefix=/home/sparkuser/mesos && make && make check && make
install
```

与 Spark 的 standalone 模式的配置类似，这里也需要检查不同的 Mesos 节点能否找到其他的节点。将 master 的 hostname 添加到 mesossprefix/var/mesos/deploy/masters 中，将 worker 节点的 hostname 添加到 mesossprefix/var/mesos/deploy/slaves 中。然后在 master 的 mesossprefix/var/mesos/conf/mesos.conf 文件下设置 master 与其他节点通信的地址和端口。

一旦 Mesos 构建完毕，就该配置 Spark 和 Mesos 一起协同工作了。将 conf/spark-env.sh.template 复制成 conf/spark-env.sh，然后将 MESOS_NATIVE_LIBRARY 设置为 Mesos 的安装目录。下一小节将提到关于 spark-env.sh 中有关设置的更多信息。

需要在集群中的每台机器上都安装 Mesos 和 Spark，一旦在一台机器上配置了 Mesos 和 Spark，可以用 pscp 将构建复制到所有的节点上：

```
pscp -v -r -h  -l sparkuser ./mesos /home/sparkuser/mesos
```

顺利的话，接下来可以使用 mesosprefix/sbin/mesos-startcluster.sh 脚本开启 Mesos 集群，并且通过 mesos://[host]:5050 作为 master 调度 Mesos 上的 Spark。

1.6 在 Yarn 上部署 Spark

Yarn 是 Apache Hadoop 的第二代 MapReduce。你一旦构建了 Spark 的 assembly jar 包，就可以利用 Spark 提供的简单方式在

Yarn 上调度作业。你建立的 Spark 作业需要使用一个 standalone 模式下 master 的 URL。Spark 应用都是从命令行的参数中读取 master 的 URL，所以指定 `--args` 为 standalone。

运行一下 GroupByTest.scala 样例作业，如下：

```
sbt/sbt assembly #Build the assembly

SPARK_JAR=./core/target/spark-core-assembly-0.7.0.jar ./run spark.deploy.
yarn.Client --jar examples/target/scala-2.9.2/spark-examples_2.9.2-
0.7.0.jar --class spark.examples.GroupByTest --args standalone --num-
workers 2 --worker-memory 1g --worker-cores 1
```

1.7 通过 SSH 部署集群

如果集群未安装任何集群管理软件，你能用一些方便 Spark 部署的脚本通过 SSH 部署 Spark。这种方式在 Spark 文档中叫“standalone 模式”。如果只有一个 master 和一个 worker，可以在相应的机器上分别用 `./run spark.deploy.master.Master` 和 `./run spark.deploy.worker.Worker spark://MASTERIP:PORT` 启动相应的实例，master 的默认端口是 8080。即使集群由大量机器组成，也不需要到每一台机器上手动地执行命令来启动。bin 目录下提供了大量的启动服务器的脚本。

使用这些脚本的前提是 master 能够无密码登录到所有 worker。推荐创建一个新用户运行 Spark，并让它成为 Spark 的专用用户。本书用 sparkuser 作为 Spark 的用户。在 master 节点上运行 `ssh-keygen` 生成一个 SSH 密钥，在输入密码时提供空密码。一旦生成了密钥，将公钥（如果使用 RSA 密钥，它一般会被默认存储到 `~/.ssh/id_rsa.pub` 中）添加到每个节点的 `./ssh/authorized_keys` 文件中。

Spark 的管理脚本需要你的用户名匹配，如果不匹配的话，你可以在 ~/.ssh/config 配置另一个用户名。

master 能无密码登录到其他机器后，开始配置 Spark。[filepath]conf/spark-env.sh.template[/filepath] 是一个简便的配置文件模板，将它拷贝到 [filepath]conf/spark-env.sh[/filepath]。修改文件中的 SCALA_HOME 为 Scala 的安装目录，同时根据具体环境配置一些（或者所有的）环境变量，如表 1-1 所示。

表 1-1

名　称	意　义	默认值
MESOS_NATIVE_LIBRARY	指向 Mesos 根目录	无
SCALA_HOME	指向 Scala 根目录	无，非空
SPARK_MASTER_IP	master 节点 IP 地址，用于监听和与其他 worker 节点通信	Spark 启动所在的节点 IP
SPARK_MASTER_PORT	Sparkmaster 节点监听端口号	7077
SPARK_MASTER_WEBUI_PORT	master 节点上 Web 界面端口号	8080
SPARK_WORKER_CORES	worker 节点可使用 CPU 核个数	全部使用
SPARK_WORKER_MEMORY	worker 节点可使用内存量	系统最大内存：1GB（或者 512MB）
SPARK_WORKER_PORT	worker 节点运行所在端口号	随机
SPARK_WEBUI_PORT	worker 节点上 Web 界面端口号	8081
SPARK_WORKER_DIR	worker 节点产生文件所在的位置	SPARK_HOME/work_dir

一旦完成了所有的配置，就可以开始启动和运行集群了。用户需要复制相应版本的 Spark 以及构建好的配置到所有的节点

上。安装 PSSH 可以简便部署过程，分布式 SSH 工具 PSSH 包含 PSCP 功能。PSCP 极大地简便了将文件复制到多个目标节点的过程，尽管拷贝过程需要花费一些时间，使用示例如下：

```
pscp -v -r -h conf/slaves -l sparkuser ../spark-0.7.0 ~/
```

如果配置经过改变，需要将改变后的配置传输到所有的 worker 节点上，如下：

```
pscp -v -r -h conf/slaves -l sparkuser conf/spark-env.sh ~/
spark-0.7.0/conf/spark-env.sh
```

如果集群上使用一个共享的 NFS，尽管 Spark 会默认的命名日志文件，但最好区分地指定 worker 的目录，否则它会写到同一个目录中，如果想在共享 NFS 上设置 worker 的日志目录，请考虑添加 hostname 来区别，例如 `SPARK_WORKER_DIR=~/work-'hostname'`。

为了得到更好的性能，也可以考虑将日志文件放到一个独立目录中。

如果 worker 节点上没有安装 Scala，可以通过 pssh 安装：

```
pssh -P -i -h conf/slaves -l sparkuser "wget http://www.scala-lang.org/
downloads/distrib/files/scala-2.9.3.tgz && tar -xvf scala-2.9.3.tgz && cd
scala-2.9.3 && export PATH=$PATH:`pwd`/bin && export SCALA_HOME=`pwd` &&
echo \"export PATH=`pwd`/bin:\\\\$PATH && export SCALA_HOME=`pwd`\" >>
~/.bashrc"
```

接下来启动集群，`start-all.sh` 和 `start-master.sh` 都是在集群的 master 上运行的脚本。启动脚本都是守护进程，所以终端不会输出大量的运行日志。

```
ssh master bin/start-all.sh
```

如果遇到无法找到一个类的错误，类似于 `java.lang.NoClassDefFoundError: scala/ScalaObject`，请检查是否已经安装并且正确地设置了 SCALA_HOME。

Spark 的脚本假设 Spark 在 master 和 worker 节点上的安装目录是一样的。如果安装目录不同，应该编辑 `bin/spark-config.sh`，将其设置为合适的目录。

Spark 提供的帮助管理集群的命令如表 1-2 所示。

表 1-2

命　令	用　途
`bin/slaves.sh <command>`	command 是可以在任意 worker 节点上运行的命令，比如 `bin/slaves.sh uptime` 能够显示每个 worker 节点运行的总时间
`bin/start-all.sh`	启动 master 节点和所有 worker 节点，在 master 节点上执行
`bin/start-master.sh`	启动 master 节点，在 master 节点上执行
`bin/start-slaves.sh`	启动所有的 worker 节点
`bin/start-slave.sh`	启动一个 worker 节点
`bin/stop-all.sh`	停止 master 节点和所有 worker 节点
`bin/stop-master.sh`	停止 master 节点
`bin/stop-slaves.sh`	停止所有的 worker 节点

spark 启动脚本运行之后，用户可以通过浏览器获得屏幕截图上显示的界面（图 1-2）。在 master 节点的 8080 端口有个友好的 web 界面；通过 8081 端口可以访问和切换到所有的 worker 节点上。界面中包含正在运行的 worker 的信息，以及现在正在运行和已经完成的作业信息。

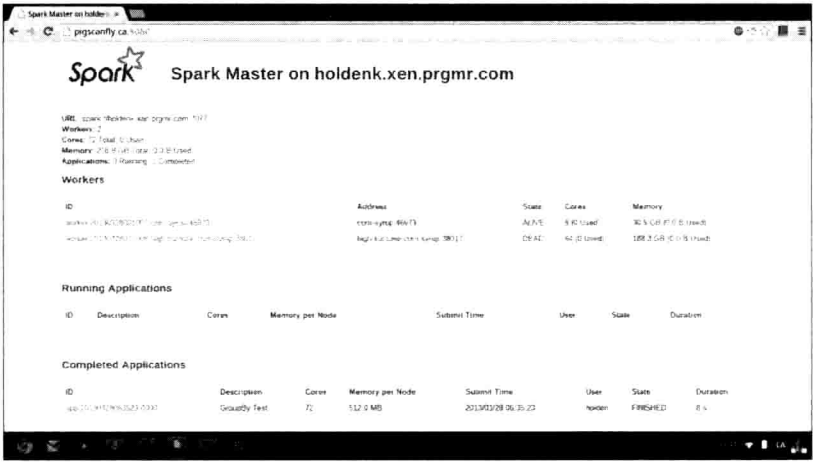


图 1-2

至此 Spark 集群就完成了启动。就像在本地模式下一样，可以使用提供的 run 脚本执行 Spark 作业。所有的样例作业都在 examples/src/main/scala/spark/examples/ 中，这些作业都需要一个参数指向集群的主节点。假设你在 master 节点上，可以试着运行一个样例作业：

```
./run spark.examples.GroupByTest spark://`hostname`:7077
```

如果遇到了下面这个问题 java.lang.UnsupportedClassVersionError，请升级 JDK，如果使用的是 Spark 编译后的版本，需要重新编译源码，Spark 0.7 版本需要 JDK 1.7 版编译，检查一下 JRE 的命令如下：

```
java -verbose -classpath ./core/target/scala-2.9.2/
classes/
spark.SparkFiles | head -n 20
```

注意 49 是 JDK1.5，50 是 JDK1.6，60 是 JDK1.7。

如果不能连接到 localhost，请确认是否已将 master 配置为监

听所有的 IP 地址（或者你没有将 localhost 配置为 master 监听的 ip 地址）。

如果万事俱备，你将看到屏幕上输出一些日志信息。如下：

```
13/03/28 06:35:31 INFO spark.SparkContext: Job finished: count at
GroupByTest.scala:35, took 2.482816756 s
2000
```

1.8 链接和参考

- http://archive09.linux.com/feature/151340
- http://spark-project.org/docs/latest/spark-standalone.html
- https://github.com/mesos/spark/blob/master/core/src/main/scala/spark/deploy/worker/WorkerArguments.scala
- http://bickson.blogspot.com/2012/10/deploying-graphlabsparkmesos-cluster-on.html
- http://www.ibm.com/developerworks/library/os-spark/
- http://mesos.apache.org/
- http://aws.amazon.com/articles/Elastic-MapReduce/4926593393724923
- http://spark-project.org/docs/latest/ec2-scripts.html

1.9 小结

本章介绍了怎样部署 Spark 的本地模式和集群模式，现在可以开始运行 Spark 应用了。在下章中，我们将要学习怎样使用 Spark shell。

第 2 章
Spark shell 的使用

Spark shell是一个特别适合快速开发Spark原型程序的工具，可以帮助我们熟悉Scala语言。即使你对Scala不熟悉，仍然可以使用这个工具。Spark shell使得用户可以和Spark集群交互，提交查询，这便于调试，也便于初学者使用Spark。前一章介绍了运行Spark实例之前的准备工作，现在你可以开启一个Spark shell，然后用下面的命令连接你的集群：

```
MASTER=spark://`hostname`:7077 ./spark-shell
```

如果在本地模式下运行Spark，而且在没有Spark实例运行的前提下，直接执行./spark-shell将以单线程启动，多线程需追加`local[n]`。

2.1 加载一个简单的text文件

Spark shell一旦连接上一个Spark集群，终端会输出一些日志信息，这些信息中含有指定app ID的内容，类似于`Connected to Spark cluster with app ID app-20130330015119-0001`。随后Web用户界面（默认8080端口）上正在运行的application中会显示该app ID。接下来下载一个数据集来做些实验，为《The Elements of Statistical Learning》这本书准备的大量数据集，都是以非常便于使用的格式给出的。获得垃圾链接数据集的命令如下：

```
wget http://www-stat.stanford.edu/~tibs/ElemStatLearn/datasets/spam.data
```

在Spark shell中输入以下语句，作用是将`spam.data`当作文本文件加载到Spark中：

```
scala> val inFile = sc.textFile("./spam.data")
```

上面的语句将 spam.data 文件中的每行作为一个 RDD（Resilient Distributed Datasets）中的单独元素加载到 Spark 中，并返回一个名为 inFile 的 RDD。

注意当你连接到 Spark 的 master 之后，若集群中没有分布式文件系统，Spark 会在集群中每一台机器上加载数据，所以要确保集群中的每个节点上都有完整数据。通常可以选择把数据放到 HDFS、S3 或者类似的分布式文件系统去避免这个问题。在本地模式下，可以将文件从本地直接加载，例如 sc.textFile([filepah])，想让文件在所有机器上都有备份，请使用 SparkContext 类中的 addFile 函数，代码如下：

```
scala> import spark.SparkFiles;
scala> val file = sc.addFile("spam.data")
scala> val inFile = sc.textFile(SparkFiles.get("spam.data"))
```

与大多数的 shell 一样，Spark shell 也有命令的历史记录。按“up 键”可得到前一条执行过的命令。如果你不想完整地输入一条命令或者不确信调用一个对象的什么方法，按“Tab 键”，Spark shell 能够尽可能地自动补全你的代码。

对于逻辑回归的例子，RDD 要是以文件中的每行来组织记录就不是十分有用了，因为逻辑回归需要由空格分割的数值作为输入数据。直接在 RDD 上做映射，能够较快地获得特定格式的数据（注意 _.toDouble 和 x=>x.toDouble 等价）：

```
scala> val nums = inFile.map(x => x.split(' ').map(_.toDouble))
```

然后比较 nums 和 inFile 这两个 RDD，确认一下两种数据的内容是一致的。通过在两个 RDD 上调用 first() 函数查看这两个

RDD 中的第一个元素：

```
scala> inFile.first()
[...]
res2: String = 0 0.64 0.64 0 0.32 0 0 0 0 0 0 0.64 0 0 0 0.32 0 1.29
1.93 0 0.96 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0.778 0 0 3.756 61 278 1

scala> nums.first()
[...]
res3: Array[Double] = Array(0.0, 0.64, 0.64, 0.0, 0.32, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.64, 0.0, 0.0, 0.0, 0.32, 0.0, 1.29, 1.93, 0.0, 0.96,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.778, 0.0, 0.0, 3.756, 61.0, 278.0, 1.0)
```

2.2 用 Spark shell 运行逻辑回归

当执行一条不包含左半部（比如省略 `val x=y` 中的 `val=x`）的语句时，Spark shell 会以 `res[number]` 命名结果值（number 默认是递增的），若显式地写出 `val res[number]=y`，`res[number]` 将用于命名结果值。之前我们获得了更利于使用的格式的数据集，可以用它来做点更有意义的事情，用 Spark 在垃圾链接数据集上运行逻辑回归程序的例子如下：

```
scala> import spark.util.Vector
import spark.util.Vector

scala> case class DataPoint(x: Vector, y: Double)
defined class DataPoint

scala> def parsePoint(x: Array[Double]): DataPoint = {
      DataPoint(new Vector(x.slice(0,x.size-2)) , x(x.size-1))
      }
parsePoint: (x: Array[Double])this.DataPoint

scala> val points = nums.map(parsePoint(_))
points: spark.RDD[this.DataPoint] = MappedRDD[3] at map at
<console>:24

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

```
scala> val rand = new Random(53)
rand: java.util.Random = java.util.Random@3f4c24
scala> var w = Vector(nums.first.size-2, _ => rand.nextDouble)
13/03/31 00:57:30 INFO spark.SparkContext: Starting job: first at
<console>:20
...
13/03/31 00:57:30 INFO spark.SparkContext: Job finished: first at
<console>:20, took 0.01272858 s
w: spark.util.Vector = (0.7290865701603526, 0.8009687428076777,
0.6136632797111822, 0.9783178194773176, 0.3719683631485643,
0.46409291255379836, 0.5340172959927323, 0.04034252433669905,
0.3074428389716637, 0.8537414030626244, 0.8415816118493813,
0.719935849109521, 0.2431646830671812, 0.17139348575456848,
0.5005137792223062, 0.8915164469396641, 0.7679331873447098,
0.7887571495335223, 0.7263187438977023, 0.40877063468941244,
0.7794519914671199, 0.1651264689613885, 0.1807006937030201,
0.3227972103818231, 0.2777324549716147, 0.20466985600105037,
0.5823059390134582, 0.4489508737465665, 0.44030858771499415,
0.6419366305419459, 0.5191533842209496, 0.43170678028084863,
0.9237523536173182, 0.5175019655845213, 0.47999523211827544,
0.25862648071479444, 0.020548000101787922, 0.185553327397141 37, 0....

scala> val iterations = 100
iterations: Int = 100

scala> import scala.math._

scala> for (i <- 1 to iterations) {
        val gradient = points.map(p =>
          (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
        ).reduce(_ + _)
        w -= gradient
      }
[....]

scala> w
res27: spark.util.Vector = (0.2912515190246098, 1.05257972144256,
1.1620192443948825, 0.764385365541841, 1.3340446477767611,
0.6142105091995632, 0.8561985593740342, 0.7221556020229336,
0.40692442223198366, 0.8025693176035453, 0.7013618380649754,
0.943828424041885, 0.4009868306348856, 0.6287356973527756,
0.3675755379524898, 1.2488466496117185, 0.8557220216380228,
0.7633511642942988, 6.389181646047163, 1.43344096405385,
1.729216408954399, 0.4079709812689015, 0.3706358251228279,
0.8683036382227542, 0.36992902312625897, 0.3918455398419239,
0.2840295056632881, 0.7757126171768894, 0.4564171647415838,
0.6960856181900357, 0.6556402580635656, 0.060307680034745986,
0.31278587054264356, 0.9273189009376189, 0.0538302050535121,
0.545536066902774, 0.9298009485403773, 0.922750704590723,
0.072339496591
```

如果顺利，你已经使用Spark运行了逻辑回归程序。至此，我们已经学到了很多，学到了定义类，创建RDD以及创建函数。感受到Spark shell是如此的方便，因为它很大程度上基于Scala REPL（Scala交互式shell，即Scala解释器），并且继承了Scala REPL（Read-Evaluate-Print-Loop）（读取－求值－打印－循环）的所有功能。Spark shell虽然强大，但多数时候你还是会运行编译后的代码，而不是使用REPL环境。

2.3 交互式地从S3加载数据

本节开始Spark shell上的第二个实战。Spark支持从S3加载数据，S3是Amazon EMR的一部分，它提供了一些Wikipedia的浏览统计数据，这些浏览数据的格式便于Spark测试。为了访问数据，首先需要将AWS访问证书设置为shell的参数。对于EC2的注册指令和建立shell的参数，请参照第1章中的“在EC2上用脚本运行Spark”这一小节（访问S3，还需要额外的密钥`fs.s3n.awsAccessKeyId/awsSecretAccessKey`或者使用这种格式`s3n://user:pw@`）。以上步骤完成后，开始加载S3上的数据，然后查看一下数据的第一行：

```
scala> val file = sc.textFile("s3n://bigdatademo/sample/wiki/")
13/04/21 21:26:14 INFO storage.MemoryStore: ensureFreeSpace(37539)
called with curMem=37531, maxMem=339585269
13/04/21 21:26:14 INFO storage.MemoryStore: Block broadcast_1 stored
as values to memory (estimated size 36.7 KB, free 323.8 MB)
file: spark.RDD[String] = MappedRDD[3] at textFile at <console>:12

scala> file.take(1)
13/04/21 21:26:17 INFO mapred.FileInputFormat: Total input paths to
process : 1
...
13/04/21 21:26:17 INFO spark.SparkContext: Job finished: take at
<console>:15, took 0.533611079 s
res1: Array[String] = Array(aa.b Pecial:Listusers/sysop 1 4695)
```

如果不将 AWS 访问证书设置为 shell 的参数，访问方式就需要这样写：s3n://<AWS ACCESS ID>:<AWS SECRET>@bucket/path。查看一下第一行的数据十分重要，因为除非我们强制使 Spark 去物化数据，否则数据并不会真正被加载。Amazon 提供了一个小的数据集给开发者使用。数据是从稍大的 http://aws.amazon.com/datasets/4182 提供的数据集上拉取的。当以交互式模式开发的时候，通常想得到快速的结果反馈，因此这种实战非常有用。但如果抽样数据太大，执行时间太长，还是通过嵌入在 Spark shell 中的 sample 函数削减 RDD 的大小比较好。

```
scala> val seed  = (100*math.random).toInt
seed: Int = 8
scala> file.sample(false,1/10.,seed)
res10: spark.RDD[String] = SampledRDD[4] at sample at <console>:17

//If you wanted to rerun on the sampled data later, you could write it
back to S3
scala> res10.saveAsTextFile("s3n://mysparkbucket/test")
13/04/21 22:46:18 INFO spark.PairRDDFunctions: Saving as hadoop file
of type (NullWritable, Text)
....
13/04/21 22:47:46 INFO spark.SparkContext: Job finished:
saveAsTextFile at <console>:19, took 87.462236222 s
```

完成数据加载之后，我们开始在样本数据中寻找最受欢迎的文章。首先对每行数据进行解析，将其拆分为 < 文章名，计数 > 键值对。其次，由于同一个文章名可以出现多次，因此需要按照文章名对数据进行一次规约，将同一个文章的多个计数进行累加。最后，规约产生的键值对的键和值将进行一次交换，对交换后的数据执行排序操作后，我们就获得了访问量最多的文章。具体过程如下：

```
scala> val parsed = file.sample(false,1/10.,seed).map(x => x.split("
")).map(x => (x(1), x(2).toInt))
parsed: spark.RDD[(java.lang.String, Int)] = MappedRDD[5] at map at
<console>:16

scala> val reduced = parsed.reduceByKey(_+_)
13/04/21 23:21:49 WARN util.NativeCodeLoader: Unable to load native-
hadoop library for your platform... using builtin-java classes where
applicable
13/04/21 23:21:49 WARN snappy.LoadSnappy: Snappy native library not
loaded
13/04/21 23:21:50 INFO mapred.FileInputFormat: Total input paths to
process : 1
reduced: spark.RDD[(java.lang.String, Int)] = MapPartitionsRDD[8] at
reduceByKey at <console>:18

scala> val countThenTitle = reduced.map(x => (x._2, x._1))
countThenTitle: spark.RDD[(Int, java.lang.String)] = MappedRDD[9] at
map at <console>:20

scala> countThenTitle.sortByKey(false).take(10)
13/04/21 23:22:08 INFO spark.SparkContext: Starting job: take at
<console>:23
....
13/04/21 23:23:15 INFO spark.SparkContext: Job finished: take at
<console>:23, took 66.815676564 s
res1: Array[(Int, java.lang.String)] = Array((213652,Main_Page),
(14851,Special:Search), (9528,Special:Export/Can_You_Hear_Me),
(6454,Wikipedia:Hauptseite), (4189,Special:Watchlist), (3520,%E7
%89%B9%E5%88%A5:%E3%81%8A%E3%81%BE%E3%81%8B%E3%81%9B%E8%A1%A8%E7
%A4%BA), (2857,Special:AutoLogin), (2416,P%C3%A1gina_principal),
(1990,Survivor_(TV_series)), (1953,Asperger_syndrome))
```

此外也可以利用 Python 和 Spark 交互，通过运行 `./pySpark` 即可。

2.4 小结

本章涵盖了怎样使用 Spark shell 并利用它加载数据，也引导你做了一个简单的机器学习实验。现在你已经了解 Spark 的交互式终端是如何工作的，下章将会讲解怎样在更传统的环境下建立 Spark 作业。

第 3 章
构建并运行 Spark 应用

用 Spark shell 运行 Spark 的交互式模式在代码持久化方面有所局限，而且 Spark shell 不支持 Java 语言。构建一个 Spark 作业比构建一个普通的应用更需要技巧，因为所有要依赖的 jar 包都必须被拷贝到集群中的每台机器上。本章将涵盖用使用 Maven 或者 sbt 构建一个 Java 和 Scala 实现的 Spark 作业，以及用 non-maven-aware 构建系统来构建 Spark 作业。

3.1 用 sbt 构建 Spark 作业

sbt(simple build tool) 是一个流行的 Scala 构建工具，它同时支持 Scala 项目和 Java 项目的构建。用 sbt 构建 Spark 项目是非常好的选择，因为 Spark 本身就是使用 sbt 构建的。使用它，依赖的添加将变得非常容易（这点对 Spark 尤其重要），同时它也支持把所有的东西打成一个可部署的 jar 包。当前，为了简化安装过程，利用 sbt 构建项目时通常都是利用 shell 脚本来引导项目自带的特定版本的 sbt，使构建变得简单。

不妨练下手，拿一个已经可以运行的 Spark 作业，按照以下步骤为它创建一个构建文件。在 Spark 的根目录中，将实例 `GroupByTest` 拷贝到一个新的目录中，如下：

```
mkdir -p example-scala-build/src/main/scala/spark/examples/
cp -af sbt example-scala-build/
cp examples/src/main/scala/spark/examples/GroupByTest.scala example-
scala-build/src/main/scala/spark/examples/
```

由于我们的 jar 包要上传至所有节点，所有的依赖包也必须一并上传。你可以挨个儿上传一大把 jar 包，也可以使用一个方便的 sbt 插件 `sbt-assembly` 将所有依赖集合进一个 jar 包中。如果你的项目不涉及错综复杂的依赖关系，那就不必动用 `assembly`

插件了。这种情况下，就必须要在 Spark 的根目录下运行 sbt/sbt assembly，然后把生成的 jar 包 core/target/spark-core-assembly-0.7.0.jar 加到 classpath 环境变量中。sbt-assembly 包是一个非常好的工具，有了它就不用手工管理一大堆 jar 文件了。将以下代码加入构建文件 project/plugins.sbt 中，即可启用 assembly 插件。

```
resolvers += Resolver.url("artifactory",
url("http://scalasbt.artifactoryonline.com/scalasbt/
sbt-plugin-releases"))(Resolver.ivyStylePatterns)

resolvers += "Typesafe Repository" at
"http://repo.typesafe.com/typesafe/releases/"

resolvers += "Spray Repository" at "http://repo.spray.cc/"
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.8.7")
```

sbt 利用 resolver 来定位包的位置，你可以把它们当作额外的 APT(Advanced Package Tool) PPA(Personal Package Archive) 源，只不过其作用范围仅限于待构建的包。在浏览器中打开 resolver 中列出的 URL，大部分情况下你都会看到一堆目录，从中可以看见该 resolver 提供了哪些包。上述 resolver 中的 URL 都指向网络位置，但是也有些指向本地路径，这些 resolver 一般用于简化开发过程。顾名思义，addSbtPlugin 指令的意思是从 com.eed3si9n 引入版本号为 0.8.7 的 sbt-assembly 包，同时也隐式地指定了 Scala 的版本和 sbt 的版本。最后记得执行 sbt reload clean update 安装新插件。

下面是其中一个样例作业 GroupByTest.scala 的构建文件，将以下代码加到 ./build.sbt 中：

```
//Next two lines only needed if you decide to use the assembly plugin
import AssemblyKeys._
assemblySettings

scalaVersion := "2.9.2"
```

```
name := "groupbytest"

libraryDependencies ++= Seq(
  "org.spark-project" % "spark-core_2.9.2" % "0.7.0"
)

resolvers ++= Seq(
  "JBoss Repository" at
  "http://repository.jboss.org/nexus/content/
   repositories/releases/",
  "Spray Repository" at "http://repo.spray.cc/",
  "Cloudera Repository" at
  "https://repository.cloudera.com/artifactory/cloudera-repos/",
  "Akka Repository" at "http://repo.akka.io/releases/",
  "Twitter4J Repository" at "http://twitter4j.org/maven2/"
)
//Only include if using assembly
mergeStrategy in assembly <<= (mergeStrategy in assembly) {
  (old) =>
  {
    case PathList("javax", "servlet", xs @ _*) => MergeStrategy.first
    case PathList("org", "apache", xs @ _*) => MergeStrategy.first
    case "about.html"  => MergeStrategy.rename
    case x => old(x)
  }
}
```

如你所见，build.sbt 构建文件的格式和 plugins.sbt 的格式是一样的。关于这个构建文件的独特之处，值得一提的是，就像在 plugins.sbt 文件中一样，也必须加一些 resolvers 确保 sbt 能够找到所有的依赖。注意必须指定 spark-core 的版本，如 "org.spark-project" % "spark-core_2.9.2" % "0.7.0" 而不是简单的 "org.spark-project" %% "spark-core" % "0.7.0"。可能的话，你应该尽量使用 %% 格式，这样能够自动指定 Scala 版本。上述构建文件中的另一要点是用到了 mergeStrategy。由于多个依赖的 jar 包可能会包含相同的文件，当在将所有文件合并到单个 jar 包中的时候，需要告诉 assembly 插件怎样处理这种情况。除了 mergeStrategy 和手工指定 Spark 对应的 Scala 版本之外，上述构建文件就没什么好解释的了。

如果你在 master 的 JDK 版本和 worker 上的 JDK 不一致，你可以通过在构建文件中加入下面的语句切换到目标 JDK，以便它们保持一致：

```
javacOptions ++= Seq("-target", "1.6")
```

至此构建文件就被定义好了，然后构建 GroupByTest 这个 Spark 作业：

```
sbt/sbt clean compile package
```

这将产生 target/scala-2.9.2/groupbytest_2.9.2-0.1-SNAPSHOT.jar。

在 Spark 根目录中运行 sbt/sbt assembly，检查是否生成 assembly 的 jar 包并将它加入到 classpath 环境变量中。example 程序需要指定 Spark 的 SPARK_HOME 的路径和 SPARK_EXAMPLES_JAR 的路径。也需要用 -cp 指定针对本地 Scala 版本构建得到的 jar 包的 classpath 环境变量。运行下面的例子：

```
SPARK_HOME="../"  SPARK_EXAMPLES_JAR="./target/scala-2.9.2/
groupbytest-assembly-0.1-SNAPSHOT.jar"  scala -cp/users/
sparkuser/spark-0.7.0/example-scala-build/target/scala-2.9.2/
groupbytest_2.9.2-0.1-SNAPSHOT.jar:/users/sparkuser/spark-0.7.0/
core/target/
spark-core-assembly-0.7.0.jar spark.examples.GroupByTest local[1]
```

如果将所有的依赖用 assembly 插件都打到一个简单的 jar 包中，需要这样调用：

```
sbt/sbt assembly
```

这样能够产生一个 assembly 的 jar 包 target/scala-2.9.2/groupbytest-assembly-0.1-SNAPSHOT.jar，然后能够以不指定 spark-core-assembly 的方式运行一遍：

```
SPARK_HOME="../" \ SPARK_EXAMPLES_JAR="./target/scala-2.9.2/groupbytest-
assembly-0.1-SNAPSHOT.jar" \
 scala -cp /users/sparkuser/spark-0.7.0/example-scala-build/target/
scala-2.9.2/groupbytest-assembly-0.1-SNAPSHOT.jar spark.examples.
GroupByTest local[1]
```

随着版本的演进，你今后可能还会碰到各种 jar 包合并相关的问题；真到那时候，在网上随便搜索一下，应该就能找到靠谱的解决办法，本书就不操心那些尚未出现的问题了。一般来说，`MergeStrategy.first` 就够用了。

跑通前面的代码并不意味着万事大吉。sbt 会查询本地缓存，依赖解析过程中定位到的依赖包可能是由其他项目写入缓存的，因此在一台机器上能构建成功并不意味着在别的机器上也能成功。保险起见，开始构建之前应该删除本地的 `ivy` 缓存并执行 `sbt clean`。如果发现某些依赖包下载失败，请检查 Spark 的 resolver 列表，并在 `build.sbt` 中自行添加缺失条目。

一些有用的链接如下：

- `http://www.scala-sbt.org/`
- `https://github.com/sbt/sbt-assembly`
- `http://spark-project.org/docs/latest/scala-programming-guide.html`

3.2 用 Maven 构建 Spark 作业

Maven 是一个开源的 Apache 项目，它能构建 Java 版本和 Scala 版本的 Spark 作业。和 sbt 一样，可以通过 Maven 包含 Spark 所有的依赖关系简化构建进程，Maven 能够用插件将 Spark 和所有的依赖捆绑为一个简单的 jar 包，和 `sbt/sbt assembly`

的功能是一样的。

为了描述用 Maven 构建 Spark 作业的过程，本小节以一个 Java 样例作业为例子，因为 Maven 更常用来构建 Java 任务。第一步，用一个可以运行的 Spark 作业，将创建构建文件的步骤过一遍。先把 GroupByTest 例子复制到新的目录中，生成 maven 的模板如下：

```
mkdir example-java-build/; cd example-java-build
mvn archetype:generate \
   -DarchetypeGroupId=org.apache.maven.archetypes \
   -DgroupId=spark.examples \
   -DartifactId=JavaWordCount \
   -Dfilter=org.apache.maven.archetypes:maven-archetype-quickstart
cp ../examples/src/main/java/spark/examples/JavaWordCount.java
JavaWordCount/src/main/java/spark/examples/JavaWordCount.java
```

第二步：更新 Maven 的 pom.xml，让它包含正在使用的 Spark 的版本信息。由于运行的这个例子文件需要 JDK1.5，将需要更新 Maven 所要使用的 Java 版本；在写本书的时候，它默认是 1.3 版本，在 <projects> 标签之间，需要添加下面代码：

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.spark-project</groupId>
    <artifactId>spark-core_2.9.2</artifactId>
    <version>0.7.0</version>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.5</source>
```

```
        <target>1.5</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

现在用 Maven 的 package 功能构建 jar 包，运行下面的命令：

```
SPARK_HOME="../" SPARK_EXAMPLES_JAR="./target/JavaWordCount-1.0-
SNAPSHOT.jar" java -cp ./target/JavaWordCount-1.0-SNAPSHOT.jar:../../
core/target/spark-core-assembly-0.7.0.jar spark.examples.JavaWordCount
local[1] ../../README
```

就像 sbt 一样，可以用一个插件集合所有的依赖进一个 jar 包中，在 `<plugins>` 标签中，添加以下代码：

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>1.7</version>
  <configuration>
    <!-- This transform is used so that merging of akka configuration
files works -->
    <transformers>
      <transformer implementation=
  "org.apache.maven.plugins.shade.resource
  .ApacheLicenseResourceTransformer">
      </transformer>
      <transformer implementation=
  "org.apache.maven.plugins.shade
  .resource.AppendingTransformer">
        <resource>reference.conf</resource>
      </transformer>
    </transformers>
  </configuration>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

然后执行 `mvn assembly`，产生 jar 包能够直接拿来执行上节的代码，同时可以省略对 Spark 的 assembly jar 包的指定。

一些有用的链接如下：

- http://maven.apache.org/guides/getting-started/
- http://maven.apache.org/plugins/maven-compiler-plugin/examples/set-compiler-source-and-target.html
- http://maven.apache.org/plugins/maven-dependency-plugin/

3.3 用其他工具构建 Spark 作业

如果 sbt 或者 Maven 对你都不是很合适，你可以选择自己习惯的构建系统。还好，Spark 可以将 Spark 依赖的所有 jar 包集合成一个扁平的 jar 包，方便你利用自己的构建系统进行构建。很简单，在 Spark 的根目录中运行一下 `sbt/sbt assembly`，将生成的 jar 包从 `core/target/spark-core-assembly-0.7.0.jar` 拷到你的构建依赖中。

不管你的构建系统是什么，你时不时都会需要使用 Spark 的某个补丁版本。在这种情况下，可以将 Spark 的库文件部署在本地。我建议你给自用的补丁版本单独分配一个版本号，以确保 `sbt/maven` 不会找错文件。编辑 `project/SparkBuild.scala`，调整“`version:=`”这部分代码即可修改版本号。如果用的是 sbt，请务必运行 `sbt/sbt update`。对于其他的构建系统，只要确保构建时用的是新的 assembly jar 包就可以了。

3.4 小结

现在你已经可以用 Maven、sbt 或者其他的构建系统构建 Spark 作业，之后可以钻进 Spark 系统去干点更有意思的事情了，下章将会介绍怎样创建 SparkContext。

第 4 章
创建 SparkContext

本章介绍怎样创建一个适合特定集群的 SparkContext 实例。SparkContext 实例可以与一个 Spark 集群连接，也提供与 Spark 系统交互的入口。创建一个 SparkContext 实例的目的是能够和 Spark 交互，并提交作业。在第 2 章中，我们通过 Spark shell 和 Spark 交互并通过它创建了 SparkContext 实例，现在能够创建 RDD、广播变量、累加器，等等，利用你拥有的数据做一些有趣的事情。Spark shell 也作为一个通过 SparkContext 与 Spark 交互的例子出现在 ./repl/src/main/scala/spark/repl/SparkILoop.scala。

下面的一小段代码将用 MASTER 环境变量（本地模式下不需设置）创建名为”Spark shell”的 SparkContext 实例，也没有指定任何依赖。这是由于 Spark shell 已经被内置到 Spark 中，因此没有任何需要分布式传输到各个 worker 节点上的 jar 包。

```
def createSparkContext(): SparkContext = {
  val master = this.master match {
    case Some(m) => m
    case None => {
     val prop = System.getenv("MASTER")
     if (prop != null) prop else "local"
    }
  }
  sparkContext = new SparkContext(master, "Spark shell")
  sparkContext
  }
```

为了使客户端能够建立与 Spark 集群的连接，SparkContext 对象需要如下基本信息：

- master: master URL 可以是如下格式之一：
  - local[n]：本地模式。
  - spark://[sparkip]：指向一个 Spark 集群。
  - mesos://：如果 Spark 部署在一个 mesos 集群上则指向一个 mesos 路径。

- application name：可读的应用名字。
- sparkHome：Spark 的根目录路径。
- jars：提交作业所要依赖的 jar 包路径。

4.1 Scala

在 Scala 程序中，可以通过以下语句创建 SparkContext 实例：

```
val spar kContext = new SparkContext(master_path, "application
name", ["optional spark home path"],["optional list of jars"])
```

虽然所有的参数值都可以在代码中硬编码，但更好的方式是从环境变量中读取对应的值，并且提供默认设置。后者允许你在环境变化时，不需要重新编译代码就能运行，因此灵活度较大。用 local 作为默认值便于在本地测试环境中加载应用。选择合适的默认值，能较大限度地避免再次指定它。如下例：

```
import spark.sparkContext
import spark.sparkContext._
import scala.util.Properties

val master = Properties.envOrElse("MASTER","local")
val sparkHome = Properties.get("SPARK_HOME")
val myJars = Seq(System.get("JARS")
val sparkContext = new SparkContext(master, "my app", sparkHome,
myJars)
```

4.2 Java

使用 java 创建一个 SparkContext 实例，请参照以下代码：

```
import spark.api.java.JavaSparkContext;

JavaSparkContext ctx = new JavaSparkContext("master_url",
"application name", ["path_to_spark_home", "path_to_jars"]);
```

为了确保以上代码能够正确运行，代码中对应的参数需要修改为针对目标 Spark 环境的正确参数值。一旦任何参数值需要修

改，都需要更新代码。因此更明智的做法是采用类似 Scala 示例代码中的方式，为每个参数设置默认值并允许它们被覆盖。以下代码展示了如何使用环境变量。

```
String master = System.getEnv("MASTER");
if (master == null) {
  master = "local";
}
String sparkHome = System.getEnv("SPARK_HOME");
if (sparkHome == null) {
  sparkHome = "./";
}
String jars = System.getEnv("JARS");
JavaSparkContext ctx = new JavaSparkContext(System.getenv("MASTER"),
"my Java app",
System.getenv("SPARK_HOME"), System.getenv("JARS"));
```

4.3 Java 和 Scala 共享的 API

一旦创建 SparkContext 实例，它将提供 Spark 的主入口。下章将介绍怎样使用创建的 SparkContext 实例加载和保存数据，如何使用它去提交更多的 Spark 作业，添加和删除依赖。表 4-1 给出了一些 SparkContext 提供的非数据驱动的方法。

表 4-1 SparkContext 提供的方法

方 法	用 途
addJar(path)	向 SparkContext 实例添加在这个实例上将要运行的作业所依赖的 jar 包
addFile(path)	将 path 指向的文件下载到集群中的每个节点上
stop()	关闭 SparkContext 连接
clearFiles()	删除由 addFiles 所添加的文件，新加入的节点不会再获得这些文件
clearJars()	删除由 addJars 所添加的 jar 包，新提交的作业不会再依赖这些 jar 包

4.4 Python

Python 版的 SparkContext 实例有别于 Scala 版本和 Java 版本，它不使用 jar 包来指定依赖，然而项目仍可能包含一些依赖关系，这时可以将 pyFiles 设置为需要的 zip 和 py 文件（或者如果你没有任何文件分发，就让它为空）。

运行下面的代码创建一个 SparkContext 实例：

```
from pyspark import SparkContext

sc = SparkContext("master","my python app", sparkHome="sparkhome",
pyFiles="placeholderdeps.zip")
```

至此，我们已经学习了创建 SparkContext 实例连接 Spark 集群，随后将介绍如何加载数据到 Spark 中。

4.5 链接和参考

下面是一些有用的参考链接：

- http://spark-project.org/docs/latest/quick-start.html
- http://www-stat.stanford.edu/~tibs/ElemStatLearn/data.html
- https://github.com/mesos/spark/blob/master/repl/src/main/scala/spark/repl/SparkILoop.scala
- http://spark-project.org/docs/0.7.0/api/pyspark/pyspark.context.SparkContext-class.html
- http://spark-project.org/docs/0.7.0/api/core/spark/SparkContext.html
- http://spark-project.org/docs/0.7.0/api/core/spark/api/java/JavaSparkContext.html
- http://www.scala-lang.org/api/current/index.html#scala.util.Properties$

4.6 小结

本章介绍了怎样利用 SparkContext 连接到 Spark 集群，下章将介绍如何使用 SparkContext 从不同的数据源将数据加载到 Spark 中。

第 5 章
加载与保存数据

通过前几章的学习，你已经体验了 Spark shell 的使用方法，熟悉怎样建立到 Spark 集群的连接，已经构建作业用于部署。为使这些任务更具有实际用途，本章将介绍 Spark 中数据的加载与保存。RDD 是 Spark 中数据表示的主要单元，它使得对数据执行并行操作变得简便。另一些形式的数据，比如累加器和广播变量，有它们自己的表示形式。Spark 能够从很多“源”加载和保存 RDD。

5.1 RDD

RDD 能够从 Hadoop 相关的源创建，Scala、Java 或者 Python 的集合也能作为 RDD 的源，从简单的集合创建 RDD 便于测试。

在深入研究 Spark 所支持的数据源之前，首先了解一下 RDD 是什么和 RDD 不是什么。理解这点很关键：即使一个 RDD 被指定，但是它在内存中暂时还不包含具体数据。这代表当打算访问这个 RDD 中数据的时候将不会在内存中命中数据。在 RDD 中创建数据的计算仅当数据被引用时才执行，这意味着可以将多个操作链在一起，不需要担心额外的阻塞操作。这点很重要：在开发应用的时候，你可以编码，编译，然后运行作业，除非将 RDD 里面的数据物化，否则代码是不可能加载数据的。

每当物化 RDD 的时候它将被重新计算。如果打算经常使用一些数据，利用 cache 函数将 RDD 中的数据保存到内存中则能提高性能。

5.2 加载数据到 RDD 中

接下来将尝试从不同的源构造 RDD。如果你决定在 Spark shell 端运行简单例子的话，可以在 RDD 上调用 first 函数来确认数据能够被加载。在第 2 章中，你已经学习怎样从一个文本文件或者从 S3 存储系统加载数据，可见 Spark 支持从不同的源加载不同格式的数据。

直接将 Scala 集合转化为一个 RDD 是一种创建 RDD 的最简单方式。SparkContext 提供一个 parallelize 函数，这个函数以 Scala 集合为参数，然后将它转化为一个 RDD，这个 RDD 中单个元素的数据类型和 Scala 集合中的数据类型相同。

- **Scala**:

```
val dataRDD = sc.parallelize(List(1,2,4))
```

- **Java**:

```
JavaRDD<Integer> dataRDD = sc.parallelize(Arrays.asList(1,2,4));
```

- **Python**:

```
rdd = sc.parallelize([1,2,3])
```

加载文本文件中的数据是加载外部数据最简单的方式，需要数据在每个节点上都存有备份。本地模式无需考虑这个问题，但在分布式模式下，可以用 Spark 的 addFile 函数拷贝数据到集群中每个节点上，假设 SparkContext 实例叫做 sc，可以像下面这样加载文本文件数据（当然首先得创建这个文本文件或者它已经存在）：

- **Scala**:

```
import spark.SparkFiles;
...
sc.addFile("spam.data")
val inFile = sc.textFile(SparkFiles.get("spam.data"))
```

- **Java**:

```
import spark.Sparkfiles;

sc.addFile("spam.data");
JavaRDD<String> lines = sc.textFile(SparkFiles.get("spam.data"));
```

- **Python**:

```
from pyspark.files import SparkFiles

sc.addFile("spam.data")
sc.textFile(SparkFiles.get("spam.data"))
```

结果RDD是一个字符串数据集，文件中的每行是RDD中的一个单独元素。通常情况下，输入文件是CSV或者TSV文件，可以用一个标准的CSV库解析。在第2章中，提到了用`split`函数和`toDouble`函数解析数据，但是对于更加复杂的CSV文件，还是推荐使用CSV解析库。在第3章中，你已经学习了怎样构建Spark作业，因此你可以将`build.sbt`中的`libraryDependencies`改为：

```
libraryDependencies ++= Seq(
    "org.spark-project" % "spark-core_2.9.2" % "0.7.0",
    "net.sf.opencsv" % "opencsv" % "2.0"
)
```

现在可以用到CVS解析器了。本章为了简略而使用`opencvs`演示例子，但是根据具体要解析什么，也许还有更适合你需求的CVS解析器。下面这个例子能够解析输入的CVS文件，然后累加每一行中的值：

```
package pandaspark.examples

import spark.SparkContext
import spark.SparkContext._
import spark.SparkFiles;
import au.com.bytecode.opencsv.CSVReader
import java.io.StringReader

object LoadCsvExample {
  def main(args: Array[String]) {
```

```
    if (args.length != 2) {
      System.err.println("Usage: LoadCsvExample <master>
    <inputfile>")
      System.exit(1)
    }
    val master = args(0)
    val inputFile = args(1)
    val sc = new SparkContext(master, "Load CSV Example",
                System.getenv("SPARK_HOME"),
                Seq(System.getenv("JARS")))
    sc.addFile(inputFile)
    val inFile = sc.textFile(inputFile)
    val splitLines = inFile.map(line => {
      val reader = new CSVReader(new StringReader(line))
      reader.readNext()
    })
    val numericData = splitLines.map(line => line.map(_.toDouble))
    val summedData = numericData.map(row => row.sum)
    println(summedData.collect().mkString(","))
  }

}
```

上面的代码同时也描述了从 Spark 中获得数据（将数据从 Spark 中提取出来）的一种方式：你能够用 `collect` 函数将数据转化为一个 Scala 数组。和 `parallelize` 函数一样，`collect` 函数便于测试。`collect` 函数只有在当数据能够完全放在一个节点的内存中的时候才有用，因此它将使得单点成为瓶颈。

将文本文件加载到 Spark 中固然很简单，但将大量的数据存储成本地磁盘上的文本文件并不是好的策略。Spark 支持从不同的 Hadoop 格式（如 sequuence files，文本文件等）文件加载数据，也支持从所有支持 Hadoop 的存储源（HDFS、S3、HBase 等）中加载数据。如果有需要，可以用快加载工具（如 ImportTsv）加载 CSV 数据到 HBase 或者从 HBase 获得 CSV 数据。本节接下来将介绍另一些加载数据的方法，0.7 版本中，PySpark 不支持这些方法。

Sequence files是包含键值对的平坦型二进制文件，它是一种常用的存储数据供Hadoop使用的方式。加载sequence files到Spark中和加载文本文件类似，但是必须人为地定义所处理的键值对的类型。类型必须继承Hadoop的Writable类，或者可以隐式的转化为这种类型。对于Scala的用户，一些Scala中的数据类型能通过WritableConverter隐式转换为Writable类。在Spark的0.7版本中，已经在WritableConverter标准化的数据类型有Int、Long、Double、Float、Boolean、Array和String。

接下来，让我们看看怎样加载一个包含有String和Integer的sequence file。

- **Scala**:

```
val data = sc.sequenceFile[String, Int](inputFile)
```

- **Java**:

```
JavaPairRDD<Text, IntWritable> dataRDD = sc.sequenceFile(file,
Text.class, IntWritable.class);
JavaPairRDD<String, Integer> cleanData = dataRDD.map(new
PairFunction<Tuple2<Text, IntWritable>, String, Integer>() {
 @Override
public Tuple2<String, Integer> call(Tuple2<Text, IntWritable>
pair) {
return new Tuple2<String, Integer>(pair._1().toString(),
pair._2().get());
}
});
```

注意在前面的例子中，像文本数据的输入，文件不必是传统的文件，它也能够存在S3、HDFS上，等等。同时也要注意对于Java，你不能依赖在类型之间利用隐式转换。

HBase是一个基于Hadoop的数据库，它支持对记录进行随机读写。从HBase中加载数据相比于文本文件和sequence file会

有点不同。对于 HBase，应该以一种不同的方式向 Spark 指定类型信息。由于 HBase 没有作为 Spark 的依赖被包含进去，所以应该像前面处理 opencsv 一样将 `org.apache.hbase" % "hbase" % "0.94.6` 加入到你的构建文件中。

如果你遇到不能解析的依赖，请将 maven 库中 Apache HBase 的版本库 `https://repository.apache.org/content/repositories/releases` 加到你的 resolver 中。

让我们看一下 HBase 数据库在 Spark 中的用法：

- **Scala**:

```
import spark._
import org.apache.hadoop.hbase.{HBaseConfiguration,
HTableDescriptor}
import org.apache.hadoop.hbase.client.HBaseAdmin
import org.apache.hadoop.hbase.mapreduce.TableInputFormat
…
val conf = HBaseConfiguration.create()
conf.set(TableInputFormat.INPUT_TABLE, input_table)
 // Initialize hBase table if necessary
val admin = new HBaseAdmin(conf)
if(!admin.isTableAvailable(input_table)) {
  val tableDesc = new HTableDescriptor(input_table)
  admin.createTable(tableDesc)
}
val hBaseRDD = sc.newAPIHadoopRDD(conf,
                  classOf[TableInputFormat],
                  classOf[org.apache.hadoop.
                  hbase.io.ImmutableBytesWritable],
                  classOf[org.apache.hadoop.
                  hbase.client.Result])
```

- **Java**:

```
import spark.api.java.JavaPairRDD;
import spark.api.java.JavaSparkContext;
import spark.api.java.function.FlatMapFunction;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HTableDescriptor;
```

```
import org.apache.hadoop.hbase.client.HBaseAdmin;
import org.apache.hadoop.hbase.mapreduce.TableInputFormat;
import org.apache.hadoop.hbase.io.ImmutableBytesWritable;
import org.apache.hadoop.hbase.client.Result;
...
JavaSparkContext sc = new JavaSparkContext(args[0], "sequence
load", System.getenv("SPARK_HOME"), System.getenv("JARS"));
Configuration conf = HBaseConfiguration.create();
conf.set(TableInputFormat.INPUT_TABLE, args[1]);
//Initialize HBase table if necessary
HBaseAdmin admin = new HBaseAdmin(conf);
if(!admin.isTableAvailable(args[1])) {
    HTableDescriptor tableDesc = new
       HTableDescriptor(args[1]);
    admin.createTable(tableDesc);
}
JavaPairRDD<ImmutableBytesWritable, Result> hBaseRDD
= sc.newAPIHadoopRDD( conf, TableInputFormat.class,
ImmutableBytesWritable.class, Result.class);
```

加载 HBase 中数据的方法可推广到加载所有类型的 Hadoop 数据，如果现在 Spark 中还不存在加载某种类型数据的方法，只需要简单地创建一个指定怎样加载数据的配置，然后将它传到 `newAPIHadoopRDD` 方法中。对于文本文件和 sequence files 的相应加载方法是存在的，类似于 Sequence File API 的 Hadoop 文件加载方法也是存在的。

5.3 保存数据

分布式计算作业很有趣，当合理存储它的结果数据时会更有价值。当 `SparkContext` 中存在有一个加载 RDD 数据的方法时，保存 RDD 数据的方法也就被定义到了相应的 RDD 类中。用 Scala，隐式转化可以将 RDD 中的数据转化为合适的类型以便存储为 sequence file，如果用 Java，应该用显示转化。

下面是保存 RDD 中数据的不同方式：

- **Scala**:

```
rddOfStrings.saveAsTextFile("out.txt")
keyValueRdd.saveAsSequenceFile("sequenceOut")
```

- **Java**:

```
rddOfStrings.saveAsTextFile("out.txt")
keyValueRdd.saveAsSequenceFile("sequenceOut")
```

- **Python**:

```
rddOfStrings.saveAsTextFile("out.txt")
```

5.4 连接和参考

- http://spark-project.org/docs/latest/scala-programming-guide.html#hadoop-datasets
- http://opencsv.sourceforge.net/
- http://commons.apache.org/proper/commons-csv/
- http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/SequenceFileInputFormat.html
- http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/InputFormat.html
- http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/
- http://spark-project.org/docs/latest/api/pyspark/index.html
- http://wiki.apache.org/hadoop/SequenceFile
- http://hbase.apache.org/book/quickstart.html
- http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/mapreduce/TableInputFormat.html
- http://spark-project.org/docs/latest/api/core/index.html#spark.api.java.JavaPairRDD

5.5 小结

在本章中，我们了解了怎样从不同的源加载数据的各种方式，也学习了从输入的文本文件中解析数据。了解了怎样将数据加载到 RDD 之后，下章将讨论对数据做不同的处理。

第 6 章
操作 RDD

前面几章介绍了让Spark运行起来的必要基础知识，以及多种加载和储存数据的方法，那么现在要进行综合运用了：操作数据。在几种不同的编程语言下操作RDD的API基本相似，但亦不完全一致，所以，每一种编程语言下的API都会用独立的章节来说明，读者只需选择自己感兴趣的语言相关的部分阅读。需要说明的是，Python API的实现与Scala/Java API相比在一些特征上不对等，但是截至0.7版，Python支持大部分基本功能，并且已经有计划在未来版本中使其与其他语言更统一。

6.1 用Scala和Java操作RDD

在Scala中操作RDD非常简单，特别是如果你熟悉Scala集合库(collection library)的话。Spark的RDD对象接口兼容很多标准的函数式列表方法（functional list functions），这使得把现有的单机Scala代码改写成支持分布式的代码要比改Java或Python代码要容易得多。需要注意的是，改写之后不能再假设这些方法会在同一台机器上执行。

用Java操作RDD也比较简单，只偶尔比Scala麻烦一点，这是因为Java不支持隐式类型转换，必须显式地指定类型。Java处理RDD返回值很自然，而Scala则需要用`Tuple2`类来处理键值对（key-value pairs）类型的返回值。

`map`和`reduce`是MapReduce的两个标志性操作。前面的章节中已经介绍了`map`函数，它的工作原理是把一个函数作用在每一个从RDD读入的元素上，每次函数输出是一个新元素。例如把一个RDD中的每个元素上加1以形成一个新的RDD，只需要用一句Scala代码：`rdd.map(xv=>vx+1)`，或者用下面

的 Java 代码：

```
rdd.map(new Function<Integer, Integer>() {
  public Integer call(Integer x) { return x+1;}
});
```

有一点要特别注意：map 函数和其他所有 Spark 函数并不改变输入 RDD 的值，而是返回一个新元素组成的新 RDD。reduce 函数的参数是一个函数，能够作用于一个数据序列，每次携带一对元素（先前的结果，以及序列中的下一个元素）进行计算，最终把序列中的所有数据聚合成一个元素。该函数必须满足结合律和交换律，即 f(a,b)==f(b,a)，且 f(a, f(b,c)) = f(f(a,b),c)。例如，将所有元素累加，使用 rdd.reduce(x,y=>x+y) 或者：

```
rdd.reduce(new Function2<Integer, Integer, Integer>(){
public Integer call(Integer x, Integer y) { return x+y;} }.
```

flatMap 是一个很有用的函数，它能让函数返回一个指向的内容被“扁平化”的迭代器。一个简单的例子是解析一组数据，其中某一部分可能无法解析，这时候用 flatMap 来输出，对于无法解析的部分会输出一个空列表。另外，还有一个与 reduce 函数对应的 reduceByKey 函数，它作用于一组 RDD，把它们作为键值对来生成一个新 RDD。不像 Scala 的 map 函数，RDD 的 map 函数会在多台机器上分布式运行，所以不能依赖数据的共享状态。

在继续学习更多的 RDD 操作函数之前，需要了解一些共享状态相关知识。在前面的例子里，把每一个数都加 1，并不需要共享任何数字之间的状态。尽管如此，即使是很简单的任务，比如分布式解析 CVS 文件，如果能有一些共享的计数器来记录被拒绝的记录数会特别方便。Spark 支持广播变量（broadcast）和累加

器变量（accumulator）。通过调用sc.broadcast(value)来创建一个共享常量，Spark会在后台自动把这个常量广播到所有节点上，并且保证只会广播一次。

另一个共享状态的方法是用累加器变量，使用sc.accumulator (initialvalue)来创建该变量，它会返回一个分布式上下文中的对象，可以通过这个对象进行累加操作，然后调用.value()来获取累加值。accumulableCollection函数能创建一个可被分布式添加的集合。尽管如此，在使用这些函数之前需要确认一下，是否真的有必要使用？能否通过自定义map函数来获取这些值呢？如果预定义的几种累加器不能满足需求，可以通过实现accumulable接口来自定义累加器。breadcast可以由所有的工作节点读取，accumulator可以由所有工作节点写，但只能由控制节点读取。

如果用Scala写与Java Spark进程交互的代码（比如测试场景），用Java Spark context中的intAccumulator和类似方法比较好，其他类型的accumulator可能不太好用。

如果发现accumulator变量的变化不符合预期，不要担心，可能是Spark的延迟更新机制导致的。Spark在计算结果输出之前不会真正去做map、reduce或其他针对RDD的计算。

还是看看前面解析CSV文件的例子，现在来把它实现得更健壮一些。在前面的例子里假设输入都是有效的，一旦遇到任何错误就立即终止整个处理流程。对于部分场景来说这么处理是正确的，但是，当处理一些第三方数据的时候，有必要对无效数据有

一定的容忍度。另一方面，又并不希望直接把无效数据当成正确的值，这么做的话会导致最终结果毫无意义。最好的解决方案是添加一个无效数据计数器：

```
package spark.examples

import spark.SparkContext
import spark.SparkContext._
import spark.SparkFiles;

import au.com.bytecode.opencsv.CSVReader

import java.io.StringReader

object LoadCsvWithCountersExample {
  def main(args: Array[String]) {
    if (args.length != 2) {
      System.err.println("Usage: LoadCsvExample <master>
                          <inputfile>")
      System.exit(1)
    }
    val master = args(0)
    val inputFile = args(1)
    val sc = new SparkContext(master, "Load CSV With Counters
             Example",
  System.getenv("SPARK_HOME"),
  Seq(System.getenv("JARS")))
    val invalidLineCounter = sc.accumulator(0)
    val invalidNumericLineCounter = sc.accumulator(0)
    sc.addFile(inputFile)
    val inFile = sc.textFile(inputFile)
    val splitLines = inFile.flatMap(line => {
      try {
        val reader = new CSVReader(new StringReader(line))
        Some(reader.readNext())
      }
      catch {
        case _ => {
          invalidLineCounter += 1
          None
        }
      }
    }
    )
    val numericData = splitLines.flatMap(line => {
      try {
        Some(line.map(_.toDouble))
```

```
      }
      catch {
        case _ => {
          invalidNumericLineCounter += 1
          None
        }
      }
    }
    )
    val summedData = numericData.map(row => row.sum)
    println(summedData.collect().mkString(","))
    println("Errors:
            "+invalidLineCounter+","+invalidNumericLineCounter)
  }

}
```

Java实现：

```
import spark.Accumulator;
import spark.api.java.JavaRDD;
import spark.api.java.JavaPairRDD;
import spark.api.java.JavaSparkContext;
import spark.api.java.function.FlatMapFunction;

import au.com.bytecode.opencsv.CSVReader;
import java.io.StringReader;
import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;

public class JavaLoadCsvCounters {
  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: JavaLoadCsvCounters <master>
                          <inputfile>");
      System.exit(1);
    }
    String master = args[0];
    String inputFile = args[1];
    JavaSparkContext sc = new JavaSparkContext(master,
                          "java load csv with counters",
                          System.getenv("SPARK_HOME"),
                          System.getenv("JARS"));
    final Accumulator<Integer> errors = sc.accumulator(0);
    JavaRDD<String> inFile = sc.textFile(inputFile);
    JavaRDD<Integer[] > splitLines = inFile.flatMap(
                                     new FlatMapFunction<String,
                                     Integer[]> (){
      public Iterable<Integer[]> call(String line) {
```

```
        ArrayList<Integer[]> result = new ArrayList<Integer[]>();
        try {
               CSVReader reader = new CSVReader(
                                  new StringReader(line));
               String[] parsedLine = reader.readNext();
               Integer[] intLine = new Integer[parsedLine.length];
               for (int i = 0; i < parsedLine.length; i++) {
                 intLine[i] = Integer.parseInt(parsedLine[i]);
               }
               result.add(intLine);
        }
        catch (Exception e) {
          errors.add(1);
        }
        return result;
        }
  }
  );
    System.out.println("Loaded data "+splitLines.collect());
    System.out.println("Error count "+errors.value());
  }
}
```

上面的代码向读者展示了如何使用 flatMap 方法。一般 flatMap 用于把 map 返回的序列（sequence）展平，而且，flatMap 还能够自动把结果中的空值 (None) 过滤掉。

统计概要在处理大规模数据的时候特别有用。在前面的例子中，数据以 Double 类型加载是为了符合 Spark 的 RDD 数据统计功能接口要求。在 Java 中，必须显式地使用 JavaDoubleRDD 类型来匹配。在下面的例子里，应该用 DoubleFunction<Interger[] >，而不能用 Function<Integer [], Double>，因为后者的结果返回类型不是 JavaDoubleRDD。Scala 用户不用关心这些细节，因为 Scala 支持自动的隐式转换。计算平均值和方差，或者边统计边计算，可以这么做：在前面函数的末尾加上下面这段代码，其中 println(summedData.stats()) 这一句负责输出统计概要。

如果用Java实现这个功能，代码如下：

```
JavaDoubleRDD summedData = splitLines.map(
                           new DoubleFunction<Integer[]>()
{
  public Double call(Integer[] in) {
    Double ret = 0.;
    for (int i = 0; i < in.length; i++) {
      ret += in[i];
    }
    return ret;
  }
});
System.out.println(summedData.stats());
```

当处理键值对的数据时，把具有相同key的元素聚合到一起处理起来会比较方便（例如，当key表示用户的时候）。groupByKey函数提供了一个简单的接口按照key来聚合数据。groupByKey是combineByKey的一种特例。PairRDD类里面有好几个函数都是基于combineByKey函数实现。在使用groupByKey或者其他从combineByKey衍生出来的函数进行数据处理的时候，应该仔细考查一下该函数是否是最合适的。在起步阶段最常用的做法是使用groupByKey，然后对结果求和，Scala代码如下：

```
groupByKey().map((x, y) => (x, y.sum))
```

Java代码如下：

```
pairData.groupByKey().mapValues(new Function<List<Integer>,
  Integer >(){
  public Integer call(List<Integer> x){
    Integer sum = 0;
    for (Integer i : x) {
      sum += i;
    }
    return sum;
  }
});
```

如果用 reduceByKey 代码可以简化成：reduceByKey((x, y) => x+y)

对应的 Java 代码为：

```
pairData.reduceByKey(new Function2<Integer, Integer, Integer>() {
  public Integer call (Integer a, Integer b){
    return a+b;
  }
});
```

foldByKey(zeroValue) (function) 方法与传统的 fold 操作比较相似。传统的 fold 方法作用于一个 list 的过程如下：首先传入的 function 函数会作用于传入的初始值 zeroValue 和 list 中第一个元素，得到一个结果后，将 function 函数继续作用于结果与 list 中的下一个元素，得到下一个结果。这需要顺序遍历整个 list。foldByKey 的行为略有不同，在本节末尾的表格里总结了一组 PairRDD 的函数供读者参考。部分 PairRDD 的功能只在 0.7.2 版本中才添加到 Java API。

有时候只希望更新键值对中的值部分，PairRDD 就是一个例子。对于 Scala 开发者来说，如果希望实现传统的 fold 行为，可以用先用 groupByKey，然后在结果集上用 fold 函数。这里有一个例子，只关心改变 RDD 的 value 部分，不关心 key：

```
rdd.groupByKey().mapValues(x => {x.fold(0)((a,b) => a+b)})
```

通常一份完整的数据必须由多个数据源 join 起来得到，可以用 coGroup 来完成。把多份带交易数据的访问日志联合起来，甚至是两份基于相同数据源的计算结果联合起来，都可以用 coGroup。假设两组 RDD 数据有相同的 key，可以用 rdd.coGroup(otherRdd) 来实现联合。针对不同用途，有好几个

join 函数，都可以在本节末尾的表格中找到。

接下来的任务是学习如何在集群间分发文件，通过给前面章节中梯度下降的例子扩展 GeoIP 功能来演示具体做法。发布某些库的时候需要同时发布它的依赖文件，通常做法是把所有需要的文件打包到一个 Jar 包里，但 Spark 提供了一种极简单的方法来分发依赖文件，只需要调用一个 addFile() 函数，代码如下所示：

```
package pandaspark.examples
import scala.math

import spark.SparkContext
import spark.SparkContext._
import spark.SparkFiles;
import spark.util.Vector

import au.com.bytecode.opencsv.CSVReader

import java.util.Random
import java.io.StringReader
import java.io.File

import com.snowplowanalytics.maxmind.geoip.IpGeo

case class DataPoint(x: Vector, y: Double)

object GeoIpExample {

    def main(args: Array[String]) {
      if (args.length != 2) {
        System.err.println("Usage: GeoIpExample <master>
                            <inputfile>")
        System.exit(1)
    }
    val master = args(0)
    val inputFile = args(1)
    val iterations = 100
    val maxMindPath = "GeoLiteCity.dat"
    val sc = new SparkContext(master, "GeoIpExample",
                System.getenv("SPARK_HOME"),
                Seq(System.getenv("JARS")))
    val invalidLineCounter = sc.accumulator(0)
    val inFile = sc.textFile(inputFile)
    val parsedInput = inFile.flatMap(line => {
```

```
      try {
            val row = (new CSVReader(
              new StringReader(line))).readNext()
                  Some((row(0),row.drop(1).map(_.toDouble)))
      }
      catch {
        case _ => {
          invalidLineCounter += 1
          None
        }
      }
    })
    val geoFile = sc.addFile(maxMindPath)
    //getLocation gives back an option so we use flatMap to only
output if it's a some type
    val ipCountries = parsedInput.flatMapWith(_ => IpGeo
     (dbFile = SparkFiles.get(maxMindPath)))
     ((pair, ipGeo) => {
     ipGeo.getLocation(pair._1).map(c => (pair._1,
      c.countryCode)).toSeq
     })
    ipCountries.cache()
    val countries = ipCountries.values.distinct().collect()
    val countriesBc = sc.broadcast(countries)
    val countriesSignal = ipCountries.mapValues(
      country => countriesBc.value.map(
      s => if (country == s) 1
           else 0
    ))
    val dataPoints = parsedInput.join(countriesSignal)
       .map(input => {
          input._2 match {
            case (countryData, originalData) => DataPoint(
              new Vector(countryData++originalData
              .slice(1,originalData.size-2)),
              originalData(originalData.size-1))
          }
       })
    countriesSignal.cache()
    dataPoints.cache()
    val rand = new Random(53)
    var w = Vector(dataPoints.first.x.length, _ =>
      rand.nextDouble)
    for (i <- 1 to iterations) {
      val gradient = dataPoints.map(p =>
        (1 / (1 + math.exp(-p.y*(w dot p.x))) - 1) * p.y *
        p.x).reduce(_ + _)
        w -= gradient
    }
    println("Final w: "+w)
  }
}
```

在老版本中addFile函数在本地模式下有问题，遇到这种情况时可以采用Shark（一种基于Spark的工具，后面会讲到）开发者用到的一种解决方案（如果addFile失败就直接使用原始文件）。了解更多，请访问：

```
https://github.com/amplab/shark/commit/47c21f55
621acd5afb412f54a45c68e141240030.
```

在前面的代码中出现了多次Spark计算，第一次是决定数据覆盖了哪些国家，这样就可以把每一个国家映射到一个二值特征上；第二次是用一组公共代理的访问延迟来估算延迟。这里也用到了mapWith，它用来给每一个分区创建资源，例如创建一个到后端的连接或创建一个随机数生成器。一些元素无法在节点之间序列化传输（例如本例中的IpCountries），所以需要在本地处理。在本例中还可以看到，多个RDD都被缓存，无需重复计算。

处理多个RDD有几种语言选择，下面分述之。

6.1.1 Scala RDD函数

PairRDD函数都是基于combineByKey实现，所有对[K,V]类型RDD的操作请看表6-1。

表 6-1

函　数	参数选项	解　释	返回类型
foldByKey	(zeroValue) (func(V,V)=>V) (zeroValue,partitioner) (func(V,V=>V) (zeroValue,partitions) (func(V,V)=>V)	用提供的函数来合并值。与传统的作用于list的fold函数不同之处在于，zeroValue可以被加任意多次	RDD[K,V]

（续）

函　数	参数选项	解　释	返回类型
reduceByKey	(func(V,V)=>V) (func(V,V)=>V,numTasks)	reduce 的并行版本，把有相同 key 的值用提供的函数进行合并，返回一个 RDD	RDD[K,V]
groupByKey	() (numPartitions)	按 key 把元素分组	RDD[K, Seq[V]]

6.1.2 PairRDD 连接函数

对于两个或多个键值对的 RDD 处理，连接操作很有用。有几种不同的方法进行连接操作，具体选取哪一种取决于具体需求，如表 6-2 所示。

表 6-2

函　数	参数选项	解　释	返回类型
cogroup	(otherRDD[K,W] …)	根据 key 连接两个或多个 RDD。如果某个元素只出现在其中一个 RDD 中，那么结果之一 Seq 为空	RDD[(K,(Seq[V], Seq[W] …))]
join	(otherRDD[K, W]) (otherRDD[K,W], partitioner) (otherRDD[K,W], numPartitions)	连接两个 RDD，结果集中只包含 key 同时出现在两个 RDD 中的行	RDD[(K, (V, W))]
subtractKey	(otherRDD[K, W]) (otherRDD[K, W],partitioner) (otherRDD[K, W],numPartitions)	返回一个 RDD，结果集中的 key 都不存在于另一个 RDD 中	RDD[(K, V])]

6.1.3 其他 PairRDD 函数

一些函数仅作用于键值对，如表 6-3 所示。

表 6-3

函 数	参数选项	解 释	返回类型
lookup	(key : K)	在RDD中查找特定元素。使用RDD的分区函数确定查询哪个（几个）分区	Seq[V]
mapValues	(f: V=>U)	PairRDD map的特化版本，通过一个作用于值的map函数修改键值对中的值。如果需要基于Key和Value一起来确定映射关系，只能用普通的RDD map函数	RDD[(K,U)]
collectAsMap	()	将RDD转化成一个map。RDD必须能够完全放入内存	Map[K,V]
countByKey	()	计算每一个key下面的元素个数	Map[K, Long]
partitionBy	(partitioner: Partitioner, mapSideCombine: Boolean)	返回一个通过Partitioner重新分区的RDD。mapSide Combine控制具有相同key值的分组，默认为false；如果数据中存在大量重复key可以设置成true	RDD[(K,V)]
flatMapValues	(f: V=>TraversableOnce[U])	类似于mapValues，是PairRDD flatMap的特化版本，通过一个作用于值的map函数修改键值对中的值。输出结果被“扁平化”，即：输出的不是Seq[V]，而是Seq[Seq[V]]。如果需要基于Key和Value一起来确定映射关系，只能用普通的RDD map函数	RDD[(K,U)]

保存 PairRDD 的方法前一章已经叙述，本章不再赘述。

6.1.4 DoubleRDD 函数

Spark 定义了一些好用的函数，可用于 RDD 中的 double 数据类型，如表 6-4 所示。

表 6-4

函 数	参 数	返回值
mean	()	所有 RDD 中的元素的平均值
sampleStdev	()	抽样的标准差
Stats	()	均值、方差、计数，作为一个 StatCounter 返回
Stdev	()	总体的标准差
Sum	()	所有 RDD 中的元素的和
variance	()	所有 RDD 中的元素的方差

6.1.5 通用 RDD 函数

接下来，所有的 RDD 函数都可以作用于所有类型的 RDD。如表 6-5 所示。

表 6-5

函 数	参 数	返回值
aggregate	(zeroValue: U)(seqOp: (U,T) => T, combOp (U, U) => U)	聚集 RDD 上每一个分区的元素，然后用 combOp 进行组合。zeroValue 参数取 0 或 1，0 表示 +，1 表示 *

（续）

函　数	参　数	返回值
cache	()	缓存RDD，下次访问时无需再次计算。等价于persist(StorageLevel.MEMORY_ONLY)
collect	()	RDD中的所有元素以数组形式返回
count	()	RDD中元素总个数
countByValue	()	一个map，记录每个value出现的次数
distinct	() (partitions: Int)	去掉重复元素的RDD
filter	(f: T=>Boolean)	符合条件f的元素组成的RDD
filterWith	(construct A: Int=>A)(f: (T,A)=>Boolean)	类似filter，但条件f增加了一个由constructA生成的额外参数，分别作用于每一个分区。提供该函数的原始动机是为每个分区提供一个随机数生成器
first	()	RDD里的第一个元素
flatMap	(f: T=>TraversableOnce[U])	一个类型为U的RDD
fold	(zeroValue: T)(op: (T,T)=>T)	用提供的操作对值进行合并，首先合并每个分区，然后合并每个分区的结果
foreach	(f: T=>Unit)	将操作f作用在每个元素上
groupBy	(f: T => K) (f: T => K, p:Partitioner) (f: T => K,numPartitions:Int)	以一个RDD作为输入，输出是类型为(K, Seq[T])的键值对组成的一个RDD，其中key是f作用在每个元素上计算出的结果

（续）

函　数	参　数	返回值
keyBy	(f: T => K) (f: T => K, p:Partitioner) (f: T => K,numPartitions:Int)	与 groupBy 一样，但不把有相同 key 的元组合并到一起。返回一个类型为 (K, T) 的 RDD
map	(f: T => U)	将操作 f 作用在输入 RDD 的每个元素上得到的结果组成的 RDD
mapPartitions	(f: Iterator[T] => Iterator[U])	类似于 map，但提供的函数 f 输入输出都是一个 Iterator，并且 f 是分别作用于每个分区
mapPartitions- WithIndex	(f: (Int,Iterator[T]) => Iterator[U], preservePartitions)	类似 mapPartitons，还额外提供了一个参数 index，用于指示输入分区的下标
mapWith	(constructA: Int => A) (f: (T, A) => U)	类似 map，但 f 增加了一个由 constructA 生成的额外参数，分别作用于每一个分区。提供该函数的原始动机是为每个分区提供一个随机数生成器
persist	() (newLevel:StorageLevel)	设置 RDD 的存储级别，可以缓存 RDD 计算结果。存储级别的定义在 StorageLevel.scala 中（常用的有 NONE, DISK_ONLY, MEMORY_ONLY, MEMORY_AND_DISK）
pipe	(command:Seq[String]) (command: Seq[String], env:Map[String, String])	让 RDD 中的每个元素都流过一个外部命令定义的管道，输出一个 String 类型的 RDD

（续）

函　数	参　数	返回值
sample	(withReplacement:Boolean, fraction:Double, seed: Int)	采样元素组成的RDD
takeSample	(withReplacement:Boolean, num: Int, seed: Int)	返回指定个数的采样元素。其工作原理是先过采样RDD，然后提取一个子集
toDebugString	()	辅助调试函数，输出该RDD的衍生链
union	(other: RDD[T])	取两个RDD的并集，重复元素被合并
unpersist	()	从内存或磁盘移除RDD
zip	(other: RDD[U])	要求两个RDD有相同的分区数，每个分区的大小一致。返回一个类型为[T, U]键值对的RDD

6.1.6 Java RDD 函数

大部分Java版RDD函数与Scala版十分相似，但类型签名略有不同。

6.1.7 Spark Java 函数类

Java RDD API都需要通过扩展几个基础函数类来实现，表6-6展示了这些Spark Java函数。

表 6-6

函 数	参 数	用 途
Function<T,R>	R apply(T t)	该函数输入类型为 T，输出类型为 R。常用于映射类函数
DoubleFunction<T>	Double apply(T t)	与 Function<T, Double> 一样，但类 map 函数调用返回的类型是 JavaDoubleRDD（用于统计）
PairFunction<T, K, V>	Tuple2<K, V> apply(T t)	该函数的输出是一个 JavaPairRDD。如果该函数作用于一个 JavaPairRDD<A,B>，那么 T 的类型是 Tuple2<A,B>
FlatMapFunction<T, R>	Iterable<R> apply(T t)	该函数用于通过 flatMap 函数生成一个 RDD
PairFlatMapFunction<T, K, V>	Iterable<Tuple2<K,V>> apply(T t)	该函数的输出是一个 JavaPairRDD。如果该函数作用于一个 JavaPairRDD<A,B>，那么 T 的类型是 Tuple2<A,B>
DoubleFlatMapFunction<T>	Iterable<Double> apply(T t)	类似 FlatMapFunction<T, Double>，但类 map 函数调用返回的类型是 JavaDoubleRDD（用于统计）
Function2<T1, T2, R>	R apply(T1 t1, T2 t2)	该函数有两个输入参数，一个输出结果。用于合并以及类似函数

6.1.8 常用 Java RDD 函数

表 6-7 中的 RDD 函数对任意类型 RDD 均可用。

表 6-7

函 数	参 数	返回值 / 用途
cache	()	让 RDD 保持在内存中
coalesce	numPartitions: Int	返回一个新的 RDD，分区数为 numPartitions

（续）

函　数	参　数	返回值/用途
collect	()	将RDD转成一个List返回
count	()	RDD中元素总个数
countByValue	()	一个map，记录每个value出现的次数。
distinct	() (Int numPartitions)	去掉重复元素的RDD，且生成的RDD的分区数为numPartitions
filter	(Function<T, Boolean> f)	符合条件f的元素组成的RDD
first	()	RDD里的第一个元素
flatMap	(FlatMapFunction<T, U> f) (DoubleFlatMapFunction <T> f) (PairFlatMapFunction<T, K, V> f)	对应地分别返回一个类型为U，Double，和Pair<K, V>的RDD
fold	(T zeroValue, Function2<T,T,T> f)	用提供的操作对值进行合并，首先合并每个分区，然后合并每个分区的结果
foreach	(VoidFunction<T> f)	将操作f作用在每个元素上
groupBy	(Function<T, K> f) (Function<T, K> f, Int numPartitions)	返回分组后的元素，类型为JavaPairRDD
map	(DoubleFunction<T> f) (PairFunction<T, K2, V2> f) (Function<T, U> f)	将操作f作用在输入RDD的每个元素上得到的结果组成的RDD
mapPartitions	(DoubleFunction <Iterator<T>> f) (PairFunction <Iterator<T>, K2, V2> f) (Function<Iterator <T>, U> f)	类似于map，但函数f是作用于每个分区，而不是每个元素。当需要对每一个分区进行一些初始化工作的时候该函数十分有用
reduce	(Function2<T, T, T> f)	用函数f合并所有元素

（续）

函　数	参　数	返回值 / 用途
sample	(Boolean withReplacement, Double fraction, Int seed)	指定比例的采样元素组成的 RDD

6.1.9 JavaPairRDD 合并函数

有很多不同函数可用于合并多个 RDD，如表 6-8 所示。

表 6-8

函　数	参　数	返回值 / 用途
subtract	(JavaRDD<T> oth er) (JavaRDD<T> other, Partitioner p) (JavaRDD<T> other, Int numPartitions)	返回两个 RDD 的差集
union	(JavaRDD<T> other)	返回两个 RDD 的并集
zip	(JavaRDD<U> other)	返回一个键值对 RDD[T, U] 类型的 RDD 注意：本函数要求所有 RDD 的分区数相同，每个分区的大小相同

6.1.10 JavaPairRDD 函数

表 6-9 中介绍了一些仅针对键值对 RDD 的函数。

表 6-9

函 数	参 数	返回值/用途
cogroup	(JavaPairRDD<K, W> other) (JavaPairRDD<K, W> other, Int numPartitions) (JavaPairRDD<K, W> other1, JavaPairRDD<K, W> other2) (JavaPairRDD<K, W> other1, JavaPairRDD<K, W> other2, Int numPartitions)	根据key来连接两个或多个RDD。注意：如果一个元素存在于一个RDD中，但不存在于另一个RDD中，那么其中一个列表会为空
combineByKey	(Function<V, C> createCombiner, Function2<C, V, C> mergeValue, Function2<C, C, C> mergeCombiners)	按照key来合并元素的通用函数。参数createCombiner负责把类型V转换成类型C，mergeCombiners负责把两个C类型的值合并成一个C类型的值
collectAsMap	()	返回一个键值对的map
countByKey	()	返回一个map，记录每一个key下面的元素个数
flatMapValues	(Function[T] f, Iterable[V] v)	返回一个类型为V的RDD
join	(JavaPairRDD<K, W> other) (JavaPairRDD<K, W> other, Int integers)	连接两个RDD，结果集中只包含key同时出现在两个RDD中的行
keys	()	返回一个只包含key的RDD
lookup	(Key k)	在RDD中查找特定元素。该函数会利用RDD的分区函数来计算应该在哪个或哪些分区中查找元素
reduceByKey	(Function2[V,V,V] f)	reduce函数的并行版本，使用f来合并每个key对应的values，最终返回一个合并后的RDD

（续）

函　数	参　数	返回值 / 用途
sortByKey	(Comparator [K] comp, Boolean ascending) (Comparator [K] comp) (Boolean ascending)	按照 key 对 RDD 排序，最终每个分区对应一个连续范围的元素
values	()	返回一个只包含 value 的 RDD

6.2 用 Python 操作 RDD

Python 环境下 Spark 支持的函数比 Java 和 Scala 的要少，尽管如此，Spark 的大部分核心功能都还是支持的。

MapReduce 系统下标志性的两个命令式：map 和 reduce。map 函数前面的章节中已经介绍过了，它的工作方式就是对输入 RDD 中的一个元素调用一个函数，该函数输出一个新元素。例如，给一个 RDD 上的所有元素都加一，生成一个新的 RDD，可以这样实现：rdd.map(lambda x: x+1)。特别要注意的，map 函数以及其他 Spark 函数都不会改变当前 RDD 的值，而是会返回一个由新值组成的新 RDD。reduce 函数对所有元素对调用一个函数，该函数能把所有数据合并到一起，最终的合并结果返回给调用程序。对于一个让所有元素相加的程序来说，可以这么写：rdd.reduce(lambda x, y: x+y)。

flatMap 函数是个有用的工具，允许用户写一个函数，其返回结果是一个指向“扁平”值的 Iterable 对象。一个简单的例子是：希望解析所有数据，但其中的部分数据可能没有被解析。flatMap 函数在解析失败的时候能输出一个空列表，成功的时候能输出一个正常的列表。除了 reduce，还有一个对应的 reduceByKey 函数，

专门针对键值对类型的RDD，计算生成一个新的RDD。

许多针对RDD的映射操作也定义了针对分区的变体。这种情况下，函数的输入和输出都是一个代表分区内所有元素的Iterator对象。当需要针对每个分区做大量操作，例如，向后端服务器发起一个连接时，这些变体函数十分有用。

通常，数据能通过键值映射的形式表达，所以Python RDD类中定义了许多专门针对数据类型为键值对的RDD函数。如果只希望更新键值对中的值部分，可以使用mapValues函数。

另一个传统map函数的变体是mapPartitions，它的操作对象是一个个的分区。使用mapPartitions的主要理由是它可以用于给map函数做初始化工作。一个很好的例子是在初始化阶段创建一个复杂的连接到后端服务器，或者解析一些复杂的输入。

```
def f(iterator):
      //Expensive work goes here
     for i in iterator:
       yield per_element_function(i)
```

除了针对数据的简单操作，Spark还提供了broadcast以及accumulator支持。broadcast能用于广播只读数据到所有的分区上，以避免把一个值反复序列化多次。accumulator允许所有分区往上累加值，结果能从主节点中读取。创建累加器的方法是counter = sc.accumulator(initialValue)。如果希望增加自定义行为，可以提供一个AccumulatorParam作为accumulator函数的参数。返回的counter可以在任意工作节点上用counter += x的方法增加其值。结果值通过counter.value()读取到。broadcast值通过bc = sc. broadcast(value)创建，通过bc.value()在工作节点上读取。accumulator值只能在主节点上读到，broadcast值能在

所有分区上读到。

6.2.1 标准 RDD 函数

表 6-10 介绍了 Python 中一些可以作用于所有类型 RDD 的函数。

表 6-10

函　数	参　数	用　途
flatMap	(f, preservesPartitioning=False)	以一个函数为参数，该函数作用于每一个输入类型为 T 的元素，返回一个类型为 U 的 Iterator 对象。flatMap 返回一个类型为 U 的扁平化 RDD
mapPartitions	(f, preservesPartitioning=False)	以一个函数为参数，该函数输入为类型为 T 的 Iterator，输出是类型为 U 的 Iterator。最终结果是一个类型为 U 的 RDD。例如，提供的函数以整数 iterator 为输入，字符串 iterator 为输出，它作用于一个整数型 RDD 之后的输出是一个字符串类型的 RDD。对于需要复杂初始化操作的 map 操作有用
filter	(f)	以一个函数为参数，返回的 RDD 中包含所有被函数结算后结果为 true 的元素
distinct	()	返回去掉了重复元素的 RDD(例如，输入是 1，1，2，输出是 1，2)
union	(other)	返回两个 RDD 的并集
cartesian	(other)	返回两个 RDD 的笛卡尔积
groupBy	(f, numPartitions=None)	返回值是 f 的输出经过聚合后得到的元素组成的 RDD

（续）

函　数	参　数	用　途
pipe	(command, env={})	让 RDD 中的每个元素都流过一个外部命令定义的管道，输出一个 RDD
foreach	f	将操作 f 作用在每个元素上
reduce	f	用给定的函数合并所有元素
fold	zeroValue, op	用提供的操作对值进行合并，首先合并每个分区，然后合并每个分区的结果
countByValue	()	返回一个字典映射，记录每个值及其在 RDD 中出现的次数
take	num	返回含有 num 个元素的列表。如果 num 值很大，该函数会很慢。如果想读取整个 RDD，请使用 collect 函数
partitionBy	(numPartitions, partitionFunc=hash	用新的分区函数来分区 RDD。partitionFunc 参数只需要简单地把输入映射到一个整数空间上，partitionBy 将整数与 numPartitions 取模即可

6.2.2 PairRDD 函数

表 6-11 中介绍了一些仅针对键值对 RDD 的函数。

表　6-11

函　数	参　数	用　途
collectAsMap	()	返回一个由 RDD 的所有键值对组成的字典
reduceByKey	(func, numPartitions=None)	reduceByKey 函数是 reduce 函数的并行版，用提供的 func 函数把每个 key 下的所有 value 都合并起来生成一个 RDD

（续）

函　数	参　数	用　途
countByKey	()	返回一个字典映射，记录每个键及其在RDD中出现的次数
join	(other, numPartitions=None)	连接两个RDD，结果集中只包含key同时出现在两个RDD中的行。每一个key对应结果是一个由两个value组成的元组
rightOuterJoin	(other, numPartitions=None)	连接两个RDD，结果集中的key都来自other，如果key在源RDD中不存在，那么结果元组中的第一个值就为None
leftOuterJoin	(other, numPartitions=None)	连接两个RDD，结果集中的key都来自源RDD，如果key在other中不存在，那么结果元组中的第二个值就为None
combineByKey	(createCombiner, mergeValues, mergeCombiers)	按照key来合并元素。本函数以一个类型为(K, V)的RDD为输入，以一个类型为(K, C)的RDD为输出。参数createCombiner负责把类型V转换成类型C，mergeCombiners负责把两个C类型的值合并成一个C类型的值
groupByKey	(numPartitions=None)	按照key来对RDD中的元素进行分组
cogroup	(other, numPartitions=None)	根据key来连接两个或多个RDD。注意：如果一个元素存在于一个RDD中，但不存在于另一个RDD中，那么其中一个列表会为空

6.3 链接和参考

下面列出一些有用的链接和参考：

- http://www.scala-lang.org/api/current/index.html#scala.collection.immutable.List
- http://spark.incubator.apache.org/docs/latest/api/core/index.html#spark.api.java.JavaRDD
- http://spark.incubator.apache.org/docs/latest/api/core/index.html#spark.api.java.JavaPairRDD
- http://spark.incubator.apache.org/docs/latest/api/core/index.html#spark.SparkContext
- http://spark.incubator.apache.org/docs/latest/api/core/index.html#packa

6.4 小结

本章介绍了当数据加载到RDD后如何以一种分布式的方式对数据进行计算，综合读写RDD的知识，现在能够用Spark编写分布式程序了。在下一章，将介绍如何综合运用Spark和Hive。

第 7 章
Shark-Hive 和 Spark 的综合运用

本章介绍如何综合使用 Hive 和 Spark，把 Hive 的查询语句集成到 Spark 程序中。跳过本章不会影响对后继章节内容的理解，所以，如果你不希望了解 Hive，可以直接跳过本章。

本章涵盖以下主题：

- 使用 Hive/Shark
- 如何安装 Shark
- 加载数据到 Shark
- 运行 Shark
- 在 Spark 程序中运行 HiveQL 查询

7.1 为什么用 Hive/Shark

Hive 是一个很火的 Hadoop 项目，允许对大数据集进行级联查询。Hive 上的查询语言叫 HiveQL，支持大部分的 SQL 和一些扩展功能。设计 Shark 之初就充分考虑了兼容 Hive 查询语言、序列化格式等需求，人们选择使用 Shark 的主要原因是因为它处理多条查询的能力比传统的 Hive 和 Hadoop 要强。本章不讲 Hive，而是讲如何把 HiveQL 集成到 Spark 程序中，以及如何设置 Shark。一般来说，HiveQL 与 SQL 比较像，所以对 SQL 熟悉的话，理解本章的内容应该很轻松。

7.2 安装 Shark

编写本章时 Shark 的最新版本是 0.7.0，它依赖于 Spark 0.7.2 和最新版的 JVM（Open JK7 或 Oracle HotSpot JDK7）。Shark 针对 Hadoop 1 和 Hadoop 2 的预编译版下载地址分别是 http://spark-

project.org/download/shark-0.7.0-hadoop1-bin.tgz 和 http://spark-project.org/download/shark-0.7.0-hadoop2-bin.tgz。下载和解压好 Shark 后，就可以配置了。本例假设 Shark 被解压到了 /home/spark/ 目录下。Shark 的配置独立于 Spark，位于 shark-0.7.0/conf/shark-env.sh，对于本地模式，需要像下面这样设置 HIVE_HOME 和 SPARK_HOME：

```
export HIVE_HOME=/home/spark/hive-0.9.0-bin
export SPARK_HOME=/home/park/spark-0.7.2
source $SPARK_HOME/conf/spark-env.sh
```

在本地模式下需要创建一个目录来存 Hive 的文件，默认目录是 /user/hive/warehouse。为了保证目录可访问，可以用 chown 命令，如下所示：

```
mkdir -p /user/hive/warehouse && chown [your-spark-user] /user/hive/
warehouse
```

如果在 Spark 集群中使用 Shark，还需要设置 MASTER 和 HADOOP_HOME 变量。如果 Hive 在原来的环境中预安装好了，那么需要设置 HIVE_CONF_DIR 变量，指向 Hive XML 配置文件所在目录。如果上面这几个变量定义在了 source... 这一行之后的话，还可以在这几个变量中直接引用 Spark 中的变量，例如：

```
export HADOOP_HOME=/path/to/hadoop
export MASTER=spark://$SPARK_MASTER_IP:7077
```

Shark 安装设置好之后，需要把 Shark 和配置好的 Hive 拷贝到所有的工作节点上，方法如下：

```
pscp -v -r -h ./spark-0.7.2/conf/slaves -l sparkuser
./shark-0.7.0 ~/
pscp -v -r -h ./spark-0.7.2/conf/slaves -l sparkuser
./hive-0.9.0-bin ~/
```

如果是在 EC2 上设置 Shark，那么设置这一步就免了吧，直

接使用最新的 AMI，上面已经安装好了 Shark。

7.3 运行 Shark

不管用的是哪种安装配置 Shark 的方法，启动 Shark CLI 的方法都一样。Shark 的 bin 目录提供了对应不同日志级别的三种 shark 变体，默认是 `./bin/shark`，适合大多数情况；遇到问题需要调试时可以使用 `./bin/shark-withinfo`，需要更多调试信息的话可以使用 `./bin/shark-withdebug`。如果把 Shark 连接到 Spark 集群，则可以通过浏览器界面的“running jobs”下看到 Spark 作业。（如果看不到，可能的原因是 Shark 作业依然运行在一个本地 Spark 上，所以，请反复确认配置无误。）

7.4 加载数据

Hive 默认带了一个示例数据集，位于 `~/hive-0.9.0-bin/example/`，可以用它来验证 Shark 是否能正常工作。用下面的命令加载数据到 Shark：

```
shark> CREATE TABLE src(key INT, value STRING);
shark> LOAD DATA LOCAL INPATH '${env:HIVE_HOME}/examples/files/in1.txt'
INTO TABLE src;
shark> SELECT src.key, src.value FROM src WHERE src.key < 100;
```

以上命令成功执行后会有类似下面这样的输出：

```
OK
48
Time taken: 3.02 seconds
```

Shark 也可以从 S3 上以与 Hive 一样的方式加载数据。作为测试，使用 HiveAWS 手册中提供的公开数据集，例如，`s3n://`

data.s3ndemo.hive/kv，这是一个键值对数据集。为了访问该数据，需要把 AWS 账号的用户名、密码配置到 ~/hive-0.9.0-bin/conf/hive-site.xml(若不存在则创建一个) 中，如下所示：

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>

<property>
    <name>fs.s3n.awsAccessKeyId</name>
    <value>accesskeygoeshere</value>
</property>
<property>
    <name>fs.s3n.awsSecretAccessKey</name>
    <value>yoursecretkeygoeshere</value>
</property>
</configuration>
```

还有一种方式是把 AWS 用户名密码写在数据集路径中：s3n://username:passwor@[…]。假设已经把 AWS 用户名密码配置在 hive-site.xml 中了，可以这样为数据集创建一个表格：create external table kv (key int, values string) location 's3n://data.s3ndemo.hive/kv';。若希望加载更复杂的数据，请参考 HiveAWS 手册。

Shark 还提供了一个独立的 Shark shell 界面，与 Spark 的 shell 比较相似。在该界面下可以编写 Scala 的代码与 Shark 交互。容易混淆的是：Shark 的上下文是用 sc 表示，Spark 的 shell 里面也是用 sc 来指代 SparkContext。

7.5 在 Spark 程序中运行 HiveQL 查询

如果希望把一个 Hive 查询集成到 Spark 项目中，可以

用 sql2rdd。需要注意的是，返回值类型为 RDD[shark.api.Row]，所以即使用了 sql2rdd 也还是要做一些转换工作，使得返回数据能被普通的 Spark 代码使用。行 API 提供了 get[Type](rowName)，支持所有已知类型。从一行数据里面取出一个整数可以这样写：row.getInt("key")，其中 key 是某个整数列的列名。

Shark 最初是一个独立的项目，最近才“Maven 化”，这使得在类似于 Spark 环境中引用 Shark 模块变得容易。尽管如此，Shark 目前还没有部署到任何 Maven 库中。不过用户可以自己动手，把它部署到本地 Maven 中，具体做法是在 Spark 的目录下运行 stb/sbt publish-local。

如果运行 sbt/sbt publish-local 时出现“不安全和未预料的参数个数”错误提示，可能是 sbt 使用的是老 JDK。可以通过设置 java_home 来指定 JDK 版本。

在 Shark shell 交互模式下做一下实验，下面是综合运用 Shark 和 Spark 的例子：

```
//Basic Shark example in Scala

package com.pandaspark.examples

import shark._
import spark.SparkContext._

object BasicSharkExample {
  def main(args: Array[String]) {
    val sc = SharkEnv.initWithSharkContext("BasicSharkExample")
    println("Starting shark requests");
    sc.sql("drop table if exists src");
    sc.sql("CREATE TABLE src(key INT, value STRING)")
    sc.sql("LOAD DATA LOCAL INPATH
           '${env:HIVE_HOME}/examples/files/in1.txt'
           INTO TABLE src")
```

```
    val rdd = sc.sql2rdd("SELECT src.key, src.value
                          FROM src WHERE src.key < 100")
    rdd.cache()
    println("Found "+rdd.count()+" num rows")
    val normalRDD = rdd.map(x => (x.getInt("src.key"),
                            x.getString("src.value")))
    println("Formatted as "+normalRDD.collect().mkString(","))
  }
}
```

对应的 Java 版实现：

```
//Basic Shark example in Java

package com.pandaspark.examples;

import spark.api.java.JavaRDD;
import spark.api.java.JavaPairRDD;
import spark.api.java.function.PairFunction;

import scala.Tuple2;

import shark.SharkEnv;
import shark.api.Row;
import shark.api.JavaSharkContext;
import shark.api.JavaTableRDD;

public class BasicJavaSharkExample {
  public static void main(String[] args) {
    JavaSharkContext sc = SharkEnv.
        initWithJavaSharkContext("BasicSharkExample");
    sc.sql("drop table if exists src");
    sc.sql("CREATE TABLE src(key INT, value STRING)");
    sc.sql("LOAD DATA LOCAL INPATH
           '${env:HIVE_HOME}/examples/files/in1.txt'
            INTO TABLE src");
    JavaTableRDD rdd = sc.sql2rdd("SELECT src.key,
                                  src.value FROM src
                                  WHERE src.key < 100");
    rdd.cache();
    System.out.println("Found "+rdd.count()+" num rows");
    JavaPairRDD<Integer, String> normalRDD = rdd.map(new
      PairFunction<Row, Integer, String>() {
      @Override
      public Tuple2<Integer, String> call(Row x) {
        return new Tuple2<Integer,String>(x.getInt("key"),
                   x.getString("value"));
      }
    });
```

```
    System.out.println("Collected: "+normalRDD.collect());
  }
}
```

为了避免直接依赖 Spark，会这样修改 build.sbt：

```
libraryDependencies ++= Seq(
    "edu.berkeley.cs.amplab" % "shark_2.9.3" % "0.7.0"
)
```

还需要包含随 Shark 发行的补丁版 Hive，具体方法是在 build 文件中添加如下内容：

```
unmanagedJars in Compile <++= baseDirectory map {
  base => val hiveFile = file(System.getenv("HIVE_HOME")) / "lib"
          val baseDirectories = (base / "lib") +++ (hiveFile)
          val customJars = (baseDirectories ** "*.jar")
//Hive uses an old version of guava that doesn't have what we want
  customJars.classpath.filter(!_.toString.contains("guava"))
}
```

运行这样一个作业与运行普通的 Spark 作业略有不同。在 SharkEnv 初始化逻辑中，会搜索多个环境变量用于启动设置。所以，确保所有环境变量都被正确设置的最简单办法是用系统提供的脚本，只需要像下面一样设置好 CLASSPATH：

```
CLASSPATH=/home/spark/fastdataprocessingwithspark-sharkexamples/target/
scala-2.9.3/fastdataprocessingwithspark-sharkexamples-assembly-0.1-
SNAPSHOT.jar ./run com.pandaspark.examples.BasicSharkExample
```

7.6 链接和参考

下面列出一些有用的链接和参考：

- http://hive.apache.org/
- https://github.com/amplab/shark/wiki/Shark-User-Guide
- https://github.com/amplab/shark/wiki
- https://github.com/amplab/shark/wiki/Running-Shark-Locally
- https://github.com/amplab/shark/wiki/Running-Shark-on-a-Cluster
- https://cwiki.apache.org/confluence/display/Hive/

HiveAws+HivingS3nRemotely

- https://github.com/amplab/shark/wiki#developer-documentation

7.7 小结

本章介绍了如何设置 Shark 以及如何把 Shark 集成到 Spark 程序中。下一章将介绍如何编写简单的单元测试。

第8章
测　试

不经测试就能写出高效代码是很有挑战的。高效的测试能极大提高开发者的效率，特别是遇到类似分布式系统这种端到端执行比较耗时的情况。不过本章的重点不是说服你必须进行测试，自觉艺高人胆大，那也行。

8.1 用 Java 和 Scala 测试

为简单起见，本章只使用 ScalaTest 和 JUnit 测试库。ScalaTest 既能用于 Scala 代码，也能用于 Java 代码，目前 Spark 用的就是它。JUnit 是流行的 Java 测试框架。

8.1.1 为测试而重构

对于已经与 RDD 交互或 SparkContext 交互隔离开了的代码，可以使用标准的测试方法进行测试。用匿名函数来编写 Spark 代码固然很方便，但如果能顺便给这些匿名函数命个名字，测试起来则会轻松得多，因为这样就不需要去很罗嗦地设置 SparkContext。举个例子，前面实现的 Scala CSV 解析器代码，测试起来就很难：

```
val splitLines = inFile.map(line => {
    val reader = new CSVReader(new StringReader(line))
    reader.readNext().map(_.toDouble)
}
```

或者是 Java 代码：

```
JavaRDD<Integer[]> splitLines = inFile.flatMap(
new FlatMapFunction<String, Integer[]> (){
  public Iterable<Integer[]> call(String line) {
    ArrayList<Integer[]> result = new ArrayList<Integer[]>();
    try {
    CSVReader reader = new CSVReader(new StringReader(line));
    String[] parsedLine = reader.readNext();
```

```
    Integer[] intLine = new Integer[parsedLine.length];
    for (int i = 0; i < parsedLine.length; i++) {
      intLine[i] = Integer.parseInt(parsedLine[i]);
    result.add(intLine);
    }
    catch (Exception e) {
      errors.add(1);
    }
    return result;
    }
  }
);
```

在 Scala 里，可以把这个 CSV 解析器写成一个单独的函数：

```
def parseLine(line: String): Array[Double] = {
    val reader = new CSVReader(new StringReader(line))
    reader.readNext().map(_.toDouble)
}
```

或者在 Java 里可以这样写：

```
public class JavaLoadCsvTestable {
    public static class ParseLine extends Function<String, Integer[]>
{
      public Integer[] call(String line) throws Exception {
       CSVReader reader = new CSVReader(new StringReader(line));
       String[] parsedLine = reader.readNext();
       Integer[] intLine = new Integer[parsedLine.length];
       for (int i = 0; i < parsedLine.length; i++) {
         intLine[i] = Integer.parseInt(parsedLine[i]);
       }
      return intLine;
      }
    }
```

然后在测试 Java 代码的过程中根本不需要担心 Spark 相关的逻辑处理：

```
package pandaspark.examples

import org.scalatest.FunSuite
import org.scalatest.matchers.ShouldMatchers

class TestableLoadCsvExampleSuite extends FunSuite with ShouldMatchers
{
    test("should parse a csv line with numbers") {
```

```
      TestableLoadCsvExample.parseLine("1,2") should equal
(Array[Double](1.0,2.0))
      TestableLoadCsvExample.parseLine("100,-1,1,2,2.5")
  should equal (Array[Double](100,-1,1.0,2.0,2.5))
    }
    test("should error if there is a non-number") {
      evaluating {
        TestableLoadCsvExample.parseLine("pandas")
      } should produce [NumberFormatException]
    }
}
```

或者在测试 Java 代码的时候还可以这样写：

```
class JavaLoadCsvExampleSuite extends FunSuite with ShouldMatchers {

    test("should parse a csv line with numbers") {
      val parseLine = new JavaLoadCsvTestable.ParseLine();
      parseLine.call("1,2") should equal (Array[Integer](1,2))
      parseLine.call("100,-1,1,2,2") should equal
  (Array[Integer](100,-1,1,2,2))
    }
    test("should error if there is a non-integer") {
      val parseLine = new JavaLoadCsvTestable.ParseLine();
      evaluating { parseLine.call("pandas")  } should produce
  [NumberFormatException]
      evaluating {parseLine.call("100,-1,1,2.2,2") should equal
  (Array[Integer](100,-1,1,2,2)) } should produce
  [NumberFormatException]
    }
}
```

上面的测试使用的都是 ScalaTest，别着急，等一下再看如何用 JUnit。

8.1.2 与 SparkContext 交互测试

前面的代码中，为了优雅地处理失败，CSV 解析器被扩展成可以记录非法输入次数。为了验证这一部分逻辑，可以提供一个 mock counter 和其他相关 mock 对象来模拟出用到的 Spark 组件，但无形之中也施加了一个限制：只能测试不依赖于 Spark 的代码。

为了解决这一问题，可以重构代码，让核心部分不依赖于 Spark 而变得可测，同时支持外部传入 SparkContext 来进行更完整的测试，如下面的代码所示：

这么做有一个很大的副作用，要求测试必须串行执行，否则 sbt 会尝试同时启动多个 SparkContext，导致输出很多诡异的错误日志。可以通过在测试中设置 `parallelExecution` 为 `false` 强制测试程序在 sbt 中顺序执行。

```
object MoreTestableLoadCsvExample {
  def parseLine(line: String): Array[Double] = {
    val reader = new CSVReader(new StringReader(line))
    reader.readNext().map(_.toDouble)
  }
  def handleInput(invalidLineCounter: Accumulator[Int],
  inFile: RDD[String]): RDD[Double] = {
    val numericData = inFile.flatMap(line => {
      try {
        Some(parseLine(line))
      }
      catch {
        case _ => {
        invalidLineCounter += 1
        None
        }
      }
    })
    numericData.map(row => row.sum)
  }

  def main(args: Array[String]) {
    if (args.length != 2) {
      System.err.println("Usage: TestableLoadCsvExample
  <master> <inputfile>")
      System.exit(1)
    }
    val master = args(0)
    val inputFile = args(1)
    val sc = new SparkContext(master, "Load CSV Example",
  System.getenv("SPARK_HOME"),
  Seq(System.getenv("JARS")))
```

```
    sc.addFile(inputFile)
    val inFile = sc.textFile(inputFile)
    val invalidLineCounter = sc.accumulator(0)
    val summedData = handleInput(invalidLineCounter, inFile)
    println(summedData.collect().mkString(","))
    println("Errors: "+invalidLineCounter)
    println(summedData.stats())
  }

}
```

用下面的代码测试这个类：

```
import spark._
import spark.SparkContext._
import org.scalatest.FunSuite
import org.scalatest.matchers.ShouldMatchers

class MoreTestableLoadCsvExampleSuite extends FunSuite with
ShouldMatchers {
  test("summ data on input") {
    val sc = new SparkContext("local", "Load CSV Example")
    val counter = sc.accumulator(0)
    val input = sc.parallelize(List("1,2","1,3"))
    val result = MoreTestableLoadCsvExample.handleInput(counter,
  input)
    result.collect() should equal (Array[Int](3,4))
  }
  test("should parse a csv line with numbers") {
    MoreTestableLoadCsvExample.parseLine("1,2") should equal
  (Array[Double](1.0,2.0))
    MoreTestableLoadCsvExample.parseLine("100,-1,1,2,2.5")
  should equal (Array[Double](100,-1,1.0,2.0,2.5))
  }
  test("should error if there is a non-number") {
    evaluating { MoreTestableLoadCsvExample.parseLine("pandas") }
  should produce [NumberFormatException]
  }
}
```

用 Java 实现：

```
public class JavaLoadCsvMoreTestable {
    public static class ParseLineWithAcc extends
  FlatMapFunction<String, Integer[]> {
    Accumulator<Integer> acc;
    ParseLineWithAcc(Accumulator<Integer> acc) {
      this.acc = acc;
```

```
    }
    public Iterable<Integer[]> call(String line) throws Exception{
      ArrayList<Integer[]> result = new ArrayList<Integer[]>();
      try {
        CSVReader reader = new CSVReader(new StringReader(line));
        String[] parsedLine = reader.readNext();
        Integer[] intLine = new Integer[parsedLine.length];
        for (int i = 0; i < parsedLine.length; i++) {
          intLine[i] = Integer.parseInt(parsedLine[i]);
        }
        result.add(intLine);
      }
      catch (Exception e) {
        acc.add(1);
      }
      return result;
    }
  }
    public static JavaDoubleRDD processData(
  Accumulator<Integer> acc, JavaRDD<String> input) {
      JavaRDD<Integer[]> splitLines = input.flatMap(
  new ParseLineWithAcc(acc));
      JavaDoubleRDD summedData = splitLines.map(
  new DoubleFunction<Integer[]>() {
        public Double call(Integer[] in) {
          Double ret = 0.;
          for (int i = 0; i < in.length; i++) {
            ret += in[i];
          }
        return ret;
        }
      }
      );
    return summedData;
}
```

然后在 Scala 中用下面的代码测试：

```
class JavaLoadCsvMoreTestableSuite extends FunSuite
with ShouldMatchers {
  test("sum data on input") {
    val sc = new JavaSparkContext("local", "Load Java CSV test")
    val counter: Accumulator[Integer] = sc.intAccumulator(0)
    val input: JavaRDD[String] = sc.parallelize
  (List("1,2","1,3","murh"))
    val javaLoadCsvMoreTestable = new JavaLoadCsvMoreTestable();
    val resultRDD = JavaLoadCsvMoreTestable.
  processData(counter,input)
    resultRDD.cache();
```

```
    val resultCount = resultRDD.count()
    val result = resultRDD.collect().toArray()
    resultCount should equal (2)
    result should equal (Array[Double](3.0, 4.0))
    counter.value should equal (1)
    sc.stop()
  }
}
```

注意，上面的测试数据中增加了一个非法输入。

在 Java 中，使用 JUnit4 测试：

```
package pandaspark.examples;

import spark.*;
import spark.api.java.JavaSparkContext;
import spark.api.java.JavaRDD;
import spark.api.java.JavaDoubleRDD;
import org.scalatest.FunSuite;
import org.scalatest.matchers.ShouldMatchers;

import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.Ignore;
import org.junit.runner.RunWith;
import org.junit.runners.JUnit4;

import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;
@RunWith(JUnit4.class)
public class JavaLoadCsvMoreTestableSuiteJunit {
    @Test
    public void testSumDataOnInput() {
      JavaSparkContext sc = new JavaSparkContext("local",
  "Load Java CSV test");
      Accumulator<Integer> counter = sc.intAccumulator(0);
      String[] inputArray = {"1,2","1,3","murh"};
      JavaRDD<String> input = sc.parallelize
  (Arrays.asList(inputArray));
      JavaDoubleRDD resultRDD = JavaLoadCsvMoreTestable.
  processData(counter, input);
      long resultCount = resultRDD.count();
      assertEquals(resultCount, 2);
      int errors = counter.value();
      assertEquals(errors, 1);
```

```
        sc.stop();
    }
}
```

8.2 用 Python 测试

Spark 的 Python 测试在概念上是相似的，差别在于测试库略有不同。PySpark 使用 doctest 和 unittest 来测试自己。doctest 库让写测试变得很容易，它的工作原理是将程序输出的文本与期望结果做比对。运行测试的方法是：pyspark -m doctest [pathtocode]，以 Spark 提供的 wordcount.py 中提取出的 countWords 为例，用 doctest 来测试 word count 函数：

```
"""
>>> from pyspark.context import SparkContext
>>> sc = SparkContext('local', 'test')
>>> b = sc.parallelize(["pandas are awesome", "and ninjas are also
awesome"])
>>> countWords(b)
[('also', 1), ('and', 1), ('are', 2), ('awesome', 2), ('ninjas', 1),
('pandas', 1)]
"""

import sys
from operator import add

from pyspark import SparkContext
def countWords(lines):
    counts = lines.flatMap(lambda x: x.split(' ')) \
                  .map(lambda x: (x, 1)) \
                  .reduceByKey(add)
    return sorted(counts.collect())

if __name__ == "__main__":
    if len(sys.argv) < 3:
      print >> sys.stderr, "Usage: PythonWordCount
  <master> <file>"
      exit(-1)
    sc = SparkContext(sys.argv[1], "PythonWordCount")
    lines = sc.textFile(sys.argv[2], 1)
```

```
    output = countWords(lines)
    for (word, count) in output:
      print "%s : %i" % (word, count)
```

也可以像下面这样测试类似于前面 Java 和 Scala 程序测试的内容，具体代码如下所示：

```
"""
>>> from pyspark.context import SparkContext
>>> sc = SparkContext('local', 'test')
>>> b = sc.parallelize(["1,2","1,3"])
>>> handleInput(b)
[3, 4]
"""

import sys
from operator import add

from pyspark import SparkContext
def handleInput(lines):
    data = lines.map(lambda x: sum(map(int, x.split(','))))
    return sorted(data.collect())

if __name__ == "__main__":
    if len(sys.argv) < 3:
        print >> sys.stderr, "Usage: PythonLoadCsv
  <master> <file>"
        exit(-1)
    sc = SparkContext(sys.argv[1], "PythonLoadCsv")
    lines = sc.textFile(sys.argv[2], 1)
    output = handleInput(lines)
    for sum in output:
      print sum
```

8.3 链接和参考

下面列出一些有用的链接和参考：

- http://blog.quantifind.com/posts/spark-unit-test/
- http://www.scalatest.org/
- http://junit.org/
- http://docs.python.org/2/library/unittest.html
- http://docs.python.org/2/library/doctest.html

8.4 小结

本章介绍了如何组织代码使其具备可测性，还介绍了 Spark 中使用的测试框架。有效的测试能够节省大量调试时间，特别对于分布式系统来说，调试是极其痛苦的。在下一章，将会介绍一些诸如 Spark 的调试和安全之类的技巧和窍门。

第 9 章
技巧和窍门

你已经学会了如何构建和测试 Spark 作业，并且能够让它们在 Spark 集群上运行起来，现在开始学习如何成为一个专业的 Spark 开发者。

9.1 日志位置

Spark 和 Shark 都会记录运行日志，遇到问题的时候可以通过分析这些日志来定位问题。当运行的程序用到 `sql2rdd` 或基于 Shark 的工具时，理想的调试入口点就是通过日志看一看系统正在执行一个什么样的 HiveQL，一般显示在执行的 Spark 程序的控制台日志里，搜索类似于这样的一行文本 `Hive history file=/tmp/spark/hive_job_log_spark_201306090132_919529074.txt`。Spark 还在每台机器上保存了一份本机日志，默认路径是 Spark 目录下的 logs 子目录。Spark 还提供了 Web 界面来查看每个作业的 `stdout`（标准输出）和 `stderr`（标准错误）文件。

9.2 并发限制

Spark 的并发度受分区（partition）数量的限制，反过来说，分区数过多就会导致启动过多的任务，带来不必要的开销。当分区数过多的时候应该使用 `coalesce(count)` 方法来收缩分区数量。当创建新的 RDD 时可以设置分区切分数量，针对多个 RDD 的 `grouping` 或者 `joining` 方法可以使用这个设置。新建 RDD 的默认分区数受 `spark.default.parallelism` 变量控制，这个变量还控制了 `groupByKey` 运算符以及其他 `shuffle` 运算符能够启动的任务数。

9.3 内存使用与垃圾回收

可以通过让JVM打印垃圾回收细节来了解垃圾回收对Spark的影响，具体做法是在conf/spark-env.sh文件的SPARK_JAVA_OPTS环境变量后面添加如下选项：-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps。添加这些选项后就能在作业运行时从标准输出看到垃圾回收细节了。查看标准输出的方法请参考本章第一节“日志位置”。

如果发现Spark集群耗费过多时间在垃圾回收上，可以通过spark.storage.memeoyFaction调低RDD缓存的用量，这个值默认是0.66。如果计划在一个集群上运行一个耗时很久的Spark作业，可以通过设定spark.cleaner.ttl为一个非零值n，表示每隔n秒清理一次元数据。默认情况下Spark不会清理任何元数据。

如果发现Spark集群耗费过多内存，可以通过控制RDD的存储级别来调整内存用量。如果RDD在内存里放不下，但依然希望把所有数据都缓存在内存里，可以尝试以下不同的存储级别：

- MEMORY_ONLY：默认存储级别，会尽可能把整个RDD都缓存在内存中。
- MEMORY_AND_DISK：尽可能把每一个分区都缓存在内存中，内存不够时才选择存储在磁盘中。
- DISK_ONLY：无论内存是否足够，所有分区都存储在磁盘中。

在RDD上调用持久化函数时会应用这些选项，默认情况下RDD都以非序列化模式存储，这样可以节省读取数据时的反序列

化开销，如果存储级别后面添加了 _SER 后缀（如 MEMORY_AND_DISK_SER——译者注），Spark 会在存储时对数据进行序列化，这样可以节省一些存储空间。

9.4 序列化

出于对运行速度、空间利用率和完整支持 Java 对象几个方面的权衡考虑，Spark 支持数种不同的序列化方法。如果使用序列器来缓存 RDD，强烈建议使用一个快速序列器。默认序列器使用了 Java 的默认序列化算法，KyroSerialier 序列器比默认序列器快得多，算法需要的内存是默认序列器的 1/10。可以通过将 spark.serializer 改成 spark.KryoSerializer 来切换序列器。使用 KyroSeriializer 有一个前提：类必须是可被 KryoSerializer 序列化的。

Spark 提供了一个 KryoRegistrator 接口，只要实现该接口就可以 KryoSerializer 序列化。

```
class MyReigstrator extends spark.KyroRegistrator {
   override def registerClasses(kyro: Kyro) {
    kyro.register(classOf[MyClass])
   }
}
```

访问 https://code.google.com/p/kryo/#Quick-start 了解更多关于如何实定制序列器的知识。通过自定义序列器，可以极大降低对象序列化后占用的空间。例如，可以通过调用 kyro.register(classOf[MyClass],100) 给类赋一个整数 ID，而不是把完整的类名序列化。

9.5 IDE 集成环境

作者作为一个 Emacs 用户，发现 ENhanced Scala Interaction Mode（ensime）设置对开发很有帮助。你可以从 https://github.com/aemoncannon/ensime/downloads 安装最新版的 ensime（要求下载的版本号与 Scala 版本相匹配）。

```
wget https://github.com/downloads/aemoncannon/ensime/ensime_2.9.2-
0.9.8.1.tar.gz
tar -xvf ensime_2.9.2-0.9.8.1.tar.gz
```

在 .emacs 文件中，添加：

```
;; Load the ensime lisp code...
(add-to-list 'load-path "ENSIME_ROOT/elisp/")
(require 'ensime);;
This step causes the ensime-mode to be started whenever
;; scala-mode is started for a buffer. You may have to customize this
step
;; if you're not using the standard scala mode.
(add-hook 'scala-mode-hook 'ensime-scala-mode-hook)
```

然后添加 ensime sbt 插件到工程中（在 project/plugins.sbt）：

```
addSbtPlugin("org.ensime" % "ensime-sbt-cmd" % "0.1.0")
```

然后运行插件：

```
sbt
> ensime generate
```

如果使用了 git，还可以把 .ensime 添加到 .gitignore 文件中。

如果使用了 IntelliJ（一个类似于 sbt-idea 的插件，可以用于自动生成 IntelliJ IDEA 文件），也可以把 IntelliJ sbt 插件添加到项目中（在 project/plugins.sbt）：

```
addSbtPlugin("com.github.mpeltonen" % "sbt-idea" % "1.5.1")
```

然后运行插件：

```
sbt
> gen-idea
```

这会生成可以由 IntelliJ 加载的 IDEA 工程文件。

Eclipse 用户也可以用 sbt 来生成 Eclipse 工程文件，具体是用 sbteclipse 插件。可以把这个插件也添加到项目中（在 project/plugin.sbt）：

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" %
"2.3.0")
```

然后运行插件：

```
sbt
> eclipse
```

这会生成 Eclipse 项目文件，可以通过 Eclipse 的项目导入向导把它们导入到 Eclipse 中。另外，spark-plug 项目对 Eclipse 用户来说也很有用，它支持从 Eclipse 中直接启动 Spark 集群。

9.6 Spark 与其他语言

如果需要用 Scala 之外的语言来使用 RDD，有如下几个选择：Java 可以使用 JNI，Python 可以使用 FFI。除此之外，可能只有编译好的二进制程序，或者某种非 C 的语言。对于这种情况，最简单的做法是用 Java/Scala/Python API 提供的管道接口，其工作原理是先读取 RDD，然后序列化成字符串，最后通过管道推送给目标程序。对于纯字符串类的数据，这种方式是很合适的；对于其

他类型的数据，需要采用两边都能理解的序列化协议。如果数据具有一定的结构化特征，采用 JSON 或者 Protocol Buffer 作为协议是不错的选择。

9.7 安全提示

安全是 Spark 设置中需要重点考虑的问题。如果使用默认脚本在 EC2 上运行 Spark，会发现访问 Spark 集群时会受限。这种安全策略很好，即使不是在 EC2 上，因为一般来说，运行在 Spark 上的数据都是不应该公开给外部访问的，你也不乐意任何陌生人能在自己的集群上运行任意代码。如果 Spark 集群位于一个私有网络中，那是最好了；如果不是，则应该让系统管理员设置一些 IP 规则来限制访问。

9.8 邮件列表

作为技巧篇的结尾，也许最有用的一个小提示就是把 Spark 邮件列表充分利用起来，那里有其他人关于 Spark 的最新经验。通过下面的地址订阅 Spark 邮件列表：https://groups.google.com/forum/?fromgroups#!forum/spark-users（不久会迁移到 http://mail-archives. apache.org/mod_mbox/incubator-spark-user/）。遇到问题的时候可以搜一下邮件列表里的历史邮件，也许其他人已经遇到过相似的问题。

9.9 链接和参考

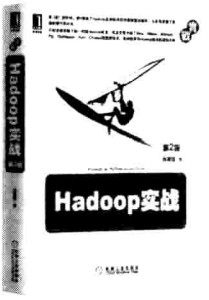
下面列出一些有用的链接和参考：

- http://blog.quantifind.com/posts/logging-post/
- http://jawher.net/2011/01/17/scala-development-environment-emacs-sbt-ensime/
- https://www.assembla.com/spaces/liftweb/wiki/Emacs-ENSIME
- http://syndeticlogic.net/?p=311
- http://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf
- https://github.com/shivaram/spark-ec2/blob/master/ganglia/init.sh
- http://spark-project.org/docs/0.7.2/tuning.html
- https://github.com/mesos/spark/blob/master/docs/configuration.md
- http://kryo.googlecode.com/svn/api/v2/index.html
- https://code.google.com/p/kryo/
- http://scala-ide.org/download/current.html
- http://syndeticlogic.net/?p=311
- http://mail-archives.apache.org/mod_mbox/incubator-spark-user/
- https://groups.google.com/forum/?fromgroups#!forum/spark-users

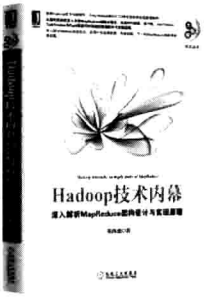
9.10 小结

本章总结了一些常见的内容，有助于提升 Spark 开发的经验。祝你在 Spark 项目中好运！现在，去解决一些有意思的问题吧！

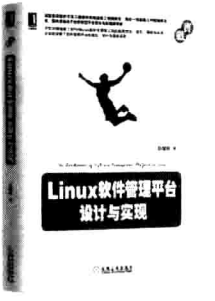
推荐阅读


Hadoop实战
第2版

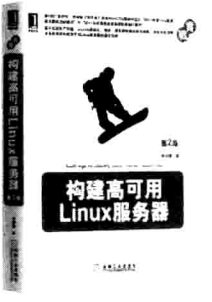

Hadoop技术内幕
深入解析Hadoop Common和
HDFS架构设计与实现原理

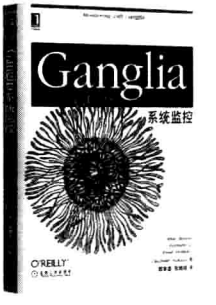

Hadoop技术内幕
深入解析MapReduce架构设计与实现原理


Hadoop技术内幕
深入解析YARN架构设计与实现原理


Linux软件管理平台
设计与实现

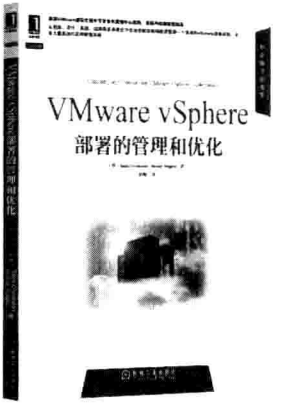

高性能Linux服务器构建实战
运维监控、性能调优与集群应用


构建高可用
Linux服务器


Ganglia
系统监控
O'REILLY

推荐阅读


vSphere 5.0
存储设计与实现


VMware vSphere
部署的管理和优化


VMware、Citrix和Microsoft
虚拟化技术详解与应用实践


虚拟化技术指南

本书将带领你踏上用Spark编写分布式Map/Reduce程序之旅，从搭建集群、交互式使用API到在集群上部署Spark作业，一步步引导你写出高效的分布式应用程序。本书涵盖广泛，包括在各种环境下搭建Spark集群（Standalone、EC2等）、使用交互式Shell编写分布式代码，并且书中还穿插讲述了如何使用Java、Scala和Python编写并部署分布式作业，如何使用交互式Shell快速编写分布式原型程序，以及如何使用Spark API。此外，本书还详细介绍了如何操作RDD，怎样结合Spark和Hive，使用Shark编写类SQL查询。

本书主要内容：

- 如何用Spark Shell快速开发原型分布式程序
- 与Spark分布式数据（RDD）进行交互的多种方法
- 如何从各种数据源加载数据
- 如何使用类SQL语句查询Spark
- 如何集成Spark程序和Shark查询
- 如何高效测试分布式程序
- Spark安装调优方法
- 如何在你的集群上安装及构建Spark系统
- 如何高效处理大数据集

[PACKT] PUBLISHING

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/大数据

ISBN 978-7-111-46311-5

9 787111 463115 >

定价：29.00元