Programming Pig

Pig编程指南

[美] *Alan Gates* 著
曹坤 译

O'REILLY®
人民邮电出版社
POSTS & TELECOM PRESS

O'REILLY®

Pig 编程指南

[美] Alan Gates 著

曹坤 译

人 民 邮 电 出 版 社

北 京

图书在版编目（CIP）数据

Pig编程指南 /（美）盖茨（Gates,A.）著 ；曹坤译
. -- 北京 ：人民邮电出版社，2013.2
ISBN 978-7-115-30111-6

Ⅰ. ①P… Ⅱ. ①盖… ②曹… Ⅲ. ①程序语言－程序设计－指南 Ⅳ. ①TP312-62

中国版本图书馆CIP数据核字(2012)第278366号

版权声明

Pig 编程指南

- 著　[美] Alan Gates
  译　曹　坤
  责任编辑　汪　振
- 人民邮电出版社出版发行　北京市崇文区夕照寺街 14 号
  邮编　100061　电子邮件　315@ptpress.com.cn
  网址　http://www.ptpress.com.cn
  三河市海波印务有限公司印刷
- 开本：787×1000　1/16
  印张：13
  字数：262 千字　2013 年 2 月第 1 版
  印数：1–3 000 册　2013 年 2 月河北第 1 次印刷

著作权合同登记号　图字：01-2012-6404 号
ISBN 978-7-115-30111-6

定价：49.00 元

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154
广告经营许可证：京崇工商广字第 0021 号

内 容 提 要

本书不仅为初学者讲解 Apache Pig 的基础知识，同时也向有一定使用经验的高级用户介绍更加综合全面的 Pig 重要特性，如 PigLatin 脚本语言、控制台 shell 交互命令以及用于对 Pig 进行拓展的用户自定义函数（UDF）等。当读者有大数据处理需求时，本书提供了如何更高效地使用 Pig 来完成需求的方法。

本书适合各个层次的 Pig 用户及开发人员阅读使用。

谨以此书献给我的妻子 Barbara，以及我的儿子 Adam 和 Joel。

他们给予我支持和鼓励，并允许我占用许多个周末的时间投入写作，才使得本书最终成为可能。

前言

数据成瘾

数据是可以令人上瘾的。在过去数十年里我们收集和存储数据的能力已经大幅度提高。然而我们对于获取更多数据的欲望依旧没有得到满足。科学家们希望能够存储更多的数据以建立可以用来描述这个世界的更好的数学模型。市场营销者想通过数据了解他们的客户的意愿和消费习惯。金融分析师想通过数据更好地了解他们的市场运作情况。还有每个人都想保留各自的数码照片、电影、电子邮件等。

计算机和互联网的多次革命大大提高了我们收集和存储数据的能力。在这些革命之前，美国国会图书馆是世界上数据收集量最大的机构之一。曾经估算过它的印刷类书籍刊物包含的信息量大约能达到 10TB。如今大型互联网公司一天的数据量就可以达到这么多。同时并非只有网络程序正以惊人的速率产生着数据。例如,计划在智利建造的大口径全景巡天望远镜（LSST）预计将会每天产生 20TB 的数据。

导致数据这种海量增长的部分原因是我们具有收集更多数据的能力。每当用户在网站上点击网页，Web 服务器就会记录下该用户目前停留在哪个页面以及点击了哪些链接这样的信息。高速公路上每当一辆车驶过传感器的时候，车的行驶速度就会被记录下来。但是数据能够如此海量增长的更多原因是我们具备了海量数据存储的能力。十年前，望远镜每天晚上都会对星空拍摄很多的照片，但是这些照片并非全部都能以同一级别的分辨率存储，一些数据信息会被丢弃，因为根本没有足够的空间来存储它们。但是 LSST 正式运作后，这将不再是个问题。收集和存储海量数据的能力只是满足了我们的数据癖好。

在过去几十年里计算机系统中通常使用的数据存储和数据处理的工具之一就是关系型数据库管理系统（RDBMS）。但是随着数据集的不断增大，只有更高级的（也因此更加昂贵的）关系型数据库管理系统才能够达到目前很多用户所期望的标准。同时，许多涉及处理数据的工程师和科学家们意识到他们并不需要 RDBMS 提供的每一项功能。这些系统确实很强大而且功能很多，但是许多需要处理 TB 或者 PB 级别数据的用户只需要使用其中的一小部分功能。

RDBMS 的高成本和非必需的功能导致了一些非主流的数据处理系统的出现，Apache Hadoop 就是这些非主流的系统之一。Hadoop 是由 Doug Cutting 发起的一个开源项目。它是基于 Google 发表的多篇论文而实现的,这些论文描述了 Google 的工程师们是如何

对他们收集的海量数据进行存储和处理的。在过去的几年，雅虎和其他一些互联网公司共同推动了 Hadoop 的发展。如果想了解 Hadoop 的发展历史，请阅读 Tom White(O'Reilly 出版社出版)所著的 <<Hadoop: 编程指南>>一书。Hadoop 需要被部署到集群机器中，它可以提供一种利用集群力量进行存储和处理的方法。

像 Hadoop 这样的新型数据处理系统的发展促使了已经存在的工具和语言对 Hadoop 的移植，以及新工具（如 Apache Pig）的构造。像 Pig 这样的工具为数据使用者提供了一种较高层次的抽象，为使用者提供了一种可以使用到 Hadoop 强大的处理能力和灵活性的方式，而并不需要他们用低层 Java 代码来编写大量的数据处理程序。

谁应该读这本书

这本书是为 Pig 开发者准备的，无论是新手还是具有一定经验的开发者。之前从没有接触过 Pig 的人会在书中发现一些有用的入门性介绍，包括如何运行 Pig 以及怎么开始书写 Pig Latin 脚本。对于经验丰富的 Pig 使用者，这本书介绍了 Pig 几乎所有的功能：它可以从使用哪些不同的模式运行，Pig Latin 语言的详细介绍，以及如何在 Pig 中开发用户自定义函数（UDF）。甚至对于那些使用 Pig 很久的用户也可能会在这本书里发现之前他们从未使用过的功能。

作为一个相对年轻的项目，在过去的 4 年里 Pig 有了相当大的变化和成长。在这段时间里，我们从发布 0.1 版本一直到发布 0.9 版本。本书是以 Pig 0.7 版本为基础展开的。对于 0.8 或 0.9 版本中特有的功能，书中在介绍的时候会进行说明。0.6 版本到 0.7 版本的最大不同是对加载函数接口和存储函数接口这两个接口进行了重写，因此第 11 章的内容对于使用 0.6 或更低版本的用户是不适用的。当然,本书其他章节对于较早版本同样适用。

对于读者和 Pig 使用者来说，如果对 Hadoop 有所了解，那么阅读本书是有帮助的。附录 B 对 Hadoop 进行了简要的介绍而且说明了它是如何工作的。第 1.1.1 小节“Pig 是基于 Hadoop 的”提供了一个简单的 Hadoop 任务的例子。对于那些不熟悉 Hadoop 的用户,这几节介绍的内容将会有很大帮助。

本书中有几部分用到了 Java、Python 和 SQL 的程序片段。使用 Pig 并非要求一定要了解这些语言，但是对于 Pig 的一些高级功能还是需要对 Python 和 Java 有所了解。有使用 SQL 背景的读者可以在 1.1.2 小节下面的“查询语言和数据流语言的比较”这一小节看到 Pig Latin 和 SQL 的相似和不同之处。

本书中所使用的惯例

本书中使用到了如下几种印刷体：

斜体字

表明是新的术语、URL、电子邮件地址、文件名以及文件扩展名。

等宽字体

用于程序列表，以及段落中使用到的程序片段，例如变量或者函数名称、数据库、数据类型、环境变量、语句和关键字。

等宽粗体

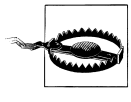
表示命令或者其他需要用户进行输入的文本。

等宽斜体

表示该文本需要用户提供对应的值替换或者需要通过上下文才能获取到的值。

提示

这个图标表明是一个小技巧、建议或者一般性的注释。



警告

这个图标表明是个警告或者警示。

本书中的代码样例

本书中使用到的样例脚本、用户自定义函数（UDF）和数据集大多可以从我的 GitHub 资源库中下载。资源库中也包含了一些 *README* 文档，这些文档可以指导用户如何编译 UDF 并且帮助用户理解数据文件中的内容。文本中的每一个可以在 GitHub 中获取到的样例脚本，在开始的时候注释内容中都包含了文件名。Pig Latin 样例脚本和 Python 样例脚本在 *examples* 目录下按章分类存放。用户自定义函数（UDF），包括使用 Java 编写的和使用 Python 编写的都放在另外一个目录 *udfs* 下。所有的数据集都放在 *data* 目录下。

为简明起见，每一个脚本都假定输入和输出都是本地文件目录。因此，当使用本地模式运行时，用户应该在需要输入的数据文件所在的目录下执行 Pig 脚本。而当在集群上运行时，用户需要将输入的数据文件放置在集群文件系统的用户根目录下。

本书提供的样例脚本都是在 Pig 0.8.0 版本或 0.8.1 版本下测试通过的，当然除了一些使用到 0.9 版本中新增功能的脚本，这些脚本是在 0.9 分支版本下测试通过的，因为在写

本书的时候 Pig 还没正式发布 0.9 版本（译者注：Pig 0.9.2 发行版对于这些新增功能也是适用的）。

样例中所使用的 3 个数据集虽然都相当小，但它们都是真实的数据集。文件名为 *baseball* 的文件包含的是棒球运动员的统计信息。第二个数据集是纽约股票交易数据，有两个文件，文件名分别是：*NYSE_daily* 和 *NYSE_dividends*。这份数据被简化为只包含 2009 年以后股票代码以 C 开头的数据信息，这是为了使数据量足够小以方便用户下载。当然，数据的模式并没有改变。如果用户想下载完整的数据集然后把它放到集群中（只需要几个节点就可以），那将是一个对 Pig 和 Hadoop 更加逼真的演示。在 *README* 文件中有如何下载这些数据集的说明。第三个数据集是从 Pig 官网的网页抓取的简单爬虫信息。

使用代码样例

本书的目的是帮助用户完成他们的任务。通常情况下，用户可以在自己的程序和文档中使用书中的代码。这时不需要联系我们以获取许可，除非有明显复制了代码的大部分内容的时候。例如，写程序用到了本书中几个代码片段是不需要获得许可的，但是如果销售或者传播包含了 O'Reilly 系列书籍中的例子的 CD 光盘，那么就一定要获得我们的许可才行。在回答问题时引用到本书或以本书中的例子为引证时不需要获得许可。合并书中大量样例代码到用户的产品文档中时一定需要获得我们的许可才行。

虽然并非是必需的，但如果可以注明出处，我们将十分感激。出处一般包括书名、作者、出版商和 ISBN。例如："*Programming Pig* by Alan Gates (O'Reilly).Copyright 2011 Yahoo!, Inc., 978-1-449-30264-1."

如果读者感觉到没有合理地或者在如上所述的许可范围内使用本书中代码样例的话，请尽管通过 *permissions@oreilly.com* 联系我们。

Safari®图书在线

Safari 图书在线是一个按需服务的数字图书馆。使用它用户可以轻松地通过检索超过 7,500 本技术和创意参考书以及视频教程从而快速获得用户想知道的答案。

通过订阅，用户可以从我们的在线图书馆中阅读每一篇文章和观看每一段视频，还可以通过手机和其他移动设备看书。在书还没有印刷前就可以事先看到书目，还可以独享看到正在进行中的书籍草稿并将意见反馈给作者。复制粘贴代码样例，组织用户的收藏夹，下载一些章节，对关键章节做标记，创建笔记，打印书籍内容，还有可以因其他众多的省时功能而受益。

O'Reilly Media 已经将本书上传到 Safari 图书在线服务了。想获得对这本书的完整数据

版访问权限以及来源于 O’Reilly 和其他出版商的对相同话题的讨论，请通过网址 http://my.safaribooksonline.com 免费注册账户。

如何联系我们

请将对于本书的评论和问题通过如下联系方式发送给出版商：

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

本书有一个 Web 页面，在此页面上我们列举了勘误表、例子和其他附加信息。用户可以通过如下链接访问这个页面：

http://shop.oreilly.com/product/0636920018087.do

可以通过如下 E-mail 地址将你对本书的评论或一些技术方面的问题发送给我们：

bookquestions@oreilly.com

想获得更多的关于我们的系列书籍、会议、资源中心和 O’Reilly 网络的信息，请通过如下网站查看：

http://www.oreilly.com

鸣谢

一本书就像一支职业橄榄球队，大部分的荣耀都给了四分卫或跑卫。但是如果这个队防线很差，那么四分卫就不会有机会将球扔出去。接球员必须有能力接住球，防卫必须有能力阻止其他队得分。简而言之，整个队伍必须通力合作才能取胜。在这些背后在场地上还有一批教练、训练员以及准备和引导整个队伍的经理。这本书的形式也是同样的道理。书的封面上写的是我的名字，但是如果没有这群由开发者、研究者、测试者、文档书写者以及对 Pig 项目有贡献的人组成的了不起的团队，那将没什么有价值的信息来写成本书。

特别地，我想感谢 Pig 开发贡献者和使用者对于本书的贡献和反馈。Chris Olston、Ben Reed、Richard Ding、Olga Natkovitch、Thejas Nair、Daniel Dai 和 Dmitriy Ryaboy 对本书一遍又一遍的草稿都提供了非常有帮助的反馈。Julien Le Dem 提供了在 Python 中嵌

入使用 Pig 的代码样例。Jeremy Hanna 书写了 Pig 和 Cassandra 那个章节。她的反馈使本书的讲解更加清楚和正确。

还要感谢 Tom White 在我下定决心写这本书时对我的鼓励，以及对于写这本书所需要的时间和努力所给予的理性的提醒。Hadoop 项目的 Chris Douglas 在涉及 Hadoop 和 MapReduce 的章节给了我非常有帮助的反馈。

我还想感谢 Mike Loukides 和在 O’Reilly 的整个团队。他们让我觉得在写第一本书时是一段快乐的和令人愉悦的经历。最后，感谢 Yahoo!培育了 Pig 以及投入了超过 25 个工程年（同时还在增加）的努力，同时让我有时间来写这本书。

目录

第 1 章

初识 Pig

1.1 Pig 是什么？

Pig 提供了一个基于 Hadoop 的并行地执行数据流处理的引擎。它包含了一种脚本语言，称为 Pig Latin，用来描述这些数据流。Pig Latin 本身提供了许多传统的数据操作（如 join、sort、filter 等），同时允许用户自己开发一些自定义函数用来读取、处理和写数据。

Pig 是一个 Apache 开源项目。这意味着用户可以免费下载源码或者二进制包，自由使用它，对这个项目贡献自己的代码，同时也可以在 Apache License 的许可范围下将 Pig 用到自己的产品中或者在需要的时候修改代码来满足特定需求。

1.1.1 Pig 是基于 Hadoop 的

Pig 运行于 Hadoop 之上，它同时使用到 Hadoop 分布式文件系统 *HDFS* 和 Hadoop 处理系统 *MapReduce*。

HDFS 是一个分布式文件系统，它将文件存储到 Hadoop 集群的各个节点上。它负责将文件分割成许多数据块然后分发到不同的节点机器上，其中包括对每个数据块进行多份冗余备份，这样可以避免因为某台机器宕掉而造成数据丢失。HDFS 提供了一种类似 POSIX 的用户交互形式给用户。默认情况下，Pig 从 HDFS 中读取输入文件，使用 HDFS 来存放 MapReduce 任务所生成的中间数据，最终将输出写入 HDFS 中。在第 11 章，用户将看到 Pig 不只是可以从 HDFS 中读取输入文件或将输出文件写入 HDFS 的。

MapReduce 是一个简单而强大的并行数据处理算法。MapReduce 计算框架下的每个任务都由 3 个主要阶段组成：map 阶段、shuffle 阶段和 reduce 阶段。在 map 阶段，程序可以并行独立操作输入数据中的每一条记录。因为可以同时运行多个 map 任务，所以

即使输入的数据量达到 GB 或者 TB 级别，只要有足够多的机器，map 阶段通常在 1 分钟内就可以完成。

MapReduce 任务的一个特别之处在于需要确定数据是根据哪个键进行收集的。例如，假设用户在处理一个网站的 Web 服务器日志，而且这个网站需要用户登录后才能操作，那么用户就可能会使用用户 ID 作为数据收集的键，因为通过这个用户 ID 就可以知道每个用户在这个网站上的对应的所有操作。map 阶段后紧跟着就是 shuffle 阶段，在这个阶段数据已经根据用户指定的键收集起来并且分发到不同的机器上去了，这是为 reduce 阶段做准备。包含同一键的所有记录将会交由同一个 reducer 处理。

在 reduce 阶段，程序将提取每个键以及包含该键的所有记录。这个过程也是在多台机器上并行执行完成的。当处理完所有组时，reducer 就可以写输出了。下面我们将通过一个简单的 MapReduce 程序进行演示。想更多地了解 MapReduce 是如何工作的，请看附录 B“MapReduce 介绍”。

MapReduce 演示程序

假设现在有一个 MapReduce 程序对一个文本文件进行词频统计。该程序本身是 MapReduce 提供的演示程序。在这个例子中，map 阶段会从文本文件中一次读取一行，然后分割出每个词作为一个字符串，之后对于分割出的每个单词，会输出单词本身以及数字 1，数字 1 表示这个单词出现过 1 次。在 shuffle 阶段，将使用单词作为键，哈希分发对应的记录到不同的 reducer 中去。在 reduce 阶段会将相同的单词对应的出现次数相加，并最终将求和后的数值和单词本身一起输出。以童谣“Mary Had a Little Lamb”为例，输入将是：

```
Mary had a little lamb
its fleece was white as snow
and everywhere that Mary went
the lamb was sure to go.
```

这里假设每一行都被发送到不同的 map 任务中去了。当然事实上，每个 map 任务处理的数据要远远大于这个数量，这里只是为了后面更好地去描述。MapReduce 整个过程的数据流如图 1-1 所示。

map 阶段一旦结束，shuffle 阶段将会把包含相同单词的所有记录提交到同一个 reducer 中。对于这个例子我们假设有两个 reducer:以 A～L 开头的单词提交到第一个 reducer 中，而以 M～Z 开头的单词提交到第二个 reducer 中。这两个 reducer 最终将会把每个单词的出现次数分别相加然后输出。

Pig 的所有数据处理过程都是使用 MapReduce 来执行的。Pig 将用户所写的 Pig Latin 脚本编译成一个或者多个 MapReduce 任务，然后在 Hadoop 上执行。例子 1-1 展示了如何使用 Pig Latin 脚本来对童谣“Mary Had a Little Lamb”进行词频统计。

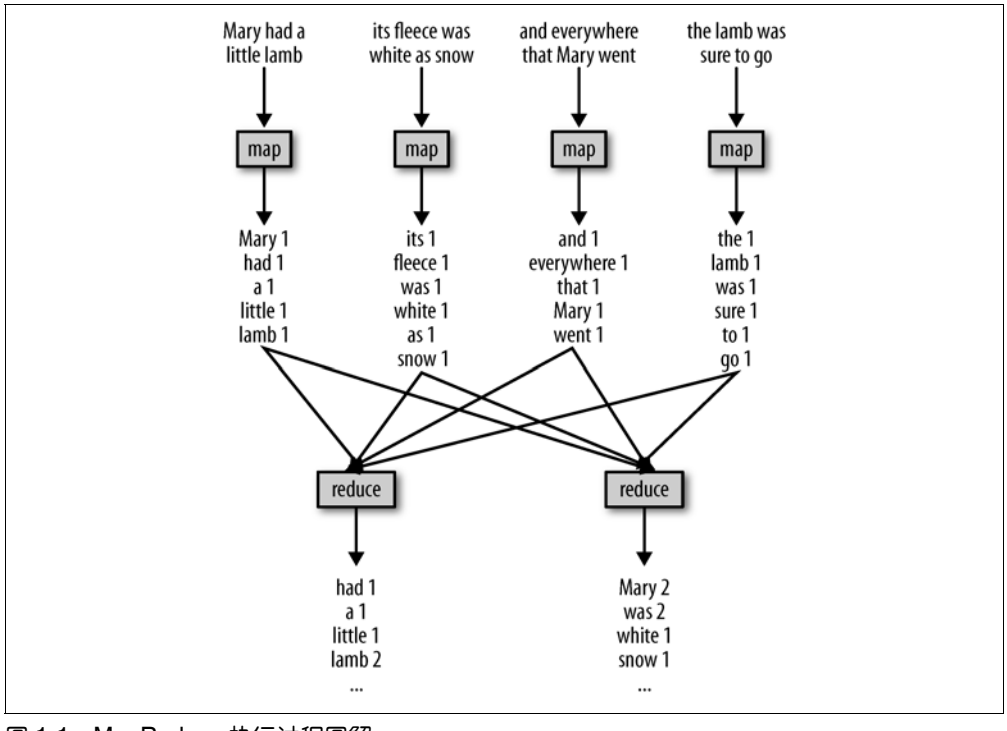


图 1-1　MapReduce 执行过程图解

例 1-1　使用 Pig 对童谣“Mary 和她的羔羊”进行词频统计

```
--加载文件名为 Mary 的文件，
--并将记录中的唯一字段命名为 'line'。
input = load 'mary' as (line);

--TOKENIZE 将 line 按单词分割成列
--flatten 接受 TOKENIZE 操作后产生的记录集合然后分开成独立的列，
-- 这个独立的列称为 word
words = foreach input generate flatten(TOKENIZE(line)) as word;

-- 现在按照 word 进行分组
grpd = group words by word;

-- 计数
cntd = foreach grpd gengerate group, COUNT(words);
-- 打印结果
dump cntd;
```

在使用 Pig 时无须去过度关注 map、shuffle 和 reduce 阶段，因为 Pig 会将脚本中的操作解析成相应的 MapReduce 阶段。

1.1.2 Pig Latin，一种并行数据流语言

Pig Latin 是一种数据流语言，这意味着它允许用户去描述如何从一个或多个数据源并行读取数据，然后并行地进行处理，最后将处理结果并行地输出到一个或多个结果集中。这些数据流可以像前面提到的那个词频统计例子一样是个简单的线性流。同时它们也可以是复杂的工作流，其中可以包含一些加入多个输入的节点，也可以包含一些将输入数据分割成多个流的节点，这些节点都是通过不同的操作符来处理的。用数学语言来描述的话，Pig Latin 描述的是一个有向无环图（DAG），在这个图中，节点代表处理数据的操作符，节点间的向量代表数据流。

这意味着 Pig Latin 和用户之前见过的许多编程语言会有所不同。在 Pig Latin 中没有 **if** 语句，也没有 for 循环操作。这是因为传统的过程语言和面向对象语言描述的是控制流，而数据流只处于一个从属地位。而 Pig Latin 更专注于数据流。想了解 Pig Latin 脚本中如何在处理数据流的同时加入控制流，请阅读第 9 章。

查询语言和数据流语言的比较

大体一瞥，人们会说 Pig Latin 不过是 SQL 的一个面向过程化的版本。尽管确实有一定的相似性，但是其实两者具有非常多的差异。SQL 是一种查询语言，它关注于允许用户构造查询，它允许用户去描述他们想得到什么问题的答案，而不是如何给出问题的答案。然而在 Pig Latin 中，用户可以详细描述如何对输入的数据进行处理。

Pig Latin 和 SQL 的另一个主要区别是 SQL 面向的是回答一个问题，因此当用户想同时进行多个数据操作时，他们要么使用多个查询语句，这时需要将一些查询的中间数据存放到临时表中；要么写一个大的包含子查询的查询语句，将一些初始的处理过程由子查询来完成。然而，很多用户发现子查询令人困惑而且也并非那么容易去构建。同时，子查询使用的是由内而外的设计，也就是说，在数据管道最里面的子查询会最先执行。

Pig 被设计为实现知道将要进行的一系列的数据操作，因此不需要通过颠倒顺序的子查询的方式来写数据管道，也无需使用临时表来存放中间数据。这点将通过例子 1-2 和例子 1-3 来进行演示。

现在假设有个用户想先按某个键对表进行 group 分组操作，然后和第二张表进行 join 连接操作。在 SQL 查询中，因为 join 操作发生在 group 操作之后，所以要么使用子查询，要么写两个查询语句，同时将中间结果保存到临时表中。例子 1-3 用到了一个临时表，因为这样可读性要好些。

例 1-2　SQL 中先进行分组然后进行连接操作

```
CREATE TEMP TABLE t1 AS
SELECT customer, sum(purchase) AS total_purchases
FROM transactions
```

```
GROUP BY customer;

SELECT customer, total_purchases, zipcode
FROM t1, customer_profile
WHERE t1.customer = customer_profile.customer;
```

在 Pig Latin 中，是另外一种方式，如例 1-3 所示。

例 1-3　Pig Latin 中先进行分组然后进行连接操作

```
-- 加载汇报文件，按照 customer 字段进行分组，然后计算他们的总购物金额
txns = load 'transactions' as (customer, purchase);
grouped = group txnx by customer;
total = foreach grouped generate group, SUM(txns.purchase) as tp;
-- 加载 customer_profile 文件
profile = load 'customer_profile'as (customer, zipcode);
-- 对已经分好组并进行了累加计算的汇报文件数据和 customer_profile 文件进行连接
answer = join total by group, profile by customer;
-- 将结果输出到控制台
dump answer;
```

此外，SQL 和 Pig Latin 各因不同的应用场景而生。SQL 的应用场景是 RDBMS，在这种场景下，数据是标准化的，并且加上了模式和其他一些特有的约束（例如，null 值也是不可以脱离约束单独存在的等）。Pig 是为 Hadoop 数据处理环境而设计的，在这种环境下，模式有时是未知的或不一致的，数据可能没有进行恰当的约束而且很少进行数据标准化。基于这些不同，Pig 不需要将数据事先导入表中，当数据导入 HDFS 中后，它就可以直接操作这些存放在 HDFS 的数据。

如果语言和文化类似，那么融入一个新的环境可能会更加容易些。我和妻子一起去过法国几次。我会讲很少的法语，但是因为是商业语言（或许是因为美国人和大不列颠人喜欢到法国度假），对于我来说法语中已经包含了足够多的英语口语，使我足够应付得了。而我的妻子，她会讲法语。她在法国有朋友去拜访时，她可以和他们很好地交谈。她可以去那些不在通常的旅游线路上的其他景区探险。她的法国经历比我要多得多，因为她会讲当地本土语言——法语。

在数据处理范畴里，SQL 就是英语。它有个非常好的特点就是无论是人还是工具都认识它，也就是说它的入门门槛很低。我们的目标是使 Pig 成为像 Hadoop 这样的并行数据处理系统范畴里的母语。尽管这可能要求用户需要进行一定的学习才能使用，但是它可以让用户更加充分地利用 Hadoop 提供的计算能力。

Pig 和 MapReduce 的区别是什么

我刚刚声明 Pig 团队的一个目标是使 Pig Latin 成为像 Hadoop 这样的并行数据处理环境的母语。但是难道 MapReduce 提供的还不够吗？有必要使用 Pig 吗？

Pig 比直接使用 MapReduce 有几个优点。Pig Latin 提供了所有标准的数据处理操作，

例如 join、filter、group by、order by、union 等。MapReduce 直接提供了 group by 操作（也就是 shuffle 和 reduce 两个阶段做的事情），同时通过实现分组操作间接地提供了 order by 操作。过滤器操作和推测执行操作可以在 map 阶段进行简单实现。但是其他的操作，特别是 join 操作无法提供，所以必须由用户自己进行代码实现。

Pig 提供了一些对这些标准的数据操作的复杂的、完备的实现。例如，因为每个键对应的记录的个数很少是均匀地分布在集群中的，所以提交给 reducer 的数据经常会产生数据倾斜。也就是说，有的 reducer 需要比别的 reducer 处理 10 倍或更多倍的数据。Pig 具有 join 和 order by 操作可以处理这种情况，而且（在一些情况下）可以重新均衡 reducer 负荷。这些需要 Pig 团队花费好几个月的时间编写 MapReduce 程序，然后再重构代码，这确实耗费时间。

在 MapReduce 中，在 map 阶段和 reduce 阶段的内部的数据处理对于系统来说是不透明的。这意味着 MapReduce 没有机会优化或者检查用户的代码。Pig 另一方面，可以通过分析 Pig Latin 脚本来了解用户描述的数据流。这意味着 Pig 可以在早期进行错误检查（例如用户是否将一个 string 类型的字段放到一个 integer 类型的字段中？）和进行优化（例如这两个 group 操作是否可以合并？）

MapReduce 没有一个类型系统，这是有意这么设计的，因为这样可以给用户更大的自由度去使用他们自己的数据类型和序列化框架。但这样就产生了一个不好的问题，就是限制了系统在运行前和运行时对用户代码进行检查的能力。

这几个方面都表明 Pig Latin 相对于 MapReduce Java 代码更容易编写和维护。我做了一个并非科学的实验，对于同一个操作我分别使用 Pig Latin 和 MapReduce 进行实现。假设有个文件存有用户数据，另一文件存放了对于某个网站的点击数据，例子 1-4 所示的 Pig Latin 脚本将找到年龄为 18～25 岁的用户访问最多的 5 个页面。

例 1-4　查找访问次数最多的前 5 个 URL

```
Users = load 'users' as (name, age);
Fltrd = filter Users by age >= 18 and age <=25;
Pages = load 'pages' as (user, url);
Jnd   = join Fltrd by name, Pages by user;
Grpd  = group Jnd by url;
Smmd  = foreach Grpd generate group, COUNT(Jnd) as clicks;
Srtd  = order Smmd by clicks desc;
Top5  = limit Srtd 5;
store Top5 into 'top5sites';
```

这段脚本的第 1 行表示加载文件名为 *users* 的文件，同时声明这份数据有两个字段：**name** 和 **age**，而且为这个输入取别名为 **Users**。第 2 行是个过滤器，将 **Users** 中 **age** 这个字段值大于等于 18 而且小于等于 25 的记录过滤出来，不满足条件的数据将被忽略。经过过滤器后，留下的数据就是在我们感兴趣的年龄范围内的了。我们将这个过滤器

的结果取别名为 **Fltrd**。

第 3 行是第 2 个 **load** 加载数据语句，这个语句加载了文件 *Pages*，并取别名为 **Pages**，它声明了两个字段：**user** 和 **url**。

“**Jnd = join**”这一行以 **Fltrd.name** 和 **Pages.user** 为键，对 **Fltrd** 和 **Pages** 进行 join 连接操作。通过这次 join 操作我们就可以得到每个用户访问过的所有 URL 链接了。

“**Grpd = group**”这一行按照 URL 进行分组。结果是每一个 url，例如 **pignews.com/frontpage**，都对应着一组 **url** 字段中包含了对应值的所有记录。紧跟着的下一行会统计每个 URL 对应的记录个数。这一行后我们就知道了，对于每个 URL，被年龄为 18～25 岁的用户访问了多少次。

之后的一件事就是按访问次数从访问最多到访问最少进行排序。“**Srtd = order**”这一行就是根据前一行的统计结果进行 **desc**（降序）排列。因此，最大值将在第 1 行。因为最终我们只需要最前面的 5 条记录，所以最后一行将统计结果限制在前 5 行。最后的结果重新存放到 HDFS 中一个叫做 *top5sites* 的文件中。

在 Pig Latin 中整个处理过程需要写 9 行代码，耗时在 15 分钟左右，其中包括写代码和对代码进行调试的时间。如果以 MapReduce（这里省略了）来写的话，需要差不多 170 行的代码而且花费了我 4 个小时的时间才调试成功。Pig Latin 同样利于维护，因为这段代码，对于后来的其他开发者同样是容易理解和方便修改的。

当然 Pig 所带来的这些便利同样是有代价的。通过 MapReduce 框架可以开发一些算法，在 Pig 中却很难实现。同时对于开发者，他们需要放弃一个层次的控制权。一名优秀的工程师，只有给予足够的时间，总是可以将一个普通的系统做得足够好。因此对于不常见的算法或者是对于性能要求很高的话，这种情况下使用 MapReduce 仍然是正确的选择。基本上这种情况也和选择 Java 编码而不选择使用像 Python 这样的脚本语言的情形是一样的。Java 功能强大，但是因为它是高级程序语言，所以使用它开发需要比脚本语言花费更多的时间。开发者需要根据实际情况选择合适的工具。

1.1.3 Pig 的用途

以我的经验，Pig Latin 的使用场景可以分为独立的三大类：传统的抽取转换加载（ETL）数据流、原生数据研究和迭代处理。

最大的使用场景就是数据流了。一个通常的例子就是网络公司从他们的 Web 服务器上收集到日志，进行数据清洗，之后进行简单的聚合预计算，然后导入数据仓库中。在这种情况下，数据被加载到计算网格中，之后使用 Pig 从数据泥潭中清理出有价值的数据。同时还可以使用 Pig 将用户网页操作数据和用户数据库信息进行 join 连接，这样可以将用户 cookie 和已知的用户信息关联起来。

另外一个数据流应用的例子是使用 Pig 处理离线数据来建立用户行为预测模型。Pig 被

用来扫描所有的用户和网站的交互数据，最终将用户分为各种各样的群组。然后，对于每个群组会生成一个数学模型，根据该模型可以预知这个群组的用户对各种类型的广告或者新闻文章的反映是什么样子的。通过这种方式，网站可以知道展示什么样的广告可以更有可能获得更多的点击，或者发布什么样的新闻故事可以更有可能吸引用户和挽留用户再次访问。

传统上，使用像 SQL 这样的语言执行点对点的查询可以快速地为问题准备好相应的数据。然而，对于原始数据的研究，一些用户还是偏向使用 Pig Latin 脚本。因为 Pig 可以在无模式，模式信息不全，或者模式不一致的情况下进行操作，同时因为 Pig 可以很容易地控制封装的数据，因此对于那些期望在数据没有进行清洗也没有写入数据仓库的情况下，分析数据的研究人员经常更偏好于使用 Pig。经常处理大规模数据集的研究人员经常会使用像 Perl 或者 Python 这样的脚本语言进行处理。具有这些使用背景的用户通常更喜欢使用 Pig 这样的数据流范式而非像 SQL 那样的声明式查询语言。

创建迭代处理模型的用户也开始使用 Pig。假设有一个新闻门户网站，它保留了一个它跟踪的关于该网站的所有新闻故事的图。在这个图中每个新闻故事都是一个节点，节点间的连线表示的是相关故事间的关系。例如，所有关于即将来临的选举的故事都是联系到一起的。每 5 分钟都有一组新的故事进来，这时数据处理引擎需要将这组故事增加到图中。这些故事中有一些是新的，有一些是对之前的故事进行的更新，还有一些是替代之前已经存储的一些故事的。这时需要对整个故事图做一些数据处理步骤。例如，对于建立行为目的模型的处理过程就需要将用户数据和整个故事图进行连接。每 5 分钟重新运行整个图是不可行的，因为对于适当数量的硬件资源来说在 5 分钟内运行出结果是不可能的。但是模型创建者不想只是每天更新一次这些模型，因为那意味着会错过一整天的时间来提供机会。

为了应付这个问题，有必要定期地首先对整个图进行连接，例如可以按照天来进行连接。然后，每 5 分钟一旦有数据进来，就可以立即完成对新进来的数据进行连接操作，同时这个结果是可以和对整个图做连接的结果整合在一起的。这个组合步骤并不容易，因为需要在 5 分钟内完成对整个图进行插入、更新和删除操作。使用 Pig Latin 来表达这种组合关系是可以的并且是相当方便的。

目前所说的一切都隐含着一点：Pig（与 MapReduce 一样）是面向数据批处理的。如果需要处理的是 GB 或者 TB 数量级的数据，那么 Pig 是个不错的选择。但是因为它期望的是序列地读取一个文件中的所有记录然后序列地将输出写入存储中，因此对于那些需要写单条或者少量记录，或者查询随机序列下的多条不同记录这样的任务，Pig（与 MapReduce 一样）并非是个好选择。关于在这些情况下选用什么样的软件才是合理的更多讨论请查看第 12.3 节“NoSQL 数据库”。

1.1.4 Pig 的设计思想

在早期，作为潜在贡献者加入 Pig 项目的人们并非了解这个项目究竟是关于什么的。他

们并不清楚怎样做才是最好的贡献或者哪些贡献会被接受以及哪些不会被接受。因此，Pig 团队发布了一个项目设计思想声明，其内容总结为 Pig 渴望成为：

Pig 什么都吃

不管数据是否有元数据，Pig 都可以操作。不管数据是关系型的、嵌套型的，或者是非结构化的，Pig 也同样可以操作。而且它还可以很容易地通过扩展，不单单可以操作文件，还可以操作 key/value 型的存储，以及数据库等。

Pig 无处不在

Pig 期望成为一种并行数据处理语言。它不会局限于是一种特殊的并行处理框架。它首先是基于 Hadoop 之上的实现，但是我们期望它并非只能在 Hadoop 平台上使用。

Pig 是家畜

Pig 被设计为可以让用户很容易地控制和修改的语言。

Pig 允许用户随时整合加入他们的代码，因此目前它支持用户自定义字段类型转换函数、用户自定义聚合方法函数和用户定义条件式函数。这些函数可以使用 Java 来写也可以使用最终可以编译成 Java 代码的脚本语言（例如 Jython）编写。Pig 支持用户定义的加载和存储函数。Pig 通过自己的 **stream** 命令和需要 MapReduce 相关的 JAR 包的 **mapreduce** 命令可以执行外部的执行命令。Pig 同样允许用户为自己的特定使用场景提供一个用户自定义的分区方法函数，使他们执行的任务在 reduce 阶段可以达到一个均衡的负荷。

Pig 有一个优化器，它可以将 Pig Latin 脚本中的操作过程进行重新排列以达到更好的性能，例如将 MapReduce 任务进行合并等。但是，如果对于某种情形下这种优化是不必要的话，用户可以很容易地将最优控制器关闭，这样执行过程就不会发生改变。

Pig 会飞

Pig 处理数据很快。我们会持续地优化性能，同时不会增加一些使 Pig 显得较重而降低性能的新功能。

1.2 Pig 发展简史

Pig 最初是作为 Yahoo！的一个探索性的项目，Yahoo!的科学家们设计了 Pig 并且给出了一个原型实现。正如 2008 年发表在数据管理专业委员会（SIGMOD[1]）杂志的一篇论文所描述的，研究者认为 Hadoop 所描述的 MapReduce 框架“过于底层和严格，需要

[1] Christopher Olston et al, “Pig Latin: A Not-So-Foreign Language for Data Processing,”可以通过如下链接获得 *http://portal.acm.org/citation.cfm?id=1376726*。

用户花费大量时间编写代码，而且很难维护和重用。”同时他们注意到 MapReduce 用户对像 SQL 这样的声明式语言并不熟悉。因此他们着手开发“一种叫做 Pig Latin 的新语言，这种语言被设计为在像 SQL 这样的声明式类型的语言和像 MapReduce 这种较底层的过程式的语言之间达到一个非常好的平衡点。”

最初 Yahoo!的 Hadoop 使用者开始采用 Pig。之后一个开发工程师团队开始接手 Pig 的最初原型并将 Pig 原型开发成一个达到产品级别的可用产品。在这个时间点左右，也就是 2007 年的秋天，Pig 通过 Apache 孵化器进行开源。一年后也就是 2008 年的 9 月，Pig 的第一个发布版本出现了。同年的晚些时候，Pig 从孵化器中毕业，正式提升为 Apache Hadoop 项目的一个子项目。

2009 年早期其他公司在他们的数据处理中开始使用 Pig。Amazon 也将 Pig 加入它的弹性 MapReduce 服务中的一部分。2009 年末，Yahoo！公司所运行的 Hadoop 任务有一半是 Pig 任务。在 2010 年，Pig 发展持续增长，这一年 Pig 从 Hadoop 的子项目中脱离出来，自己成为了一个最高级别的 Apache 项目。

为什么叫做 Pig

有一个问题经常被问起，那就是“为什么把它命名为 Pig？”人们还想知道是否 Pig 这个词是由缩写字母组成的。回答是否定的。事实上最初开发这个项目的研究人员只是简单地把它称为“那种语言。”不过他们终究需要为它起个名字。一名研究人员的脑海中出现了 Pig 这个单词，然后就提议称它为 Pig，这个名字一下子就让大家惊住了。这个名字古怪但是容易记也容易读。虽然有些人暗示这个名字听起来有点别扭或者傻，但是因这个名字促使了我们一个非常不错的命名方法，例如把这种语言命名为 Pig Latin，把终端交互命名为 Grunt，把一个共享资源库性质的分支成为 Piggybank。

第 2 章

安装和运行 Pig

2.1 下载和安装 Pig

用户需要下载和安装 Pig，才能在本地机器或者 Hadoop 集群上使用它。如果用户已经了解并完成了这一操作，那么可以直接跳到 2.2 节“运行 Pig”进行阅读。

用户既可以直接下载 Pig 安装包也可以先下载源代码然后自行进行编译。当然同样也可以以 Hadoop 分支的方式获得 Pig。

2.1.1 从 Apache 下载 Pig 软件包

这个是 Apache Pig 的官方版本。它是一个软件包，里面包含了运行 Pig 所需的所有 JAR 包。用户可以通过访问 Pig 发布页面进行下载。

Pig 不需要安装到 Hadoop 集群中去。它运行在用户提交 Hadoop 任务的那台机器上。尽管用户可以从个人笔记本电脑或者台式计算机运行 Pig，但是在实际操作中，大部分的集群管理员会配置好一到多台可以访问他们 Hadoop 集群的服务器，虽然这些服务器可以不是集群的一部分（也就是说它们既不是数据节点也不是计算节点）。通过这种方式，管理员可以方便地升级 Pig 和将工具集成在一起，同时也可以很好地限制人员对于集群的访问。这些机器被称为网关机或者缝边机。本书中我称之为网关机。

用户需要在这些网关机上安装 Pig。如果用户是通过个人台式计算机或者笔记本电脑访问 Hadoop 集群的话，那么同样需要在个人台式计算机或者笔记本电脑上安装 Pig。当然，如果用户想通过本地模式使用 Pig 的话，那么也可以将 Pig 安装到个人的本地机器上。

Pig 的核心是用 Java 语言编写的，因此它是跨平台的。启动 Pig 的 shell 脚本是个 bash 脚本，所以它需要一个 Unix 环境。Pig 所基于的 Hadoop，即使使用的是本地模式，也需要是 UNIX 环境的，因为 Hadoop 的文件操作是基于 UNIX 的。实际情况是，大部分的 Hadoop 集群采用的是 Linux 系统。很多 Pig 开发者是在 Mac OS X 系统上开发和测

试 Pig 的。

Pig 需要 Java 1.6 支持，Pig 版式从 0.5 到 0.9 需要 Hadoop 0.20 版本支持。对于以后的版本，请在下载页面确认该版本需要对应的 Hadoop 版本是多少。下载的 Pig 中会包含与它对应的 Hadoop。如果用户想在本地模式下使用 Pig 或者想把它安装到一个没有安装 Hadoop 的网关机上，那么就不需要额外去下载 Hadoop。

一旦下载完 Pig，可以把它放在自己喜欢的任何位置，Pig 无需依赖一个特定位置。安装第一步，将压缩包放在用户期望的目录下然后执行：

```
tar xzf filename
```

其中 *filename* 是用户所下载的 TAR 压缩包文件。

安装第二步是确定环境变量 **JAVA_HOME** 是否设置指向到包含一个 Java 发布版的目录，这一步是为执行 Pig 做环境准备。如果这个环境变量没有设置，那么 Pig 会立即执行失败。用户可以通过 shell 命令设置该环境变量，也可以在调用 Pig 的时候通过命令行指定该环境变量，或者在位于用户刚才解压后的文件 *bin* 目录下的 **pig** 脚本文件中显式地指定好 **JAVA_HOME** 环境变量。用户可以通过执行 **which java** 命令查看当前 java 所在路径，把这条命令返回的结果后面的 **bin/java** 去掉，就是当前环境中 **JAVA_HOME** 的值。

2.1.2 从 Cloudera 下载 Pig

除了官方的 Apache 版本之外，同样有一些其他的公司会重新包装和分发 Hadoop 以及与其相关的工具。当前这些公司中最受欢迎的是 Cloudera，它为 Red Hat 系列的系统开发相应的 RPM 包以及为 Debian 系统开发相应的 APT 包。对于不可以使用这些包管理器的其他系统，Cloudera 同样提供了 tar 压缩包文件。使用像 Cloudera 这样的发行版的一个优点是与 Hadoop 相关的所有工具都是打包在一起并且在一起测试完成的。同样，如果用户需要专业的技术支持，它也是提供的。缺点是用户将受制于其使用的发行版的提供商的发行速度。Apache 推出一个新版本后，然后到各个分支给出不同的发行版，这中间会有一定的时间延迟。

如果想获得从 Cloudera 下载和安装 Hadoop 和 Pig 的详细说明，请访问 Cloudera 下载页面。需要注意的是用户需要单独下载 Pig，因为 Hadoop 包中没有包含 Pig。

2.1.3 使用 Maven 下载 Pig

除了可以从 Pig 的 Apache 官方网站上下载发行版之外，用户还可以通过 Apache 的 Maven 资源库下载 Pig。这个站点包含了 Pig 需要的所有 JAR 文件，包括源代码 jar 包文件，Javadocs jar 包文件，同时还包括一个定义了 Pig 所依赖的 JAR 包的 POM 文件。如果用户使用 **maven** 或者 **ant** 进行编译，在编译过程中也会自动从这个资源库中下载相关的 JAR 包。

2.1.4 下载 Pig 源码

当用户从 Apache 下载 Pig 的时候，同时也会得到所有 Pig 的源代码。这样可以方便用户调试其下载的那个版本的 Pig 或者只是下载代码下来看看 Pig 是如何工作的。但是如果用户想一直跟随在最前沿，想在正式版本发行前尝试新的功能或者进行 bug 修复，那么可以通过 Apache 的版本管理资源库中下载源代码。用户需要可以访问 Pig 的问题跟踪系统并将一些补丁文件打到自己使用的版本中，这些补丁文件往往还没有来得及合并到代码资源库中。如何通过 svn 或者 git 获得 Pig 可以在 Pig 的版本控制页面这个网页看到。

2.2 运行 Pig

用户可以在本地运行 Pig 也可以在集群系统上运行 Pig，还可以将 Pig 作为 Amazon 的弹性 MapReduce 服务系统的一个组成部分运行。

2.2.1 本地单机运行 Pig

本地单机运行 Pig 按 Pig 的定义说法是以 local 模式运行。Local 模式对于样机研究和调试用户所写的 Pig Latin 脚本是非常有用的。有些人在处理小文件的时候也采用这种模式，尽快处理流程和处理大数据量是相同的——这样可以保证无论处理什么数量级别的数据时都采用一致的数据流，但是他们不希望集群为处理小的文件和小的任务而浪费资源。

在 0.6 版本和更早的版本中，Pig 自身是以 local 本地模式执行脚本的。从版本 0.7 开始，它开始使用 Hadoop 的类 LocalJobRunnr，这个类可以读取本地文件系统并且可以在本地执行 MapReduce 任务。这样执行的优点是在本地运行 Pig 任务和在集群中执行是相同的，同时它们都是以相同的流程执行，从而使得调试更加方便。缺点是太慢，启动一个本地的 Hadoop 实例需要将近 20 秒的时间，即使是非常小的任务也需要耗时这么久。[1]

下面我们以 local 本地模式运行一个 Pig Latin 脚本。查看前言的“本书中的代码样例”部分获得如何下载本例中所需要的数据和 Pig Latin 代码样例脚本源码。例 2-1 所示是个简单的例子，其加载了文件 NYSE_dividends，并按股票交易码进行分组，然后计算每个交易码对应的平均股息。

例 2-1 以本地模式运行 Pig

```
-- average_dividend.pig
```

[1] 本地模式下同样切换使用 MapReduce 过程的另一个原因是，Pig 每一个新增的功能都更多地使用到 MapReduce 的特性，这些特性在本地模式下非常困难或者几乎不可能复制下来。因此本地模式和 MapReduce 模式具有不同的功能集合。

```
-- 从 NYSE_dividends 中加载数据，并声明其有 4 个字段
dividends = load 'NYSE_dividends' as (exchange, symbol, date, dividend);
-- 按照股票交易号对行进行分组
grouped  = group dividends by symbol;
-- 计算每个股票交易号对应的平均股息
avg      = foreach grouped generate group, AVG(dividends.dividend);
-- 将结果存储到文件 average_dividend 中
store  avg into 'average_dividend';
```

如果用户使用 head -5 这个 shell 命令来查看 NYSE_dividends 文件，那么将会看到：

```
NYSE CPO      2009-12-30      0.14
NYSE CPO      2009-09-28      0.14
NYSE CPO      2009-06-26      0.14
NYSE CPO      2009-03-27      0.14
NYSE CPO      2009-01-06      0.14
```

这与我们在 Pig Latin 脚本中所声明的结构是一致的。第 1 个字段是这支股票所属的交易所的名称，第 2 个字段是股票交易码，第 3 个字段是所支付股息的日期，第 4 个字段是股息的大小。

提示

记住，在运行 Pig 前用户需要设置 **JAVA_HOME** 环境变量。这个值等于用户使用的 Java 发行版所在的目录路径。

切换到 NYSE_dividends 文件所在的目录，然后用户就可以通过如下方式以本地模式的方式运行 Pig 了：

```
pig_path/bin/pig –x local average_dividend.pig
```

其中 *pig_path* 是指用户本地机器 Pig 安装所在的目录。

执行上述命令后用户应该能看到屏幕上有很多输出。大部分输出是 MapReduce 的 **LocalJobRunner** 所生成的日志，但其中也有一些是 Pig 告诉用户它将如何执行这个脚本，以及执行过程中的状态信息等。在输出的最底端用户应该能看到一个简短的提示信息 **Success!**。这意味着一切进展顺利。该脚本将输出存储到 average_dividend 文件中，因此也许用户期望能在自己的本地目录下发现一个名字为 average_dividend 的文件。事实上用户会发现一个这个名字的目录，该目录下有一个名字为 part-r-00000 的文件。这是因为 Hadoop 是一个分布式系统，一般是以并行的方式处理数据的，当它将输出数据存放到一个“文件”中的时候，它就会使用这个“文件”的名称创建一个目录，每一个写处理过程就会在这个文件夹中生成一个独立的文件片段。在这个例子中我们只有一个写处理过程，因此我们只得到了一个文件片段。用户可以通过如下命令查看这个结果文件中的内容：

```
cat average_dividend/part-r-00000 | head -5
```

结果是：

```
CA      0.04
CB      0.35
CE      0.04
CF      0.1
CI      0.04
```

2.2.2 在 Hadoop 集群上运行 Pig

大部分的时间用户将在 Hadoop 集群上运行 Pig。正如我们在 2.1 节“下载和安装 Pig”中所描述的那样，Pig 可以在用户个人机器或者网关机上以 local 本地模式运行。所有的语法解析、执行校验以及制定执行计划都是在本地完成的。Pig 然后才会在集群上执行 MapReduce 任务。

提示

当我提到“用户的网关机”的时候，我的意思是那台用户可以提交 Pig 任务的机器。通常情况是一台或多台可以访问 Hadoop 集群的机器。然后，看用户机器的配置了，它也可以是用户的本地机器。

为了能在用户的机器中使用 Pig，只需要告诉 Pig 两个信息：集群的 Namenode 和 JobTracker 所在的位置。NameNode 是 HDFS 的控制者，JobTracker 协调 MapReduce 任务。在 Hadoop 0.18 和更早版本中，这些位置用户可以在配置文件 hadoop-site.xml 中找到。在 Hadoop 0.20 版本和之后版本中，它们在不同的配置文件中：core-site.xml、hdfs-site.xml 和 mapred-site.xml。

如果用户已经通过 MapReduce 或者其他工具在用户的网关机上运行了 Hadoop 任务，那么用户很有可能已经看到这些文件了。如果没有，那么最好的办法是从集群节点机器中复制一份相同的配置文件到用户的网关机上。这样可以保证得到的是正确的地址和其他一些相关的配置信息。

如果不管是什么原因，无法从集群中复制这些文件，那么用户可以自行创建一个 hadoop-site.xml 文件。文件内容如下所示：

```
<configuration>
<property>
  <name>fs.default.name</name>
  <value>namenode_hostname:port</value>
</property>

<property>
  <name>mapred.job.tracker </name>
```

```
  <value>jobtracker_hostname:port</value>
</property>
</configuration>
```

用户需要从他的集群管理员那里获得 NameNode 和 JobTracker 对应的服务器名和端口名。

一但用户找到了，复制了或者自己创建了这些配置文件，之后需要告诉 Pig 这些配置文件在哪个目录下，用户需要设置 **PIG_CLASSPATH** 这个环境变量到那个目录下。需要注意的是需要指向的是那些 XML 配置文件所在的目录，而不是这些配置文件本身。Pig 会自动加载读取那个目录下的所有 XML 和 properties 类型的文件。

让我们再重新运行下这个之前在本地模式运行过的例子（例 2-1）。如果用户是第一次使用这个集群，那么最好创建一个根目录。这个过程可以通过 Pig 来完成：

```
PIG_CLASSPATH=hadoop_conf_dir pig_path/bin/pig –e fs –mkdir /user/
username
```

其中 hadoop_conf_dir 是包含 hadoop-site.xm 或者 core-site.xml、hdfs-site.xml 和 mared-site.xml 文件的那个目录；pig_path 是指用户的网关机上 Pig 所在的路径；usename 是用户在网关机上的用户名。如果用户使用的 0.5 或者更早的版本，那么需要将命令语句中的“**fs –mkdir**”改为“**mkdir**”。

提示

记住，在执行任何 Pig 命令前，都需要设置好 JAVA_HOME 环境变量。详细信息请查看 2.1.1 小节“从 Apache 下载 Pig 软件包”。

为了可以在用户集群中执行这个例子，需要将数据文件复制到集群中：

```
PIG_CLASSPATH=hadoop_conf_dir pig_path/bin/pig –e fs –copyFromLocal
NYSE_dividends NYSE_dividends
```

如果用户使用的是 Pig 0.5 版本或者更早的版本，需要将上述命令中的“**fs –copyFromLocal**”换成“**copyFromLocal**”。

现在已经准备好执行这个 Pig Latin 脚本了：

```
PIG_CLASSPATH=hadoop_conf_dir pig_path/bin/pig average_dividend.pig
```

输出的前几行告诉用户 Pig 是如何连接到集群中的。这个信息之后会描述执行该脚本的进度信息。用户需要通过检查 Pig 报告的信息和集群的信息是否一致来确定 Pig 连接到了正确的文件系统和 JobTracker，这点很重要。如果文件系统是以 file:///开头或者 JobTracker 显示的是 *localhost*，那么表示 Pig 并没有连接到集群中。如果这样，用户需要检查配置文件中是否正确地设置了值，还有设置的 PIG_CLASSPATH 指向的目录是

否包含了那些配置文件。

在靠近输出信息的最后面，应该有一行输出信息为 **Success!**。这意味着脚本成功执行了。之后用户可以通过如下命令查看结果：

```
PIG_CLASSPATH=hadoop_conf_dir pig_path/bin/pig –e cat average_dividend
```

执行上述这条命令应该给用户相同的连接信息，最终会输出所有的股票交易码和它们的平均股息值。

用户可能注意到在例子 2-1 中我曾指出 average_dividend 是一个文件夹，因此用户需要使用 cat 查看这个文件夹下的片段文件。然而在这个例子中，我直接使用命令 cat 来查看 average_dividend 这个目录。如果用户使用 list 命令查看 average_dividend 这个目录，可以发现这和之前一样确实是个文件夹，但是在 Pig 中，命令 cat 是可以操作文件夹的。在第 3 章中会对此进行讨论。

2.2.3 在云服务上运行 Pig

云计算[1]伴随着软件即服务（SaaS）模式的出现已经好几年了。像 Hadoop 这样的硬件密集型的应用的出现是偶然的。搭建和维护一个 Hadoop 集群是一个需要在硬件获取、设备成本以及维护和管理等方面进行昂贵投入的命题。很多用户发现如果通过租硬件的方式会比较便宜一些。

无论用户或者用户的组织决定是使用云端的 Hadoop 和 Pig 还是使用自己拥有和操作的机器，在用户所使用的集群上运行 Pig 的使用说明都是一样的，具体操作请查看 2.2.2 小节“在 Hadoop 集群上运行 Pig”。

然而，Amazon 的弹性 MapReduce（EMR）云服务提供的方式有所不同。EMR 允许用户租借虚拟 Hadoop 集群，而不是允许用户租借机器用于任何类型的处理（像 Amazon 的弹性云计算[EC2]服务和其他云服务）。这些集群从 Amazon 的简单存储服务（S3）读取和存储数据到 S3。这意味着用户甚至不用自己去建立他们自己的 Hadoop 集群，而如果使用 EC2 或其他类似服务时他们需要自己去建立 Hadoop 集群。

EMR 用户可以通过他们的浏览器，SSH 或者一个 Web 服务 API 访问他们租借的 Hadoop 集群。想获得关于 EMR 的更多信息，请访问网址：http://aws.amazon.com/elasticmapreduce。不过，我建议从这个不错的向导开始，向用户介绍该服务的一些相关信息。

2.2.4 命令行使用以及配置选项介绍

Pig 具有许多用户可以使用的命令行选项。用户可以通过输入 **pig -h** 命令查看完整的选

[1] 作为当前最火的技术，术语“云计算”已被用来描述所有使用多台计算机而不仅仅只使用一台计算机的应用。而本章中所说的云计算是指使用计算机集群并部署特定的软件进行运算的活动。

项列表。大部分的命令行选项我们稍后会在使用了这些选项功能的那些章节中讨论。在本小节中我们将讨论如下几个不同的命令行选项：

`-e 或者 –execute`

在 Pig 中单独执行一条命令。例如：**pig -e fs –ls** 将会列举出用户根目录下的文件。

`-h 或者 –help`

列举出可用的命令行选项。

`-h properties`

列举出 Pig 将要使用的属性值，如果用户设置了这些属性值。

`-P 或者 –propertyFile`

指定一个 Pig 应该读取的属性值配置文件。

`-version`

打印出 Pig 的版本信息

Pig 同样会使用很多的 Java 属性值。属性值信息的完整列表可以通过执行 **pig –h properties** 命令打印出来。在之后使用到相关属性值的章节中再讨论如何指定哪些属性值。

Hadoop 同样具有许多用于决定其行为的 Java 属性。例如，用户可以通过设置属性 mapred.child、java.opts 来对运行 map 和 reduce 任务的 JVM 传送一些属性值。在 Pig 0.8 版本和之后版本中，这些 Java 属性也可以传送给 Pig，Pig 然后会在调用到 Hadoop 的时候再将这些参数传送给 Hadoop。对于 0.8 版本之前的其他版本，这些属性值必须要在 hadoop-site.xml 配置文件中配置才可以，这样 Hadoop 客户端会自己加载这些配置信息。

这些属性值可以通过命令行选项 **–D** 传送给 Pig，传送的格式和普通的 Java 属性一样。例如，**bin/pig –D exectype=local**。当使用命令行的方式加载这些属性的时候，必须要在 Pig 专有的命令行选项（例如，**-x local**）前定义这些属性。用户也可以在用户所使用的 Pig 发行版目录下的 *conf/pig.properties* 这个文件中定义用户的属性值。最后，用户还可以通过 **–P** 命令指定另一个不同的配置文件。如果同时使用了命令行的方式和属性文件的方式设置属性值，优先以命令行设置的值为准。

2.2.5 返回码

Pig 使用如表 2-1 所示的返回码来传达是成功还是失败。

表 2-1　　Pig 的返回码

值	含　义	备　注
0	成功	
1	失败，但还可以重试	
2	失败	
3	部分失败	Mulitquery 下使用，请参考 6.3 节“非线性数据流”
4	传递给 Pig 是非法参数	
5	抛出 IOException 异常	通常是由 UDF 抛出的
6	抛出 PigException 异常	通常意味着有 Python UDF 抛出了异常
7	抛出 ParseException 异常（参数代入完成后进行解析时可能会抛出这样的异常）	
8	抛出 Throwable 异常（一种未知意外异常）	

第 3 章

命令行交互工具 Grunt

Grunt 是 Pig 的命令行交互工具。它允许用户交互地输入 Pig Latin 脚本以及以交互的方式操作 HDFS。

不需要执行任何脚本和命令直接调用 Pig，就可以进入 Grunt。命令如下：

```
pig -x local
```

立刻会有如下信息：

```
grunt>
```

这样就开启了一个访问用户本地文件系统的 Grunt shell 界面。如果用户在上述命令中省略–x local 而且同时在他的 PIG_C LASSPATH 中包含了集群的配置信息，那么用户将会开启一个访问用户集群的 HDFS 文件系统的 Grunt shell 界面。

正如你所想的，Grunt 与其他 shell 界面一样也会提供一个命令行操作历史也可以进行编辑，同时也是以 Tab 键进行自动补全命令。但是没有提供通过 Tab 键补全文件名的功能。也就是说，如果用户输入 **kil** 然后再按 **Tab** 键，那么就会自动补全命令为 **kill**。但是如果在用户的本地文件系统中有一个文件名称为 *foo*，用户输入 **ls fo**，之后再按 **Tab** 键，不会自动补全命令为 **ls foo**。这是因为通过 HDFS 去连接然后再确认文件是否存在耗时太久，以至于根本没有使用的必要。

尽管 Grunt 是个很有用的 shell 工具，但请记住它并非包含了普通 shell 具有的所有功能。它没有提供标准 Unix shell 所提供的很多功能，例如管道、重定向和后台执行等功能。

用户可以通过输入 **quit** 或者按 Ctrl-D 组合键退出 Grunt。

3.1 在 Grunt 中输入 Pig Latin 脚本

Grunt 的主要用途之一就是以交互式会话的方式输入 Pig Latin 脚本。这对于快速地对数

据进行抽样以及原型设计更好的 Pig Latin 脚本是非常有用的。

用户可以在 Grunt 中直接输入 Pig Latin 脚本。Pig 不会执行用户输入的 Pig Latin 脚本，直到它发现用户输入 **store** 或者 **dump** 命令。不过，它会做一些基本的语法和语义检查，这样可以方便用户快速捕获到错误。如果用户在 Grunt 中有一行 Pig Latin 脚本输入错误了，那么可以采用相同的别名重新输入那条脚本，Pig 会以最后那行用户输入的实例为准。例如：

```
   pig –x local
   grunt> dividends = load 'NYSE_dividends' as (exchange, symbol, date,
dividend);
   grunt> symbols = foreach dividends generate symbl;
   … Error during parsing. Invalid alias: symbl …
   grunt> symbols = foreach A generate symbol;
   …
```

3.2 在 Grunt 中使用 HDFS 命令

除了可以交互式地输入 Pig Latin 脚本，Grunt 的另一个主要用途是作为访问 HDFS 的一个 shell 端口。在 Pig 0.5 版本以及之后的版本中，所有的 **hadoop fs** shell 命令都可以在 Grunt 中使用。这些命令可以通过关键字 **fs** 访问。**hadoop fs** 命令后面所跟的横线（-）在 Grunt 中同样是需要的：

```
   grunt>fs -ls
```

用户可以在网址：http://hadoop.apache.org/common/docs/r0.20.2/hdfs_shell.html 中查看到所有的 hdfs shell 操作命令。其中许多命令与 Unix shell 中的相同而且功能也是类似的：chgrp、chmod、chown、cp、du、ls、mkdir、mv、rm 和 stat。其他的一些要么看起来和 Unix 命令相似，要么功能有点不同，要么有点不像，包括：

`cat filename`

将一个文件的内容打印到标准输出 stdout。用户也可以使这个命令后面跟的是一个文件夹，那么它会迭代输出该文件夹下的每一个文件。

`copyFromLocal localfile hdfsfile`

从用户的本地磁盘中复制一个文件到 HDFS 中。该命令是以串行而非并行的方式处理的。

`copyToLocal hdfsfile localfile`

从 HDFS 文件系统中复制一个文件到用户的本地磁盘。该命令是以串行而非并行的方式处理的。

`rmr filename`

递归地移除文件。该命令相当于 Unix 中的 rm –r 命令。要慎重使用此命令。

在 Pig 0.5 版本前的其他版本中，还没有提供 hadoop fs 系列命令，而是 Grunt 本身实现了如下一些命令：

cat、cd、copyFromLocal、copyToLocal、cp、ls、mkdir、mv、pwd、rm（该命令与 Hadoop 中的 **rmr** 命令效果一样，而不是 Hadoop 中的 **rm**）和 **rmf**。在 Pig 0.8 版本中，所有这些命令都还是存在的。但是除了 **cd** 和 **pwd** 两个命令外，其他都不推荐使用，而且它们在未来可能会被移除，所以推荐使用 **hadoop fs**。

在 0.8 版本，Grunt 中新增了一个命令：**sh。**该命令允许用户使用本地的 shell 命令，就像 **fs** 提供给用户访问 HDFS 的功能一样。没有使用管道或者重定向的简单的 shell 命令都可以被执行。因为 **sh** 并非总是能正确地获取到当前的工作目录，所以最好还是使用绝对路径。

3.3 在 Grunt 中控制 Pig

Grunt 同样提供了控制 Pig 和 MapReduce 的命令：

`kill` *`jobid`*

终止指定 *jobid* 的 MapReduce 任务。产生任务的 **pig** 命令在输出信息中会列举出它所产生的所有任务的 ID。用户也可以通过 Hadoop 的 JobTracker 的 GUI（图形用户界面）查看到所有任务的 ID 号。在 GUI 中，会列出当前集群中正在运行的所有任务信息。需要注意的是该命令只会终止特定的那个 MapReduce 任务。如果用户的 Pig 任务包含了其他的不依赖用户所终止的那个任务的其他 MapReduce 任务时，那么这些任务还会继续执行下去。如果用户想终止一个特定的 Pig 任务所触发的所有的 MapReduce 任务，那么最好的办法是终止这个 Pig 进程，然后再使用那个命令终止所有真正执行的 MapReduce 任务。请确定使用 Ctrl-C 或者 Unix 中 kill 命令来终止 Pig 进程，而不要使用 Unix 中的 kill -9 命令，因为这个命令不会触发 Pig 进行临时文件清理，这样会在用户的集群中留下垃圾文件。

```
exec [[-param param_name = param_value]] [[-param_file filename]] script
```

执行 Pig Latin 脚本文件 *script*。*script* 这个文件中的别名不会被传入到 Grunt 中。当在一个 Grunt 会话中测试用户的 Pig Latin 脚本时，这个命令非常有用。想了解更多关于选项–param 和 –param_file 的信息，请查看 6.5.1 小节“参数传入”。

```
run [[-param param_name = param_value]] [[-param_file filename]] script
```

在当前 Grunt shell 中执行 Pig Latin 脚本 *script*。因此 *script* 脚本文件中使用的所有别名在 Grunt 中都是有效的，同时 *script* 文件中的命令也是通过 shell 历史记录可以查看到的。这是另一种方式在一个 Grunt 会话中测试用户的 Pig Latin 脚本。想了解更多关于选项–param 和–param_file 的信息，请查看 6.5.1 小节“参数传入”。

第 4 章

Pig 数据模型

在我们了解 Pig Latin 提供了哪些操作符之前，我们需要先了解 Pig 的数据模型。这些内容包括 Pig 的数据类型，它的处理思想是什么，例如是如何处理缺失数据的，以及用户需要通过何种方式描述用户导入 Pig 中的数据。

4.1 数据类型

Pig 的数据类型可以分为两大类：只包含一个值的基本类型（scalar type）和包含其他类型的复杂类型（complex type）。

4.1.1 基本类型

Pig 的基本类型是大多数编程语言都有的简单类型。除了 bytearray 这个类型外，其他类型都是通过调用 java.lang 类的 Pig 接口实现的，这样很容易在用户自定义函数 UDF 中使用。

`int`

整数。它是通过 java.lang.Integer 实现的。它存储一个 4 个字节大小的带符号整数。整型常数通常以整数数值来表示，如 42。

`long`

长整型。它是通过 java.lang.Long 实现的。它存储一个 8 个字节大小的带符号整数。长整型（long）常数是以一个结尾为 L 的整数来表示的，如 5000000000L。

`float`

浮点数。它是通过 java.lang.Float 实现的。它用 4 个字节存储值。取值范围与 Java 的 Float 类型的值的取值范围相同，用户可以在如下网址查看：

http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#4.2.3。需要注意的是，因为这是个浮点数，所以在一些计算中它会丢失精度。对于一些要求不能丢失精度的运算，用户应该使用 int 或者 long 数据类型。浮点型（float）常量可以通过一个浮点数加上 f 来表示。浮点数值可以使用简单的格式表示，例如 3.14f；或者以指数的形式表示，例如 6.022e23f。

`double`

双精度浮点数。它是通过 java.lang.Double 实现的。它使用 8 个字节存储值。取值范围与 Java 的 Double 类型的值的取值范围相同，用户可以在如下网址查看：http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#4.2.3。需要注意的是，因为这是个浮点数，所以在一些计算中它会丢失精度。对于一些要求不能丢失精度的运算，用户应该选择 int 或者 long 数据类型。双精度浮点型（double）常量可以使用简单的格式表示，例如 2.71828；或者以指数的形式表示，例如 6.626e-34。

`chararray`

字符串或者字符数组。它是通过 java.lang.String 实现的。chararray 常量是以加单引号的一系列字符来表示的，例如'fred'。除了标准的字母数字和符号字符之外，用户还可以通过转义符反斜杠以 chararray 的形式来表示一些特定的字符，例如\t 表示 Tab,\n 表示回车返回。Unicode 字符可以通过\u 加上它们的 4 位十六进制 Unicode 值来表示。例如 Ctrl-A 对应的值为\u0001。

`bytearray`

一团或一组字节。bytearray 是通过封装了 Java 的 byte[]的 DataByteArray Java 类来实现的。没有办法去定义一个 bytearray 常量。

4.1.2 复杂类型

Pig 有 3 个复杂数据类型：map、tuple 和 bag。这 3 种类型都可以包含任意类型的数据，包括其他复杂类型的数据。所以如果有一个 map，它的值字段是 bag 类型，这个 bag 包含了一个 tuple，而该 tuple 的字段是 map，这种情况是可以存在的。

`Map`

Pig 中的 map 是一种 chararray 和数据元素之间的键值对映射，其中数据元素可以是任意的 Pig 类型，包括复杂类型。其中的 chararray 被称为键（key），它作为查找对应数据元素的索引，相应的数据元素被称为值（value）。

因为 Pig 不知道值的类型，那么它就会假设值为 bytearray 类型，尽管实际的值可能为其他类型。如果用户想知道真实的数据类型是什么（或者用户想让它成为什么数据类型），用户可以对它进行类型转换，相关信息请查看 4.2.1 节“类型转换”。用户如果没有显式地对值进行类型转换，那么 Pig 将会根据用户在脚本中如何使用这个值将其

转换成一个最有可能的类型。如果值是 bytearray 外的其他类型，那么 Pig 会在运行时获得数据类型然后进行处理。关于 Pig 如何处理未知数据类型的更多信息请查看 4.2 节“模式”。

默认情况下并不要求一个 map 中的所有值具有相同的数据类型。一个 map 包含两个键：name 和 age，其中 name 对应的值是 chararray 类型，而 age 对应的值是 int 类型，像这种情况是合法的。从 Pig 0.9 版本开始，map 可以将它的值声明为具有相同的数据类型。这个新功能是非常有帮助的，因为如果用户事先就知道 map 中所有的值都是具有相同的数据类型的话，那么就可以避免进行类型转换，而且 Pig 也就无需在执行阶段运行时对数据类型进行控制。

map 常量通过方括号来划定 map 结构，键和值间是一个#号，键值对之间使用逗号分隔。例如：['name'#'bob','age'#55]将创建一个包含“name”和“age”两个键的 map。第一个值是 chararray 类型的，第二个值是一个整数。

Tuple

tuple 是一个定长的，包含有序 Pig 数据元素的集合。tuple 可以分为多个字段，每个字段对应着一个数据元素。这些数据元素可以是任意的数据类型，它们无须是相同的数据类型。一个 tuple 相当于 SQL 中的一行，而 tuple 的字段就相当于 SQL 中的列。因为 tuple 中的字段是有序的，所以可以通过位置来获得字段。详细信息请查看第 5.3.1 小节中的“foreach 语句中的表达式”部分。一个 tuple 可以，但非必须，有一个和它对应的模式描述每一个字段的名称和数据类型是什么。这样可以让 Pig 检查 tuple 中的数据与用户期待的数据是否一致，而且这样可以允许用户通过字段的名称对数据进行引用。

tuple 常量使用圆括号来指示 tuple 结构，使用逗号来划分 tuple 中的字段。例如：('bob', 55）描述的是一个包含了两个字段的 tuple 常量。

Bag

bag 是一个无序的 tuple 集合。因为它是无序的，所以无法通过位置获得 bag 中的 tuple。与 tuple 一样，一个 bag 可以但非必须，有一个和它对应的模式。对于 bag 而言，模式描述它其中每一个 tuple 的名称以及数据类型是什么。

bag 常量是通过花括号进行划分的，bag 中的 tuple 用逗号来分隔。例如：{('bob',55),('sally',52),('john',25)}构造了一个包含 3 个 tuple 的 bag,每个 tuple 包含了两个字段。

Pig 用户经常注意到 Pig 没有提供可以存放任意数据类型的 list 或者 set 数据类型。其实可以使用 bag 来模拟 set 数据类型，只需要将需要的类型作为 tuple 的唯一的一个字段就可以了。例如，如果用户想存储一组整数，用户可以创建一个包含一个 tuple 的 bag，而 tuple 只有一个字段存放的是 int 型数据。这可能有点麻烦，但是很有效。

bag 是 Pig 中一种不需要加载到内存中的数据类型，稍后用户会看到，因为在进行分组

时不会使用 bag 存放集合，所以 bag 可以变得非常大。在需要的时候 Pig 可以将 bag 中的数据写入磁盘，而只保留 bag 的部分信息在内存中。bag 的大小取决于本地磁盘提供给 bag 临时的存放空间的大小。

Pig 不同数据类型对内存的需求情况

在之前的章节中我经常提到每种数据类型的值占用的存储空间大小（例如整数需要占用 4 个字节，long 类型需要占用 8 个字节等），从而告诉用户这些数据类型可以拥有多大（或多小）的值。然而，这并没有告诉用户这些数据类型的对象实际使用多少的内存空间。因为 Pig 在内部是使用 Java 对象来表示这些值的，所以会有一个额外的开销。这个额外开销取决于用户的 JVM，但通常情况下是每个对象占 8 个字节。对于 chararray 类型可能会更多，因为 Java 中 String 对于每个字符占用 2 个字节而不是 1 个字节。

因此，如果用户试图想算出 Pig 需要消耗多少内存才能处理好用户的数据（例如，假设用户要进行一个 join 操作而需要将一个哈希表放到内存中），不可以假定磁盘上数据的字节数等同于用户需要的内存大小。磁盘和内存之间的倍增系数取决于用户的数据，例如在磁盘上数据是否是压缩的、用户磁盘的存储格式等。一般说来，对于没有压缩的数据，将会占用 4 倍于本地磁盘空间的内存空间。

4.1.3 NULL 值

Pig 中有数据元素为 null 的概念。任何数据类型的数据都可以为 null。很重要的一点是 Pig 中 null 的概念与 SQL 中所说的 null 的概念是一样的，而与 C、Java、Python 等语言中的 null 的概念完全不同。在 Pig 中 null 值所表达的含义是这个值是未知的。这可能是因为数据缺失，或者在处理数据时发生了错误等原因造成的。在大多数的程序语言中，当一个值没有被赋值或者没有指向一个有效的地址或者对象的时候，我们称这个值为 null。这里对于 null 值概念的不同非常重要而且会影响到 Pig 处理 null 数据的方式，特别是如何操作这些值的方式。在 5.3.1 节“foreach”、5.3.3 节“Group”和 5.3.6 节“Join”中可以查看到 Pig 在表达式和关系式中是如何处理 null 值的。

与 SQL 不同，在 Pig 中没有数据约束这个概念。在 null 的定义的背景下，这意味着任何一个数据元素可以一直为 null。在用户写 Pig Latin 脚本和用户自定义函数 UDF 时，需要牢记这一点。

4.2 模式

Pig 对于模式是非常宽松的，这和 Pig“什么都吃”的理念相通。如果用户为数据定义

了一个模式，那么 Pig 会使用这个模式，为的是既可以进行预先的错误检查也可以用于执行过程的优化。但是如果用户没有为数据提供一个模式，Pig 仍然可以处理数据，它会根据脚本是如何处理数据的做出一个最合理的猜测。首先，我们先看看用户可以使用哪些方式将模式传达给 Pig；然后我们看一下当用户没有提供模式给 Pig 时，它是如何处理的。

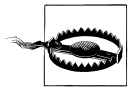
将用户数据的模式传达给 Pig 的最简单的方式是在加载数据的时候显式地告诉 Pig 用户加载的数据是什么样子的：

```
dividends = load 'NYSE_dividends' as
(exchange:chararray, symbol:chararray, date:chararray, dividend:float);
```

Pig 现在预期到用户的数据具有 4 个字段。如果数据本身多于 4 个字段，那么 Pig 会把多出的字段截掉不要。如果数据本身小于 4 个字段，那么 Pig 会使用 null 值进行补充。

在指定数据的模式时同样可以不给出明确的数据类型。在这种情况下，数据类型被认为是 bytearray:

```
dividends = load 'NYSE_dividends' as (exchange, symbol, date, dividend);
```



警告

用户应该可以预测到这样同样会将用户的数据转换为包含 4 个字段的 tuple，而无论实际输入的字段的个数有多少个，就像之前的那个用户既指定了字段的别名也指定了字段的类型时一样。在 Pig 0.9 版本中，确实是这样。但是在 0.8 版本或更早的版本中并非如此，当用户没有显式地指定字段的类型时，既不会裁剪多余的数据也不会用 null 值补齐字段值。

同样，当用户声明一个模式的时候，也并非一定要声明这个模式中的复杂数据类型，当然如果用户愿意也是可以声明的。例如，假设用户的数据包含一个 tuple，那么就可以声明该字段是个 tuple，同时可以不声明这个 tuple 中包含了哪些字段。当然用户也可以声明这个 tuple 具有 3 个字段，而且全都是整数。表 4-1 给出了如何在一个模式声明中定义各种数据类型的语法。

表 4-1 模式语法

数 据 类 型	语　　法	举　　例
int	int	as (a:int)
long	long	as (a:long)
float	float	as (a:float)
double	double	as (a:double)
chararray	chararray	as (a:chararray)

续表

数据类型	语　法	举　例
bytearray	bytearray	as (a:bytearray)
map	map[]或者 map[*type*]，其中 *type* 必须是一个合法的数据类型。这就声明了 map 中所有的值都是这个数据类型的	as (a:map[], b:map[int])
tuple	tuple() 或者 tuple(list_of_fields)，其中 list_of_fields 是一组通过逗号分隔的字段声明	as (a:tuple(),b:tuple (x:int, y:int))
bag	bag{} 或者 bag{t: (*list_of_fields*)}，其中 *list_of_fields* 是一组通过逗号分隔的字段声明。需要注意的是，bag 中的 tuple 必须要有一个名称，这里指定的名称为 t，虽然用户可能永远无法直接访问 t 这个 tuple	(a:bag{}, b:bag{t:(x:int,y:int)})

在运行时声明模式也不错，这使得用户无需事先将数据加载到一个元数据系统而可以直接操作数据。这也意味着如果用户只对数据中前几个字段感兴趣，那么可以只声明这几个需要的字段。

但是对于每小时或每天都要操作相同数据的生产系统来说，这种方式有几个比较显著的缺点。其一，一旦用户的数据发生改变，那么用户就需要改写他的 Pig Latin 脚本。其二，尽管这对于有 5 个列的数据处理起来很好，但是如果数据有 100 个列，那将是非常痛苦的。为了解决这些问题，还有另外一种在 Pig 中加载模式的方式。

如果用户所使用的加载（load）函数事先已经知道数据的模式，该函数就可以将此模式传达给 Pig（加载函数是指 Pig 是如何读取数据的，请查看 5.2.1 节“加载”获得更详细的信息）。加载函数可能事先知道模式，因为模式可以存放到像 HCatalog 这样的元数据库中，或者它可以存在于数据本身中（假设，例如数据以 JSON 格式存储）。在这种情况下，用户不需要在 load 语句中声明模式，而且用户还可以通过字段的别名进行引用，这是因为 Pig 会在对用户的脚本进行错误检查时从加载函数中获得模式信息：

```
mdata = load 'mydata' using HCatLoader();
cleansed = filter mdata by name is not null;
…
```

但是当用户遇到数据流的时候怎么办？如果用户定义了一个模式但是加载函数又返回了一个模式那该怎么办？如果它们是完全相同的，一切都好。如果它们并非完全相同，Pig 会先适配加载函数返回的模式和用户定义的模式。例如，如果用户定义一个字段的类型为 long 但是加载函数返回的是 int，Pig 就会将 int 格式转换为 long。然而，如果 Pig 无法找到方法使加载函数返回的模式和用户定义的模式相适配，那么它会报错。在

4.2.1 节“类型转换”，用户可以查看到为使两个模式匹配工作，Pig 可以进行哪些以及不可以进行哪些类型转换。

现在我们来看看当用户和加载函数都没有告诉 Pig 数据的模式的时候会怎样。除了可以通过名称进行引用外，字段还可以通过位置进行引用，位置的初始值是 0。语法是美元符号，位置：$0 是指第 1 个字段。因此很容易告诉 Pig 哪一个字段才是用户想使用的。但是 Pig 怎么知道数据类型呢？它确实不知道，所以刚开始的时候，它假定所有的字段都是 bytearray 类型的。之后 Pig 会根据用户在脚本中是怎样使用这些字段的，推测出用户认为这些字段是什么以及用户想怎么使用它们。请读者思考如下语句：

```
   --no_schema.pig
   daily = load 'NYSE_daily';
   calcs = foreach daily generate $7 / 1000, $3 * 100.0, SUBSTRING($0, 0, 1),
$6 - $3;
```

表达式中$7/1000,1000 是一个整数，因此可以猜测到 NYSE_daily 文件的第 8 个字段是整型的或者是可以转换成整型的类型。以同样的方式，$3*100.0 表明$3 是个 double 型的，同时在一个将 chararray 作为参数的函数中发现了$0，表明$0 是 chararray 类型的。但是最后一个表达式$6-$3 怎么办呢？在 Pig 中减号（-）是用于两个数值型数据之间的，因此 Pig 可以很容易猜测到$3 和$6 是数值型的。但是应该是整型的还是浮点型的呢？这里 Pig 为保证正确性会认为它们都是浮点型的，会将它们的类型转换为 double 型。这是更安全的猜测，因为如果它们是整型的，它们也是可以使用浮点型数据表示的，但是反过来就不行了。然而，因为浮点型数据运算起来会慢很多而且会损失精度，所以如果这些值只是整数，用户还是应该进行类型转换，在本例中也就是应该转换为整型。

也有一些情况下 Pig 不能给出一个聪明的猜测：

```
   --no_schema_filter
   daily = load 'NYSE_daily';
   fltrd = filter daily by $6 > $3;
```

在 Pig Latin 中，大于号（>）对于数值型、chararray 型和 bytearray 型都是合法的操作符。因此，Pig 无法进行推测。在这种情况下，Pig 会将它们都作为 bytearray 类型处理，也就是说会对这些字段内的数据进行一个字节一个字节的比较。

Pig 同样要处理当它推测的是错误的而需要随时去改变的情况。仔细看如下的语句：

```
   --unintended_walks.pig
   player     = load 'baseball' as (name:chararray, team:chararray，
                pos:bag{t:(p:chararray)}, bat:map[]);
   unintended = foreach player generate bat#’base_on_balls’ – bat#’ibbs’;
```

因为 map 中的值可以为任意的数据类型，Pig 并不知道 bat# 'base_on_balls'和 bat#'ibbs'

的数据类型是什么。正如之前所描述的那样，Pig 会假定它们是 double 类型的。但是我们知道它们从内部实际上被证明是整型的 [1]。在那种情况下，Pig 需要在运行时对类型进行匹配，将它所认为的类型从 bytearray 转换为 double 类型，将 int 型转换为 double 型。需要注意的是这仍会产生一个 double 输出而不是一个 int 输出。这看上去并非凭直觉就可获得，请查看本章最后的“Pig 是强类型吗？”部分以了解为什么会这样。需要注意的是在 Pig 0.8 版本以及之前的版本中，这种运行时匹配的代码大多是不完善和经常失败的。在 0.9 版本中，大部分的问题被修复了。但是如果用户正在使用一个较老的版本，他也许需要正确地对数据进行类型转换以保证获得正确的结果。

最后，Pig 对模式的认知在 Pig Latin 脚本执行的不同阶段可能是不同的。在前面所有的例子中，我们没有定义模式直接加载数据然后把它传送给 **foreach** 语句，数据没有模式而直接出发。但是在 **foreach** 语句之后，模式是已知的了。同样地，Pig 可以一开始就知道模式，但是如果数据和其他没有模式的数据混合在一起，这个模式也就被丢弃了。也就是说，没有模式是具有感染性的：

```
   --no_schema_join.pig
   divs  = load 'NYSE_dividends' as (exchange, stock_symbol, date,
dividends);
   daily = load 'NYSE_daily';
   jnd   = join divs by stock_symbol, daily by $1;
```

在这个例子中，因为 Pig 并不知道 *daily* 的模式，所以它也就不知道 join 语句中 divs 和 daily 的模式。

类型转换

前一节提到了类型转换但是没有繁琐地说明类型转换到底是如何工作的。在 Pig 中进行类型转换的语法与 Java 中的语法是一样的，即类型名称放在圆括号里面，后面跟的是值：

```
   --unintended_walks_cast.pig
   player     = load 'baseball' as (name:chararray, team:chararray,
                pos:bag{t:(p:chararray)}, bat:map[]);
   unintended = foreach player generate (int)bat# 'base_on_balls'- (int)bat#
'ibbs';
```

通过类型转换的方式指定类型与在模式中指定它们的方式是一样的，正如前面的表 4-1 所示。

并非所有用户想象到的类型转换在 Pig 中都是允许的。表 4-2 描述了基本类型间哪些类型转换是允许的。因为 Pig 不知道如何采用二进制的方式来展示各种各样的数据类型，所有类型不允许转换为 bytearray 型。将 bytearray 类型转换为其他类型是允许的。转换

[1] 这并不是样例数据中所展示的那种情况。对于那种情况，用户需要使用一个加载函数将那个别名为 bat、值为整数的 map 加载进来。

成复杂数据类型或者将复杂数据类型转换为其他类型目前也是不允许，当然除了可以从 bytearray 类型转换为其他复杂数据类型，尽管从概念上在一些情况下转换应该是可以的。

表 4-2　　支持的数据类型转换

	转换为 int	转换为 long	转换为 float	转换为 double	转换为 chararray
从类型 int		可以	可以	可以	可以
从类型 long	可以。所有大于 2^{31} 或者小于 -2^{31} 的值会被截断		可以	可以	可以
从类型 float	可以。数值会被截断为 int 类型数值	可以。数值会被截断为 long 类型数值		可以	可以
从类型 double	可以。数值会被截断为 int 类型数值	可以。数值会被截断为 long 类型数值	可以。超过 float 类型精度的值会被截断		可以
从类型 chararray	可以。非数值的字符转换后结果为 null	可以。非数值的字符转换后结果为 null	可以。非数值的字符转换后结果为 null	可以。非数值的字符转换后结果为 null	

有一种需要特别处理的类型转换，那就是将 bytearray 转换为其他类型。因为 bytearray 表示的是一串字节，所以 Pig 并不知道如何将它的内容转换为其他任何一种数据类型。继续前面的那个例子，bat#'base_on_balls'和 bat#'ibbs'两个都是被加载为 bytearray 类型的，通过脚本中的上下文知道用户是想把它们的类型转换为 int 型来处理。

Pig 并不知道 *baseball* 中存储的整数是以 ASCII 字符串形式，还是 Java 序列化值形式，还是二进制编码的十进制形式，或者其他的存储格式存储的，所以它会向加载函数获取相关信息，因为加载函数有责任实现将 bytearray 转换为其他数据类型。通常这样是没有问题的，但是这也确实会在一些特殊的情况下导致 Pig 无法确定如何去对 bytearray 类型的值进行类型转换。特别是在用户自定义函数 UDF 返回的是一个 bytearray 类型的值的情况下，Pig 就不知道如何对这个值进行类型转换了，因为这个值并不是由加载函数产生的。

在结束“类型转换”这一小节之前，我们需要考虑下 Pig 在执行过程中自动为用户进行类型转换的情况。与用户显式地指定需要转换的类型相比，这些类型转换是隐式的。思考如下脚本：

```
--total_trade_estimate.pig
daily = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
```

```
            date:chararray, open:float, high:float, low:float, close:float,
            volume:int, adj_close:float);
rough = foreach daily generate volume * close;
```

在这种情况下，Pig 会将第 2 行优化为（float）volume*close，这样进行了类型转换而同时不会丢失精度。通常来说，Pig 在需要隐式地进行类型转换时，会向范围更大的类型转换。因此，int 和 long 一起的话结果会为 long 型；int 或者 long 和 float 一起的话结果会为 float 类型；同样的 int、long 或者 float 和 double 在一起的时候结果会是 double 型。数值类型和 chararray 以及其他类型间没有隐式的类型转换。

Pig 是强类型吗？

在强类型语言（例如 Java）中，用户必须明确地声明所有变量的类型。在弱类型语言（例如 Perl）中，变量可以是不同的数据类型，这要通过其应用的上下文来确定。那么 Pig 是哪一种呢？总体来说它是强类型的。如果用户描述了其数据的模式，那么 Pig 会按照用户所声明的模式来处理用户的数据。但是当 Pig 并不知道数据对应的模式的时候，它会在运行时为数据指定好数据类型。（也许我们应该称 Pig 是“轻类型”的。它确实对类型要求严格但是如果没有明确定义类型也是可以处理数据的。）为了能看到这两种情况之间的区别，我们再来看个例子：

```
--unintended_walks.pig
player     = load 'baseball' as (name:chararray, team:chararray,
               pos:bag{t:(p:chararray)}, bat:map[]);
unintended = foreach player generate bat# 'base_on_balls'- bat# 'ibbs';
```

在这个例子中，我们假设 base_on_balls 和 ibbs 的值被认为是在内部当作整数处理的（也就是说加载函数将它们构造成整数）。假如 Pig 是弱数据类型，那么 *unintended* 的输出将是只有一个整型字段的记录。而事实上，Pig 输出的结果是一个只有一个 double 型字段的记录。Pig 会进行推测，然后尽可能地将数据转换为它所推测的类型。

Pig 的缺点是使擅长使用弱数据类型语言的用户当数据没有按他们所预测的那样输出的时候，会对此感到十分惊讶甚至不知所措。然而，从好的一方面来说，在这些情况下，用户只通过观察 Pig Latin 脚本就可能知道输出数据的数据类型是什么而无需知道输入数据是什么样的。

第 5 章

Pig Latin 介绍

是时候开始研究 Pig Latin 了。本章将向用户介绍 Pig Latin 的基础知识，足以帮助用户自己编写第一个可以使用的脚本了。关于 Pig Latin 更多的高级应用将在第 6 章进行介绍。

5.1 基础知识

Pig Latin 是一种数据流语言，每个处理步骤都会产生一个新的数据集，或者产生一个新的*关系*。在 input = load 'data'这句脚本中， **input** 是加载数据集 *data* 后结果的关系名称。这里所说的关系名称也就是我们通常所说的*别名*。关系名称看起来和变量的概念相似，但是它们不是变量。一旦声明了，那么这个分配就是不变的了。关系名称是可以被重用的，例如，如下的用法是被允许的：

```
A = load 'NYSE_dividends' (exchange, symbol, date, dividends);
A = filter A by dividends > 0;
A = foreach A generate UPPER(symbol);
```

但是，这种做法是不推荐的。这里看上去好像是在重定义 A，其实是创建了一个新的关系并命名为 A，这也就丢掉了和之前那个叫做 A 的关系的联系了。虽然 Pig 会尽可能地保持关系，但是这仍然不是一个好的做法。这会导致在阅读用户脚本程序和查看错误日志的时候让人感到困惑（例如会问：我指定的到底是哪个 A？）。

除了关系名称外，Pig Latin 还有一个概念就是字段名称。它代表的是一个关系所包含的字段（或者称为列）的名称。例如，在前面提到的 Pig Latin 脚本片段中， dividends 和 symbol 就是字段名称。这与变量当在数据管道中传送时对于不同的记录包含不同的值这一点有些相像，但是无法对它们进行赋值。

无论是关系名称还是字段名称都必须以字母字符开头，之后可以跟上零个或多个字母、数字或者下划线（_）。名称中的所有字符必须都是 ASCII 码。

5.1.1 大小写敏感

不幸地是，Pig Latin 无法明确说是否是大小写敏感的。Pig Latin 中的关键字是非大小写敏感的，例如，LOAD 和 load 是等价的。但是关系名称和字段名称是大小写敏感的。因此 A = load 'foo';和 a = load 'foo';是不等价的。用户自定义函数 UDF 的名称也是大小写敏感的，因此 COUNT 和 count 所指的并非是同一个 UDF。

5.1.2 注释

Pig Latin 具有两种注释方式：SQL 样式的单行注释（--）和 Java 样式的多行注释（/* */）。例如：

```
A = load 'foo';--这个是单行注释
/*
 * 这个是多行注释
 */
B = load /* 这个是中间的注释*/ 'bar';
```

5.2 输入和输出

在用户可以做一些自身感兴趣的事情之前，需要知道如何为数据流增加输入和输出。

5.2.1 加载

任何一种数据流的第一步都是要指定输入。在 Pig Latin 中是通过 load 语句来完成的。默认情况下，load 使用默认加载函数 PigStorage 加载存放在 HDFS 中并且以制表键进行分割的文件。例如，语句 divs = load '/data/examples/NYSE_dividends'；会在文件夹目录 /data/examples 下查找文件名为 *NYSE_dividends* 的文件。当然也可以写相对路径。默认情况下，Pig 任务会在当前用户的 HDFS 中的用户目录下执行，例如/user/yourlogin。除非用户改变了这个目录，否则所有的相对路径都以那个用户目录为基准。用户也可以指定一个完整的 URL 路径，例如，hdfs://nn.acme.com/data/examples/NYSE_dividends ，这可以从 NameNode 为 nn.acme.com 的 HDFS 实例中读取文件。

实际上，用户的大部分数据并非是使用制表键做分隔符的文本文件，也有可能需要从其他非 HDFS 的存储系统中加载数据。Pig 允许用户在加载数据时通过 using 句式指定其他加载函数。例如，如果想从 HBase 中加载数据，那么用户需要使用为 HBase 准备的加载函数：

```
divs = load 'NYSE_dividends' using HBaseStorage();
```

如果没有指定加载函数，那么会使用内置的加载函数 PigStorage。用户同样可以通过 using 句式为使用的加载函数指定参数。例如，如果想读取以逗号分隔的文本文件数据，那么 PigStorge 会接受一个指定分隔符的参数：

```
divs = load 'NYSE_dividends' using PigStorage(',');
```

load 语句中也可以有 as 句式，这个句式可以为用户加载的数据指定模式（4.2 节“模式”对在 Pig Latin 中声明模式的语法和语义已有详细的介绍）。

```
divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
```

当从 HDFS 访问指定“文件”的时候，用户也可以指定文件夹。在这种情况下，Pig 会遍历用户指定的文件夹下的所有文件并将它们作为 load 语句的输入。因此，如果有一个名称为 *input* 的文件夹，里面包含两个文件 *today* 和 *yesterday*，那么用户可以将文件夹 *input* 作为加载的输入文件，Pig 会将 *today* 和 *yesterday* 两个文件同时作为输入进行处理。如果指定的文件夹下还有其他文件夹，所有这些文件夹下的文件也都是会被包括的。

PigStorage 和 TextLoader，这两个内置的可以操作 HDFS 文件的 Pig 加载函数，是支持模式匹配的。[1] 通过模式匹配，用户可以读取不在同一个文件夹下的多个文件或者读取一个文件夹下的部分文件。表 5-1 描述的是在 Hadoop 0.20 中所提供的合法的模式匹配语法。需要注意的是这些正则字符的含义是由 Pig 下面的 HDFS 决定的，因此这些正则字符是否可以工作取决于用户使用的是什么版本的 HDFS。同样的，如果用户是从 Unix shell 命令行运行 Pig latin 命令，那么用户还需要对这些正则字符进行转义以防止正则字符被分拆为多个部分。

表 5-1　　Hadoop 0.20 中提供的正则匹配语法

正 则 字 符	描　　述
?	匹配任何单个字符
*	匹配零个或多个字符
[abc]	匹配字符集合{a,b,c}所包含的任意一个字符
[a-z]	匹配指定范围内的任意字符
[^abc]	匹配未包含的任意字符，其中^符号匹配输入字符串的开始位置
[^a-z]	匹配任何不在指定范围内的任意字符
\c	移除（转义）字符 c 的所表达的特殊含义
{ab,cd}	配置字符串集合{ab,cd}中的任一字符串

5.2.2　存储

当用户处理完数据之后，需要把结果写到某个地方。Pig 提供了 store 语句来进行写数

[1] 所有使用 FileInputFormat 作为它的 InputFormat 的加载器都是支持模式匹配的。大多数从 HDFS 中加载数据的加载器是使用这个 InputFormat 的。

据。在许多方面，它是 load 语法的镜像。默认情况下，Pig 使用 PigStorage [1] 将结果数据以制表键作为分隔符存储到 HDFS 中：

```
store processed into '/data/examples/processed';
```

上面的语句，Pig 会将处理完的结果数据存储到*/data/example* 路径下的 *processed* 文件夹下。当然用户可以指定相对路径，同样也可以指定完整的 URL 路径，例如 *hdfs://nn.acme.com/data/example/processed*。

如果用户并没有显式地指定存储函数，那么将会默认使用 PigStorge。用户可以使用 using 语句指定一个不同的存储函数：

```
store processed into 'processed' using
      HBaseStorage();
```

用户也可以传参数给其使用的存储函数。例如，如果想将数据存储为以逗号分隔的文本数据，PigStorage 会接受一个指定分隔符的参数：

```
store processed into 'processed' using PigStorage(',');
```

正如 2.2 节“运行 Pig”所提及的，当写到文件系统中后，*processed* 是一个包含多个部分文件的文件夹，而不是一个文件。但是至于会生成多少个部分文件，这要取决于执行 store 前的最后一个任务的并行数，该值由为这个任务所设置的并行级别所决定。更多信息请查看 5.3.9 小节“Parallel”。如果这是一个 map-only 的任务，那么生成的文件个数取决于 map 任务的个数，这是由 Hadoop 而非 Pig 控制的。

5.2.3 输出

在大多数情况下当处理完数据后用户会希望将结果存储到某个地方，但是用户偶尔也会想在屏幕上看看这些结果数据。这对于调试阶段和原型研究阶段是特别有用的。对于一些特殊的运行很快的任务也是有用的。dump 语句可以将用户的脚本输出打印到屏幕上。

```
dump processed;
```

一直到 Pig 0.7 版本，**dump** 的输出数据是符合 Pig Latin 中定义的常量格式的。因此，long 类型的值会以 **L** 结尾，float 类型的数据会以 **F** 结尾，map 用[]（方括号）分隔，tuple 使用()（圆括号），bag 使用{}(花括号)。从 0.8 版本开始，long 值的 **L** 和 float 类型值的 **F** 被移除了，然而复杂数据类型的表示方式被保留了下来。null 表示数据缺失值，字段是以逗号分隔的。因为输出中的每条记录都是一个 tuple，所以它是被（）（圆括号）包围的。

[1] 加载和存储函数可以为同一个函数，例如 PigStorage 就是这样的。

5.3 关系操作

关系操作符是 Pig Latin 提供给用户用于操作数据的主要工具。使用关系操作符可以让用户对数据进行排序、分组、连接、推测和过滤等转换。本节将介绍一些基本的关系操作符。对于这些操作符更多的高级功能以及其他一些高级关系操作符的介绍，将在 6.1 节“高级关系操作”中进行介绍。本节所介绍的内容已经足以让用户能够使用 Pig Latin 进行编程了。

5.3.1 foreach

foreach 语句接受的是一组表达式，然后在数据管道中将它们应用到每条记录中，因此命名为 foreach。通过这些表达式，它会产生出新的数据并传送给下一个操作符。对于那些熟悉数据库术语的人来说，这个就是 Pig 的推测操作符。例如，如下代码加载了完整的记录，然后对于每条记录只保留 user 和 id 两个字段：

```
   A = load 'input' as (user:chararray, id:long, address:chararray,
phone:chararray, preferences:map[]);
   B = foreach A generate user, id;
```

foreach 语句中的表达式

foreach 支持大量的表达式，最简单的就是常量和字段引用。常量的语法在 4.1 节“数据类型”中已经讨论过了。字段引用可以通过别名进行引用（在之前的例子中有过演示）或者通过位置引用。位置引用是通过**$**（美元符号）加上从 0 开始的整数构成的：

```
   prices = load 'NYSE_daily' as (exchange, symbol, date, open, high, low,
close, volume, adj_close);
   gain = foreach prices generate close - open;
   gain2 = foreach prices generate $6 - $3;
```

上面的脚本中，关系名称 gain 和 gain2 会存储相同的值。位置引用这种方式在不知道模式或者没有声明模式的时候会很有用。

除了使用别名和位置，用户可以使用*（星号）代表所有字段，这会创建一个包含所有字段的 tuple。从 0.9 版本开始，用户也可以使用..（两个点）来指定字段区间。当字段很多而且不想在 foreach 命令中重写一遍时，这是非常好用的：

```
   prices = load 'NYSE_daily' as (exchange, symbol, date, open,
               high, low, close, volume, adj_close);
   beginning = foreach prices generate ..open; -- produces exchange, symbol,
date, open
   middle = foreach prices generate open..close; -- produces open, high, low,
close
   end = foreach prices generate volume..; -- produces volume, adj_close
```

对于整数和浮点数，标准的算术操作符也是支持的：+表示加法，-表示减法，*表示乘法，/表示除法。这些操作符会根据数值本身确定返回值类型，因此 5/2 是 2，然而 5.0/2.0 的值是 2.5。除此之外，对于整数，还支持取模操作符%。对于整数和浮点数，同样支持一元负值操作符（-）。Pig Latin 遵循标准的数学执行优先级规则。想了解不同数据类型间使用算术操作符（例如，5/2.0）会做如何处理，请查看 4.2.1 节“类型转换”以获得更多信息。

null 值对于所有的算术操作符都是抵消的。也就是说，x+null=null，这里不管 x 是什么类型的值都是成立的。

Pig 同样提供了一个三元条件操作符，经常被称为 bincond。它以一个布尔表达式开始，后面跟着一个？（问号），然后跟着是如果前面布尔表达式为真的时候返回的值，后面再跟着：（冒号），最后跟着如果前面布尔表达式为假时应返回的值。如果前面的布尔表达式返回 null，那么整个表达式返回 null。表达式中的？后的两个值应该是相同的数据类型：

```
2 == 2 ? 1 : 4   -- 返回 1
2 == 3 ? 1 : 4   -- 返回 4
null == 2 ? 1 :  4    -- 返回 null
2 == 2 ? 1 : 'fred' -- 类型错误；冒号两边的值应该是相同的数据类型
```

为了从复杂的数据类型中提取出值，需要使用投射运算符。对于 map 是使用#（也就是散列表），然后后面跟着一个字符串类型的键的名称。需要注意的是一个键对应的值可以是任意数据类型的。如果用户使用的键在 map 中不存在，那么结果是 null:

```
bball = load 'baseball' as (name:chararray, team:chararray,
          position:bag{t:(p:chararray)}, bat:map[]);
avg = foreach bball generate bat#'batting_average';
```

tuple 的映射是用.（点操作符）符号的。最外层的数据记录，可以通过字段名指定一个字段（如果对于这个 tuple 用户定义了一个模式）。对 tuple 通过位置进行引用，如果该位置不存在，则会返回 null。如果使用字段名进行引用，而 tuple 中没有这个字段名，那么将会产生一个错误：

```
A = load 'input' as (t:tuple(x:int, y:int));
B = foreach A generate t.x, t.$1;
```

对于 bag 的映射并不像 map 和 tuple 映射那么简单明了。bag 并不会保证它内部存放的 tuple 是以有序的方式存放的，因此对一个 bag 中的 tuple 进行映射是没有意义的。反之，当用户需要映射一个 bag 中的字段的时候，可以通过创建一个包含用户需要的字段的 bag：

```
A = load 'input' as (b:bag{t:(x:int, y:int)});
B = foreach A generate b.x;
```

这会创建一个新的 bag，其中只有字段 x 在里面。用户可以通过使用圆括号中标明逗号分隔的字段名称的方式映射多个字段：

```
A = load 'input' as (b:bag{t:(x:int, y:int)});
B = foreach A generate b.(x, y);
```

这有一点区别是 b.x 是一个 bag 而不是一个可以进行计算的数量值。请思考如下的 Pig Latin 脚本，这个脚本是错误的：

```
A = load 'foo' as (x:chararray, y:int, z:int);
B = group A by x;-- 产生包含对于 x 给定的值对应的所有记录的 bag A
C = foreach B generate SUM(A.y + A.z);
```

这里我们很清楚编程人员想做什么。但是因为 A.y 和 B.y 是 bag，而 bag 类型间是没有加法操作的，因此这里会报错。[1]在 Pig Latin 中正确的进行这个计算的方式如下：

```
A = load 'foo' as (x:chararray, y:int, z:int);
A1 = foreach A generate x, y + z as yz;
B = group A1 by x;
C = foreach B generate SUM(A1.yz);
```

foreach 语句中的 **UDF**

在 **foreach** 句式中可以使用用户自定义函数（UDF），这些 UDF 被称为求值函数。因为它们是作为 **foreach** 语句的组成部分，所以这些 UDF 每次读取一条记录然后产生一条输出。需要记住的是这里不管是输入还是输出都可以是一个 bag，因此这里的一条记录可以包含一个 bag 单元的记录：

```
--udf_in_foreach.pig
divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
  -- 确保所有的字符串都是大写的
upped = foreach divs generate UPPER(symbol) as symbol, dividends;
grpd = group upped by symbol; -- 为每个 symbol 值都输出一个 upped bag
-- 接受一个 bag 的整数，为每个分组产生一个结果
sums = foreach grpd generate group, SUM(upped.dividends);
```

除此之外，评估函数可以接受*作为参数，这个符号表示将整个记录传送给该函数。即使函数本身是无参数的，它们也是可以被引用的。

读者可以在附录 A 中查看到 Pig 提供的完整的 UDF 列表。在 5.4 节“用户自定义函数 UDF”，我们将讨论如何调用非 Pig 自带的 UDF。

[1] 用户可能注意到并会说这里 Pig 是可以算出预期并会去执行的，既然 SUM(A.y + A.z)可以分解为“foreach A 中的记录，将 y 和 z 相加然后获得这个求和结果。”确实是这样。但是当我们将 group 换作 cogroup，这样就会引入 A 和 B 两个 bag（6.1.3 小节“cogroup”）同时将求和操作改为 SUM(A.y + B.z)，因为 A 和 B 都无法保证是有序的，所以这不是个很好定义的操作。在设计这门语言的时候，我们始终认为规定 bag 无论在什么情况下都不可以进行相加计算会比允许在一些情况下可以进行相加而另一些情况下不可以进行相加要好一些。

foreach 语句中的字段命名

每一个 **foreach** 语句的结果都是一个新的 tuple，这种情况下输出结果的模式与 **foreach** 的输入数据的模式是不同的。Pig 可以从 **foreach** 语句中推断出输出结果的各个字段的数据类型，但是它并非总能够推断出这些字段的名称。对于那些没有使用其他操作符的简单的映射的字段来说，Pig 会保留它们之前使用的名称：

```
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
          date:chararray, dividends:float);
sym = foreach divs generate symbol;
describe sym;

sym: {symbol: chararray}
```

一旦其中有一个使用的不是简单的映射操作，Pig 就不会对这个字段重新赋一个新的名称。如果用户不是显式地给这个字段起个名称，那么该字段就没有名称，也就只能通过位置参数来获得，例如，$0。用户可以用 **as** 句式为字段命名：

```
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
            date:chararray, dividends:float);
in_cents = foreach divs generate dividends * 100.0 as dividend, dividends
* 100.0;describe in_cents;

in_cents: {dividend: double,double}
```

需要注意的是在 **foreach** 中 **as** 关键字是跟在每个表达式之后的，这点与 **load** 句式指定字段名的方式不同。在 load 句式中字段定义语句是跟在整个语句之后的。通过 6.1.1 节下面的“flatten”小节的介绍，用户将会明白为什么采用两种不同的方式。

5.3.2 Filter

通过 **filter** 语句用户可以选择将哪些数据保留到用户的数据流里。filter 中包含了一个断言。对于一条记录，断言如果为 true 那么这条记录就会在数据流中传下去，否则就不会向下传。

断言式可以包含等值比较操作符，包括判断是否相等的==，以及!=、>、>=和<=。这些比较操作符都可以用于基本数据类型。==和!=也可以用于 map 和 tuple 这样复杂数据类型的比较。如果在两个 tuple 间进行比较，那么还要求这两个 tuple 要么有相同的模式要么都没有模式。所有上述的等值判断操作符都不可以用于 bag。

Pig Latin 遵循大多数程序语言都支持的标准运算符操作优先级规则，其中有一项就是算术计算操作符的优先级要大于等值比较操作符的优先级。因此，**x+y == a + b** 和 **(x+y) == (a + b)**是等价的。

对于 chararray 数据类型，使用者可以判断这个 chararray 是否符合指定的正则表达式：

```
--filter_matches.pig
divs    = load 'NYSE_dividends' as (exchange:chararray,
symbol:chararray,
             date:chararray, dividends:float);
startswithcm = filter divs by symbol matches 'CM.* ';
```

提示

Pig 使用的是 Java 的正则表达式格式。这种表达式格式要求整个 chararray 都必须是匹配的，而不是像 Perl 类型的那样表达式只要求部分匹配。例如，如果用户想找所有包含字符串"fred"的字段，用户必须采用'**.*fred.***'这种格式，而不是'**fred**'，后者之后匹配 chararray 类型的 **fred** 这个字符串。

用户可以通过在语句前面加上关键字 **no**，来查找那些不满足一个指定正则表达式的 chararray 类型数据：

```
-- filter_not_matches.pig
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
         date:chararray, dividends:float);
notstartswithcm = filter divs by not symbol matches 'CM.*';
```

用户可以将多个断言表达式通过布尔操作符 **and** 和 **or**，以及用于取相反的不符合条件的数据的布尔操作符 **not** 联合在一起使用。正如通常的标准规则，布尔操作符的执行优先级从高到低依次是:not、and、or。因此 a and b or not c 和（a and b）or (not c)是等价的。

Pig 会在一些情况下对布尔操作符缩短判断路径。如果对于 and 操作符左边的第一个表达式结果是 false，and 后面的判断就会被忽略。因此对于 **1==2 and udf(x)**这个表达式，该 UDF 函数永远也不会被调用。同样的，如果一个 or 的左边表达式结果是 true，那么 or 的右边也不会被采用。因此对于 **1==1 or udf(x)**这个表达式，该 UDF 函数也同样永远不会被调用。

对于布尔操作符，null 值遵循 SQL 的三元判断逻辑。也就是说 **x==null** 的结果是一个 **null** 值，而不是 **true**（即使当 x 也是 null）或者 **false**。过滤器 filter 只会允许布尔值判断为 **true** 的值通过。因此如果一个字段包含了 **2**、**null** 和 **4** 3 个值，如果用户的过滤条件是 **x==2**，那么只有第一个值也就是值 2 可以通过这个过滤器。然而，对于过滤条件 **x != 2** 将只会返回最后一条记录也就是值 **4**。而查找 null 值的正确方式是使用 **is null** 操作符，当值为 null 时该表达式会返回 **true**。如果想得到值不为 null 的记录，则应使用 **is not null**。

同样的，对于任何正则表达式，**null** 既不会匹配上也不会失败。

正如求值函数中使用到的一些 UDF 一样，也有一些专门处理过滤数据的 UDF，我们称之为过滤函数。这些函数可以返回一个布尔值并且也可以在 **filter** 语句中调用。注意过滤函数不可以在 **foreach** 语句中使用。

5.3.3 Group

group 语句可以把具有相同键值的数据聚合在一起，这是我们在本书中遇到的第一个与在 SQL 中语法相同的操作符，但是很重要的一点是我们意识到 Pig Latin 中的 group 操作与在 SQL 中的 group 操作有着本质的区别。在 SQL 中 **group by** 子句创建的组必须直接注入一个或多个聚合函数中。在 Pig Latin 中 **group** 和聚合函数之间没有直接的关系。然而，**group** 关键字正如它字面所表达的：将包含了特定的键所对应的值的所有记录封装到一个 bag 中。之后，用户可以将这个结果传递给一个聚合函数或者使用它做其他一些处理。

```
-- count.pig
daily = load 'NYSE_daily' as (exchange, stock);
grpd = group daily by stock;
cnt = foreach grpd generate group, COUNT(daily);
```

上面这个例子展示了以 **stock** 为键的值进行分组然后做计数统计。在进行 group 分组操作后直接存储到文件系统中用于后期的再处理，在 Pig 中，也是合法的操作：

```
-- group.pig
daily = load 'NYSE_daily' as (exchange, stock);
grpd = group daily by stock;
store grpd into 'by_group';
```

group by 语句的输出结果包含两个字段，一个是键，另一个是包含了聚集的记录的 bag。存放键的字段别名为 group[1]。而 bag 的别名和被分组的那条语句的别名相同，因此在前面的例子中别名应该为 daily 同时和关系 daily 具有相同的模式。如果关系 daily 没有模式，那么数据包 daily 也将是没有模式的。对于分组中的每一条记录，整个记录（包括那个键）都在这个数据包 bag 中。将上述脚本中最后一行的 store grpd …改为 describe grpd;将会可以查看这个关系 grpd 的模式：

```
grpd: {group: bytearray,daily: {exchange: bytearray,stock: bytearray}}
```

用户也可以对多个键进行分组，但是这些键必须包含在一组圆括号内。结果记录同样包含两个字段。在这种情况下，**group** 字段将是包含每个键值的 tuple。

```
--twokey.pig
daily = load 'NYSE_daily' as (exchange, stock, date, dividends);
grpd = group daily by (exchange, stock);
avg = foreach grpd generate group, AVG(daily.dividends);
describe grpd;
grpd: {group: (exchange: bytearray,stock: bytearray),daily: {exchange:
bytearray, stock: bytearray,date: bytearray,dividends: bytearray}}
```

用户也可以使用关键字 all 对用户的数据流中所有字段进行分组：

[1] 由于 Pig Latin 中重载了关键字 group，这是令人遗憾和困惑的，但是也是现在很难去改变的了。

```
--countall.pig
daily = load 'NYSE_daily' as (exchange, stock);
grpd = group daily all;
cnt = foreach grpd generate COUNT(daily);
```

group all 的输出记录是以字符串文字 all 为键的值。一般情况下这没有什么问题，因为通常会把数据包 bag 直接传送给像 COUNT 这样的聚集函数。但是如果用户计划把记录存储起来或者用于其他用途，那么用户可能需要事先知道全部的字段。

group 是我们到目前为止遇到的第一个通常会触发一个 reduce 过程的操作符。分组就意味着收集所有键中都包含相同的值的记录。如果数据流处于一个 map 阶段，那么就会迫使它先进行 shuffle 然后再进行 reduce。如果数据流处于一个 reduce 阶段，那么就会迫使它先进行 map 然后再进行 shuffle，最后再进入 reduce 阶段。

因为分组收集了所有键中都包含相同的值的记录，所以用户经常会获得数据倾斜的结果。也就是说，恰恰因此用户已经指定了用户的任务具有 100 个 reducer，那么也就没有理由期望每个键对应的值会按照个数均匀地分发出去。它们可能是一个高斯分布或幂次法则分布 [1]。例如，假设用户有一些网页索引数据，同时用户按照根 URL 进行分组。一些特定的值，例如 yahoo.com 会比大多数的其他根 URL 具有更多的记录，这也就意味着一些 reducer 会比其他的 reducer 获得更多的数据。因为用户的 MapReduce 任务是直到所有的 reducer 都完成时才会结束，那么这个数据倾斜会在相当大的程度上拖慢用户的处理过程。在一些情况下这也并非是一个 reduce，可以处理的了那样大的数据的。

Pig 具有一系列的处理方法来控制数据倾斜，以达到 reducer 间的负荷均衡。Hadoop 的组合器是适用于分组的。对 Hadoop 的组合器是如何工作的，请参见附录 B 的“Combiner 阶段”。这不会移除所有的数据倾斜，但是会在上面加上限制。同时因为对于大多数的任务 mapper 的个数都是数以万计的，即使 reducer 获得了一些倾斜的数据，每个 reducer 所处理的实际记录数也会非常小以足够这些 reducer 能够快速处理好。

不幸的是，并非所有的计算都是可以通过这个组合器完成的。对于可以分解成任意数量的处理步骤的计算，例如 sum 运算，也就是所谓的可分布处理的计算可以很好地使用组合器运算。而那些可以分解为一个初始化过程，任意数量的中间处理过程，最后会有一个结束处理过程的计算，我们称之为代数运算。例如 count 就是这样的一类函数，其中初始阶段是一个计数过程，中间处理过程和结束阶段是求和过程。可分布处理的计算是代数计算的一种特殊情况：初始过程、中间处理过程和结束处理过程都是相同的操作。追踪用户在网站上的行为的会话分析是一个非代数运算的例子。在开始分析用户和网站的交互行为之前用户必须对所有的记录按照时间戳进行排序处理。

[1] 以我的经验，绝大部分的人类数据跟踪行为是遵循幂次法则分布的。

Pig 的操作符和内置的 UDF 当在需要的时候都会使用这个组合器，因为它可以降低数据倾斜的特性以及在处理早期进行聚合处理的操作大大降低了网络间的数据传输和写磁盘的数据量，因此可以显著提高处理效率。用户自定义函数 UDF 可以通过实现代数计算接口来决定在什么时候使用这个组合器。关于如何使用户的 UDF 来使用这个组合器的更多详细的介绍，请查看 10.2 节“代数运算接口”。

关于在执行 group 操作时如何确定并行处理的级别的更多信息，请查看 5.3.9 节“Parallel”。同时，需要记住的是当使用 **group all** 的时候，用户有必要序列化用户的数据流。也就是说，这个步骤和此步骤后其他所有步骤，在用户没有将当前包含了所有记录的数据包分离出来之前都是无法以并行的方式运行的。

最后需要说明的是，**group** 处理 null 值的方式与 SQL 处理 null 值的方式是一样的：将以 null 作为键的所有记录汇集到相同的组里面。需要注意的是这与表达式处理 null 值的方式（无论 null == null 还是 null != null 的返回值都不是 true）以及 join（请查看 5.3.6 节“Join”）处理 null 值的方式是截然不同的。

5.3.4 Order by

order 语句是对用户的数据进行排序，产生一个全排序的输出结果。全排序意味着不仅是将每个部分的数据进行排序，同时也会保证对于 n 个部分文件那么第 n 个文件中的记录序号要比第 n–1 个部分文件中的少。当用户的数据存放在 HDFS 中的时候，每个部分都是一个部分文件，这意味着使用 **cat** 命令将会使用户的数据全部以有序的方式输出。

Order 语句的语法与 **group** 语句的语法相似。用户需要指定想按照哪一个或者多个键对数据进行排序。一个比较明显的区别是在 **order** 语句中指定键不需要使用圆括号：

```
--order.pig
daily = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
         date:chararray, open:float, high:float, low:float, close:float,
         volume:int, adj_close:float);
bydate = order daily by date;

--order2key.pig
daily = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
          date:chararray, open:float, high:float, low:float,
          close:float, volume:int, adj_close:float);
bydatensymbol = order daily by date, symbol;
```

在指定的键后面加上 **desc** 关键字可以使结果按照降序排列。如果 **order** 语句中指定了多个键，那么 **desc** 关键字之后对仅靠着它的那个键起作用。其他的键还会按照升序排序：

```
--orderdesc.pig
daily = load 'NYSE_daily' as (exchange:chararry, symbol:chararray,
         date:chararray, open:float, high:float, low:float, close:float,
         volume:int, adj_close:float);
```

```
byclose = order daily by close desc, open;
dump byclose; -- 字段 open 还是按照升序排序存储的
```

数据的排序方式取决于所指定的字段的类型：数值会按照数字顺序排序，chararray 类型的字符串字段会按照字典顺序排序，bytearray 类型的字段也按照字典顺序排序，但是按照字节值而不是字符值排序。对 map、tuple 和 bag 类型的字段进行排序会报错。对于所有的数据类型，null 都是最小的，因此常常是最先显示的（或者在使用 **desc** 排序时是最后显示的）。

在 5.3.3 节“Group”中讨论过，数据倾斜在数据中是很常见的。这只有在使用了 **group** 的时候才会影响到 **order**，会导致一些 reducer 会比其他的 reducer 花费明显长的时间。为了解决这个问题，Pig 会在 reducer 间均衡输出。它的处理方法是对于 **order** 语句的输入先进行取样以获得键值分布情况的预算。基于这个样本，Pig 会创建一个分割器用于产生一个均衡的全排序（可以查看附录 B“Shuffle 阶段”以了解什么是分割器）。举个例子，假设用户要对一个包含 **a**、**b**、**e**、**e**、**e**、**e**、**e**、**e**、**m**、**q**、**r**、**z** 这几个值的字符串类型的字段进行排序，同时用户有 3 个 reducer。在这个例子中分割器会按照如下方式分割数据：值为 **a～e** 的传送给第 1 个 reducer，值为 **e** 的传送给第 2 个 reducer，值为 **m～z** 传送给第 3 个 reducer。我们可以注意到 **e** 可以传送给第 1 个 reducer 也可以传送给第 2 个 reducer。这样使得分割器可以将数据平均地分发出去。在实际操作中，当使用这个算法时很少看到 reducer 之间消耗时间的方差超过 10%。

Pig 为减少数据倾斜而采用的这种数据分发方式所产生的弊端之一，是它打破了同一个指定键所对应的所有实例都传送到同一个部分文件中这个 MapReduce 规则。如果用户需要其他基于这一规则的处理过程，那么就不要使用 Pig 的 **order** 语句去对数据排序。

order 总是会引起用户的数据流通过一个 reduce 阶段，因此将所有相等的记录收集到一起是有必要的。同时，为了做抽样，Pig 也会在用户的数据流中增加一个额外的 MapReduce 任务。因为这个抽样是非常轻量的（它之后读取每个数据块的第一条记录），通常只会占用少于整个任务的 5%的时间。

5.3.5 Distinct

distinct 语句非常简单，它会将重复值去掉，只会对整个记录进行处理，而不是对字段级别进行计算：

```
--distinct.pig
-- 为每笔交易生成一个去掉重重后的股票代码列表
-- 这个加载过程截断记录，只会选择前两个字段
daily = load 'NYSE_daily' as (exchange:chararray, symbol:chararray);
uniq  = distinct daily;
```

因此它需要将相似的记录收集到一起来判断它们是否有重复，所有 **distinct** 会触发一个 reduce 处理过程。在 map 阶段也确实可以通过那个组合器将一些重复的数值去除掉。

这里所展示的 **distinct** 用法与 SQL 中的 **select distinct x** 用法是相同的。6.1.1 节中的“内嵌 foreach”小节会介绍怎样做才能达到与 SQL 中的 select count(distinct x)语句一样的效果。

5.3.6 Join

join 是数据处理中非常重要的操作之一，同时是最有可能在很多的 Pig Latin 脚本中使用到的操作。join 可以将一个输入中的记录和另一个输入中的数据放在一起。通过指定每个输入的键可以达到中介目的。当这些键值相等[1]的时候，数据就会被连接在一起。没有被匹配到的数据会被去掉：

```
   --join.pig
   daily = load 'NYSE_daily' as (exchange, symbol, date, open, high, low,
close,
              volume, adj_close);
   divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
   jnd = join daily by symbol, divs by symbol;
```

用户也可以指定多个键来使用 join。每个语句中都有相同个数的键，同时它们必须是相同的或者是同一类的数据类型（同一类的数据类型是指可以通过隐式的类型转换得到的，相关内容请参考 4.2.1 小节“类型转换”）：

```
   -- join2key.pig
   daily = load 'NYSE_daily' as (exchange, symbol, date, open, high, low,
close,
             volume, adj_close);
   divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
   jnd = join daily by (symbol, date), divs by (symbol, date);
```

与 **foreach** 一样，**join** 也会保留传递给它的输入数据的字段的别名，同时它也会保留这个字段来自的关系的别名，可以通过**::**符号指定。在上面的那个例子的最后一行加上 **describe jnd** 语句，会产生如下结果：

```
   jnd: {daily::exchange: bytearray,daily::symbol: bytearray,daily::date:
bytearray,
   daily::open: bytearray,daily::high: bytearray,daily::low: bytearray,
   daily::close: bytearray,daily::volume: bytearray,daily::adj_close:
bytearray,
   divs::exchange: bytearray,divs::symbol: bytearray,divs::date:
bytearray,
   divs::dividends: bytearray}
```

只有当字段名和记录中的字段不再相同时才需要使用 **daily::**这个前缀。在这个例子中，如果用户想在 join 操作之后引用这两个 **date** 字段中的任一个，用户需要使用 daily::date

[1] 事实上，join 并非只有在等值条件下才能使用，但是 Pig 只支持等值 join 操作（称为等值 join）。请查看 6.1.5 节“cross”以获得在 Pig 中如何进行非等值 join 操作的相关内容。

和 divs::date 方式指定。但是像 **open** 和 **divs** 这样的字段就不需要使用那个前缀了，因为它们不会引起混淆。

Pig 同时也是支持 outer join 的。在 outer join 中，没有和另一方匹配的数据也会被留下来，同时使用 null 值填补缺失的字段。outer join 分为 left、right 和 full 3 种形式。left outer join 意味着左边的数据会被全部保留下来即使在右边没有匹配的值。同样的，right outer join 意味着右边的数据被全部保留下来即使在左边没有匹配的值。full out join 意味着即使两边没有匹配的值两边的值也全部都会保留下来：

```
   --leftjoin.pig
   daily = load 'NYSE_daily' as (exchange, symbol, date, open, high, low,
close,
             volume, adj_close);
   divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
   jnd = join daily by (symbol, date) left outer, divs by (symbol, date);
```

outer 是一个多余的关键字，是可以省略的。与一些 SQL 实现不同的是，**full** 关键字是不可以省略的。**C= join A by x outer, B by u;**将会产生一个语法错误，而不是一个 full outer join。

只有当 Pig 知道数据需要填补 null 值的那边或那两边的模式的时候才支持使用 outer join。因此对于 left outer join，Pig 必须知道右边的模式；对于 right outer join，Pig 需要知道左边的模式；而对于 full outer join，Pig 必须要知道两边的模式。这是因为如果没有模式，Pig 就不会知道需要填补多少个 null 值 [1]。

和 SQL 中一样，键所对应的 null 值和什么都不会匹配上，即使和另一边输入中的 null 值也无法匹配。因此，对于内部连接 inner join，所有包含 null 键值的记录会被去除。对于外部连接 outer join，它们会被保留下来但是不会和另一边输入的任何记录相匹配。

Pig 也支持在一个操作下做多个 join 操作，只要它们全部使用相同的键进行连接即可。以下只有在做内部连接 inner join 时才能使用：

```
A = load 'input1' as (x, y);
B = load 'input2' as (u, v);
C = load 'input3' as (e, f);
alpha = join A by x, B by u, C by e;
```

自连接也是支持的，尽管这时数据需要被加载两次：

```
-- selfjoin.pig
```

[1] 用户可能会提出一个反对的看法，因为可能会认为 Pig 应该可以通过查看 join 连接中的其他记录从而推断出有多少个字段。然而，其实这并不可行，有两个原因。第一，当没有提供模式的时候，Pig 不会强加一个语法要求记录必须具有相同的模式。因此，假设 Pig 可以从一条记录推断另一条记录是不正确的。第二，join 连接中可能是没有任何数据是匹配的，也因此 Pig 可能根本没有数据作为依据去推断。

```
-- 对于每支股票，找出在两个日期范围内股息都是增加的记录
divs1 = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
            date:chararray, dividends);
divs2 = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
            date:chararray, dividends);
jnd = join divs1 by symbol, divs2 by symbol;
increased = filter jnd by divs1::date < divs2::date and
              divs1::dividends < divs2::dividends;
```

如果将上面的代码按照如下方式进行更改，则会执行失败：

```
-- selfjoin.pig
-- 对于每支股票，找出在两个日期范围内股息都是增加的记录
divs1 = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
           date:chararray, dividends);
jnd = join divs1 by symbol, divs1 by symbol;
increased = filter jnd by divs1::date < divs2::date and
               divs1::dividends < divs2::dividends;
```

看上去这段代码应该是可以执行的，虽然 Pig 可以将 divs1 数据集进行分割然后传送给 join 两次，但是问题在于连接操作后字段名会变得模糊不清，因此 **load** 语句必须要执行两次。下一个要做的事情是执行这两个 load 语句，因为输入数据是相同的，所以只需要执行一个 load 语句，但是目前来说还没有做这个优化。

Pig 通过在 map 处理阶段标注出每条记录是来自哪一个输入的 MapReduce 方式进行这些 join 操作，之后它再使用 join 的键作为 shuffle 阶段的键。因此 join 会触发一个新的 reduce 处理过程。一旦对于指定键具有相同值的所有记录被收集到一起的时候，Pig 会对两条输入的记录进行一次交叉乘积运算。为了减少内存使用，它使用 map 处理阶段增加的输入标注对进入 reducer 的记录进行 MapReduce 排序。因此，所有记录中来自左边输入的记录会先到达，Pig 会将这些记录缓存在内存中，而右边的输入数据会在之后紧跟而来。每当这些数据来一条就会和左边的数据进行一次交叉运算然后产生一条输出记录。在一个多路连接操作中，左边 n-1 个输入都会被放入内存中，同时第 n 条记录直接通过。如果用户知道输入记录中对于选定的键每个值对应着多条记录，那么在 Pig 查询中写 join 语句时牢记这一点将会非常重要。将这样的输入记录放到用户的 join 语句的右边将会降低内存的使用率，也许能够提高用户脚本的执行效率。

5.3.7 Limit

有时用户只是想从结果中拿出几条数据看看，limit 语句就可以满足此需求：

```
--limit.pig
divs = load 'NYSE_dividends';
first10 = limit divs 10;
```

这个例子将只会返回最多 10 行记录（如果用户的输入小于 10 行，那么会返回所有的

记录)。需要注意的是除了 order 外的其他所有操作符，Pig 都不会保证产生的数据是按照一定次序排序的。因为 NYSE_dividends 文件记录大于 10 条，所以上述样例脚本每次运行可能返回不同的结果。在 limit 前如果加上 order 语句就可以保证每次执行返回的结果是一样的。

limit 会产生一个额外的 reduce 过程，因为它需要将记录收集起来以计算它应该返回多少记录。Pig 通过在每个 map 过程限制输出条数然后再对 reducer 限制输出条数的方式来优化这一执行过程。当 limit 和 order 一起使用的时候，在 map 和 reduce 阶段这两个处理过程会同时进行。也就是说，在 map 这一边，使用 MapReduce 对记录进行排序然后在组合器中限制输出条数。shuffle 阶段会有一部分是用 MapReduce 再次进行排序，Pig 这时对 reduce 阶段再次进行输出条数限制。

这里 Pig 没有提供一个可能的优化是在早期读取输入数据时，一旦达到 limit 后指定的条数时就停止读入。因此，在这个例子中，如果用户期望使用此优化的方式去读取一小片输入数据，那么将会感到失望，因为 Pig 仍然会读取全部数据。

5.3.8 Sample

sample 语法提供了一种方式用于抽取样本数据。它会读取所有的数据然后返回一定百分比的行数的数据。可以通过一个 0 和 1 间的 double 值指定百分比。因此，在如下的例子中，0.1 表示 10%：

```
--sample.pig
divs = load 'NYSE_dividends';
some = sample divs 0.1;
```

当前这个抽样函数非常简单。sample A by 0.1 其实被重写成 filter A by random() <= 0.1。很明显这个是非精确的，因此使用 sample 的脚本每次执行的结果都是不同的。同时，百分比也并非是精确的，但是一定是近似的。对于增加一些高级的抽样技术是有讨论的，但是到目前为止还没有完成。

5.3.9 Parallel

Pig 的核心声明之一就是它将提供一种并行数据处理语言。Pig 理念中的一个原则是 Pig 是家畜（请查看 1.1.4 小节“Pig 的设计思想”），因此 Pig 喜欢用户告诉如何进行并行。为了做这件事，Pig 提供了 **parallel** 语句。

parallel 语句可以附加到 Pig Latin 中任一个关系操作符后面。然后，它只会控制 reduce 阶段的并行，因此只有对于可以触发 reduce 过程的操作符使用才有意义。可以触发 reduce 过程的操作符有：**group***、**order**、**distinct**、**join***、**cogroup***和 **cross**。使用星号标记的操作符具有多个不同的实现，这些实现中有的会触发一个 reduce 过程而有的则不会。对于此问题以及本章节没有覆盖到的操作符，请查看第 6 章以获得详细介绍。在本地模式下 **parallel** 会被忽略，原因是本地模式下所有的操作符都是串行执行的：

```
   --parallel.pig
   daily = load 'NYSE_daily' as (exchange, symbol, date, open, high, low,
close,
             volume, adj_close);
   bysymbl = group daily by symbol parallel 10;
```

在这个例子中，parallel 会使 Pig 触发的 MapReduce 任务具有 10 个 reducer。parallel 只会对后面附加上它的语句起作用，它们并非是在整个脚本中一直起作用的。因此如果这里的 group 后面跟的是一个 order 语句，用户需要单独为这个 order 语句设置 parallel 值。因为 group 操作最有可能显著地降低数据量大小，所以用户可能需要更改并行参数的值：

```
   --parallel.pig
   daily = load 'NYSE_daily' as (exchange, symbol, date, open, high, low,
close,
             volume, adj_close);
   bysymbl = group daily by symbol parallel 10;
   average = foreach bysymbl generate group, AVG(daily.close) as avg;
   sorted = order average by avg desc parallel 2;
```

然而，如果用户不想为脚本中每一个会触发 reduce 过程的操作符单独设置并行数值，那么用户可以通过 set 命令设置一个脚本范围内有效的参数值：

```
   --defaultparallel.pig
   set default_parallel 10;
   daily = load 'NYSE_daily' as (exchange, symbol, date, open, high, low,
close,
              volume, adj_close);
   bysymbl = group daily by symbol;
   average = foreach bysymbl generate group, AVG(daily.close) as avg;
   sorted = order average by avg desc;
```

在这个脚本中，所有的 MapReduce 任务会在 10 个 reducer 中完成。当用户设置了一个默认的并行值的时候，仍然可以为脚本中的语句增加一个 parallel 语句来重写这个默认值。因此可以设置一个大多数情况使用的默认值，然后对于需要一个其他值的操作符通过 parallel 关键字显式地重新设置一个并行值。

然后，所有这些都是静态的，如果对于不同特征的输入数据执行同一个脚本那么会怎么样呢？或者如果用户的输入数据有时差异性非常大那么会出现什么情况？用户当然不希望每次都去修改脚本。使用参数替换，用户可以在写并行语句的时候使用变量，在运行时再为这些变量赋值。相关内容请查看 6.5.1 小节“参数传入”。

到目前为止我们都假设用户知道并行值设置为多少是合理的。8.1.6 小节“选择合适的并行值”对如何确定这个值有详细的介绍。

最后还有个问题，那就是如果没有设置并行值会是什么情况？在 0.8 版本之前，Pig 对

于这种情况是让 MapReduce 来设置并行值。而 MapReduce 的默认并行值是通过集群的配置文件控制的。安装的默认值是 1，大多数人是不会更改这个值的。也就是说很有可能用户只会以一个 reducer 来运行，这绝对不是用户期望的。

为了避免这种情况，Pig 在 0.8 版本中增加了一个试探性的做法，就是如果这个值没有设置就通过一个粗略的估算给出一个并行值。它根据初始的输入值，减少数据量大小不会改变，然后每 1GB 的数据就分配一个 reducer。需要强调的是这并非是一个好算法，这只是为了防止因为一些设置错误导致脚本运行很慢，同时，在一些极端的情况下，因一些错误导致 MapReduce 本身就有问题。这是一个安全网，而不是一个优化。

Map 阶段的并行执行

parallel 仅允许用户设置 reduce 阶段的并行执行，那么 Map 阶段可以进行并行执行吗？MapReduce 只允许用户设置 reduce 阶段的并行值：它自身控制 map 阶段的并行。因为 Pig 不能控制 map 阶段的并行执行，所以它也就无法为用户提供这个功能。

在 MapReduce 中，数据是通过一个叫做 InputFormat 的类读取的。InputFormat 的部分作用就是告诉 MapReduce 需要执行多少个 map 任务，同时它也会建议在哪里应该执行。

尽管 Pig 不能给用户对执行多少个 map 任务进行直接控制，它却提供了方式允许用户编写自己的加载函数，对这个加载函数用户可以重写和运行自己的 InputFormat 方法。

5.4 用户自定义函数 UDF

Pig 的一大特色在于它允许用户通过 UDF 将 Pig 和操作符以及用户的代码或其他人提供的代码合并在一起使用。在 0.7 版本之前，所有的 UDF 必须是使用 Java 语言编写并实现为 Java 类[1]。这使得用户很容易通过写 Java 类向 Pig 中增加新 UDF 并且告诉 Pig 用户编写的 JAR 文件在什么地方。

在 0.8 版本中，也可以使用 Python 编写 UDF。Pig 使用 Jython 来执行 Python 编写的 UDF，因此这些 UDF 必须使用 Python 2.5 进行编译，这样也就不能使用 Python 3 中的功能了。

Pig 本身也包含了一些 UDF。0.8 版本之前，本身只带了少量的 UDF，只包括标准的 SQL 聚合函数和其他一些 UDF。在 0.8 版本中，增加了大量的标准字符串处理函数、数学函数和复杂类型 UDF。附录 A 中的“内置 UDF”章节有这些内置 UDF 的完整的列表和描述。

Piggybank 是和 Pig 一起发布的，其包含了用户贡献的 UDF 集合。Piggbank 中的 UDF

[1] 这就是 UDF 名称为什么是大小写敏感的原因了。

并不包含在 Pig 的 JAR 中，因此如果用户想使用这些 UDF 的话需要在脚本中进行注册。附录 A 中的“Piggybank”章节会有更详尽的介绍。

用户可以自己写 UDF 当然也可以使用这些其他用户提供的 UDF。在第 10 章将会介绍如何写 UDF。最后需要说明的是，一些 Java 静态函数也是可以作为 UDF 的。

5.4.1 注册 UDF

当用户需要使用一个非 Pig 内置的 UDF 时，需要告诉 Pig 到哪里去查找那个 UDF。可以通过 **register** 命令完成这个过程。例如，假设用户想使用 Piggybank（附录 A 中的“Piggybank”章节介绍了从哪里可以获得 Piggybank JAR）中提供的 Reverse 这个 UDF：

```
--register.pig
register 'your_path_to_piggybank/piggybank.jar';
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
          date:chararray, dividends:float);
backwards = foreach divs generate
          org.apache.pig.piggybank.evaluation.string.Reverse(symbol);
```

这个例子告诉了 Pig 在生成 JAR 传送给 Hadoop 的时候需要包含 *your_path_to_piggybank/piggybank.jar* 中的内容。Pig 会打开所有这些注册进来的 JAR，取出所有文件，然后把它们全部打入那个需要传送给 Hadoop 执行用户的任务的 JAR 中。

在这个例子中，我们需要告诉 Pig 完整的包名和 UDF 函数的类名。有两种方式可以简化这种冗长的使用方式。第 1 种方式是使用 **define** 命令（5.4.2 小节“define 命令和 UDF”会详细介绍）。第 2 种方式是在命令行为 Pig 指定一组路径用于查找需要的 UDF。因此不是以 **pig register.pig** 方式调用 Pig，而是采用 **pig –Dudf.import.list=org.apache.pig.piggybank.evaluation.string register.pig**，把脚本修改成如下形式：

```
register 'your_path_to_piggybank/piggybank.jar';
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
          date:chararray, dividends:float);
backwards = foreach divs generate Reverse(symbol);
```

使用另一种属性，我们也可以不用先注册命令。如果我们在命令行增加 **–Dpig.additional.jars=/usr/local/pig/piggybank/piggybank.jar**，那么根本就不需要使用注册命令了。

很多情况下在脚本中使用 register 和 define 命令显式地注册和定义 UDF 要比通过设置属性值的方式加载好。否则，使用脚本的用户也必须知道在命令行如何配置这些属性。然而，在一些情况下用户的脚本总是使用相同的一组 JAR，同时总是在一组相同的位置去查找它们。比如，用户可能有一组全公司都使用的 JAR。在这种情况下，把这些属性信息放到一个共享的属性配置文件中，然后使用这个配置运行 Pig 脚本将会使得共享这些 UDF 变得容易，同时也利于保证大家都在使用相同的正确版本的 UDF。

在 0.8 以及之后的版本中，**register** 命令也是可以接受 HDFS 路径的。如果用户的 JAR 存放在 HDFS 中，用户可以通过 register 'hdfs://user/jar/acme.jar'的方式加载。从 0.9 版本开始，register 也是可以接受正则匹配的。因此如果用户的 JAR 都放在一个目录下，那么也可以通过如下方式把它们都加载进来：register '/usr/local/share/pig/udfs/*.jar'。

注册 Python UDF

register 也可以加载用户的 Pig Latin 脚本中用到的 Python 编写的 UDF 所需要的资源。在这种情况下，用户不是注册一个 JAR 文件，而是注册一个包含用户的 UDF 的 Python 脚本文件。这个 Python 脚本必须在用户的当前目录下。使用样例代码中提供的例子，将 *udfs/python/production.py* 复制到数据目录下，如下所示：

```
--batting_production.pig
register 'production.py' using jython as bballudfs;
players = load 'baseball' as (name:chararray, team:chararray,
              pos:bag{t:(p:chararray)}, bat:map[]);
nonnull = filter players by bat#'slugging_percentage' is not null and
              bat#'on_base_percentage' is not null;
calcprod = foreach nonnull generate name, bballudfs.production(
               (float)bat#'slugging_percentage',
               (float)bat#'on_base_percentage');
```

这里一个重要的不同点是 register 语句中的 using jython 和 as bballudfs 部分。using jython 告诉 Pig 这个 UDF 是使用 Python 编写的，而不是使用 Java 编写，需要使用 Jython 对这个 UDF 进行编译。Pig 并不知道在用户的系统中 Jython 解释器所在的路径，因此当调用 Pig 时用户需要保证 *jython.jar* 已经放到类路径中了。用户可以通过设置 **PIG_CLASSPATH** 环境变量来指定该路径。

as bballudfs 这句话为从这个文件中加载的 UDF 定义了一个命名空间。所有这个文件定义的 UDF 目前都需要通过 **bballudfs.***udfname* 的方式进行调用。用户加载的每一个 Python 文件都需要给予一个不同的命名空间，这样可以避免用户注册两个 Python 脚本中相同函数名所产生的冲突。

这里有个警告：Pig 不会追踪用户的 Python 脚本中的依赖关系而会向用户的 Hadoop 集群中传送需要的 Python 模块。用户需要确保所使用到的 Python 模块在用户的集群中的任务节点是存在的，而且在这些节点上已经设置好了 **PYTHONPATH** 这个环境变量，只有这样才能保证用户的 Python UDF 可以正确地把这些模块加载进去。0.9 版本之后，此问题已经解决了，但是截止到写本章时还没有正式发布 [1]。

5.4.2 define 命令和 UDF

正如之前提到的，**define** 命令可用于为用户的 Java UDF 定义一个别名，那样用户就不

[1] Pig 0.9*. 版本当使用 Hadoop 0.23 版本时还是存在 Bug 的，Pig 0.10.1 版本才完全解决这个问题。

需要写那么冗长的包名全路径了。它也可以为用户的 UDF 的构造函数提供参数。**define** 命令同样可用于定义 streaming 命令，但是本小节我们只讨论 UDF 相关的功能。对于如何在 streaming 中使用 **define** 命令的详细信息，请查看 6.2.1 节“stream”。以下是一个使用 define 为 org.apache.pig.piggybank.evluation.string.Reverse 定义一个别名的例子：

```
--define.pig
register 'your_path_to_piggybank/piggybank.jar';
define reverse org.apache.pig.piggybank.evaluation.string.Reverse();
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
          date:chararray, dividends:float);
backwards = foreach divs generate reverse(symbol);
```

数学函数和过滤函数也是可以有一个或多个字符串类型的构造函数参数。如果用户使用的是一个接受构造函数参数的 UDF，**define** 命令后面就可以放置这些参数。例如，假设一个方法 CurrencyConverter 的构造函数需要两个参数，第 1 个参数是需要被转换的货币类型，第 2 个参数是需要转换成的货币类型：

```
--define_constructor_args.pig
register 'acme.jar';
define convert com.acme.financial.CurrencyConverter('dollar', 'euro');
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
           date:chararray, dividends:float);
backwards = foreach divs generate convert(dividends);
```

5.4.3 调用静态 Java 函数

Java 具有丰富的工具集和函数库，因为 Pig 是使用 Java 实现的，所以 Java 中的一些函数也是可以暴露给 Pig 用户的。从 0.8 版本开始，Pig 提供了 *invoker* 方法允许用户像使用 Pig UDF 一样使用一些特定的静态 Java 函数。

所有的没有参数或者有 int、long、float、double、String 或者 array 类型的参数[1]，同时有 int、long、float、double 或 String 类型返回值的静态 Java 函数，都是可以通过这种方式进行调用的。

因为 Pig Latin 不支持对返回值的数据类型进行重载，因此对于每一种类型有一个对应的调用方法：InvokeForInt、InvokeForLong、InvokeForFloat、InvokeForDouble 和 InvokeForString。用户需要根据期望的返回值的数据类型调用适当的调用方法。该方法有两个构造参数。第 1 个参数是完整的包名、类名和方法名。第 2 个参数是一个以空格作为分割符的参数列表，这些参数将传送给这个 Java 函数，只包含参数的类型。如果这个参数是个数组，[]（方括号）会跟在这个类型名称后面。如果这个方法不需要参数，那么第 2 个构造参数会省略。

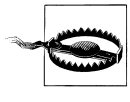
[1] 对于 int、long、float 和 double，调用方法可以调用这些以基本数据类型为输入参数的 Java 函数，但是不能接受对应的 Java 类对象（因此可以使用 int，但不可以使用 Integer 等）。

例如，如果用户想使用 Java 的 **Integer** 类将十进制的数值转换为十六进制的值，需要按如下方式使用：

```
--invoker.pig
define hex InvokeForString('java.lang.Integer.toHexString', 'int');
divs = load 'NYSE_daily' as (exchange, symbol, date, open, high, low,
          close, volume, adj_close);
nonnull = filter divs by volume is not null;
inhex = foreach nonnull generate symbol, hex((int)volume);
```

如果用户的方法接受的是一组数据类型的参数，那么 Pig 会将它封装成一个 bag，这个数据包中每个 tuple 只会有一个这种类型的字段。因此如果用户有一个 Java 方法 com.yourcompany.Stats.stdev 参数是一组 double 类型的值，那么用户可能是通过如下方式使用的：

```
define stdev InvokeForDouble('com.acme.Stats.stdev', 'double[]');
A = load 'input' as (id: int, dp:double);
B = group A by id;
C = foreach B generate group, stdev(A.dp);
```



警告

调用器不会使用累计器（Accumulator）或者代数器（Algebraic）接口，也因此可能会比专门为 Pig 所编写的 UDF 运行得要慢、占用更多的内存。这意味着在用户计划向一个调用方法传入一组参数的时候，需要仔细考虑这种降低性能的方式是否是可以接受的。10.3 节“累加器接口”和 10.2 节“代数运算接口”有详细的介绍。

采用这种方式调用 Java 函数，需要付出一些性能上的代价，因为需要使用反射机制来找到和调用这些方法。

当输入的参数是 null 的时候，调用器函数会抛出一个 **IllegalArgumentException** 异常。因此调用之前用户需要放置一个过滤器来避免异常的出现。

第 6 章

Pig Latin 高级应用

在前一章我们完成了 Pig Latin 基础知识的学习。本章我们将加大深度，讨论 Pig 是如何处理更加复杂的数据流的。在本章最后，将介绍如何使用宏命令和模板对用户的脚本进行模板化。

6.1 高级关系操作

现在我们讨论更高级的 Pig Latin 操作符，也就是上一章介绍的操作符的其他一些操作方法。

6.1.1 foreach 的高级功能

在前面对 **foreach** 的简单介绍中，我们讨论了在数据流中它是如何从每条记录中获得一系列的表达式进行输出的。现在我们来看看它可以采用哪些方式将数据流中的记录条数进行分解，以及它是如何被用于一组操作符从而对每条记录进行处理的。

flatten

有时用户的数据是存放在 bag 或者 tuple 中的，而用户想降低这个嵌套的级别。GitHub（请查看前言中的“本书中的代码样例”部分）上提供的 *baseball* 数据集可以作为一个例子。因为一个运动员可以充当多种角色，因此 **position**（角色）这个字段是以 bag 类型存储的。这样可以允许我们在 *baseball* 这个文件中仍然保证一个运行员对应一个条目[1]。但是当用户想快速地浏览数据然后根据一个特定的角色进行分组的时候，用户需要一种将这些条目从 bag 中分离出来的方法。为完成这个需求，Pig 在 **foreach** 中提供了

[1] 有数据库经验的人可能会注意到这是违反 E.F.Codd 定义的第一范式的。通常在 OLAP 系统中这种蓄意地反规范化的数据操作是非常常见的，特别是像 Hadoop 这样的大型数据处理系统。RDBMS 系统倾向于使 join 操作更普遍，因此会对它们进行优化。像 Hadoop 这样的系统，存储空间很便宜同时 join 的代价很高，通常使用嵌套的数据结构去避免进行 join 连接操作会更好些。

flatten 修饰符：

```
--flatten.pig
players = load 'baseball' as (name:chararray, team:chararray,
            position:bag{t:(p:chararray)}, bat:map[]);
pos = foreach players generate name, flatten(position) as position;
bypos = group pos by position;
```

包含一个 flatten 的 foreach 会导致 bag 中的每条记录会和其他所有 generate 语句中的表达式进行交叉乘积。我们来看下 *baseball* 文件中的第一行记录，内容如下（这里为清楚查看将制表符换成了逗号）：

```
Jorge Posada,New York Yankees,{(Catcher),(Designated_hitter)},...
```

一旦这行记录经过 **flatten** 语句处理后，将会产生两条记录：

```
Jorge Posada,Catcher
Jorge Posada,Designated_hitter
```

如果有多个 bag，而且每个都经常做 flatten 处理，那么这交叉乘积运算将使得 bag 的成员和 generate 语句中的其他表达式一样显示。因此结果将不是 n 行（这里的 n 表示一个 bag 中的记录数），而将是 n × m 行。

有一个可能会使很多用户惊讶的不好的方面是如果这个 bag 是空的，那么将不会产生任何记录。因此如果 *baseball* 文件中有一个条目是没有位置的，不管是因为 bag 是 null 还是空的，那么这条记录就不会包含在脚本 *flatten.pig* 的输出结果中。包含空 bag 的记录将会被 **foreach** 语句抛弃。采用这种处理方式的原因有两个。其一，既然 Pig 可以知道也可以不知道 bag 中数据的模式，那么它也就可能不知道怎样去为确定的字段填补什么样的值。其二，从数据角度来看，这也可能正是用户所期望的。将集合 S 和一个空集合做交叉乘积运算，结果是一个空集合。如果用户想避免这种情况，那么可以使用一个三元条件表达式将空 bag 替换成一个常量 bag：

```
--flatten_noempty.pig
players = load 'baseball' as (name:chararray, team:chararray,
          position:bag{t:(p:chararray)}, bat:map[]);
noempty = foreach players generate name,
          ((position is null or IsEmpty(position)) ? {('unknown')} :
position)
          as position;
pos = foreach noempty generate name, flatten(position) as position;
bypos = group pos by position;
```

对于 tuple 也同样可以使用 **flatten** 语句。在这种情况下，它不会产生一个交叉乘积结果，而它会使 tuple 中的每个字段转换成顶层的字段，对于空 tuple 将会移除整条记录。

如果 bag 或者 tuple 中的字段在进行 flatten 处理的时候命名了别名，那么 Pig 会一并将这些别名向数据流下个节点传。在进行 **join** 操作时，为了避免模糊不清，字段名前将加上 bag 的名称并且中间使用::符合连接。只要字段名不是模糊不清的，那么用户也并

非一定要使用 **bag 名称::** 这个前缀。

如果用户想改变这些字段的别名，或者这些字段在初始化的时候并没有别名，那么可以在 **flatten** 语句后加上 **as** 语句来指定字段别名，正如前面展示的例子中所使用的方式。如果 bag 或者 tuple 中有超过一个字段需要指定别名，那么用户需要使用圆括号将这组字段别名括起来。

最后需要说明的是，如果用户对没有模式的 bag 或者 tuple 进行 flatten 操作，同时没有使用 **as** 语句指定字段名时，那么这个 **foreach** 语句的输出结果也将是没有模式的。这是因为 Pig 并不知道 **flatten** 语句将会在结果中产生多少个字段 [1]。

内嵌 foreach

到目前为止，我们所看到的关于 **foreach** 的所有例子都是立即产生一行或多行输出。但是 **foreach** 的功能远比此强大。它也可以在用户的数据流中对每条记录进行一系列的关系操作，这就是所谓的*内嵌 foreach*，或称为*内部 foreach*。有一个例子就可以展示这种使用方式，即计算分组中不同条目的个数。例如，从数据文件 *NYSE_daily* 中查找每笔交易所对应的不同的股票交易号的个数：

```
--distinct_symbols.pig
daily = load 'NYSE_daily'as (exchange, symbol); --不关注其他字段
grpd  = group daily by exchange;
uniqcnt = foreach grpg {
          sym = daily.symbol;
          uniq_sym = distinct sym;
          generate group, COUNT(uniq_sym);
};
```

这里有几个新的知识点需要学习，后面我们将逐一介绍。在这个例子中，不是在 **foreach** 语句后立即跟着 **generate** 语句，而是在之后跟着{ (花括号)表明后面将在 foreach 语句里内嵌一些操作符。在这段内嵌代码中，传递给 foreach 的记录每次处理一条。

在这段内嵌代码的第 1 行，我们看到了一个在 foreach 语句外从没看到过的语法。事实上，**sym=daily.symbol** 这句在 foreach 语句外是非法的。这句和顶层操作语句 **sym = foreach grpd generate daily.symbol** 大体是等价的，但是因为这并非是真正的另一个 foreach 过程，所以在内嵌代码中并没有那么写。它们之间还不存在这个关系（也就是说，在这个时候还没有定义好 **grpd**）。这一行接受 **daily** 这个 bag，产生一个新的关系 **sym**，**sym** 是一个包含了只有一个 **symbol** 字段的所有 tuple 的 bag。第 2 行对关系 sym 进行 distinct 排重操作。需要注意的是即使在 foreach 内嵌语句内，关系操作符也只可以单独应用于关系，它们不可以应用于表达式。例如，语句 **unig_sym = distinct daily.symbol** 将会产生一个语法错误，因为 **daily.symbol** 是个表达式而不是关系。sym

[1] 在 0.8 和更早的版本中有一个 bug：在这种情况下不会产生一个空的模式，而是产生一个包含一个 bytearray 类型的字段的模式。这个 bug 在 0.9 版本中已经被修复了。

是一个关系。这个区别可能看上去有点武断，这是因为 Pig Latin 作为一种语言应该是有着一贯的约束定义。如果没有这种约束，那么一些奇怪的语句，例如 **C = distinct 1+2** 将是合法的。思考这个问题的方式之一是 foreach 内的赋值运算符可以连接一个表达式也可以创建一个关系，正如这个例子中所展示的。

foreach 内嵌结构内的最后一行通常或者一定是个 generate 语句。通过这个语句可以告诉 Pig 怎样接受内嵌的操作符的处理结果，以及将结果记录赋给另外一个关系（在本例中就是 uniqcnt）。因此，generate 操作符可以接受内嵌的关系操作并将它们重新转换成可供外部关系调用的表达式。也就是说，如果脚本读取 **generate group, unig_sym, uniq_sym** 这句，那么为了 generate 语句的目的这句结果会被作为 bag 处理。

理论上，任何一个 Pig Latin 关系操作符在 **foreach** 语句内都是合法的。然而，在目前情况下，只支持 **distinct、filter、limit 和 order。**

这个特点非常有用，我们可以通过一些其他的例子来看看，例如在一个数据包 bag 被传送给一个 UDF 前对数据包内的内容进行排序。这对于那些要求数据信息必须有一定次序的 UDF 来说是很方便的。假设有一个股票分析 UDF，用于按实际跟踪一支特定股票的交易信息。这个 UDF 需要输入是按照时间戳排序的：

```
--analyze_stock.pig
register 'acme.jar';
define analyze com.acme.financial.AnalyzeStock();
daily = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
          date:chararray, open:float, high:float, low:float,
          close:float, volume:int, adj_close:float);
grpd = group daily by symbol;
analyzed = foreach grpd {
             sorted = order daily by date;
             generate group, analyze(sorted);
};
```

在 Pig Latin 脚本中做排序，而不是在 UDF 中进行排序，这样做有两个重要的原因。其一，这意味着 Pig 可以使用 MapReduce 进行排序。MapReduce 在对数据进行分组的时候同时按照其他的一个键对数据进行排序。因此，这种情况下 order 语句并不需要一个额外的排序操作。其二，这也意味着用户的 UDF 并不需要所有的数据都可供使用时才开始进行处理。相反地，它可以使用 Accumulator 接口（请参考 10.3 节“累加器接口”），这个接口使内存更加高效。

该功能可用于从一组记录中找出前 k 个元素。下面的这个例子展示了查找每支股票对应支付的前 3 个股息值：

```
--hightest_dividend.pig
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
           date:chararray, dividends:float);
grpd = group divs by symbol;
```

```
top3 = foreach grpd {
            sorted = order divs by dividends desc;
            top = limit sorted 3;
            generate group, flatten(top);
};
```

目前，这些内嵌的代码部分对于每条传送给它们的记录总是连续执行的。当然 foreach 本身将会以多个 map 和 reduce 任务方式运行，但是每个 foreach 实例不会产生子任务以并行的方式来执行内嵌的那些处理过程。因此，如果我们在上面的那个例子中 grpd = group divs by symbol 语句后增加一行 parallel 10 语句，这里排序和 limit 操作将占用 10 个 reducer。但是每个对股票的分组将会串行地使用这 10 个 reducer 中的一个进行排序和取前 3 条记录操作。

当然，并不要求 foreach 内部的数据流必须是线性数据流。例如，如果用户想一起计算两个排重数，那么可以按如下方式做：

```
--double_distinct.pig
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray);
grpd = group divs all;
uniq = foreach grpd {
            exchanges = divs.exchange;
            uniq_exchanges = distinct exchanges;
            symbols = divs.symbol;
            uniq_symbols = distinct symbols;
            generate COUNT(uniq_exchanges), COUNT(uniq_symbols);
};
```

为简单起见，Pig 实际上对这里的每个表达式都使用 generate 处理一次这个数据流。这里是可行的，因为这两个数据流是完全无关的。然而，如果用户构造了一个数据流，其中进行了一次分割，同时对共享的部分使用了 UDF，那么用户将会发现它被调用的次数比想象的要多得多。

6.1.2 使用不同的 Join 实现方法

我们在前一章介绍了 **join**（请查看 5.3.6 小节“Join”），但只是介绍了 join 默认的操作方式。然而，Pig 提供了多种 join 实现方式，下面我们将做具体介绍。

在 RDBMS 系统中，传统上 SQL 优化器会为用户选择 join 实现。如果优化器选择的好那么这是非常棒的，而大多数情况下确实是那样。但是 Pig 采用了一个不同的方式。在 Pig 团队中我们喜欢说我们的优化器处于用户的座椅和键盘之间。我们授权给用户允许他们自己进行选择而非让 Pig 进行选择。因此像 join 这样有多个实现的操作符，Pig 允许用户使用 using 语句进行指定他所需要的实现。

这种处理方式与我们的 Pig 是家畜（例如，Pig 按照用户的命令做事，请查看 1.1.4 节“Pig 的设计思想”）这个设计理念是一致的。同时，作为一个相对较新的项目，Pig 还有很

多新的功能需要增加。专注于增加多个实现以供选择同时让用户进行选择使用合适的实现显得更有意义，而不是专注于做一个优化器去自己选择合适的实现。

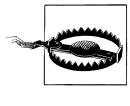
小数据和大数据进行 join 操作

使用 join 的一个通常场景是根据一个较少的输入数据进行查找。例如，设想用户正在进行将美国 ZIP 码（邮政编码）转换成对应的州和城市的数据处理。因为美国至多有 100,000 个邮政编码，这个转换表应该很容易放入内存中。不采用强制触发一个 reduc 阶段对大文件和这个小小的邮政编码转换文件进行排序，而采取将这个邮政编码转换文件分发到每台机器中，然后加载到内存中，最后再使用邮政编码文件中的每条记录对大文件进行 join 操作的方式显得更有道理。这被称为分片-重复 join（因为用户将一个文件分片而对另一个文件进行复制）：

```
--repljoin.pig
daily = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
            date:chararray, open:float, high:float, low:float,
            close:float, volume:int, adj_close:float);
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
            date:chararray, dividends:float);
jnd = join daily by (exchange, symbol), divs by (exchange, symbol)
            using 'replicated';
```

其中 using 'replicated'这句就告诉了 Pig 使用分片-复制算法执行这个 join 操作。因为没必要触发 reduce 过程，所以所有这些都可以在 map 任务阶段处理完成。

join 语句中的第 2 个输入（在例子中就是 divs）会是被加载到内存中的输入。Pig 不是事先检查指定的输入是否可以完成装入内存。如果 Pig 不能够将这些复制的输入加载入内存，那么将会产生一个错误然后执行失败。



警告

由于 Java 会在内存中存储对象，磁盘上数据大小并不等同于内存中的数据大小。请查看 4.1.2 节“Pig 不同数据类型对内存的需求情况”了解关于在 Pig 中数据到内存是如何扩张的讨论。用户会需要比在磁盘上存储冗余输入数据大的内存空间。

分片-冗余 join 只支持 inner 和 left outer join。它不可以做一个 right outer join，因为当一个指定的 map 任务在冗余输入信息中查看到一条记录和分片输入中任一条记录都不匹配的时候，它将不能确定是否可以在一个不同的分片输入中匹配到记录。因此，它不知道是否省略掉一条记录。如果用户需要一个 right 或者 full outer join，那么用户需要使用默认的 join 操作符。

分片-冗余 join 操作可用于多张表间的操作。在这种情况下，只有第一个（也就是最左边）的那个表会被读入内存中。

Pig 是通过将冗余输入放到 Hadoop 的分布式缓存中的方式来实现分片-冗余 join 的。分布式缓存是 Hadoop 提供的一个工具，它可以将一个文件预先存放到将要执行 map 或 reduce 任务的节点上，这样做有两个好处。其一，如果用户有一个分片-冗余 join 操作将执行 1,000 个 map 任务，在 HDFS 中从 1,000 个不同的机器上同时打开一个文件将会对 NameNode 已经保证这个文件数据块的 3 个节点造成严重的负担。分布式缓存是特意被设计用来控制这样的问题而不会对 namenode 造成很大负荷。其二，如果多个 map 任务都落到了同一个物理机器上，那些任务实例将可以共享使用分布式缓存中的文件，因此这可以降低文件被复制的次数。

Pig 会触发一个只有 map 过程的 MapReduce 任务对这个文件进行预处理，为把它加载到分布式缓存中做准备。如果在 load 和 join 操作符之间有过滤器 filter 或者 foreach 操作，这些将会作为初始化任务的一部分完成，因此存储到分布式缓存中的文件是越小越好。join 本身将会在第 2 个只有 map 过程的任务中完成。

对倾斜的数据进行 join 操作

正如我们在其他地方看到的一样，使用 Pig 进行处理的大部分数据在每个键的记录个数上具有明显的倾斜。例如，假如我们创建一个网站地图而且是根据 URL 中的域名（也就是作为键）进行连接操作，那么我们将会看到对于键 *yahoo.com* 对应的值具有明显的倾斜。Pig 默认的 join 连接算法对于倾斜是十分敏感的，因为它是在一个单独的 reducer 中收集关于给定的键的所有记录。在很多数据集中，一些键比其他的键具有三倍或者更多倍数数量级的记录。这会导致一两个 reducer 消耗的执行时间比其他的 reducer 要多得多。为了解决这个问题，Pig 提供了 *skew join* 连接方法。

skew join 通过先对输入数据进行样本抽样的方式来执行连接操作。在这个输入中它会识别出所有对应着非常多的值以致于 skew join 认定它将无法将它们完全载入内存中的键。然后，在第 2 个 MapReduce 任务中，进行那个 join 连接操作。除了这些在样本中确认的记录外的其他所有记录，将进行一个标准的 join 连接操作，收集相同键对应的所有记录然后传送给同一个 reducer 中。那些被认为是太大的键将会采用不同的方式处理。基于对于一个特定的键对应着记录的个数，这些记录将会被划分给一个合理的个数的 reducer 进行处理。所选择的 reducer 的个数是基于 Pig 对数据应该划分为多大，然后可以使得每个 reducer 可以将它接收的那块数据存放到内存中去进行处理而进行的评估。join 操作中对于没有进行划分的输入，被划分了的那些键将会作为冗余分发到包含那些键的 reducer 中 [1]。

例如，我们一起看看下面的这个 Pig Latin 脚本是如何执行的：

```
users = load 'users' as (name:chararray, city:chararray);
```

[1] 这个算法是在论文《并行连接操作中处理数据倾斜的一种有效办法》中提出的，这篇文章是由 David J.DeWitt、Jeffrey F.Naughton、Donovan A.Schneider 和 S.Seshadri 在第 18 届海量数据库网络大会上发表的。

```
cinfo = load 'cityinfo' as (city:chararray, population:int);
jnd = join cinfo by city, users by city using 'skewed';
```

假设 *users* 中用户所在城市的分布情况是这样的：有 20 个人居住在巴塞罗那，有 100,000 个人居住在纽约，还有 350 人居住在波特兰。我们可以进一步假设 Pig 是认为对于每个 reducer 将 75,000 条记录载入内存是合理的。但这个数据进行连接操作时，纽约将会是被认为是需要划分到不同的 reducer 之间的键。在 join 阶段，除了纽约这个键之外的其他键都将作为默认的 join 连接操作进行处理。*users* 中以纽约作为键的所有记录将会被划分到两个 reducer 中去执行。*cityinfo* 文件中以纽约为键的记录将会产生两份冗余分别传送到这些 reducer 中去。

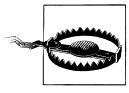
join 连接操作的第 2 个输入，在这里也就是 users，将会被进行样本抽样并且会将对应的记录的条数非常大的键分拆到各个 reducer 中。第 1 个输入会对包含这些值的进行冗余复制然后分发到各个 reducer 中。

这个算法解决了有一个输入有数据倾斜时的问题。如果两个输入都有数据倾斜的话，虽然该算法仍然可以起作用，但是会慢。采用这种处理方式的部分动机在于只要给予时间，join 连接操作最终是可以完成的。Pig 在 0.4 版本中介绍 skew join 之前，当两边的数据都发生数据倾斜时，Pig 中是无法进行 join 操作的，因为不太可能可以将任何一边的高基数的记录全部载入内存中。

skew join 可以在 inner join 或者 outer join 中完成。然而，它只能够接受两个连接输入。如果需要使用 skew join，多路 join 连接操作需要分解成一组 join 连接操作。

既然数据经常倾斜，那么为什么不总是使用 skew join 呢？原因是使用 skew join 对性能有一定的影响，这是因为输入之一必须要先进行样本抽样去发现哪些键对应了大量的记录。这通常会增加 5%的时间用于计算这个连接操作。如果用户的数据经常发生倾斜，为了避免失败或者使用默认的 join 操作时运行的非常慢以致于需要重新使用 skew join 进行执行，那么增加这 5%的时间使用 skew join 也是值得的。

正如早些时候陈述的，当要确定对哪些键需要进行划分以及划分的宽度有多大时，Pig 会评估可以将多少的数据载入内存。为了达到这次计算的目的，Pig 会查看样本中的记录大小并假设它可以使用 30%的 JVM 堆栈来具体化将要进行连接的记录。在特定的情况下可能需要增加或者减少这个堆栈百分比。使用 skew join 时，如果用户发现 join 连接操作仍然是失败的并且报内存溢出错误的话，那么用户应当减少这个值。这表明 Pig 没有预测出一个合理的内存使用，因此应该通知它使用少一些。如果概要分析显示 Pig 没有充分地使用堆栈，为了能更有效地执行 join，用户可能需要增加这个值；键值被划分的方式越少，连接操作就越高效。用户可以通过设置属性 pig.skewed.join.reduce.memusage，一个在 0 和 1 之间的值来进行调节。例如，如果用户想使用 25%而不是 30%，那么用户可以在 Pig 命令行增加 -Dpig.skewed.join.reduce.memusage=0.25 或者在属性文件中设置这个值。



警告

和 order 一样，skew join 打破了同一个键对应的所有值将由同一个 reducer 处理这个 MapReduce 惯例。这意味着包含相同键的记录可能会放在不同的文件片中。如果计划下一步处理数据的前提是要求具有相同键的所有记录需要在同一个文件片中，那么就不能使用 skew join。

对排好序的数据进行 join 操作

一个常见的数据库连接策略是首先对两边的输入按照 join 的键先进行排序操作，之后同时扫描两边的输入，进行 join 连接操作。这就是所谓的先排序-再合并连接操作。在 MapReduce 中，因为一次排序需要一个完整的 MapReduce 任务，正如 Pig 默认的连接操作一样，这个技术不会比默认的连接操作要高效多少。连接操作可以在 map 阶段通过打开并扫描两边的文件完成。Pig 称这种方式为 *merge join*，因为它是一个先排序再合并连接的操作，但是排序的过程已经完成了：

```
   --mergejoin.pig
   -- 使用脚本 sort_for_mergejoin.pig 来处理 NYSE_daily_sorted 和
NYSE_dividends_sorted
   daily = load 'NYSE_daily_sorted' as (exchange:chararray,
symbol:chararray,
             date:chararray, open:float, high:float, low:float,
             close:float, volume:int, adj_close:float);
   divs = load 'NYSE_dividends_sorted' as (exchange:chararray,
symbol:chararray,
             date:chararray, dividends:float);
   jnd = join daily by symbol, divs by symbol using 'merge';
```

执行这个连接，Pig 会首先执行一个 MapReduce 任务对第 2 个输入进行抽样，也就是 *NYSE_dividends_sorted*。这个样本会创建一个索引来告诉 Pig 进行连接的键的值，*symbol* 是每一个输入划分（通常是每个 HDFS 数据块）中的第 1 条记录。因为抽样过程只会读取每个划分的一条记录，所以这个过程是很快的。Pig 之后会执行第 2 个 MapReduce 任务接收第一个输入，也就是 *NYSE_daily_sorted* 作为输入。当每个 map 任务读取 *NYSE_daily_sorted* 划分中的第一条记录时，它会获取 symbol 的值然后在由前一个任务所创建的索引中查找这个值。它会查找比 symbol 值小的最后一个条目，然后在这个条目对应的数据块中打开 *NYSE_daily_sorted*。例如，如果索引中包含条目（CA，1），（CHY，2），（CP，3），同时 *NYSE_daily_sorted* 的一个给定的输入划分的第一个 symbol 的值是 CJA，那个 map 任务将会打开 *NYSE_daily_sorted* 第 2 个数据块。（即使 CP 曾是 *NYSE_daily_sorted* 划分中的第一个用户 ID，*NYSE_daily_sorted* 第 2 个数据块将会被打开，因为在那个数据块中可能会有键值为 CP 的记录。）一旦 *NYSE_daily_sorted* 被打开了，Pig 会不断过滤记录直到它找到 symbol 为 CJA 的记录。一旦它发现了一个匹配的，它会收集所有包含那个值的记录然后载入内存中，之后进行连接操作。然后会将第 1

个输入提前，也就是 *NYSE_daily_sorted*。如果这个键是相同的，它会再次进行连接。如果不同，它会把第 2 个输入提前，也就是 *NYSE_dividends_sorted*，同样的直到发现比第一输入，也就是 *NYSE_daily_sorted* 的下个值要大或者相等的值。如果这个值更大，则会把第 1 个输入提前并持续下去。因为两边的输入都是排好序的，所以初始化检查之后就再也不需要查询索引了。

完成所有的这些都不需要 reduce 过程，也因此这会比默认的 join 方式要更加高效。这个算法是在 0.4 版本中引入的，目前只支持两层的 inner join。

6.1.3 cogroup

cogroup 是 group 的一般化方式，不是基于一个键收集一个输入的记录，而是基于一个键收集多个输入的记录。结果是一条包含一个键和每个输入的一个数据包 bag 的记录。每个 bag 中包含了具有那个键对应的值的输入的所有记录：

```
A = load 'input1' as (id:int, val:float);
B = load 'input2' as (id:int, val2:int);
C = cogroup A by id, B by id;
describe C;

C: {group: int,A: {id: int,val: float},B: {id: int,val2: int}}
```

另外一种思考 cogroup 的方式是可以认为它是 join 的前期的一半操作。这些键被收集在一起，但是交叉乘积并没有完成。事实上，cogroup 加上 foreach，其中每个数据包都被平整过了，与 join 是等价的，只要这些键对应的值没有是 null 的。

cogroup 处理键中的 null 值的方式和 group 相似但是与 join 不同。也就是说，键对应的为 null 值的所有记录会被收集到一起。

当用户想做一个类似 join 的操作而非一个 full join 操作的时候，使用 cogroup 是有帮助的。例如，Pig Latin 并没有一个 semi-join 操作符，但是用户可以做一个 semi-join 操作：

```
--semijoin.pig
daily = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
            date:chararray, open:float, high:float, low:float,
            close:float, volume:int, adj_close:float);
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
            date:chararray, dividends:float);
grpd = cogroup daily by (exchange, symbol), divs by (exchange, symbol);
sjnd = filter grpd by not IsEmpty(divs);
final = foreach sjnd generate flatten(daily);
```

因为 cogroup 需要将具有相似键的记录收集到一起，所以这里会有一个 reduce 过程。

6.1.4 union

有时需要将两个数据集合并到一起而不是 join 连接到一起。基于这个目的，Pig Latin

提供了 union 操作符。如果想把两个文件作为输入而同时对于它们并没有一个模式描述，那么可以通过如下方式操作：

```
A = load '/user/me/data/files/input1';
B = load '/user/someoneelse/info/input2';
C = union A, B;
```

提示

与 SQL 中的 union 不同，Pig 并不要求双方输入的数据具有相同的模式。如果双方输入的数据确实具有相同的模式，那么 union 操作的输入也将具有这个模式。如果一个模式可以通过一组隐式的类型转换可以由另一个输入的模式生成的话，那么 union 的结果会具有那个类型转换后的模式。如果这两种情况都不满足，那么结果就是没有模式的（也就是说，不同的记录会具有不同的字段）。这个模式比较是包含命名比较的，因此如果字段名称如果不一致也会导致结果输出是没有模式的。可以通过在在 union 操作之前增加一个 foreach 操作来对字段进行重命名来避免这个问题。

```
A = load 'input1' as (x:int, y:float);
B = load 'input2' as (x:int, y:float);
C = union A, B;
describe C;

C: {x: int,y: float}

A = load 'input1' as (x:int, y:float);
B = load 'input2' as (x:int, y:double);
C = union A, B;
describe C;

C: {x: int,y: double}

A = load 'input1' as (x:int, y:float);
B = load 'input2' as (x:int, y:chararray);
C = union A, B;
describe C;

Schema for C unknown.
```

union 并没有执行一个数学上的集合 union 过程。也就是说，不会省略重复的记录。从这上看与 SQL 中的 union all 操作类似。同时，union 不需要一个额外的 reduce 过程。

有时随着时间数据也发生改变。如果有需要每月进行收集的数据，那么可能就需要为这个月增加一个新的列。现在因为模式是不匹配的，所以会强制禁止使用 union。如果想把这个数据合并进来同时将数据强制生成一个通用的模式，那么可以通过在 union 语句中增加 **onschema** 关键字的方式实现：

```
A = load 'input1' as (w:chararray, x:int, y:float);
B = load 'input2' as (x:int, y:double, z:chararray);
C = union onschema A, B;
describe C;

C: {w: chararray,x: int,y: double,z: chararray}
```

union onschema 需要所有的输入都要有模式，同时也要求所有的输入能够通过增加字段和隐式的类型转换产生一个共享的模式。字段键的匹配是通过名称进行的匹配，而不是通过位置匹配的。因此，在前面的例子中，从输入 *input1* 中增加了 w:chararray 字段，从输入 *input2* 中增加了 z:chararray 字段。同时，输入 *input1* 中增加了一个从 float 到 double 的类型转换，因此字段 y 是 double 类型的。通过这种方法如果不能产生一个共享的模式，那么就会返回一个错误信息。在读取数据的过程中，如果给定的输入中没有相关的字段，那么值用 null 填补。

6.1.5 cross

cross 和数学运算中的相同名称的数组操作是一致的。在如下的 Pig Latin 脚本中，cross 接收 NYSE_daily 文件中的每条记录然后和 NYSE_dividends 文件中的每条记录进行结合：

```
--cross.pig
-- 这个过程会产生大约 3GB 的数据，因此最好在集群中执行
daily    = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
             date:chararray, open:float, high:float, low:float,
             close:float, volume:int, adj_close:float);
divs     = load 'NYSE_dividends' as (exchange:chararray,
symbol:chararray,
             date:chararray, dividends:float);
tonsodata = cross daily, divs parallel 10;
```

cross 常常会产生大量的数据。如果给定的输入分别具有 n 和 m 条记录，那么 cross 的输出会产生 n × m 条记录。

Pig 确实是以并行的方式实现 cross 的。它通过生成一个综合的 join 键，对行进行冗余，然后将 cross 作为一个 join 来执行的方式实现。前面的脚本可以按如下方式重写：

```
daily = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
            date:chararray, open:float, high:float, low:float,
            close:float, volume:int, adj_close:float);
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
            date:chararray, dividends:float);
A = foreach daily generate flatten(GFCross(0, 2)), flatten(*);
B = foreach divs generate flatten(GFCross(1, 2)), flatten(*);
C = cogroup A by ($0, $1), B by ($0, $1) parallel 10;
tonsodata = foreach C generate flatten(A), flatten(B);
```

GFCross 是一个内置的 UDF。第 1 个参数是输入的编码，第 2 个参数是输入的总个数。

在这个例子中，输出是包含 4 条记录[1]的数据包。这些记录具有的模式是（int,int）。与 GFCross 第 1 个参数相同的那个字段包含了一个 0 和 3 之间的随机数。其他字段从 0 计数到 3。因此，如果假设有两条给定的记录，两个输入各包含一条，那么对于第 1 个输入的随机数是 3，对于第 2 个输入的随机数是 2，因此 GFCross 的输出看上去如下所示：

```
A {(3, 0), (3, 1), (3, 2), (3, 3)}
B {(0, 2), (1, 2), (2, 2), (3, 2)}
```

当这些记录通过 flatten 操作后，map 中会对每条输入记录产生 4 份冗余，然后它们通过合成的键进行 join 操作。对应各个输入中的每条记录，需要保证有且仅有一个合成的键对应的实例会匹配并且产生一条记录。因为对应各个记录所选择的随机数是各不相同的，所以 join 会均匀分布到 reducer 中执行。

这个算法确实实现了并行地对数据进行 cross 操作。然而，因为增加了各个输入中需要进行 shuffle 的记录数，所以会增加 shuffle 阶段的负担。同时，不管用户怎么做，cross 都会产生大量的输出数据。把这些数据写到磁盘代价是较高的，即使以并行的方式进行。

这并不是说不应该使用 cross，有些情况下使用它是不可或缺的。Pig 的 join 操作符只支持等值 join 操作，也就是说，在等值条件下进行 join 操作。因为一般的 MapReduce 的 join 实现（那些不依赖于数据需要事先排好序或者数据足够小可以载入内存）取决于将具有相同的 join 键对应的值的记录收集到同一个 reducer 中，非等值 join（也被称为 *theta* join）是很难去做的。这在 Pig 中可以通过使用 cross 以及在其后的 filter 语句来实现：

```
--thetajoin.pig
-- 同样建议在集群中执行这个脚本
daily    = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
             date:chararray, open:float, high:float, low:float,
             close:float, volume:int, adj_close:float);
divs     = load 'NYSE_dividends' as (exchange:chararray,
symbol:chararray,
             date:chararray, dividends:float);
crossed   = cross daily, divs;
tjnd     = filter crossed by daily::date < divs::date;
```

通过这种方式也可以进行模糊 join 操作，其中模糊比较在 cross 操作之后完成。然而，只要有可能，通过 UDF 将模糊的值转换成标准的值然后执行常规的 join 操作是更好的选择。例如，如果用户想按照城市来对两个输入进行 join 操作，但是想随时连接两个在之前的相同的大都市区的城市（例如，期望将“洛杉矶”和“帕萨迪纳”认为是等同的），用户可以先通过一个 UDF 对记录进行处理然后为所有在同一个大都市区的城市生成一个单独的 join 键然后再执行 join 连接操作。

[1] 在 0.8 或更早的版本中，记录的值总是 10。在 0.9 版本中，这个值变为并行因子的平方根，然后值向上取整。

6.2 在 Pig 中集成遗留代码和 MapReduce 程序

Pig 的设计思想中的一个信条是 Pig 允许用户在需要的时候将他们自己的代码整合到 Pig 中（请参考 1.1.4 节“Pig 的设计思想”）。Pig 实现这个思想的最显著的方式是通过 UDF 表现的，但是 Pig 也运行直接整合其他可执行的 MapReduce 任务。

6.2.1 stream

如果想在数据流中插入一个个性化的可执行任务，那么可以使用 stream 功能。当有一个遗留的程序同时不想去修改它或者是不能够修改它的时候，会使用到这个功能。通常也可以使用 stream 来处理经常使用的程序，或者是已经在小数据集上测试通过然后现在想在一个大数据集上执行的程序。下面我们来看一个 Perl 程序 *highdiv.pl* 的例子，这个例子用于过滤出股息少于$1.00 的所有股票：

```
-- streamsimple.pig
divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
highdivs = stream divs through `highdiv.pl` as (exchange, symbol, date,
dividends);
```

请注意 stream 命令中的 as 语句，此条语句并非是必须的。但是 Pig 并不知道这个执行过程的结果是什么，因此如果用户没有提供 as 语句，那么关系 highdivs 将没有模式。

可执行文件 *highdiv.pl* 在每个 map 任务或者 reduce 任务中都会被调用一次，不会对于每条记录都调用一次。Pig 实例化这个可执行文件然后保证数据持续输入是通过标准输入 *stdin* 实现的。同时也会持续检查标准输出 *stdout*，将所得的任意结果传送给数据流中的下一个操作符。这个可执行文件可以选择是否对每个输入产生一个输出，或者每一定数量的输入产生一个输出，或者只有当所有的输入都被接收后才会产生一个输出。

上面的例子假定集群中已经安装了 *highdiv.pl*，同时在任务机器上在工作目录下是可以执行的。如果没有满足这个条件，通常情况下也是不能执行的，那么可以通过 ship 命令将这个可执行文件加载到集群中。完成这个过程，需要使用 define 语句：

```
--streamship.pig
define hd 'highdiv.pl' ship('highdiv.pl');
divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
highdivs = stream divs through hd as (exchange, symbol, date, dividends);
```

define 语句做了两件事情。其一，它定义了将使用的可执行文件。现在在 stream 中我们通过可执行文件 *highdiv.pl* 的别名 hp 来引用它，而不是直接引用这个可执行文件。其二，它告诉了 Pig 需要加载文件*./highdiv.pl*，然后通过 ship 命令加载到 Hadoop 中作为这个任务的一部分。该文件会从用户提交这个任务的那台机器的指定位置进行加载。它将会被放置在任务机器执行任务所在的工作目录下。因此，传送给 stream 的命令必须指定的是相对于当前工作目录的相对路径，而不能通过绝对路径指定。如果用户的

可执行文件依赖于其他的模块或者文件，它们也是可以通过 ship 命令加载的。例如，如果文件 *highdiv.pl* 依赖于一个 Perl 模块 *Financial.pm*，那么可以按如下方法将它也发送到各个 task 机器上：

```
define hd 'highdiv.pl' ship('highdiv.pl', 'Financial.pm');
divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
highdivs = stream divs through hd as (exchange, symbol, date, dividends);
```

很多的脚本语言要求模板要按照它们的结构放在指定的路径上。例如，Perl 中模板 Acme::Financial 要求对应的路径应该是 Acme/Financial.pm。然而，ship 语句总是会将文件放置在当前的工作目录中，同时它不能够接受目录，因此无法使用 ship 加载目录 *Acme*。解决这个问题的办法是创建一个 TAR 文件然后使用 ship 语句加载这个文件，最后在执行过程中需要多一个过程来解压缩这个 TAR 文件。因此需要指定路径来使用模板（对于 Perl，使用-I 或者设置 PERL LIB 环境变量）以包含.（点号）。

ship 语句可以将提交任务的那台机器的文件分发到集群中去,但是有的时候集群中已经存在了这些文件。如果有一个文件需要每一个 map 或者 reduce 任务都能够访问到，这种情况下一种比较合适的访问该文件的方式是使用*分布式缓存*。分布式缓存是 Hadoop 提供的一种用于分享文件的机制。它通过将执行任务所需要的文件预先加载到对应机器的本地磁盘上的方式，从而减少了 HDFS 的加载负担。可以通过在 define 语句中的 cache 语句为可执行文件指定分布式缓存：

```
crawl = load 'webcrawl' as (url, pageid);
normalized = foreach crawl generate normalize(url);
define blc 'blacklistchecker.py' cache('/data/shared/badurls#badurls');
goodurls = stream normalized through blc as (url, pageid);
```

#号前面的部分是 HDFS 上的路径，在这个例子中，也就是/data/shared/badurls。#后面的字符串是可执行文件将读取的文件别名。因此，Hadoop 将会把/data/ shared/badurls 复制到各个 task 机器的工作目录下并命名为 badurls。

到这里为止我们都是假定可执行文件通过标准输入 *stdin* 读取数据然后通过标准输出 *stdout* 写数据，有时可能并非如此，这取决于可执行文件本身。如果可执行文件需要读取一个文件，或者写入一个文件，或者两者都是，是可以在 define 命令中指定输入和输出的。继续前面所举的例子，假定可执行文件 *blacklistchecker.py* 会通过命令行-i 指定一个输入文件然后通过-o 指定一个输出文件：

```
crawl = load 'webcrawl' as (url, pageid);
normalized = foreach crawl generate normalize(url);
define blc 'blacklistchecker.py -i urls -o good' input('urls')
output('good');
goodurls = stream normalized through blc as (url, pageid);
```

同样的，文件路径是 task 机器工作目录的相对路径。在这个例子中，Pig 将会把对于可

执行文件 *blacklistchecher.py* 指定的任务的所有输入都写入 *urls*，然后调用可执行文件，最后读取 *good* 以获得结果。同样的，每个 map 或者 reduce 任务只会调用这个可执行文件一次，因此 Pig 将会先把所有的输入写入那个文件中。

6.2.2 mapreduce

从 Pig 0.8 版本开始，通过 mapreduce 命令可以在数据流中直接添加 MapReduce 任务。当使用 MapReduce 处理比 Pig 要好但必须要和之前的 Pig 数据流相结合的时候，这个功能是很方便的。当对于新的处理想使用 Pig Latin 脚本而又想和之前旧的遗留下来的 MapReduce 程序结合使用时，这个功能也是便利的。

MapReduce 任务需要从存储设备（通常是 HDFS）上读取文件和写入文件。因此如果想把它们整合到数据流中，Pig 首先要做的就是存储数据，然后再调用 MapReduce 任务，最后重新读取数据。在 mapreduce 语句中通过 store 和 load 语句调用常规的加载和存储函数可以满足这个需求。也可以提供包含 MapReduce 任务的 JAR 文件给 Pig。

作为一个例子，我们继续之前章节所说的 URL 黑名单的例子。这里不过假设通过一个 MapReduce 任务实现而不是通过一个 Python 脚本实现：

```
crawl = load 'webcrawl' as (url, pageid);
normalized = foreach crawl generate normalize(url);
goodurls = mapreduce 'blacklistchecker.jar'
             store normalized into 'input'
             load 'output' as (url, pageid);
```

mapreduce 命令的第一个参数是包含 MapReduce 任务的 JAR 文件。load 和 store 语句表明了数据是如何从 Pig 的数据流移动到 MapReduce 任务中的。需要注意的是输入的别名是在 store 语句中定义的。和 stream 一样，Pig 对于 mapreduce 的输出是不清楚的，因此如果期望输出结果 goodurls 具有一个模式，那么需要告诉 Pig 模式具体是什么样的。这个例子也假定了 *blacklistchecker.jar* 中的 Java 代码知道从哪里读取输入和输出，同时在它的 manifest 文件中指定了缺省的可执行的类，这并非是通常的情况。在 load 语句的后面，可以通过反向的单引号指定需要传递给 Java 命令的参数：

```
crawl = load 'webcrawl' as (url, pageid);
normalized = foreach crawl generate normalize(url);
goodurls = mapreduce 'blacklistchecker.jar'
             store normalized into 'input'
             load 'output' as (url, pageid)
             'com.acmeweb.security.BlackListChecker -i input -o output';
```

上例中反向单引号中的字符串将直接传递给 MapReduce 任务。因此如果想传递一些 Java 配置项等，也可以通过这种方式。

mapreduce 命令中的 load 和 store 语句与常规的加载和存储函数的语法是一致的，因此可以使用不同的加载和存储函数，传递构造参数等。关于更多的详细信息请查看 5.2.1

节“加载”和 5.2.2 节“存储”。

6.3 非线性数据流

到这里为止所有的例子都是线性的数据流或者是树型结构的数据流。在线性数据流中，一个输入被加载，然后处理，最后存储。之前我们已经介绍了将多个数据流结合在一起的操作符：join、cogroup、union 和 cross。通过这些操作符可以建立一个树型结构将所有的输入都流向一个输出。但是在复杂的数据处理情形下，也经常需要对数据流进行划分。也就是说，一个输入将会产生多于一个的输出。也有可能是个菱形结构，数据流先被划分然后最终连接到一起。Pig 支持这种有向无环图（DAG）结构的数据流。

对数据流的划分可以是隐式的也可以是显式的。隐式划分脚本中不需要指定特殊的操作符或者语法，只需要简单的多次引用一个给定的关系就可以了。假设使用测试数据 *baseball*。例如，需要分析在同一时间一个团队的运动员所在的位置：

```
--multiquery.pig
players = load 'baseball' as (name:chararray, team:chararray,
            position:bag{t:(p:chararray)}, bat:map[]);
pwithba = foreach players generate name, team, position,
            bat#'batting_average' as batavg;
byteam = group pwithba by team;
avgbyteam = foreach byteam generate group, AVG(pwithba.batavg);
store avgbyteam into 'by_team';
flattenpos = foreach pwithba generate name, team,
            flatten(position) as position, batavg;
bypos = group flattenpos by position;
avgbypos = foreach bypos generate group, AVG(flattenpos.batavg);
store avgbypos into 'by_position';
```

关系 pwithba 通过 group 操作符被关系 byteam 和 bypos 引用。Pig 创建了一个从 pwithba 中读取数据的数据流并且将其传送到那两个 group 操作符。

还可以通过 split 操作符显式地对数据流进行划分，这可以允许用户根据需要以多种方式对数据流进行划分。我们举个例子，依据记录被创建时的日期将数据划分到不同的文件中：

```
wlogs = load 'weblogs' as (pageid, url, timestamp);
split wlogs into apr03 if timestamp < '20110404',
         apr02 if timestamp < '20110403' and timestamp > '20110401',
         apr01 if timestamp < '20110402' and timestamp > '20110331';
store apr03 into '20110403';
store apr02 into '20110402';
store apr01 into '20110401';
```

乍一看，split 看上去像是一个 switch 或者 case 语句，*但是它不是*。一条独立的记录可

能会被划分到多个分支中，因为可以通过 if 语句完成多个不同的过滤条件。同时一条记录也可以不划分到任何一个分支中去。在前面的那个例子中，如果一条记录对应的日期是 20110331，那么这条记录就会被丢弃。同时没有一个 default 缺省语句，也就是无法将任何不符合前面过滤条件的记录划分到一个特定的别名中。

split 在语义上和用户使用 filter 过滤器进行隐式划分是一致的。上面的例子也可以按如下方式重写：

```
wlogs = load 'weblogs' as (pageid, url, timestamp);
apr03 = filter wlogs by timestamp < '20110404';
apr02 = filter wlogs by timestamp < '20110403' and timestamp > '20110401';
apr01 = filter wlogs by timestamp < '20110402' and timestamp > '20110331';
store apr03 into '20110403';
store apr02 into '20110402';
store apr01 into '20110401';
```

事实上，Pig 会在内部将原先包含 split 语句的脚本重写成这种方式。

我们来看看 Pig 如何执行这些非线性的数据流。只要有可能，它都会把它们组装到单独的 MapReduce 任务中。这就是所谓的 *multiquery*。对于那种所有的操作符可以组装到一个单独的 map 任务中的情况，这是很简单的。Pig 在 map 中构建不同的数据管道之后再将合适的记录传送到每个数据管道中。那个按照日期将数据进行划分的例子也是采用这种方式执行的。

在很多种情况下，Pig 还可以将多个 group 操作符组装在一起。在本节开始的那个例子中，棒球数据是按照队伍和位置进行分组的，整个 Pig Latin 脚本将会在一个 MapReduce 任务中执行。Pig 是通过在 map 端对记录进行冗余并且对每条记录增加一个数据管道编号的方式来实现的。当在 shuffle 阶段数据被分区后，每条记录都会有一个对应的恰当的键。也就是说，按照队伍 team 进行分组的数据管道中的记录将使用队伍 team 作为 shuffle 的键，按照位置 position 进行分组的数据管道中的记录将使用位置 position 作为 shuffle 的键。这是通过声明键的类型为 tuple 并且为每条记录放置正确的值来实现的。一旦为 reducer 准备的数据收集完成，数据管道编号将作为排序的键的组成部分以保证来自每个数据管道以及组的记录能收集在一起。在 reduce 任务中，Pig 会实例化多个数据管道，每个 group 操作符都会有一个。它会根据记录的标记的数据管道编号将数据传送到合适的数据管道中。通过这种方式，只需扫描输入数据一次就可以通过多种方式进行分组操作。图 6-1 显示的例子就是数据在这样的数据管道中是怎么流动的。虽然这并没有提供线性加速的效果，但是我们看到它可以经常接近这个效果了。

有一些情况 Pig 不会将多个操作符组合成一个单独的 MapReduce 任务。对于如下操作符 Pig 不会使用 multiquery:join、union、cross 或者 cogroup。对于 order 语句也不会使用 multiquery。同样的，如果有多个 group 语句而且其中有一些使用了 Hadoop 的组合器也有一些没有使用，那么只有使用了 Hadoop 组合器的语句才会组合成一个

mulitquery。只是因为我们发现将调用了 Hadoop 组合器或没有调用 Hadoop 组合器的任务组合在一起执行效果并不好。

mulitquery 脚本通常比多次加载同一个输入更有效率，但是这种方式确实是有限制的。因为它需要在 map 阶段复制多份记录，所以会减慢 shuffle 阶段。最终对 shuffle 阶段递增的时间消耗可能会超过减少对输入数据的扫描节约下来的时间。Pig 没有办法预测什么时候这种情况会发生。当前，优化器是采用乐观的处理方式，只要能够使用 multiquery 组合任务就会使用这个策略。如果因为组合了太多的任务而导致性能下降，可以将 multiquery 优化关闭或者将 Pig Latin 脚本重写成多个脚本，这样 Pig 就不会尝试把它们组合在一起了。可以通过在命令行传入–M 或者-no_multiquery 参数关闭，或者将属性 opt.multiquery 设置为 false。

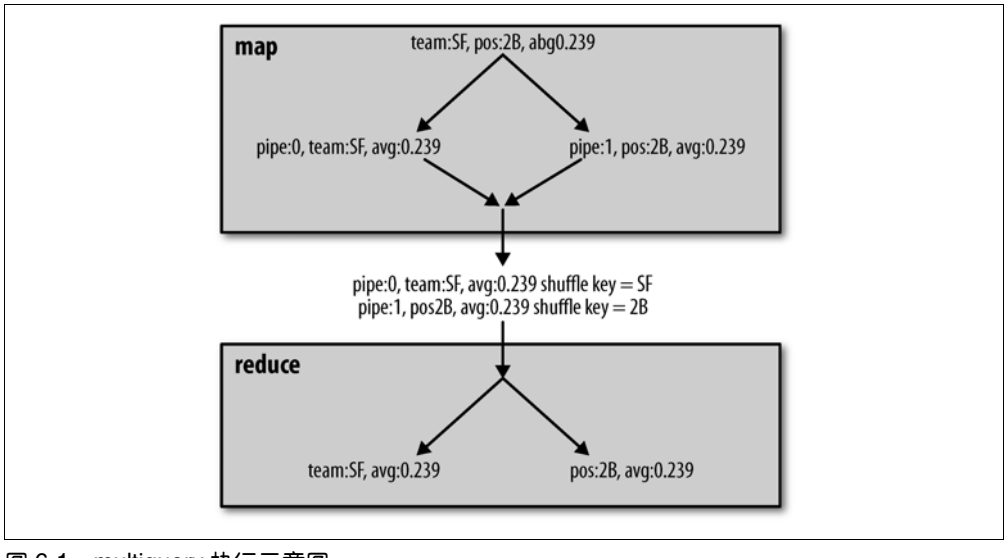


图 6-1　multiquery 执行示意图

我们还要考虑到当 multiquery 中有一个任务失败了但是其他任务都是成功的时候会怎么样。如果所有的任务都成功了，Pig 将会返回值 0 表示成功。如果所有的任务都失败了，Pig 将返回值 2。如果有一些任务成功了也有一些任务失败了，Pig 将返回值 3。默认情况下，如果有一个任务失败了，Pig 还会继续执行其他的任务。然而，如果期望 Pig 一旦发现有任务失败就停止的话，那么可以传入参数-F 或者–stop_on_failure 来控制。在这种情况下，所有没有结束的任务都会被终止，没有启动的任务都将不再启动，而已经完成的任务也不会被清理。

6.4　执行过程控制

Pig 除了提供了很多关系和数据流操作符以外，还提供了控制 MapReduce 任务如何执行

的方式。它允许用户设置参数控制环境变量以及 MapReduce 的一些配置，例如数据是怎样划分的。

6.4.1 set

set 命令用于设置 Pig 执行 MapReduce 任务的环境变量。表 6-1 描述了可以通过 set 命令设置与 Pig 相关的参数。

表 6-1 Pig 提供的 set 参数

参　数	值数据类型	描　述
debug	sring	设置日志级别是 DEBUG。与在命令行指定–debug DEBUG 效果是一样的
default_parallel	integer	为脚本中所有的 reduce 操作设置一个缺省的并发级别。详细信息请查看 5.3.9 节“Parallel”
job.name	string	为 Hadoop 任务分配一个名称。默认情况下这个名称和所执行的脚本的文件名是一样的，或者是一个为交互式进程随机生成的名称
job.priority	string	如果 Hadoop 集群是使用能力调度器对队列进行优先级控制，那么这个参数可以为 Pig 任务设置优先级。提供的值有 5 个：very_low、low、normal、high 和 very_high

例如，可以通过如下方式设置 Pig Latin 脚本的并发数和设置任务的名称为 my_job：

```
set default_parallel 10;
set job.name my_job;
users = load 'users';
```

除了这些预定义的值外，set 命令还可用于传递 Java 属性值给 Pig 和 Hadoop。Pig 和 Hadoop 都有一系列的 Java 属性用于控制它们。例如我们对于一个脚本想关闭 Java 属性，同时希望 Hadoop 在 map 端使用比常规较高的排序缓存，可以按如下设置：

```
set opt.muliquery false;
set io.sort.mb 2048; -- 设置为 2GB
```

同样也可以使用这种方式传递属性值给 UDF。所有的属性值都会在执行的时候传送给所有 Hadoop 节点上的所有任务中。它们被放置在一个叫做 JobConf 的 Hadoop 对象中。UDF 会访问这个 JobConf 对象。因此，通过 set 命令在脚本中设置的任何信息，UDF 都是可以看到的。这是一个控制 UDF 行为的方便的方式。关于如何在 UDF 中获取这些属性信息的更多内容，请参考 10.1.5 节“构造器和将数据从前端传送到后端”。

脚本中所设置的值对应脚本是全局可见的。如果在脚本中的后面重新设置了值，那么后面的值会覆盖前面的值并且整个脚本都会使用后面设置的值。

6.4.2 设置分割器

Hadoop 使用一个叫做 Partitioner 的类在 shuffle 阶段来对记录进行划分，然后传送给 reducer。关于分割器的详细信息，请查看附录 B 的“Shuffle 阶段”。Pig 没重写默认的分割器，除了 order 和 skew join。需要特殊的分割器来完成这些操作符间的平衡。

从 0.8 版本开始，Pig 允许用户自己设置分割器，除了那些已经重载了的情况。要达到这个目的，需要告诉 Pig 使用哪个 Java 类对数据进行划分。这个类必须要继承 Hadoop 的 org.apache.hadoop.mapreduce.Partitioner<KEY,VALUE>。需要注意的是，这个是 mapreduce 包下的新的 API（0.20 以及之后版本）而非之前旧的 mapred 包下的 API。

```
register acme.jar; -- 包含有分割器的 jar 包
users = load 'users' as (id,age,zip);
grp = group users by id partition by com.acme.userpartitioner parallel 100;
```

可以减少数据量的操作符可以接受 partition 语句。这些操作符有 cogroup、cross、distinct、group 和 join（同样不包括 skew join）。

6.5 Pig Latin 预处理器

在 Pig Latin 脚本被解析前会有一个预处理器。在 0.8 或更早版本中，提供了参数替换代入，粗略地和简化版的 C 语言中的#define 命令相同。从 0.9 版本开始，同样提供了其他 Pig Latin 脚本和类函数的宏定义，因此可以以模块化的方式写 Pig Latin 脚本。

6.5.1 参数传入

需要频繁调用的 Pig Latin 脚本通常含有基于它们执行的日期和位置需要进行改变的元素在里面。一个需要每天都执行的脚本很有可能在它的输入文件或者过滤器条件中包含一个日期部分。可以将这个日期作为一个参数传入，而不是每天都去修改这个脚本文件。参数传入通过简单的字符串替换实现了这个功能。参数必须以字母或下划线开头，然后可以跟上任意数量的字母、数字或下划线。参数可以通过命令行传入或者通过一个参数文件传入：

```
--daily.pig
daily = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
            date:chararray, open:float, high:float, low:float,
close:float,
            volume:int, adj_close:float);
yesterday = filter daily by date == '$DATE';
grpd = group yesterday all;
minmax = foreach grpd generate MAX(yesterday.high), MIN(yesterday.low);
```

在执行 *daily.pig* 脚本时，必须为参数 DATE 提供一个具体的值，否则将会报错并提示有没有定义的参数：

```
pig –p DATE=2009-12-17 daily.pig
```

可以按照需要指定任意多个-p 命令行。参数也可以放置在一个文件中，当有很多参数需要指定的时候这种方式会比较方便。文件内的格式是“*参数*=*值*”的形式，每行写一个。注释的内容必须以#号开头。然后可以通过-m 或者-param_file 命令指定这个参数文件：

```
pig –para_file daily.params daily.pig
```

通过命令行传入的参数要比通过文件传入的参数的优先级要高。也就是说，可以通过参数文件提供一些标准的通用的配置参数，然后根据需要通过命令行重新对一些参数赋值。

参数中可以包含其他的参数。因此，下面的参数文件内容是合法的：

```
#Param file
YEAR=2009-
MONTH=12-
DAY=17
DATE=$YEAR$MONTH$DAY
```

一个参数在被引用之前必须要先被定义。上面的参数文件中如果 DAY 那一行放置在 DATE 行之后，那么将会产生错误。另一个警告是没有特定的字符来划定一个参数的结束。任意的字母、数字或者下划线字符都将被认为是参数的一部分，然后其他字符被看作是字符本身。因此，如果有一个脚本需要每个月的月初执行，那么不可以按如下方式书写：

```
wlogs = load 'clicks/$YEAR$MONTH01' as (url, pageid, timestamp);
```

这将会把 MONTH01 作为一个参数而不是将 MONTH 作为一个参数。

当使用参数传入功能时，在预处理器结束之后所有的参数必须都要被具体的值替换。如果没有全替换，那么 Pig 将会抛出错误信息并且不会继续执行。在命令行可以通过-dryrun 标记来运行查看参数传入后的脚本。Pig 会打印出参数替换后的 Pig Latin 脚本，而不会真正执行这个脚本。

也可以在 Pig Latin 脚本内使用%declare 和%default 标记来定义参数。%declare 允许定义在脚本本身内的参数。%default 提供一个通常的默认的值，然后可以在需要的时候进行重写。考虑到大部分情况下是在 Hadoop 集群上执行脚本的，但是偶尔也会在另外一个具体不同硬件的集群上执行：

```
%default parallel_factor 10;
wlogs = load 'clicks' as (url, pageid, timestamp);
grp = group wlogs by pageid parallel $parallel_factor;
cntd = foreach grp generate group, COUNT(wlogs);
```

当使用常规的配置来执行脚本时，不需要设置参数 parallel_factor。当在一个不同的设

置的场景下，可以在命令行传入值来改变这个并发数参数。

6.5.2 宏

从 0.9 版本开始，Pig 增加了定义宏命令的功能。这使得创建用户自定义的 Pig Latin 脚本模板成为可能，同时也使得在用户间分享 Pig Latin 代码片段成为可能。这可以特别用于制定标准惯例并确保所有的数据生产者和消费者都使用这一标准。

宏是通过 define 语句定义的。一个宏会接受一组输入参数，即字符串类型的值，当宏被展开的时候会进行参数传入。按照惯例，输入的关系名称要放置在其他参数之前。输出的关系名称会在 returns 语句中给出。宏的参数都会放在{}（花括号）中。无论何处的参数，包括输出的关系名称，都是在宏的内部被引用的，它们必须以$（美元符）开头。在 Pig Latin 脚本中通过将宏指派给一个关系的方式来使用宏：

```
   -- macro.pig
   -- 给定每天的输入和指定的年，分析
   -- 在支持股息的那些天里股票价格是如何变化的
   define dividend_analysis (daily, year, daily_symbol, daily_open,
daily_close)
   returns analyzed {
       divs = load 'NYSE_dividends' as (exchange:chararray,
                 symbol:chararray, date:chararray, dividends:float);
       divsthisyear = filter divs by date matches '$year-.*';
       dailythisyear = filter $daily by date matches '$year-.*';
       jnd = join divsthisyear by symbol, dailythisyear by $daily_symbol;
       $analyzed = foreach jnd generate dailythisyear::$daily_symbol,
                       $daily_close - $daily_open;
   };

   daily = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
            date:chararray, open:float, high:float, low:float, close:float,
            volume:int, adj_close:float);
   results = dividend_analysis(daily, '2009', 'symbol', 'open', 'close');
```

同时允许宏返回的结果不是一个关系。在这种情况下，define 语句的返回部分改成了返回 void。用户需要定义一个宏来控制数据在传递给特定的输出前如何进行划分和排序，例如传递给 HBase 或者是一个数据库。

这些宏是以内联的方式展开的，这是宏和函数一个明显的不同。宏不可以递归地调用。宏可以调用其他的宏，因此宏 A 可以调用宏 B，但是宏 A 不可以调用它本身。同时一旦宏 A 调用了宏 B，那么宏 B 就可以调用宏 A。Pig 会检测出这些循环并会抛出错误信息。

不可以在宏内部使用参数传入（请查看 6.5.1 节“参数传入”）。参数需要显式地传送给宏，并且只能在最上层使用参数传入。

可以通过-dryrun 命令行参数来查看宏在内部是如何展开的。当宏被展开后，别名将会

被更改以避免和宏被展开的位置的别名形成冲突。如果我们还以前面的例子为例并且使用-dryrun 命令来查看 Pig Latin 脚本的结果，我们将会看到如下内容（为了适应页面做了适当的格式变化）：

```
   daily = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
            date:chararray, open:float, high:float, low:float, close:float,
            volume:int, adj_close:float);
   macro_dividend_analysis_divs_0 = load 'NYSE_dividends' as
(exchange:chararray,
            symbol:chararray, date:chararray, dividends:float);
   macro_dividend_analysis_divsthisyear_0 =
            filter macro_dividend_analysis_divs_0 BY (date matches
'2009-.*');

   macro_dividend_analysis_dailythisyear_0 = filter daily BY (date matches
'2009-.*');
   macro_dividend_analysis_jnd_0 =
            join macro_dividend_analysis_divsthisyear_0 by (symbol),
            macro_dividend_analysis_dailythisyear_0 by (symbol);
   results = foreach macro_dividend_analysis_jnd_0 generate
            macro_dividend_analysis_dailythisyear_0::symbol, close - open;
```

正如用户所看到的，宏内的别名展开后综合了宏的名称和调用的编号。这种方式可以提供一个唯一键从而避免其他宏使用了相同的别名，或者同一个宏被多次使用也可以保证没有重复。

6.5.3 包含其他的 Pig Latin 脚本

Pig Latin 在很长一段时间，其整个脚本需要在一个文件中保存。这样会产生令人不愉快的上千行的 Pig Latin 脚本。从 0.9 版本开始，预处理器可以用于将某个 Pig Latin 脚本加入另一个脚本中。与宏合起来(同样也是在 0.9 版本中加入的,请参考 6.5.2 节“宏”)，现在可以写出模板 Pig Latin 以方便调试和重用。

import 命令用于将某个 Pig Latin 脚本加入另一个脚本中。

```
   --main.pig
   import '../examples/ch6/dividend_analysis.pig';

   daily = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
              date:chararray, open:float, high:float, low:float,
close:float,
              volume:int, adj_close:float);
   results = dividend_analysis(daily, '2009', 'symbol', 'open', 'close');
```

import 命令会将对应的文件执行写入 import 命令所在的位置。在前面提及的例子中，脚本文件 *dividend_ananlysis.pig* 的内容会立即放置在 load 语句之前。需要注意的是，同一个文件不可以导入两次。如果希望多次使用同一功能，那么应该将其写成一个宏

然后导入包含这个宏的脚本文件。

在刚才所展示的例子中，使用了相对路径来指定了需要加入的脚本文件，也可以使用完整的绝对路径。默认情况下，运行脚本时会以 Pig 当前的工作目录为基准来获取相对路径。可以通过设置属性 pig.import.search.path 来指定搜索路径。这是一组以逗号分割的路径，用于检索文件。当前工作目录.（点号）总是在搜索路径之中的：

```
set pig.import.search.path '/usr/local/pig,/grid/pig';
import 'acme/macros.pig';
```

导入的文件不是在不同的命名空间中，这意味着所有的宏都在一个命名空间中，即使它们是通过不同的文件导入的。然而，注意应该为宏选择一个唯一的名称。

第 7 章

开发和测试 Pig Latin 脚本

最后几章会重点介绍 Pig Latin 语言。现在我们将转向开发和测试 Pig Latin 脚本的实用事项上。本章将介绍一些实用的调试工具，如 **describe** 和 **explain**，同时还将介绍几种测试脚本的方式。下一章将介绍如何提高脚本的执行效率。

7.1 开发工具

Pig 提供了几个工具和诊断操作符用于协助开发程序。在本节中我们将逐一展开介绍，同时也会介绍其他人开发的利用编辑器和集成开发环境（IDE）提高开发效率的工具。

7.1.1 语法高亮和语法检查

语法高亮通常可以帮助用户书写正确的代码，至少在语法结构上可以保证是正确的。几个常用的编辑器都提供了语法高亮包。表 7-1 中列举了这些包，这些链接都是在不同的时期添加的，因此它们是否能很好地进行高亮显示还要看当前的语法差异。

表 7-1　　Pig Latin 语法高亮相关包

工　具	URL
Eclipse	*http://code.google.com/p/pig-eclipse*
Emacs	*http://github.com/cloudera/piglatin-mode*, *http://sf.net/projects/pig-mode*
TextMate	*http://www.github.com/kevinweil/pig.tmbundle*
Vim	*http://www.vim.org/scripts/script.php?script_id=2186*

除了这些语法高亮包之外，Pig 本身也允许用户进行语法检查而不会实际执行。如果用户在命令行增加**-c** 或者**–check** 选项，Pig 就只会对脚本进行解析和执行语法检查。命令行选项**-dryrun** 也会检查语法，然后展开宏命令和 **import** 部分，以及进行参数替换。

7.1.2 describe

describe 命令会显示脚本中指定关系的模式。当用户在开发脚本时这个命令非常有帮助。而当用户正在学习 Pig Lation 时想了解不同的操作符是怎样改变数据的时候，这个命令就显得更加有作用了。**describe** 可以用于脚本中任何一个关系，一个脚本中也可以有多个 **describe** 语句：

```
--describe.pig
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,
          date:chararray, dividends:float);
trimmed = foreach divs generate symbol, dividends;
grpd = group trimmed by symbol;
avgdiv = foreach grpd generate group, AVG(trimmed.dividends);

describe trimmed;
describe grpd;
describe avgdiv;
```

```
trimmed: {symbol: chararray,dividends: float}
grpd: {group: chararray,trimmed: {(symbol: chararray,dividends: float)}}
avgdiv: {group: chararray,double}
```

describe 命令使用的是 Pig 的标准模式语法。关于这个语法的详细信息，请查看 4.2 节“模式”。在这个例子中，关系 **trimmed** 包含两个字段：字段 **symbol，**它是 chararray 类型的；字段 **dividends，**是 float 类型的。关系 **grpd** 也有两个字段，字段 **group**（Pig 经常将 group 操作结果的键命名为 group）和 bag 数据类型的字段 **trimmed，**这和 Pig 中进行分组然后产生 bag 数据类型结果的关系的命名是一致的。**trimmed** 包含的 tuple 有两个字段：**symbol** 和 **dividends。**最终，关系 **avgdiv** 有两个字段，**group** 和一个 double 值，这个值就是函数 **AVG** 的计算结果而且没有别名。

7.1.3 explain

Pig 的目标之一就是允许用户以数据流的方式考虑问题而不是以 MapReduce 的方式。但是有时用户需要深入到内部看 Pig 是如何将用户的脚本编译成 MapReduce 任务的。Pig 为此提供了 **explain** 命令。当用户试图优化脚本或者调试错误的时候，**explain** 命令就显得特别有帮助。它的存在主要是提供 Pig 开发者用于检测 Pig 是如何处理不同脚本的，因此它的输出对用户而言并非是非常友好的。但是通过一些努力，**explain** 命令可以帮助用户写出更好的 Pig Latin 脚本。

有两种方式来使用 **explain。**用户可以在 **explain** 后面跟上用户的 Pig Lation 中任何一个别名，这会显示如果用户存储这个关系那么 Pig 将会使用的执行计划。用户也可以在 Grunt 交互界面中对一个存在的 Pig Lation 脚本文件执行 **explain**。这样做有两个好处。其一，用户不需要修改他的脚本去增加 **explain** 这行代码。其二，这样处理的是脚本因此不需要指定一个特定的存储。下面的例子显示的就是 Pig 是如何执行整个脚本的：

```
--explain.pig
divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
grpd = group divs by symbol;
avgdiv = foreach grpd generate group, AVG(divs.dividends);
store avgdiv into 'average_dividend';

bin/pig -x local -e 'explain -script explain.pig'
```

这将会产生一个文本格式的图像结构输出，我们可以随时来检查输出信息。当在 Grunt 中使用 **explain** 的时候，用户也是可以将执行计划以图形化的方式展示出来的。这时用户需要在之前的命令行中增加**-dot –out *filename*** 命令。这将会打印出一个文件，内容是以 DOT 语言（一种文本图形描述语言）展现的用户的脚本将会是如何执行的。可以读取这种语言的工具可以读取该文件并产生可视化的图形结构。对于一些工具，用户可能需要将文件中的 3 个图形分割为 3 个独立的文件。

Pig 需要经历几个步骤才可以将一个 Pig Latin 脚本转换成一组 MapReduce 任务。在进行基本的解析和语义检查之后，它将会产生一个逻辑计划。这个计划描述了 Pig 在执行这个脚本时将要使用的逻辑操作符。一些优化会在这个计划中完成。例如，逻辑计划中过滤器会被尽可能地往前推[1]。前面的那个例子的逻辑计划如图 7-1 所示。为了使输出更具有可读性我将一些没有关联的片段省略了。如果用户使用的是 0.9 版本的 Pig，输出结果会有些许不同，但还是足够可以辨认的。

```
avgdiv: Store 1-27 Schema: {group: bytearray,double} Type: Unknown
|
|---avgdiv: ForEach 1-26 Schema: {group: bytearray,double} Type: bag
    |   |
    |   Project 1-22 Projections: [0]
    |   FieldSchema: group: bytearray Type: bytearray
    |   Input: grpd: CoGroup 1-16
    |   |
    |   UserFunc 1-25 function: org.apache.pig.builtin.AVG
    |   FieldSchema: double Type: double
    |   |
    |   |---Project 1-23 Projections: [3]
    |       FieldSchema: dividends: bag({dividends: bytearray}) Type: bag
    |       Input: Project 1-24 Projections: [1] Overloaded: false|
    |       |---Project 1-24 Projections: [1]
    |           FieldSchema: divs: bag({exchange: bytearray,symbol: bytearray,
    |               date: bytearray,dividends: bytearray}) Type: bag
    |           Input: grpd: CoGroup 1-16
    |
    |---grpd: CoGroup 1-16 Schema: {group: bytearray,divs: {exchange: bytearray,
        |   |   symbol: bytearray,date: bytearray,dividends: bytearray}} Type: bag
        |   |
        |   Project 1-15 Projections: [1]
        |   FieldSchema: symbol: bytearray Type: bytearray
        |   Input: divs: Load 1-14
        |
        |---divs: Load 1-14 Schema: {exchange: bytearray,symbol: bytearray,
            date: bytearray,dividends: bytearray} Type: bag
```

图 7-1　逻辑计划

[1] 或者是后推，根据用户的需要。数据库教科书经常讲后推，靠近扫描。因为 Pig Latin 脚本是在最顶端进行加载然后往下进行，所以我们更倾向于将过滤器前推到靠近加载器。

这个图表的流向是从底部到顶部的，因此 **Load** 操作符在最底部。操作符间的连接线表示流向。从图表中可以看到脚本产生的 4 个操作符（**Load**、**CoGroup**、**ForEach** 和 **Store**）。每一个操作符都有一个以标准模式语法描述的模式。**CoGroup** 和 **ForEach** 操作符还具有附属于它们的表达式（操作符下垂的线对应的部分）。在 **CoGroup** 操作符中，projection 字段表明根据哪个字段进行分组（在这里按字段 1 进行分组）。**ForEach** 操作符有一个 projection 表达式，其显示了字段 0（分组字段）和一个 UDF 表达式，还可以看出使用的 UDF 是 **org.apache.pig.builtin.AVG。**注意每一个 **Project** 表达式都有一个 **Input** 字段用于显示是从哪个操作符的结果作为它的输入的。图 7-2 展示的是当使用 **-dot** 选项时这个计划是什么样子的。

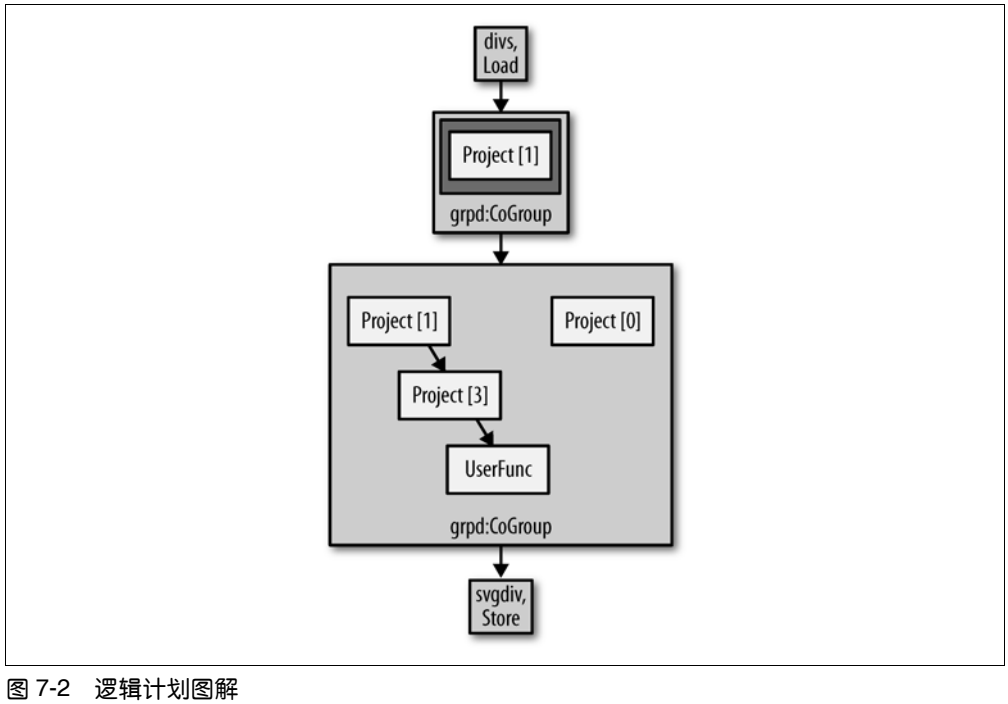


图 7-2　逻辑计划图解

对逻辑计划进行优化后，Pig 将会产生一个实体计划。实体计划描述了 Pig 在执行脚本时使用到的实体操作符，而不管它们在 MapReduce 中是如何执行的。图 7-1 所示的计划的实体计划如图 7-3 所示。

这看上去和逻辑计划有些相似，但是有一些区别。将要使用的加载和存储函数已经确定了（本例中是 **org.apache.pig.buildin.PigStorage**，这是 Pig 中默认的加载和存储函数），同时将要使用的实际路径也确定了。这个例子是以本地模式运行的，因此这里的路径指向的是本地文件。如果在集群上执行，那么路径将是形如 *hdfs:nn.machine.domain/filepath* 的。

```
avgdiv: Store(file:///home/gates/git/programmingpig/data/average_dividend:
|       org.apache.pig.builtin.PigStorage)
|
|---avgdiv: New For Each(false,false)[bag]
    |   |
    |   Project[bytearray][0]
    |   |
    |   POUserFunc(org.apache.pig.builtin.AVG)[double]
    |   |
    |   |---Project[bag][3]
    |       |
    |       |---Project[bag][1]
    |
    |---grpd: Package[tuple]{bytearray}
        |
        |---grpd: Global Rearrange[tuple]
            |
            |---grpd: Local Rearrange[tuple]{bytearray}(false)
                |   |
                |   Project[bytearray][1]
                |
                |---divs: Load(file:///home/gates/git/programmingpig/data/
                    NYSE_dividends:org.apache.pig.builtin.PigStorage)
```

图 7-3　实体计划

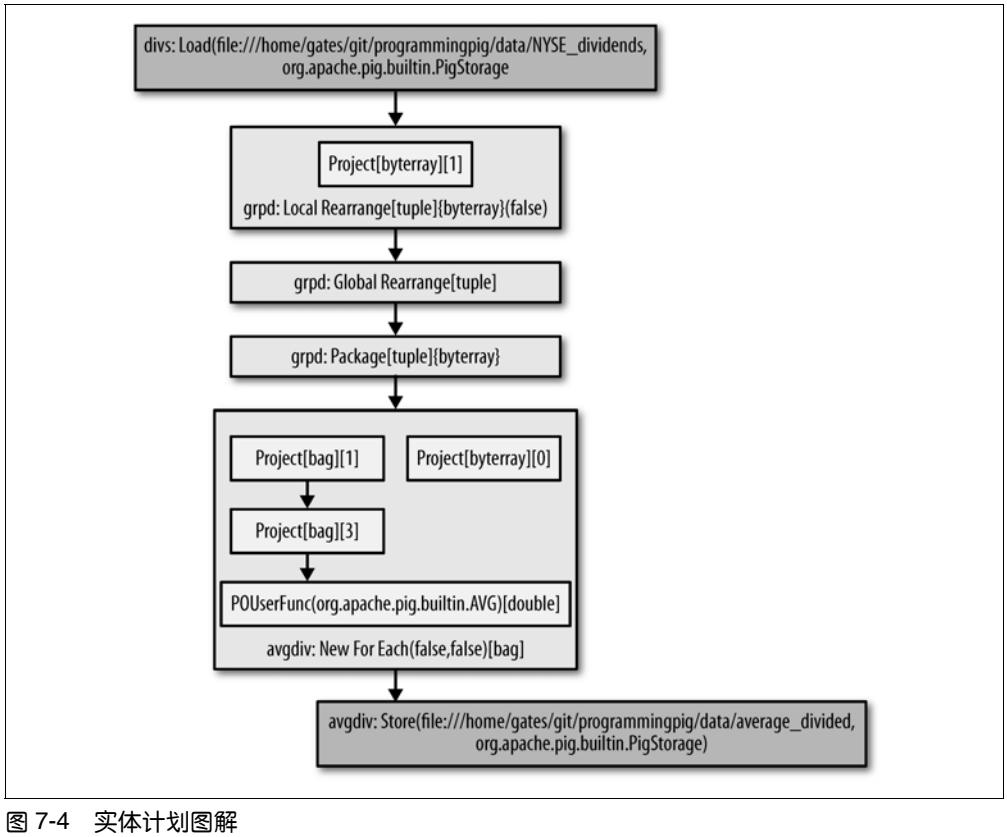


图 7-4　实体计划图解

另一个需要注意的区别是 **CoGroup** 被 3 个操作符替代了:**Local Rearrange、Global Rearrange 和 Package。Local Rearrange** 操作符是 Pig 用于通过组装键为 shuffle 阶段准备数据所使用的操作符。**Global Rearrange** 是 shuffle 任务的一个替代。**Package** 位于 reduce 阶段并且保证记录进入正确的 bag 数据包中。图 7-4 是该计划的一个图形化展示。

最后，Pig 根据实体计划决定如何将它的操作符组装到一个或多个 MapReduce 任务中。首先，它会遍历整个实体计划寻找需要一个新的 reduce 过程的所有的操作符。当有一个 **Local Rearrange、Global Rearrange 和 Package** 时这都会发生。在 Pig 完成这个过程后，它会看是否有地方可以做实体优化。例如，它会看是否有地方可以使用组合器，是否已经可以将排序过程合并到 Hadoop 在 shuffle 阶段进行的排序过程。当所有的这些都完成后，Pig 就有了一个 MapReduce 计划。该计划描述了 map 过程、组合过程和 reduce 过程，也包括了 Pig 将在每个阶段执行的实体操作符。作为实例的完结，图 7-5 展示了 MapRedcue 计划。

```
MapReduce node
Map Plan
grpd: Local Rearrange[tuple]{bytearray}(false)
|   |
|   Project[bytearray][0]
|
|---avgdiv: New For Each(false,false)[bag]
    |   |
    |   Project[bytearray][0]
    |   |
    |   POUserFunc(org.apache.pig.builtin.AVG$Initial)[tuple]
    |   |
    |   |---Project[bag][3]
    |       |
    |       |---Project[bag][1]
    |
    |---Pre Combiner Local Rearrange[tuple]{Unknown}
        |
        |---divs: Load(file:///home/gates/git/programmingpig/data/
                      NYSE_dividends:org.apache.pig.builtin.PigStorage)


Combine Plan
grpd: Local Rearrange[tuple]{bytearray}(false)
|   |
|   Project[bytearray][0]
|
|---avgdiv: New For Each(false,false)[bag]
    |   |
    |   Project[bytearray][0]
    |   |
    |   POUserFunc(org.apache.pig.builtin.AVG$Intermediate)[tuple]
    |   |
    |   |---Project[bag][1]
    |
    |---POCombinerPackage[tuple]{bytearray}


Reduce Plan
avgdiv: Store(file:///home/gates/git/programmingpig/data/average_dividend:
|         org.apache.pig.builtin.PigStorage)
|
|---avgdiv: New For Each(false,false)[bag]
    |   |
    |   Project[bytearray][0]
    |   |
    |   POUserFunc(org.apache.pig.builtin.AVG$Final)[double]
    |   |
    |   |---Project[bag][1]
    |
    |---POCombinerPackage[tuple]{bytearray}
Global sort: false
```

图 7-5　MapReduce 执行计划

这个和实体计划很相似。数据流在这里分解成 3 个执行过程：map 过程、组合过程和 reduce 过程。这里没有 **Global Rearrange** 操作符了，因为它是 shuffle 阶段的一个替代。**AVG** UDF 被分解成 3 个过程：map 阶段中的 **Initial**、组合器阶段的 **Intermediate** 和 reduce 阶段的 **Final**。如果这个例子有多个 MapReduce 任务，那么它们都会在这个输出中展现出来。MapRedcue 执行计划图解如图 7-6 所示。

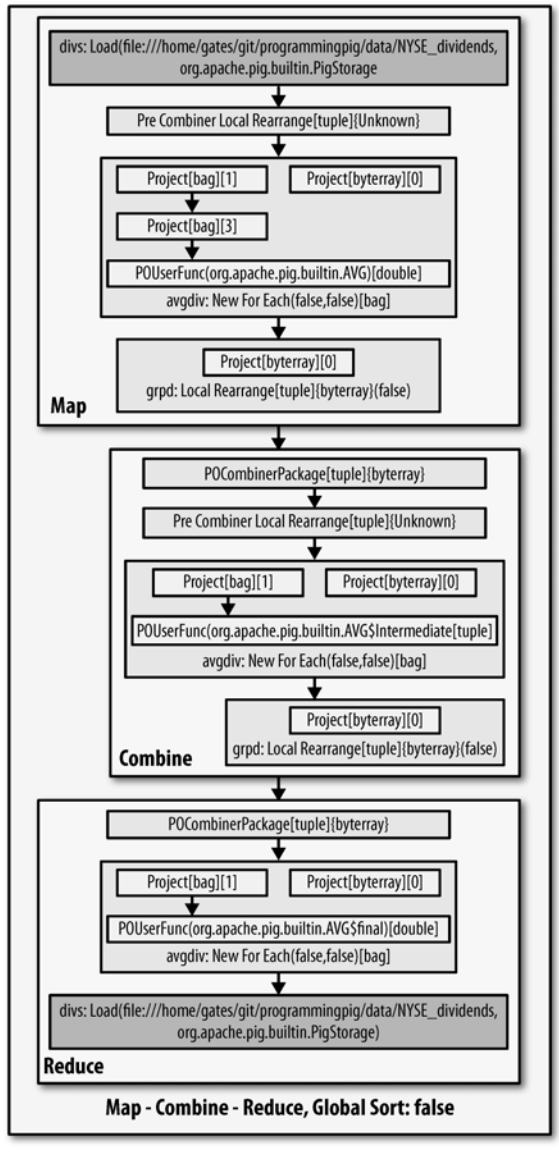


图 7-6　MapReduce 执行计划图解

7.1.4 illustrate

调试 Pig Latin 脚本的最好的方式之一就是使用它进行数据处理。但是如果用户正在使用 Pig，那么很有可能会有一个非常大的数据集。如果需要几个小时的时间处理数据，那么这将会导致一个非常长的调试周期。一个显然的解决办法是在一个样本数据上执行用户的脚本。对于简单的脚本，这样做很有效。但是抽样有另一个问题：并不能总是保证样本能很好地测试用户的脚本。例如，如果用户有一个 join 连接操作，那么他必须保证两边输入的样本数据至少有一些相同的键值。否则，用户的连接操作将不会产生结果。

为了解决这个问题，Yahoo!Research 的科学家们在 Pig 中集成了 **illustrate** 工具。**illustrate** 会对用户的数据进行抽样并且对抽样的数据执行用户的脚本，但是当它遇到一些移除数据的操作符（例如 **filter** 等）时，它会保证有一些数据可以通过这个操作符而有一些不可以。在必要的时候，它会生成一些记录而看上去就像是用户的（例如会具有相同的模式）但是并非是样本中的数据。例如，如果用户的脚本中有一行是 **B= filter A by x > 100;**而且每一条 **illustrate** 抽样产生的记录中 x 的值都是小于 100 的，对于这些记录，**illustrate** 将会至少保留一条（为了展示过滤器移除一条记录的过程），同时它将会产生一条 x 的值大于 100 的记录。

与使用 **describe** 一样，如果想使用 **illustrate**，只需要加在相应的别名之前即可。图 7-7 展示了对如下脚本进行图解的结果：

```
   --illustrate.pig
   divs = load 'NYSE_dividends' as (e:chararray, s:chararray, d:chararray,
div:float);
   recent = filter divs by d > '2009-01-01';
   trimmd = foreach recent generate s, div;
   grpd = group trimmd by s;
   avgdiv = foreach grpd generate group, AVG(trimmd.div);
   illustrate avgdiv;
```

对于这里的每一个关系，**illustrate** 为我们展示了像是从这个关系中输出的记录一样。对于 **recent = filter divs by d > '2009-01-01'**这一行，我特意选择了一个不会过滤任何数据的过滤器，来展示 **illustrate** 是如何产生一条将要被移除的记录的。在这里，也就是 **divs** 输出中所展示的最后一条记录。

和 **explain** 类似，**illustrate** 也可以作为一个命令行选项来执行而不需要修改用户的脚本文件。例如，**bin/pig –e 'illustrate –script illustrate.pig'。**

illustrate 是在 Pig 0.2 版本中加入的，但是有段时间并没有对它进行很好的维护。在 0.9 版本中，对它进行了重构。在 0.7 和 0.8 版本中，它对一些 Pig 操作符是有作用的，但并非支持所有的操作符。

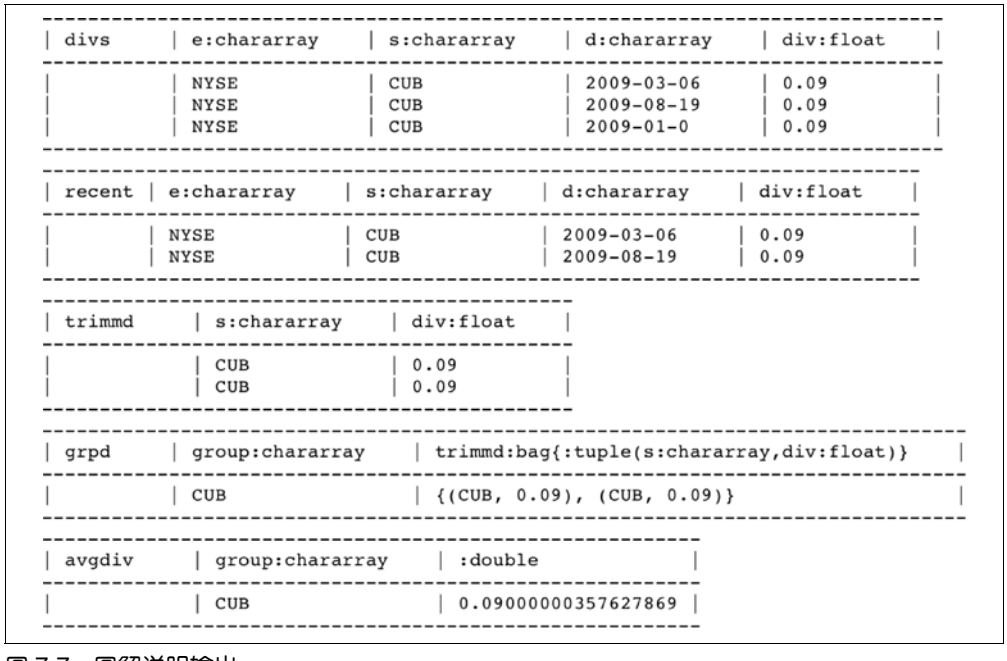


图 7-7　图解说明输出

7.1.5　Pig 统计信息

从 0.8 版本开始，在每次执行结束后 Pig 都会产生一组统计信息的总结：

```
   --stats.pig
   a = load '/user/pig/tests/data/singlefile/studenttab20m' as (name, age,
gpa);
   b = load '/user/pig/tests/data/singlefile/votertab10k'
         as (name, age, registration, contributions);
   c = filter a by age < '50';
   d = filter b by age < '50';
   e = cogroup c by (name, age), d by (name, age) parallel 20;
   f = foreach e generate flatten(c), flatten(d);
   g = group f by registration parallel 20;
   h = foreach g generate group, SUM(f.d::contributions);
   i = order h by $1, $0 parallel 20;
   store i into 'student_voter_info';
```

执行 *stats.pig* 将会产生图 7-8 所示的统计信息，为了能和页面相适应我对这些内容做了些格式调整。

前面几行给出了这个任务的一个简要的总结。**StartedAt** 是 Pig 提交这个任务的开始时间，而不是第一个任务开始在 Hadoop 集群上开始执行的时间。取决于用户的集群的繁忙程度，这个值可能有很大差异。同样的，**FinishedAt** 是 Pig 完成处理这个任务的结束

时间，这个时间会比最后一个 MapReduce 任务结束的时间稍微晚些。

```
HadoopVersion PigVersion UserId  StartedAt           FinishedAt          Features
0.20.2        0.8.1      gates   2011-05-04 18:20:34 2011-05-04 18:23:22 COGROUP,
                                                                         GROUP_BY,
                                                                         ORDER_BY,
                                                                         FILTER

Success!

Job Stats (time in seconds):
JobId  Maps Reduces Max  Min  Avg  Max    Min    Avg    Alias     Feature   Outputs
                    Map  Map  Map  Reduce Reduce Reduce
                    Time Time Time Time   Time   Time
3451   8    20      27   3    22   42     33      35    a,b,c,d,e COGROUP
3452   4    20      15   12   14   12     12      12    g,h       GROUP_BY,
                                                                  COMBINER
3453   1    1       3    3    3    12     12      12    i         SAMPLER
3454   1    20      3    3    3    12     12      12    i         ORDER_BY  bla

Input(s):
Successfully read 20000000 records from: "studenttab20m"
Successfully read 10000 records from: "votertab10k"

Output(s):
Successfully stored 6 records (186 bytes) in: "hdfs://nn.acme.com/user/gates/bla"

Counters:
Total records written : 6
Total bytes written : 186
Spillable Memory Manager spill count : 0
Total bags proactively spilled: 0
Total records proactively spilled: 0

Job DAG:
3451   ->      3452,
3452   ->      3453,
3453   ->      3454,
3454
```

图 7-8　脚本 stats.pig 的统计信息输出

以 **Job Stats** 标记的第 2 节提供了一个对每个执行的 MapReduce 任务的分解统计信息。统计信息包含每个任务具有的 map 个数和 reduce 个数，以及这些任务消耗了多少时间，以及用户的 Pig Latin 脚本和这些任务对应的别名信息。最后那个显示对应别名的功能是非常重要的，特别是当用户想知道他的脚本的哪个操作符在哪个 MapReduce 任务中执行，这对于判断为什么一个特定的任务失败了或者产生了非预期的结果是有帮助的。

Input、**Output** 和 **Counters** 这几部分的含义是不言而喻的。这是关于为了避免耗尽内存资源 Pig 对数据进行分割的次数等相关的统计信息。在本地模式中没有 **Counters** 这部分，因为 Hadoop 在本地模式中不会报告这个计数器信息。

最后的 **Job DAG** 部分描述了 MapReduce 任务之间数据是如何流向的。本例中，数据流是线性的。

7.1.6　MapReduce 任务运行状态信息

当用户在 Hadoop 集群上执行 Pig Latin 脚本时，找到任务的状态信息和日志信息是很有

挑战的。Pig 在计划和控制查询时产生的日志是存储在本地工作目录下的。当然用户也可以在命令行通过参数**–l logdir** 指定一个特定的目录来存放这些日志。然而，Hadoop 并没有提供一个获取它的任务的日志信息的方法。因此，Pig 所产生的日志文件只包含在本地机器上产生的日志。而执行过程中产生的日志文件，包括 UDF 所产生的日志文件，还是存放在 Hadoop 集群的任务节点上。所有由 map 和 reduce 任务所产生的 **stdout**（标准输出）和 **stderr**（标准错误输出）也是存放在任务节点上的。

定位到日志的第一步就是访问 JobTracker 的控制网页。该页面会给出当前在 Hadoop 集群中运行的任务的执行状态，同时也包括最近上百个左右完成的任务列表。通常来说，网址是 http://jt.acme.com:50030/jobtracker.jsp，其中 jt.acme.com 是用户在配置文件中配置的 JobTracker 的域名。图 7-9 显示的就是一个在 Linux 桌面环境下的搭建的伪分布式集群上的 JobTracker 管理网页。

从这张屏幕截图上我们可以看到，在这个伪分布式集群上最近只运行过一个任务，在页面中用户可以看到是谁提交了这个任务，任务的标号是多少以及任务名称。通过 Pig 提交的 Map/Reduce 任务，默认情况下用户名称就是运行 Pig Lation 脚本的那个用户的名称，当然允许我们在控制台通过参数指定使用其他一个用户名。由一个 Pig Lation 脚本所产生的多个 Map/Reduce 任务使用相同的用户名，而通常情况下一个 Pig 任务往往会产生多个 Map/Reduce 任务。在前面 7.1.5 节“Pig 统计信息”中我们讨论过在 Pig 任务结束的时候，会有一个统计信息输出，告诉我们在执行过程中有哪几个 Map/Reduce 任务，每个 M/R 任务使用了哪些别名和操作符。同一个 Pig 任务下的多个 M/R 任务具有相同的用户名，这非常有利于我们去定位 M/R 任务。图 7-9 所示的屏幕截图中的那个 Pig 任务，相关的统计信息一般会如下所示：

```
Job Stats (time in seconds):
JobId                ... Alias              Feature
job_201104081526_0019   daily,grpd,uniqcnt   GROUP_BY,COMBINER
```

通过提示信息中提供的这个任务 ID，我们就可以在 JobTracker 管理页面找到相应的 M/R 任务了。

需要注意的是，当这些 M/R 任务真正在集群上运行的时候，这些任务才会显示在 JobTracker 页面上。在提交到 Hadoop 集群上执行之前，Pig 需要消耗几秒钟的时间来解析 Pig 脚本，然后生成执行计划。因此一般是 Pig 提交第一任务几秒钟后才会在 Hadoop 集群上真正运行起来，也就是这个时候才会看到 M/R 任务。当然，也有可能 Hadoop 集群无法及时分配足够的资源给 Pig 任务，这种情况下，需要等待一定的时间待获得了足够的执行资源后，才能在 Hadoop 集群上触发 M/R 任务运行起来。

单击任务 ID 后会打开一个新的网页，如图 7-10 所示，这个网页会显示当前该任务的执行信息，包括任务是什么时候开始的，什么时候结束的，使用了多少个 map 资源，使用了多少个 reduce 资源，以及各种指标的结果。

如果想查看某一个特定的 map 任务的日志信息，那该怎么操作？在页面的上方有一张表格，其中统计了 map 和 reduce 各自的任务个数，单击最左边列中的“Maps”会产生作为这个 Job 一个组成部分的所有 map 任务的列表。选择任意的一个特定的 map 任务，然后点进去就可以看到这个任务是在哪些机器上执行的，它的执行状态，它的开始时间和终止时间，同时还会提供一个链接去查看它产生的日志文件。点击那个链接会（最终）让我们可以查看这个特定任务的日志。

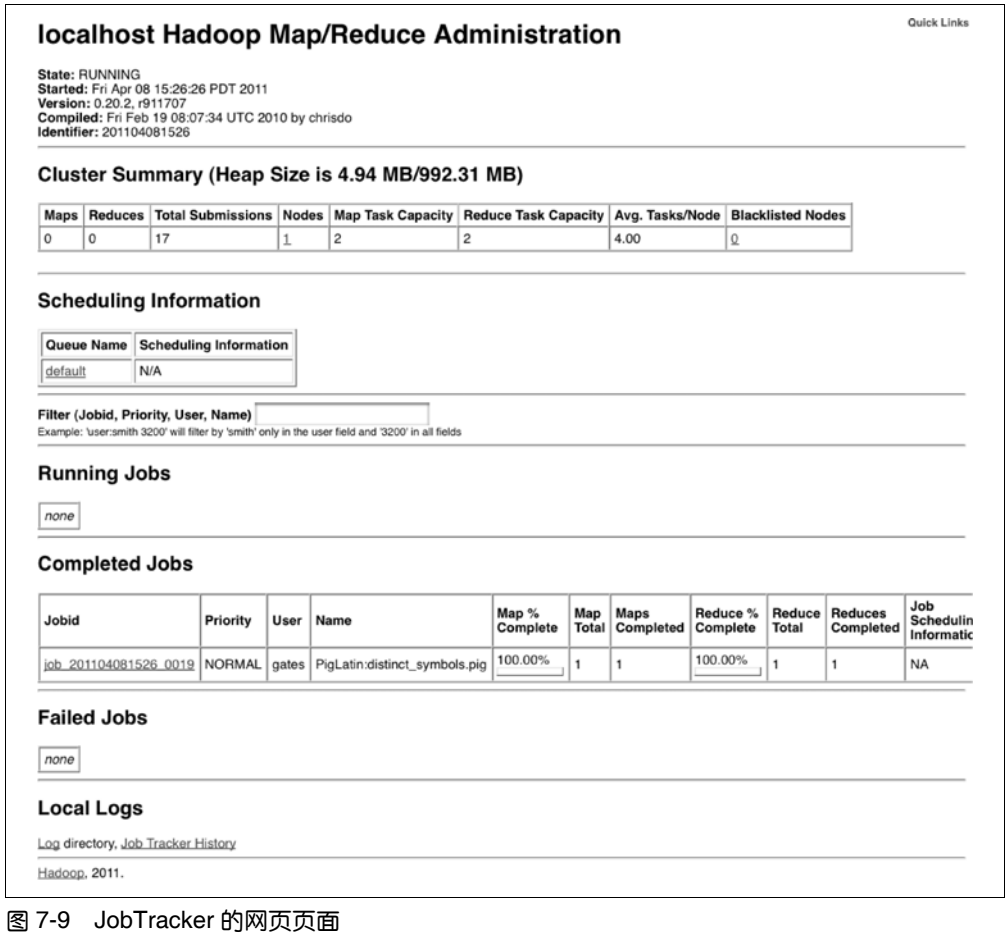
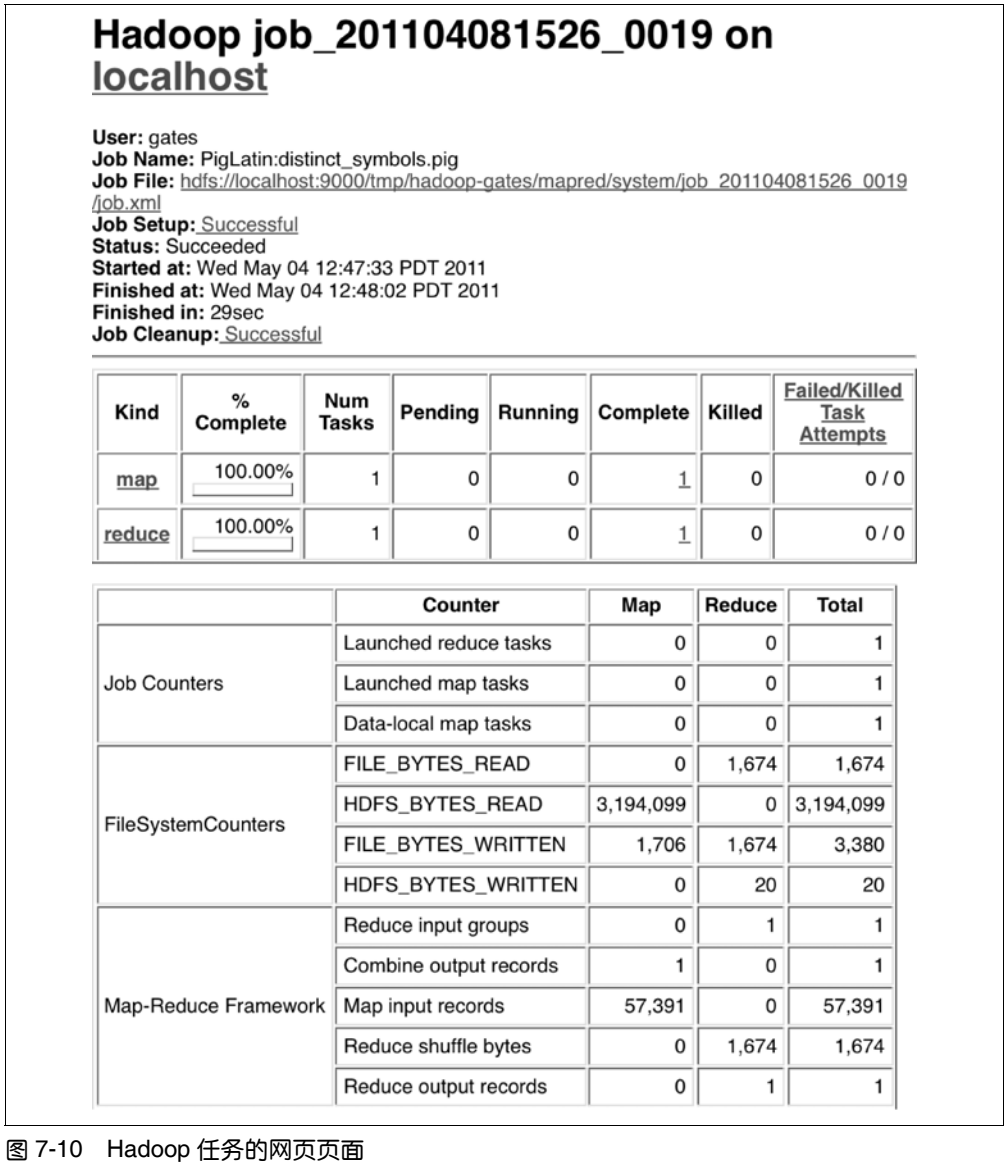


图 7-9　JobTracker 的网页页面

当然，在这个例子中，因为只有一个 map 任务所以我们就很容易找到我们需要看的 map 任务。但是如果提交的 Job 是包含了 10,000 个 map 任务呢？那么怎么判断需要去看哪个任务的日志呢？这个问题很好但通常很难有一个简明的答案。如果这些任务中偶尔有几个失败，那么可以只去检查运行失败的任务的日志。如果它们都是失败的任务，既然它们都执行相同的代码，那么完全可以任意挑选其中的一个进行查看。如果执行的 Job 看上去比预期的要慢，那么可以查看那些比其他任务消耗明显多的时间的任务。

通常查看所有的 map 任务或所有的 reduce 任务是否消耗差不多的执行时间是很有帮助的。如果消耗的时间差异很大，那么就是产生数据倾斜问题了。



Hadoop job_201104081526_0019 on localhost

User: gates
Job Name: PigLatin:distinct_symbols.pig
Job File: hdfs://localhost:9000/tmp/hadoop-gates/mapred/system/job_201104081526_0019/job.xml
Job Setup: Successful
Status: Succeeded
Started at: Wed May 04 12:47:33 PDT 2011
Finished at: Wed May 04 12:48:02 PDT 2011
Finished in: 29sec
Job Cleanup: Successful

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	1	0	0	1	0	0 / 0
reduce	100.00%	1	0	0	1	0	0 / 0

	Counter	Map	Reduce	Total
Job Counters	Launched reduce tasks	0	0	1
	Launched map tasks	0	0	1
	Data-local map tasks	0	0	1
FileSystemCounters	FILE_BYTES_READ	0	1,674	1,674
	HDFS_BYTES_READ	3,194,099	0	3,194,099
	FILE_BYTES_WRITTEN	1,706	1,674	3,380
	HDFS_BYTES_WRITTEN	0	20	20
Map-Reduce Framework	Reduce input groups	0	1	1
	Combine output records	1	0	1
	Map input records	57,391	0	57,391
	Reduce shuffle bytes	0	1,674	1,674
	Reduce output records	0	1	1

图 7-10　Hadoop 任务的网页页面

7.1.7　调试技巧

除了前面所介绍的工具之外，还有其他一些调试 Pig Latin 脚本的注意点。首先，如果 **illustrate** 提供的信息仍然无法满足用户，那么在真正放到 Hadoop 集群执行之前使用本地模式对脚本进行测试。在大多数情况下，这需要对一个样本数据进行操作，这是有

难度的，正如前面 7.1.4 节“illustrate”所解释的原因。但是使用本地模式有几个优点。其一，尽管还是慢，但是会比使用 Hadoop 网格要快，因为后者需要等待资源槽位，而且 job 初始化过程一般都要至少 30 秒的时间（相对而言本地模式下这一过程需要 10 秒钟）。其二，所有操作符产生的日志从屏幕上都可以看到，而不是放在某个任务节点上。其三，本地模式下使用的都是本地进程，这意味着用户可以对这个进程进行调试，这在调试 UDF 的时候特别有帮助。

我发现的第 2 个小技巧是有时需要关闭一些特定的功能来看是否是因为它们所造成的问题。这包括 Pig 进行的特定的优化或者没有修复所有 Bug 的新功能[1]。表 7-2 列举了可以关闭的功能。这些都是可以通过命令行参数传递给 Pig 的。

表 7-2　　可以关闭的功能列表

命令行选项	命 令 作 用	什么情况需要关闭
-t SplitFilter	阻止 Pig 对过滤器谓词进行分割，防止部分谓词前推到数据流上端	过滤器移除的行并非是预期的
-t MergeFilter	阻止 Pig 为了更高效的计算而合并临近的操作符	过滤器移除的行并非是预期的
-t PushUpFilter	阻止 Pig 将数据流中过滤器操作符前置到临近的操作符之前	过滤器移除的行并非是预期的
-t PushDownForEachFlatten	阻止 Pig 将数据流中包含 flatten 操作的 foreach 操作符前置到临近的其他操作符之前	foreach 并没有产生预期的行或者字段
-t ColumnMapKeyPrune	阻止 Pig 来预测脚本中真正使用的那些字段并让加载器只加载这些字段	加载函数返回的字符并非是预期的
-t LimitOptimizer	阻止 Pig 在数据流中将 limit 操作符前置到临近的其他操作符之前	limit 操作并没有返回预期的指定行数
-t AddForEach	阻止 Pig 在脚本中加入 foreach 操作符以移除不需要的字段	结果中没有期望的字段
-t MergeForEach	阻止 Pig 为了提高运算效率合并临近的 foreach 操作符	foreach 并没有产生预期的行或者列
-t LogicalExpressionsSimplifier	阻止 Pig 进行一些表达式简化优化	foreach 并没有产生预期的值
-t All	关闭所有逻辑优化策略。物理优化策略（例如使用组合器、multiquery 等）依旧是生效的	脚本并没有产生预期的行，同时需要判断逻辑优化器是否是产生问题的原因之一

[1] 如果用户发现当关闭一个功能后可以避免一个 Bug，请提一个 JIRA 传票到社区中，这样这个问题才能最终被解决。

续表

命令行选项	命 令 作 用	什么情况需要关闭
-D pig.usenewlogicalplan=false	阻止 Pig 使用 0.8 版本中介绍的新的逻辑计划。这只对 0.8 和 0.81 版本有效	在之前的其他版本中可以执行的脚本在 0.8 版本中无法执行
-D pig.exec.nocombiner=true	阻止 Pig 使用 Hadoop 的组合器	帮助用户检查 UDF 对 Algebraic 接口的实现是否有问题，因为只有在使用这个组合器的时候才会调用这个类
-D opt.multiquery=true	阻止 Pig 将多个数据流组装到一个 MapReduce 任务中执行	当 multiquery 脚本消耗完内存，运行性能较差，或者是其他原因导致失败
-D pig.noSplitCombination=true	阻止 Pig 为了减少 map 任务的个数对输入划分进行合并	一些输入格式，例如 HBase，是不能够对它们的划分进行合并的

提示

在 Pig 0.8.0 版本中，逻辑优化器和逻辑计划全部都重写了。在 0.8.0 版本中默认情况下使用新的优化器和逻辑计划。在 0.8.0 版本发布之后，新优化器和计划中发现了一些问题。如果用户使用的是 0.8.0 版本的 Pig，我强烈建议用户升级到 Pig 0.8.1 版本。截止到我写本章的时候，0.8.0 版本中所有已知的问题在 0.8.1 中都修复了。如果不能进行升级的话，一个变通的解决办法是在表 7-2 中描述的那些命令行选项关闭新的逻辑计划。在 Pig 0.9 版本中，旧的逻辑计划已经移除了。

7.2 使用 PigUnit 测试用户的脚本

作为开发过程的一部分，需要对 Pig Latin 脚本进行测试。甚至一旦完成脚本，常规的测试可以帮助确认对 UDF 的修改，对脚本的修改，或者用户所使用的 Pig 和 Hadoop 的版本不会导致用户的代码不可用。PigUnit 提供了一个集成到 JUnit 中的单元测试框架用于写一些可以在常规场景下执行的单元测试。PigUnit 是在 Pig 0.8 版本中加入的。

下面我们大体上看一下怎样使用 PigUnit 测试脚本。首先，我们需要一个脚本用于测试：

```
--pigunit.pig
divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
grpd = group divs all;
```

```
avgdiv = foreach grpd generate AVG(divs.dividends);
store avgdiv into 'average_dividend';
```

其次，我们需要使用到 *pigunit.jar* 这个 JAR 文件。该 JAR 并非是标准 Pig 发行版的一部分，但是用户可以从发行版的源码中编译出来。具体操作是，到发行版相关的文件夹下执行 ant jar pigunit-jar 就可以了。当编译完成后，文件夹下应该会产生两个文件：*pig.jar* 和 *pigunit.jar*。需要将这两个文件加入 classpath 中后才能运行 PigUnit 测试。

这之后，我们需要测试数据。我们可以使用一个存在的输入文件，或者在测试过程中产生一些输入数据对脚本进行测试。后面将分别展示这两种方式。

最后，我们需要写一个 JUnit 可以执行的 Java 类进行测试。我们以处理之前的那个脚本作为一个简单的例子：

```
// java/example/PigUnitExample.java
public class PigUnitExample {
    private PigTest test;
    private static Cluster cluster;

    @Test
    public void testDataInFile() throws ParseException, IOException {
        //构造一个使用脚本 pigunit.pig 的 PigTest 实例
        test = new PigTest("../pigunit.pig");

        //这里指定预期的输出结果。格式是一行一个字符串
        //本例中预期只有一行输出
        String[] output = { "(0.27305267014925455)" };

        //执行测试并检验实际输出和我们的预期输出是否一致
        //字符串"avgdiv"告诉了 PigUnit 需要对哪个别名的输出值进行检验
        //它会将那个别名对应的输出结果存放到一个文件中
        //然后读取文件内容和 output 进行对比
        test.assertOutput("avgdiv", output);
    }
}
```

用户也可以在测试代码中指定输入数据而不是依赖于一个存在的数据文件：

```
// java/example/PigUnitExample.java
    @Test
    public void testTextInput() throws ParseException, IOException {
        test = new PigTest("../pigunit.pig");

        //通过合成的方式构造输入，而不是从文件中读取内容
        //格式是每行一条记录，使用 tab 键分割
        String[] input = {
            "NYSE\tCPO\t2009-12-30\t0.14",
            "NYSE\tCPO\t2009-01-06\t0.14",
```

```
            "NYSE\tCCS\t2009-10-28\t0.414",
            "NYSE\tCCS\t2009-01-28\t0.414",
            "NYSE\tCIF\t2009-12-09\t0.029",
        };

        String[] output = { "(0.22739999999999996)" };

        //使用构造的输入来执行测试脚本而不是
        //通过 load 语句加载指定位置的文件来执行
        //"divs"是输入数据的别名
        //和前面的例子中一样，"avgdiv"是将和 output 进行
        //比较的测试结果的别名
        test.assertOutput("divs", input, "avgdiv", output);
    }
```

也可以在测试代码中指定 Pig Latin 脚本，同时也允许将输出结果和一个放有预期结果的文件进行对比：

```
// java/example/PigUnitExample.java
    @Test
    public void testFileOutput() throws ParseException, IOException {
        //这个脚本是一组字符串，每行一个字符串
         String[] script = {
           "divs = load '../../../data/NYSE_dividends' as (exchange,
symbol,
            "grpd = group divs all;",
            "avgdiv = foreach grpd generate AVG(divs.dividends);",
            "store avgdiv into 'average_dividend';",
        };
        test = new PigTest(script);

        //输出结果和一个包含了预期结果的文件进行对比
        test.assertOutput(new File("../expected.out"));
    }
```

最后，我们来看下如何将 PigUnit 和参数替换结合起来，以及如何指定预期的输出和存储下来的结果（而不是指定一个需要检测的别名）进行对比：

```
// java/example/PigUnitExample.java
    @Test
    public void testWithParams() throws ParseException, IOException {
        //在执行测试前需要事先提交的参数
        //格式是每一组键值对
        //参数名称=值
        String[] params = {
            "input=../../../data/NYSE_dividends",
            "output=average_dividend2"
        };
        test = new PigTest("../pigunitwithparams.pig", params);
```

```
        String[] output = { "(0.27305267014925455)" };

        //输出结果和预期结果进行对比
        test.assertOutput(output);
    }
```

这些列子可以通过执行包含本章中所有示例的那个 build.xml 文件运行。这些例子并非十分详尽，请查看代码本身以对其有一个完整的了解。可以查看 Pig 发行版中 *test/org/apache/pig/test/pigunit/TestPigTest.java* 位置的 PigUnit 测试类，以获得更多更深入的示例。该文件展示了 PigUnit 的大部分功能。

第 8 章

让 Pig 飞起来

谁说猪不能飞？知道如何优化 Pig Latin 脚本可以显著地提高执行性能。Pig 仍然是一个比较年轻的项目而且并没有一个精致的优化器做出最好的选择。反而，和 Pig 的用户选择这一设计思想一致，需要用户来做这些选择。除了仅仅优化脚本之外，根据投入的工作量 Pig 和 MapReduce 都可以被调优到更高效，同时也有方式对数据层进行优化。本章介绍了一些可以使 Pig 飞起来的功能。

在研究如何优化 Pig Latin 脚本之前，很有必要先了解 Pig 任务中哪些项可能造成瓶颈。

输入数据量大小

看上去一个大型的并行系统应该没有 I/O 负担约束。Hadoop 的并行架构降低了 I/O 约束但并没有移除这个约束。可以一直增加更多的 map 任务。然而，收益递减原理开始生效了。额外的 map 任务需要更多的时间去启动，而且 MapReduce 需要寻找更多的槽位来执行它们。如果 map 任务的个数需要两倍于实际具有的槽位来执行，那么将会消耗两倍于平均 map 过程执行时间来执行完所有的 map 任务。在那种情况下增加额外的一个 map 任务都会使情况更加糟糕，因为 map 所消耗的时间将增加到平均时间的 3 倍。同时，读入的每条记录都可能需要解压缩和进行反序列化。

shuffle 数据量大小

这里所说的 shuffle 数据量大小是指从 map 任务转移到 reduce 任务的数据量大小。所有这些数据需要进行序列化，排序，通过网络传输，合并和反序列化。同时，map 的个数和 reduce 的个数也是有影响的。每一个 reducer 需要访问每一个 mapper，然后从 map 的输出中找到属于它的那部分数据，然后进行复制。因此如果有 m 的 map 任务和 r 个 reduce 任务，那么 shuffle 将有 $m \times r$ 个网络连接。如果 reducer 需要在一个通道中合并太多的 map 输入，那么就必须使用多通道合并，也就是要多次读取输入然后再多次写到磁盘上（请查看附录 B 的“Combiner 阶段”获得更详细的信息）。

输出数据量大小

通过 MapReduce 任务写入的每条记录都必须进行序列化，还有可能进行压缩，之后才会写到存储器中。如果使用 HDFS 作为存储器，数据需要完成写入 3 个不同的机器中才认为是真正写入了。

中间结果数据量大小

Pig 通过将数据存储到 HDFS 中来在 MapReduce 任务间移动数据。因此这里的中间结果数据量大小受到前面所说的输入数据量大小和输出数据量大小因素的影响。

内存

一些计算需要任务在内存中存放很多的信息，例如，join 连接操作。如果 Pig 不能同时将所有的值都载入内存，那么就需要将一些值分割出来存放在磁盘上。这会导致性能明显下降，当记录需要写入磁盘和从磁盘读取的时候，可能需要多次进行读写。

8.1 编写优质的脚本

在书写 Pig Latin 脚本的时候有一些注意事项可以帮助降低前面所说的性能瓶颈。回顾下第 5 章和第 6 章哪些操作符会触发新的 MapReduce 任务将会对理解下面的内容有帮助。

8.1.1 尽早地并经常地进行过滤

尽早尽快地过滤不需要的数据将会对提升脚本执行效率有帮助。将过滤器尽可能地放置在脚本的前面可以减少 MapReduce 任务间的 shuffle 和存储的数据量大小。Pig 的逻辑优化器会尽可能地将过滤器前置。当过滤器连接着多个谓词的时候，而且一个或多个的谓词可以放置在过滤器前面的操作符上的话，Pig 就会在最后面对过滤器进行分割然后前推合适的谓词。这样允许 Pig 在无法对整个过滤器前推的时候可以对过滤器部分内容进行前推。表 8-1 描述的是一旦它们被分割后在什么情况下这些过滤器谓词会或者不会进行前推。

表 8-1 Pig 什么时候前推过滤器

前面的操作符	过滤器是否进行推测执行	注　　释
cogroup	有时进行推测执行	如果只是用于 cogroup 的一个输入而且不包含 UDF，那么过滤器会被前推
cross	有时进行推测执行	如果只是用于 cross 的一个输入那么过滤器会被前推
distinct	会进行推测执行	

续表

前面的操作符	过滤器是否进行推测执行	注　　释
filter	不会进行推测执行	会尝试将过滤器合并并且避免数据传递通过第 2 个操作符。这仅当所有的过滤器前推都完成时才会进行
foreach	有时进行推测执行	当过滤器引用到的字段是之前存在的并且 foreach 后面也有，并且 foreach 并没有改变这些字段的时候会进行过滤器前推
group	有时进行推测执行	如果不包含 UDf，那么过滤器会被前推
join	有时进行推测执行	如果过滤器只用于 join 操作的一个输入，而且如果对这个输入进行的不是外部连接，那么过滤器会被前推
load	不会进行推测执行	
mapreduce	不会进行推测执行	mapreduce 对 Pig 是不透明的，因为它无法确定进行推测执行是否是安全的
sort	会进行推测执行	
split	不会进行推测执行	
store	不会进行推测执行	
stream	不会进行推测执行	stream 对 Pig 是不透明的，因为它无法确定进行推测执行是否是安全的
union	会进行推测执行	

同时，考虑到脚本中可能隐式地增加了过滤器。例如，对于 inner join 所有的键为 null 的记录都会被过滤。如果事先知道记录中的许多记录对应的键为 null，那么对输入使用过滤器应在进行 join 之前先过滤键值为 null 的记录，这会增强 join 的性能。

8.1.2　尽早地并经常地进行映射

对于早些版本的 Pig，我们告诉用户使用 foreach 语句尽早地移除不需要使用的字段。从 0.8 版本开始，Pig 的逻辑优化器做了一件公平的事情，当它认定某些字段不再使用的时候会强有力地移除这些字段：

```
-- 这里因为脚本中并没有使用到字段 itemid，因此并不需要加载它
txnx     = load 'purchases' as (date, storied, amount, itemid);
todays   = filter txns by date == '20110513'; -- 之后不会再用到字段 date
bystore  = group todays by storied;
avgperstore = foreach bystore generate group, AVG(todays.amount);
```

然而，用户始终比 Pig 的优化器要聪明，有些情况下用户可以判定某个字段不再使用而 Pig 却无法判断出来。如果将前面的例子中 **AVG(today.amount)**改为 COUNT(todays)，Pig 不能确定，经过过滤器后，只有 id 和 amount 字段是需要存储的。Pig 不能看到 COUNT

操作不需要传递给它数据包中的所有字段。每当将整个记录传递给一个 UDF 时(udf(*))或者传递的是一整个复杂数据类型字段时，Pig 也无法判断哪些字段是需要的。在这种情况下，需要用户自己在 foreach 语句中尽早地移除不需要的字段。

8.1.3 正确并合理使用 join

join 连接操作是最普遍的时间操作之一，同时也是最消耗资源的操作之一。选择正确的 join 实现可以显著地提高整体性能。图 8-1 所示的流程图可以帮助用户做出正确的选择。

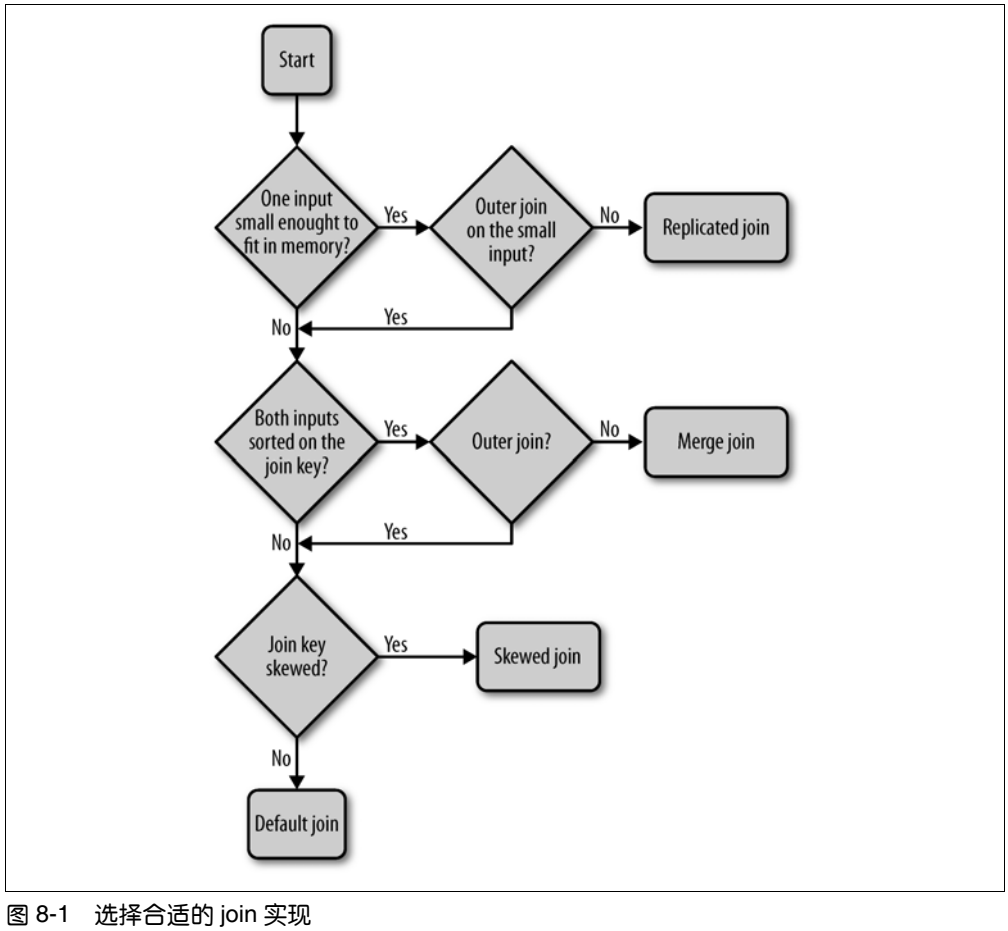


图 8-1 选择合适的 join 实现

一旦选择了采用什么样的 join 实现，还要保证输入按照正确的次序排列。对于 replicated join，小表必须是作为最后一个输入给出的。对于 skewed join，第 2 个输入是用于对大的键抽样的。对于默认的 join，最右边的输入整个记录是直接流过的，反而其他的输入对应给定键对应的记录是具体化到内存中的。因此如果知道 join 的一个输入中每个键对应的记录要比其他的多，那么应当把它放在 join 的最右边的位置。对于 merge join，

左边的输入是作为 MapReduce 任务的输入，因此开始的 map 任务个数取决于这个输入。如果一个输入比另一个输入要大，那么应该将输入大的放置在 join 的左边以此为任务获得更多的 map 任务。这也会降低为右边的输入建立索引的抽样步骤的数据量。如果想了解对于这些不同的 join 实现的完整和详细的描述，请查看 5.3.6 节“Join”和 6.1.2 节“使用不同的 Join 实现方法”。

8.1.4 适当的情况下使用 multiquery

每当做一些可以通过 multiquery 进行组装的操作的时候，例如分组和过滤，这些应当写在一个 Pig Latin 脚本中，这样 Pig 可以把它们组合在一起。虽然增加额外的操作会增加整体处理时间，但是还是会比单独执行这些任务要快。

8.1.5 选择正确的数据类型

在其他地方讨论过，有没有数据类型信息 Pig 都可以执行。如果加载函数处理的数据已经包含了数据类型信息，那么无需太多修改即可优化性能。然而，如果使用的是默认的 PigStorage 加载函数读取回车键分割的文件，那么是否使用数据类型将会影响到执行性能。

从一方面讲，将字段类型从 bytearay 转换成合适的类型会有一定的资源消耗。因此，如果不需要类型信息，那么就应当不去声明字段类型。例如，如果只是统计记录条数，省略类型声明也不会影响到脚本的输出结果。

从另一方面讲，如果做整数运算，加上类型信息可以帮助脚本执行的更好。当 Pig 被要求对 bytearray 类型的数据做数值计算的话，那么 Pig 会把 bytearray 转换成 double 类型来处理，因为这是最安全的推断。但是浮点运算在大多数的机器上比整数计算要慢得多。例如，如果对整数值做 SUM 运算，将类型声明为整数类型可以很好地提供执行性能。

8.1.6 选择合适的并行值

设置合理的并行数是比较困难的，因为有很多影响因子。在我们讨论这些影响因子之前，需要了解一些背景知识。用户一般自然而然地认为并行数越多越好，然而事实并非如此。和其他资源一样，并行会有一个网络消耗，正如在 shuffle 数据量大小性能瓶颈那段所讨论的一样。

其二，增加并行数对用户的脚本可能还存在一个潜在的问题，因为集群具有有限的 reduce 槽位，或者分配给用户的调度队列对并行数有限制。如果提供有 100 个 reduce 槽位，但是指定的语句为 **parallel 200**，那么每次也只能执行 100 个 reduce 任务。reducer 将以两个独立的组来执行。因为在开启 reduce 任务和结束 reduce 任务时有一定的资源开销，同时当并行数提高后 shuffle 阶段效率会下降，因此通常会选择比实际提供的槽位更多的 reducer 来执行，并非能够提高效率。需要留有空间让 MapReduce 重启一些失败的 reducer 和无须延长 reduce 时间来完成一些推断执行。关于推断执行的更多信息请

查看附录 B 的“故障处理”章节。

同样的，牢记并行中的数据倾斜的影响问题是很重要的。MapReduce 通常情况下可以很好地将键均分到各个 reducer 中，但是键所对应的记录数常常差异很大。因此对于那些键对应的记录数明显多得多的 reducer 会明显地比其他 reducer 执行耗时要久。除非前一个任务的所有 reducer 任务都完成了，否则 Pig 不会启动下一个 MapReduce 任务。因此最慢的那个 reducer 任务决定了任务的执行时间总长度。

如果传送给 reducer 的输入数据量为 10GB 然后设置的 parallel 参数为 10，但是有一个键对应的数据占了总数据的 50%（并非是常见的情况），那么会出现 9 个 reducer 很快结束了但是还剩余一个一直在拖后腿。这时增加并行数不会有帮助而只会浪费更多的资源。取而代之的是，需要使用 Pig 的机制来控制处理数据倾斜。

8.2 编写优质的 UDF

Pig 有一些特性试图使聚合函数执行得更快。**Algebraic** 接口允许 UDF 使用 Hadoop 的组合器（请参考附录 B 的“Combiner 阶段”章节）。**Accumulator** 接口允许 Pig 将一组记录分拆成多组然后依次将分拆后的组传递给 UDF。只有这样就可以避免需要同时将所有的记录具体化，而是当数据太多的时候进行分割并写到磁盘中。关于如何使用这些接口的详细信息，请查看 10.2 节“代表运算接口”和 10.3 节“累加器接口”。只要有可能，那么就要尽可能地在聚合函数中使用这些特性。

Pig 也具有帮助加载函数减少数据输入量的最优化方法。Pig 可以告诉加载器哪些字段是需要的和 map 中的哪些键是需要的。它也可以下推一些特定的过滤器。关于这方面的更多信息，请查看 11.1.4 节中“推测执行”部分和“加载元数据”部分的内容。

8.3 调整 Pig 和 Hadoop

乘坐商业喷气式飞机出远门的时候，你们有在驾驶舱的飞行驾驶员身边看过所有的那些刻度盘、仪表和操纵杆吗？调整 Hadoop 和这个过程类似，很多很多的选项，其中的某一些可能导致结果差异很大。但是如果没有相应的技术，就很难知道如何调节才是正确的。表 8-2 给出的是一些重要的性能调节属性。

提示

这张表是从 Tom White (O’ Reilly)的《Hadoop:The Definitive Guid，Second Edition》一书中的表 6-1 和表 6-2 精简而来，当然是在版权允许范围内的。从这两张表中可以查看到完整的参数列表。

表 8-2　　MapReduce 性能调节属性

属 性 名 称	数据类型	默认值	描　　述
io.sort.mb	int	100	对 map 的输出进行排序时使用的内存缓冲区大小，单位是 MB。增加这个值可以减少从 map 端来的划分的个数并可以使组合器更加高效，但是也因此会减少 map 任务的内存使用量
io.sort.factor	int	10	对文件进行排序时所能启动的最大数据流个数。通常会将这个值提高到 100
min.num.spills.for.combine	int	3	触发执行组合器的最小文件划分个数（从 map 端生成）
mapred.job.shuffle.input.buffer.percent	float	0.7	在 shuffle 的数据复制阶段分配给 map 输出缓冲区的内存堆总大小的最大百分比
mapred.job.shuffle.merge.percent	float	0.66	用于开启合并输出和写入磁盘的处理过程中 map 输出缓冲区的内存堆总大小的最小百分比

和 Hadoop 相比，调整 Pig 要简单得多。有几个内存相关的参数可以帮助确保 Pig 尽可能按照正确的方式使用内存，这些参数如表 8-3 所示。

表 8-3　　Pig 性能调节属性

属 性 名 称	类　型	默认值	描　　述
pig.cachedbag.memusage	float	0.1	Pig 为 map 或者 reduce 任务中所有 bag 数据能够分配的内存堆百分比。一旦 bag 占满了这个量，那么数据将会写入磁盘。增大这个值可以减少在执行过程中向磁盘写入的数据但是会增大任务导致内存溢出的可能性
pig.skewedjoin.reduce.memusage	float	0.3	在 join 过程出现数据倾斜时当尝试将一边的数据具体化到内存中去的时 Pig 能够使用的内存堆百分比。把这个值设置的大些会减少大数据量的键被划分的个数也因此影响这些键对应的记录将会被拷贝多少份，但是这样会增加 reduce 任务内存溢出的概率

Pig 和 MapReduce 的这些参数都可以在 Pig Latin 脚本中使用 set 命名进行设置值或者在命令行通过**-D** 来指定。

8.4 对计算中间结果进行压缩

现在我们可以很清楚地看到，Pig 中资源消耗最大的过程，一些是在 map 和 reduce 间和 MapReduce 任务间移动数据造成的。压缩可用于减少存储到磁盘的数据量和通过网络进行写的数据量。默认情况下，压缩功能是关闭的，无论是对于 map 任务和 reduce 任务之间的还是 MapReduce 任务之间的都是关闭的。

如果需要 map 任务和 reduce 任务之间传输的数据进行压缩，那么要使用到两个 Hadoop 参数：**mapred.compress.map.output 和 mapred.map.output.compression.codec。**如果要开启压缩，需要将 **mapred.compress.map.output** 设置为 **true**。同时还需要指定使用哪种压缩类型。最常用的压缩类型是 gzip 和 LZO。gzip 需要消耗更多 CPU 资源但是压缩效果更好。如果使用 gzip，那么需要设置 **mapred.map.output.compression.codec** 为 **org.apache.hadoop.io.compress.GzipCodec**。在大多数情况下，LZO 格式具有更好的性能优势。更详细的内容请查看下文中“在集群中配置使用 LZO 进行压缩”部分的内容。如果使用 LZO 作为编码解码器，那么需要设置 **mapred.map.output.compression.codec** 为 **com.hadoop.compression.lzo.LzopCodec**。

MapReduce 任务之间传输的数据是否进行压缩也会对 Pig 的性能有显著影响。如果 Pig 脚本中含有 join 或者其他会增大数据量的操作的时候，这种影响就更加明显了。打开压缩，需要设置 **pig.tmpfilecompression** 为 **true**。同样的，可以通过设置 **pig.tmpfilecompression.codec** 为 **gzip** 或者 **lzo** 来指定是使用 gzip 还是 LZO 压缩格式。以我们在开发这个功能时进行的测试来看，使用 LZO 时性能提高了 4 倍，使用 gzip 格式时性能有稍微的下降。

在集群中配置使用 LZO 进行压缩

LZO 是在公共通用许可证（GPL）下许可使用的，因此可以作为 Apache Hadoop 或者是 Apache Pig 的一部分进行发行。如果想使用它，需要编译和安装针对 Hadoop 的 LZO 插件同时还要配置集群。

可以访问 http://code.google.com/a/apache-extras.org/p/hadoop-gpl-compression 页面单击下载标签来下载 LZO。下载 *hadoop-gpl-compression* 压缩包到本地然后解压缩。之后需要在本地编译本地 LZO 函数库。要确保在和集群机器一致的系统上进行编译，因为这是 C 语言编写的因此不可以移植。一旦编译好本地函数库，那么下一步就是需要把它安装到集群中了。这两个过程的详细信息都可以在如下网页找到 http://code.google.com/a/apache-extras.org/p/hadoop-gpl-compression/wiki/FAQ 。这个压缩包中发现的几个 Bug 的修复方案已经在 GitHub 中提交了。对于官方的压缩包如果发现了问题，那么也可从 GitHub 中下载和编译这个版本。

8.5 数据层优化

如何设计数据对 Pig 任务的执行效率有显著的影响。从一方面讲，期望能够组织文件使得 Pig 扫描最小数据集的记录。例如，假设有一组常规数据集需要每个小时读取一次，那么每个小时的数据独立放置到一个文件中是个不错的选择。从另一方面来讲，创建的文件数越多，那么对 NameNode 造成的压力就越大。而且 MapReduce 对于大文件的操作比对小于一个 HDFS 数据块大小（默认为 64MB）的文件要高效得多。我们需要在这两个相互矛盾的因素中找到一个平衡点。

从 0.8 版本开始，当输入的是文件而且它们比一个 HDFS 数据块的一半还小的时候，Pig 在使用这个文件作为输入时会自动将较小的部分合并在一起。这样可以使 MapReduce 更高效同时会启动更少的 map 任务。这通常总是会提高集群的整体资源利用率的。但是对于单独的某个查询并非总是可以提高性能，相反，因为会丢失合并后的块的位置信息，所以 map 任务可能消耗更多的执行时间。如果想关闭该功能，可以通过命令行参数 **-Dpig.noSplitCombination=true** 设置或者在参数文件 *pig.properties* 中设置这个参数。

8.6 垃圾数据处理

在处理千兆字节或者吉字节的数据时，非常有可能出现至少存在一行错误记录或者因此导致一个非预期的结果这种情况。一个例子就是对零进行除法运算，尽管没有记录是支持分母为零进行计算的。一条垃圾数据可能会导致整个任务执行失败。为了避免产生这些失败，Pig 会插入 null 值，或给出一个警告，然后继续处理。这样一来，任务仍可以完成。在最后会有一个统计信息报告有多少个警告。应当检查下这些警告确保这个任务中一些记录的失败是否是可以接受的。如果想获得这些警告的详细信息，可以在命令行通过-w 命令关闭汇总功能。

第 9 章

在 Python 中嵌入 Pig Latin 脚本

Pig Latin 是一种数据流语言。与常规的编程语言不同，它不包含控制流结构，例如 **if** 和 **for**。对于大多数的数据处理程序而言，Pig 提供的操作符已经足够了。但是有一些类别的问题要么要求同一个数据流按照可变的次数重复执行，要么需要基于一个操作符的结果进行分流。迭代处理，即一个计算过程需要不断重复直到误差边界位于一个可接受的限制范围内才会停止，就是一个这种类别的例子。在处理开始前无法预先知道数据流需要被重复执行多少次。

很难做到以有用和直观的方式在一种语言中混合数据流和控制流。创建一种常规的语言和所有与之相关联的工具，例如 IDE 和调试器，这是值得考虑的一件大事，同时现在也不缺少这类语言。如果我们把 Pig Latin 设计成一种常规的语言，那么它将需要用户去学习一种相当复杂的语言来处理数据。因为这些原因，我们决定可以使用当前的脚本语言内嵌执行 Pig 脚本。这样可以避免去发明一种新的语言而同时可以提供给用户他们处理数据的新功能。[1]

和 UDF 一样，在 0.9 版本中我们选择 Python 作为嵌入式 Pig 的初始发布支持的语言。内嵌的接口是一个 Java 类，因此需要使用一个 Jython 解释器才能执行这些内嵌了 Pig 脚本的 Python 脚本。这意味着可以使用 Python 2.5 的功能但是不能使用 Python 3 的功能。在将来，我们希望能够将系统延伸到其他可以访问 Java 对象的脚本语言，例如 JavaScript[2] 和 JRuby。当然，既然 Pig 的基础架构全部是使用 Java 语言的，也有可能使用这个相同的接口将 Pig 脚本嵌入 Java 脚本中。

这种嵌入是以类似 JDBC 的样式进行的，也就是首先 Python 脚本会对一段 Pig Latin 脚本进行编译，然后将在 Python 中定义的变量传递给它，最后执行它。通过接口，也可

[1] 在一些文档、wiki 页面和 JIRA 问题跟踪中，嵌入式 Pig 被称为图灵机计算 Pig。这也是该项目开始时的名称，尽管我们没有使 Pig 本身具有图灵机计算功能。

[2] 0.9 中已经有了一个实验性的版本。

以做文件系统操作，注册 JAR 文件包，以及执行其他一些通用操作。这个接口的最上层类是 **org.apache.pig.scripting.Pig**。

本章通篇将使用一个从网页爬虫数据中计算页面等级的例子。这个例子可以在样例代码所在的文件夹 *examples/ch9* 中找到。这段代码通过对一组 URL 和链接数据进行迭代操作然后为每个 URL 算出一个页面等级值。[1]在 examples 文件夹下可以找到这个例子的输入——数据集 *webcrawl*。这个输入中的每一条记录都包含一个 URL，一个起始级别 1 和一个包含 tuple 的数据包，tuple 中包含了在这个 URL 下发现的所有链接地址：

```
   http://pig.apache.org/privacypolicy.html 1
{(http://www.google.com/privacy.html)}
   http://www.google.com/privacypolicy.html 1
{(http://www.google.com/faq.html)}
   http://desktop.google.com/copyrights.html 1 {}
```

尽管控制流是通过 Python 脚本完成的，但还是可以通过 Pig 的 *bin/pig* 脚本来执行。*bin/pig* 会查看#! 行的标记然后调用相应的解释器。这允许用户在期望调用 Pig Latin 脚本的系统上使用这些脚本。同时也允许 Pig 自动地从这个文件中引入 UDF 并在错误信息中能够给出错误所在行的正确位置。

为了使用 **Pig** 这个类和相关的对象，Python 脚本代码的开始需要首先声明引入它们：

```
   from org.apache.pig.scripting import *
```

9.1 编译

调用静态方法 **Pig.compile** 会促使 Pig 对代码进行预编译。因为还没有传入参数，所以这个检查并不能完全验证这段脚本。类型检查和其他语义检查在此阶段都还没有完成，仅仅完成了语法检查。**compile** 方法会返回一个可以传入一组变量的 **Pig** 对象：

```
   # pagerank.py
   P = Pig.compile("""
   previous_pagerank = load '$docs_in' as (url:chararray, pagerank:float,
                        links:{link:(url:chararray)});
   outbound_pagerank = foreach previous_pagerank generate
                        pagerank / COUNT(links) as pagerank,
                        flatten(links) as to_url;
   cogrpd = cogroup outbound_pagerank by to_url,
                        previous_pagerank by url;
   new_pagerank = foreach cogrpd generate group as url,
                         (1 - $d) + $d * SUM (outbound_pagerank.pagerank)
                        as pagerank,
                        flatten(previous_pagerank.links) as links,
                        flatten(previous_pagerank.pagerank) AS
previous_pagerank;
```

[1] 例子代码是由 Julien Le Dem 提供的。

```
    store new_pagerank into '$docs_out';
    nonulls = filter new_pagerank by previous_pagerank is not null and
                          pagerank is not null;
    pagerank_diff = foreach nonulls generate ABS (previous_pagerank -
pagerank);
    grpall = group pagerank_diff all;
    max_diff = foreach grpall generate MAX (pagerank_diff);
    store max_diff into '$max_diff';
    """)
```

这段 Pig Latin 脚本中我们之前没有见到的信息是有 4 个参数，脚本中标记为$d、$docs_in、$docs_out 和$max_diff。这些参数的语法和前面参数传入的语法是相同的。但是，Pig 期望在 **bind** 被调用时这些参数是由控制流脚本提供的。

除了例子中所示的方法外还有 3 种其他编译方法。*compile(String name，String script)* 除了需要执行的 Pig Latin 脚本的名称外还包含了一个参数 name。这个定义的 name 可以用于在其他的 Pig Latin 代码段里面引用这段代码：

```
P1 = Pig.compile("initial", """
A = load 'input';
...
""")
    P2 = Pig.compile("""
import initial;
B = load 'more_input';
...
""")
```

另外两种编译方法叫做 **compileFromFile**。这些方法与 **compile** 方法接收的参数一样，但是它们期望接收的脚本参数是包含要执行的脚本的路径，而不是脚本本身。

9.2 绑定

一旦脚本编译成功，下一步就是将控制流中的变量绑定到 Pig Latin 脚本中的变量。在我们的样例脚本中，提供了一个参数值为 map 类型的 bind 方式。其中的键是 Pig Latin 中参数的名称。下面例子中的值是脚本处理过程中不断更新的字符串值。它们也可以被 Python 变量调用：

```
# pagerank.py
params = { 'd': '0.5', 'docs_in': 'data/webcrawl' }

for i in range(10):
    out = "out/pagerank_data_" + str(i + 1)
    max_diff = "out/max_diff_" + str(i + 1)
    params["docs_out"] = out
    params["max_diff"] = max_diff
```

```
Pig.fs("rmr " + out)
Pig.fs("rmr " + max_diff)
bound = P.bind(params)
stats = bound.runSingle()
if not stats.isSuccessful():
   raise 'failed'
mdv = float(str(stats.result("max_diff").iterator().next().get(0)))
print "max_diff_value = " + str(mdv)
if mdv < 0.01:
   print "done at iteration " + str(i)
   break
params["docs_in"] = out
```

第一次执行时，Pig Latin 中的**$d** 取值为 **0.5**，**$docs_in** 取值为文件名 *webcrawl*，**$docs_out** 取值为 *out/pagerank_data_1*，**$max_diff** 取值为 *out/max_diff_1*。

bind 会返回一个 **BoundScript** 对象。该对象可以被执行、展开、描述或者说明。正如这个脚本中所展示的，单个 **Pig** 对象可以被多次绑定。只有在开始过程中才需要进行编译，然后每次使用不同的值进行绑定。

在我们的例子中，**bind** 接收的是一组 map 值。如果所有的 Python 变量和 Pig Latin 变量具有相同的名称，可以无需指定变量就调用 **bind**。这样 **bind** 自己会在 Python 内容中找具有和 Pig 脚本中相同变量名称的参数的值。如果它不能发现合适的值，那么它将会抛出一个错误信息。我们可以将样例脚本修改如下：

```
# pagerankbindnoarg.py
d = 0.5
docs_in = 'data/webcrawl'

for i in range(10):
    docs_out = "out/pagerank_data_" + str(i + 1)
    max_diff = "out/max_diff_" + str(i + 1)
    Pig.fs("rmr " + docs_out)
    Pig.fs("rmr " + max_diff)
    bound = P.bind()
    stats = bound.runSingle()
    if not stats.isSuccessful():
       raise 'failed'
    mdv = float(str(stats.result("max_diff").iterator().next().get(0)))
    print "max_diff_value = " + str(mdv)
    if mdv < 0.01:
       print "done at iteration " + str(i)
       break
    docs_in = docs_out
```

绑定多组变量

我们计算网页等级的脚本中为了迭代所有的数据对编译好的 Pig Latin 脚本多次绑定不

同的变量。这些任务是分别执行的，这也是计算页面等级需要的迭代天性所要求的。然后，有时需要同时执行一组任务，例如，假设要计算全世界所有国家的人口普查数据。这时希望对每个国家都执行相同的 Pig Latin 脚本，但是又不想分别执行它们。如果你打算每次只执行一个任务，那么采用像 Hadoop 这样的大规模并发处理系统将是毫无意义的。你想做的是告诉 Pig 去读取脚本并执行，但是不会在同一时间读取所有国家的数据。

有一种类型的 **bind** 提供了这个功能。不是接收一个参数的 map，而是接收一组包含参数的 map，同时也会返回一个 **BoundScript** 对象，但是当对这个对象调用 **run** 方法时，这个脚本所有独立的实例化对象将会一起执行：

```
#!/usr/bin/python
from org.apache.pig.scripting import *
pig = Pig.compile("""
    input = load '$country' using CensusLoader();
   ...
    store output into '$country_out';
""")

    params = [{'country': 'Afghanistan', 'country_out': 'af.out'},
   ...
              {'country': 'Zimbabwe', 'country_out': 'zw.out'}]

    bound = pig.bind(params)
    stats = bound.run()
```

9.3 运行

一旦我们获得了 **BoundScript** 对象，那么我们就可以调用 **runSingle** 方法来执行它。这会使 Pig 执行单一的 Pig Latin 脚本。当只将脚本和一组变量绑定的时候使用这个方法是合适的。**runSingle** 会返回一个 **PigStats** 对象。这个对象允许用户获得结果和检查脚本中到底发生了什么，包括状态信息，如果出现错误会有错误码和信息，以及关于执行本身的一些统计信息。表 9-1 总结了 **PigStats** 中相对重要的方法。

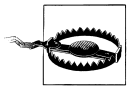
表 9-1　　PigStats 类中提供的方法

方　法	返　回　值
result(*String alias*)	参数是别名 alias，返回的是一个描述该别名对应存储的输出数据的 OutputStats 对象。从 OutputStats 对象中可以进行迭代获取结果
isSuccessful()	如果一切正常进行则返回 true，否则返回 false
getReturnCode()	获取返回码。关于返回码的详细介绍请查看表 2-1

续表

方　法	返　回　值
getErrorMessage()	如果执行失败返回错误信息。该方法会尝试获取返回错误信息中最相关的错误信息，通常是最后的那段错误信息
getAllErrorMessages()	如果执行失败，返回详细的错误信息列表
getOutputLocations()	返回脚本中存放的所有路径字符串。例如，如果脚本中有将输出写入 HDFS 中的指定文件，那么返回信息中将包括这个文件名
getOutputNames()	返回脚本中存放的别名列表
getRecordWritten()	返回这个脚本写入的总记录条数
getBytesWritten()	返回这个脚本写入的总字节数
getNumberRecords(String location)	参数是一个输出路径，返回的是写入这个输出路径的总记录条数
getNumberBytes(*String location*)	参数是一个输出路径，返回的是写入这个输出路径的总字节数
getDuration()	执行这个脚本时的时间
getNumberJobs()	这些脚本所执行的 MapReduce 任务个数

正如在例子中所看到的，**result()**方法所返回的 **OutputStats** 对象可用于递归地获得结果集。通过这个方法可以迭代访问数据，并使用 Python 脚本进行处理。标准的 **Tuple** 方法例如 **get()**可用于获得每条记录的内容。关于使用 **Tuple** 的讨论请查看 10.1.2 节“与 Pig 值进行交互”。基于迭代器所读取的结果，Python 可以决定是否停止迭代并声明执行成功，或者抛出错误，或者继续进行另一个迭代。



警告

为了可以使迭代器工作，用于存储来自别名的结果的存储函数必须也是个加载函数。Pig 会尝试使用同一个类来加载结果，尽管之前是用于存储结果的。默认的 PigStorage 对此支持得很好。

运行多个绑定

如果是将 **Pig** 对象绑定到一组包含参数的 map，那么不应当调用 **runSingle** 方法，而应当调用 **run** 方法。这可以使得 Pig 为每一个绑定开启一个线程然后执行。所有的这些任务会同时提交到 Hadoop，这将使用到 Hadoop 的并发特性。**run** 方法会返回一个 **PigStats** 对象列表。在列表中 **PigStats** 对象的次序和传递给 **bind** 方法的绑定变量的 map 的次序是一样的。因此第一个参数绑定的 map 得到的结果为 **PigStats** 列表的第一个位置，依此类推。

9.4 工具方法

除了到目前为止所提及的 **compile**、**bind** 和 **run** 方法之外，**Pig** 和 **BoundScript** 也提供了一些工具方法。

通过调用静态方法 **Pig.fs** 可以完成文件系统操作。传递给它的字符串应当是一个可以在 Grunt 控制台（参考第 3 章）执行的合法字符串。返回码与在控制台执行的返回码相同。

对于编译好的 Pig Latin 语句可以使用 **register**、**define** 和 **set** 命令，就像在非内嵌 Pig Latin 脚本中使用的方式一样。然而，有时可能需要注册一个 JAR 文件，定义一个函数别名，或者要设置一个在 Pig Latin 代码块中全局有效的值。在这些情况下可以使用表 9-2 中所示的 **Pig** 提供的静态方法。通过这些方法完成的 **register**、**define** 和 **set** 命令会影响到所有调用它们的编译好的 Pig Latin 代码：

```
   # register etc. will not affect this block.
   p1 = Pig.compile("...")

   Pig.registerJar("acme.jar")
   Pig.registerUDF("acme_python.py", "acme")
   Pig.define("d_to_e", "com.acme.financial.CurrencyConverter('dollar',
'euro'"))
   Pig.set("default_parallel", "100")

   # register etc. will affect p2 and p3
   p2 = Pig.compile("...")
   p3 = Pig.compile("...")
```

表 9-2　　Pig 提供的工具方法

方　法	参　数	等价的 Pig Latin 语句
registerJar(String jarfile)	jarfile 是将要注册的 JAR	register jarfile;
registerUDF(*String udffile, String namespace*)	udffile 是将要注册的 UDF 文件。namespace 是 UDF 将要放入的命名空间	register *udffile* using jython as *namespace*;
define(*String alias, String-definition*)	alias 是意义的名字。 definition 是将要被别名的字符串	define *alias definition*;
set(*String variable, String-value*)	variable 是需要设置的变量。value 是为变量所设置的值	set variable value;

一旦一个脚本完成了绑定而且返回了一个 **BoundScript** 对象，除了执行这个脚本外还可以使用 **describe**、**explain** 或者 **illustrate** 命令。这些与在非嵌入式 Pig Latin 脚本中的作用是完全一样的。然后，它们并非将结果输出给脚本，而是直接输出到标准输出中。（这些操作符用于进行调试而非用于直接将数据返回到脚本中。）

第 10 章

编写评估函数和过滤函数

现在是关注怎样扩展 Pig 的时候了。到目前为止我们已经了解了 Pig 提供的操作符和函数。但是 Pig 通过用户自定义函数（UDF）提供了一个增加个性化的处理逻辑的方式。这些是通过 Java 语言编写的，从 0.8 版本开始，也可以使用 Python 编写。[1] 本章将涵盖如何写求值函数，也就是操作单个元素数据或数据集合的 UDF。同时还会包括如何写过滤函数，也就是可以用作 **filter** 语句里的 UDF。

UDF 是非常强大的工具，因此接口有点复杂。在设计 Pig 的过程中，我们的目标是使简单的事情更简单使困难的事情更有可能完成。因此，最简单的 UDF 可以只要实现一个方法，但是如果想使用更高级的功能那么就需要实现一些其他方法了。本章会包含所有这些情况。

我们将通过几个可执行的 UDF 样例贯穿本章进行介绍。这里使用到的样例有一些是 Pig 内置的 UDF，它们可以在 Pig 发行版的 *src/org/apache/pig/buildin/* 目录下找到。另外一些可以在 GitHub 中的 *udfs* 目录下找到，该目录下还包含了其他一些 UDF 样例。

10.1 使用 Java 编写评估函数

Pig 和 Hadoop 都是使用 Java 语言编写的，因此自然而然地就可以使用 Java 来编写 UDF。这就允许了 UDF 可以访问 Hadoop 的 API 和使用 Pig 提供的很多功能。

在研究细节之前，有必要说明下 UDF 的命名。Pig 是通过在脚本中查找和 UDF 命名完全相同的 Java 类来定位 UDF 的。关于 Pig 会从哪些位置来查找 UDF，请参考 5.4.1 节“注册 UDF”和 5.4.2 节“define 命令和 UDF”。UDF 的命名是否要遵从全是大写，还是骆驼峰结构（例如 MyUdf），或者全是小写并没有一个公认的标准。即使是 Pig 内置

[1] 在 0.9 版本中，求值函数同样可以使用 JavaScript 编写，尽管这还处于试验阶段并没有通过完善的测试。

的 UDF 就这一点而言也并非是统一的。需要牢记的是，无论选择哪种方式，如果可以使命名简短，容易记住和书写，那么用户以及所有使用 UDF 的用户将会有更好的用户体验。

10.1.1 UDF 将在哪里执行

编写可以运行在并行处理系统上的代码是有挑战的。用户所编写的 UDF 的一个单独的实例将会被构建并运行在每个 map 或者 reduce 任务中。这些实例因为可能并非是同时执行的，因此它们之间无法共享状态。每一个 map 或者 reduce 任务中都只会有一个 UDF 实例，因此可以在上下文中共享状态。[1]

当为一个并行处理系统编码的时候，需要记住并行的力量。在串行执行程序中可以接受的操作在这里可能就是行不通的了。假设有一个 UDF，当它开始执行时，会先连接一个数据库服务器然后下载一个转换表。在一个串行执行或者低并行的环境中，这是个适当的方式。但是如果 job 中有 10,000 个 map 任务，它们同时连接数据库，那么很有可能就会有 DBA 找用户谈话了，而且谈话的内容可能并非使人愉快。

除了会在每个任务中生成一个实例之外，Pig 还会在前端生成执行计划阶段构造一份 UDF 实例，这样做有几个原因。其一，需要尽早确认它是可以构造 UDF 实例的，在生成执行计划阶段失败总比在执行过程中失败要好。其二，本章后面会讲到，它需要从 UDF 中读取一些模式和数据类型信息作为执行计划的一部分。同时这也给了 UDF 一个存储信息的机会，以便在后端提供给它本身的实例使用。

10.1.2 求值函数基本概念

所有求值函数都需要继承 Java 类 **org.apache.pig.EvalFunc**。这个类使用了 Java 中的范型，它根据 UDF 的返回值进行参数化。这个类的核心方法是 **exec**，它接收一条记录然后返回一条结果，执行的数据流中传递的每一条记录都会调用 exec 方法。它以 tuple 作为输入，其中包含了脚本传递给 UDF 的所有字段。返回值类型取决于使用哪种类型对 **EvalFunc** 进行参数化。对于简单的 UDF，这是唯一需要实现的方法。以下是一个求整数的整数倍指数然后返回值是 long 类型的 UDF 例子：

```
// java/com/acme/math/Pow.java
/**
 *一个求幂次简单的 UDF，第一个参数为底数，第二个参数为幂次数。
 *在 Pig Latin 脚本中使用方式为 Pow(x,y)，其中 x 和 y 均为整数。
 */

public class Pow extends EvalFunc<Long> {
public Long exec(Tuple input) throws IOException {
   try {
```

[1] 假设在脚本中有一个 UDF 实例。每一个对脚本中 UDF 的引用在后端都会成为一个独立的实例，即使它们放置在同一个 map 或者 reduce 任务中。

```
            /*通常情况下 UDF 的输入是一个 tuple，而不是确切的参数值。
             *UDF 需要知道其期望的是哪些参数并从 tuple 中获取到这些值。
             *如下代码的前两行分别获取到了这个 tuple 的第 1 个和第 2 个字段值。
             *因为 Tuple.get 方法返回的是对象，所以我们需要将其类型转换为整型。
             *如果类型转换失败，那么将会抛出异常。
             */
            int base = (Integer)input.get(0);
            int exponent = (Integer)input.get(1);
            long result = 1;

            /*这可能并非是最高效的实现方法 */
            for (int i = 0; i < exponent; i++) {
                long preresult = result;
                result *= base;
                if (preresult > result) {
                    //一旦溢出，将给出警告信息，而非抛出异常
                    warn("Overflow!", PigWarning.TOO_LARGE_FOR_INT);
                    //返回 null 将会告诉 Pig 失败了，但是期望仍然继续执行
                    return null;
                }
            }
            return result;
        } catch (Exception e) {
            //抛出异常将会导致这个 task 任务失败
            throw new IOException("Something bad happened!", e);
        }
    }
}
```

聚合函数中也需要实现 **EvalFunc**。因为 **group** 操作符为每个组返回一个在这个组内的所有记录的 bag，求值函数仍旧是读取一条记录然后返回一条记录。作为一个例子，我们来看下 Pig 中 **COUNT** 函数中对 **exec** 方法的实现方式。为了便于阅读，代码中去掉了对于错误信息处理的代码片段：

```
// src/org/apache/pig/builtin/COUNT.java
public Long exec(Tuple input) throws IOException {
    try {
        //数据包作为 tuple 的第一个元素传递给 UDF
        DataBag bag = (DataBag)input.get(0);
        Iterator it = bag.iterator();
        long cnt = 0;
        while (it.hasNext()){
           Tuple t = (Tuple)it.next();
           //对于 null 值和空 tuple 不用计算
           if (t != null && t.size() > 0 &&
               t.get(0) != null) {
               cnt++;
              }
```

```
            }
            return cnt;
        } catch (Exception e) {
            ...
        }
    }
```

正如 UDF 可以接受复杂数据类型的数据作为输入一样，它们也可以返回复杂数据类型的输出。例如，用户可以创建一个名为 **SetIntersection** 的 UDF 将两个 bag 类型的输入进行处理，然后返回一个 bag 数据类型的输出。

UDF 同样可以通过传递*号给 UDF 表示处理整条记录。这种情况下，用户预期的可能是传递给 UDF 的 **input Tuple** 参数，应该包含了传递给 UDF 所在的那个操作符的所有字段。但是事实上不是这样。反而，它只会包含一个字段，这个字段是 tuple 数据类型的，其中包含了所有的那些字段。假设有如下一段 Pig Latin 脚本：

```
data     = load 'input' as (x, y, z);
processed = foreach data generate myudf(*);
```

在这种情况下，**myudf.exec** 会接收到只包含一个字段的 tuple，这唯一的字段本身也是 tuple，其包含 3 个字段：x、y 和 z。如果想访问别名为 **data** 的关系中的字段 y，那么调用方法是 **t.get(0).get(1)**。

与 Pig 值进行交互

求值函数和其他 UDF 把 Pig 内部是如何处理数据类型的显示出来了。这意味着当用户读取一个字段并预期其为整数值时，用户需要命令该字段是 **java.lang.Integer** 的一个实例。关于 Pig 中数据类型和 Java 中表达方式的对应关系的详细列表，请查看 4.1 节“数据类型”。其中大部分数据类型，用户可以按照正常的方式构建合适的 Java 对象。但是，对于 tuple 和 bag 数据类型就需要做另外的处理了。这些是有接口的，但是并没有直接的构造函数。因此，用户需要使用工厂类来对这些进行处理，事实上也是这么做的。这样用户和开发者就可以根据自身的需要对 tuple 和 bag 进行不同的实现并让 Pig 使用这些实现。

TupleFactory 是用户创建 tuple 时必须使用到的单例类。用户同样是可以配置使用哪个 **TupleFactory** 的，因为既然用户提供了自己的 tuple 实现，那么用户也就需要提供生产这些 tuple 的工厂类。调用静态方法 **TupleFactory.getInstance()**，可以获得 **TupleFactory** 的一个实例来构造 tuple。

现在用户就可以使用 **newTuple()**或者 **newTuple(int size)**来创建新的 tuple 了。如果可以的话建议使用第 2 个方法，最好能够预先设置好要处理的 tuple 包含多少个字段。这样可以避免后面需要动态地增加 tuple 大小，因此也相对高效些。这个方法会创建一个具有 **size** 个数字段的 tuple，初始值全部为 null。之后用户可以通过调用 **Tuple** 中的 **set(int**

fieldNum, Object val)方法来设置字段内容。举个例子，第 11 章会有一个样例加载函数，其中创建 tuple 的代码片段如下：

```
// JsonLoader.java
private TupleFactory tupleFactory = TupleFactory.getInstance();

private Object readField(JsonParser p,
                         ResourceFieldSchema field,
                         int fieldnum) throws IOException {
    ...
        ResourceSchema s = field.getSchema();
        ResourceFieldSchema[] fs = s.getFields();
        Tuple t = tupleFactory.newTuple(fs.length);

        for (int j = 0; j < fs.length; j++) {
            t.set(j, readField(p, fs[j], j));
        }
    ...
}
```

如果在创建 tuple 的时候确实不知道它有多少个字段，那么可以是 **newTuple()**方法。这时可以使用 **Tuple** 的 **append(Object val)**方法增加字段，这个方法会将字段增加到 tuple 的尾部。

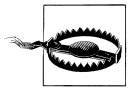
可以使用 **get(int fieldNum)**方法从 tuple 中读取数据。该方法返回的是一个 Java **Object** 数据对象，这是因为这个 tuple 并没有模式实例也并不知道每个字段的数据类型。用户需要将结果类型转换为其他合适的数据类型或者使用工具类 **org.apache.pig.data.DataType** 中提供的方法来确定数据类型。

和 tuple 一样，需要使用 **BagFactory** 来构造 bag。使用 **BagFactory.getInstance()**方法可以获得一个实例。调用 **newDefaultBag()**方法可以获得一个新的、空的 bag。这时用户可以通过调用 **DataBag** 的 **add(Tuple t)**方法来增加 tuple。用户最好使用这个方法来增加 tuple 而不是先构建好一组 tuple 然后将其传递给 **newDefaultBag(List<Tuple> listOfTuples)**方法，这是因为当数据不能完全载入内存中的时候 bag 会先将数据写入磁盘。我们再来看下加载函数 **JsonLoader** 中构造 bag 的代码片段：

```
// JsonLoader.java
private BagFactory bagFactory = BagFactory.getInstance();

private Object readField(JsonParser p,
                         ResourceFieldSchema field,
                         int fieldnum) throws IOException {
    ...
        DataBag bag = bagFactory.newDefaultBag();

        JsonToken innerTok;
```

```
            while ((innerTok = p.nextToken()) != JsonToken.END_ARRAY) {

                t = tupleFactory.newTuple(fs.length);
                for (int j = 0; j < fs.length; j++) {
                     t.set(j, readField(p, fs[j], j));
                }

                p.nextToken(); // read end of object
                bag.add(t);
            }
        ...
    }
```

可以使用 **iterator()**方法提供的迭代器来从 bag 中读取数据。该方法也实现了 Java 的 **Iterable** 方法，因此用户也是可以使用 **for(Tuple t : bag)**这种构造方式的。



警告

bag 有一个假定就是一旦从中读取数据，那么就不会再向其内写入数据。如何对数据进行划分以及如何重新读取数据都是基于这个假定的。因此一定用户调用了迭代器，那么就应该保证不再对同一个 bag 增加新的元素。

10.1.3 输入和输出模式

Pig 在执行脚本之前会进行类型检查。**EvalFunc** 类中提供了一个方法给用户，可以用于开启对 UDF 的输入和输出进行类型检查。

如果 UDF 返回的是简单数据类型，那么 Pig 会利用 Java 的反射机制来决定返回的数据类型。然而，因为 **exec** 方法接受的是一个 tuple，因此 Pig 没有办法判断用户期望 UDF 接受的输入是什么样的。当然，可以在运行时进行检查，但是如果能在编译时就进行检查，那么开发和测试过程可能会更加顺利。例如，我们可以像如下方式使用前面小节讲到的函数 **Pow** 的这个 UDF：

```
register 'acme.jar';
A = load 'input' as (x:chararray, y :int);
B = foreach A generate y, com.acme.math.Pow(x, 2);
dump B;
```

Pig 会生成并执行这个 job。但是该 job 中所有的任务都会失败，同时会获得报错信息：**ERROR 2078: Caught error from UDF: com.acme.math.Pow [Something bad happened!]**。在 Hadoop 中这样的运行时异常代价是非常大的，一方面如果集群繁忙那么调度需要消耗一些时间，另一方面每个任务都要尝试重新执行 3 次失败后才认为这个 job 失败了。下面我们对这个 UDF 做下修改让其在前面对输入信息进行检查。

声明 UDF 预期的输入模式的方法称为 **outputSchema**。这个方法这样命名是因为它返回的是所声明的 UDF 输出模式。如果用户的 UDF 并没有重载这个方法，那么 Pig 将会尝试使用用户对 **EvalFunc** 的实现的返回类型来确认它的返回类型，然后将脚本中表明的输入的模式传递给 UDF。这时如果 UDF 发现传入的模式和其预期的模式并不匹配，那么可以在这里抛出异常。如下是 **Pow** 这个方法的一种实现：

```
// java/com/acme/math/Pow.java
public Schema outputSchema(Schema input) {
    //检查字段个数是否为 2
    if (input.size() != 2) {
        throw new RuntimeException(
           "Expected (int, int), input does not have 2 fields");
    }
    try {
        //获取所有字段的类型并进行检验
        //如果字段类型错误，则判断传入的是什么数据类型并给出一个友好的错误
        //提示信息
        if (input.getField(0).type != DataType.INTEGER ||
              input.getField(1).type != DataType.INTEGER) {
           String msg = "Expected input (int, int), received schema (";
           msg += DataType.findTypeName(input.getField(0).type);
           msg += ", ";
           msg += DataType.findTypeName(input.getField(1).type);
           msg += ")";
           throw new RuntimeException(msg);
        }
    } catch (Exception e) {
        throw new RuntimeException(e);
    }

    //构造输出模式，也就是一个 long 类型的字段
    return new Schema(new FieldSchema(null, DataType.LONG));
}
```

将这个方法加入 **Pow** 中后，当我们再次调用前面那个错误的，将 charrarray 类型数据传递给 **Pow** 的脚本时，立即就会失败并报错

java.lang.RuntimeException: Expected input of (int, int), but received schema (chararray, int)。

Pig 中的这个 **Schema** 类是个复杂的类，下面我们将深入研究它所有的复杂性。下面所做的总结将足够用户参照并为 **outputSchema** 创建自己的模式。在它的核心部分，**Schema** 是由一系列的 **FieldSchema** 和别名以及 **FieldSchema** 间的映射关系组成的。每一个 **FieldSchema** 都包含了一个别名和一个类型信息。类型是以 Java 字节存储的，其大小是按照类 **org.apache.pig.data.DataType.Schema** 中定义的，这是一个递归结构。每个 **FieldSchema** 同样具有一个 **Schema** 成员。只有当数据类型是复杂数据类型时，这个成员才不是 null 值。对于 tuple 数据类型，这个成员定义了 tuple 的模式。对于 bag 数

据类型，这个成员定义了 bag 中 tuple 的模式。从 0.9 版本开始，如果是 map 数据类型，那么这个成员描述的是 map 中值的数据类型。在 0.9 版本之前，map 是没有模式的：

```
public class Schema implements Serializable, Cloneable {

    //模式中所有的字段信息列表
    private List<FieldSchema> mFields;

    //别名和字段模式的映射关系，这样可以通过别名来进行查找
    private Map<String, FieldSchema> mAliases;

    //一个 FileldSchema 对象代表这一个字段的模式信息
    public static class FieldSchema implements Serializable, Cloneable {

        //这个字段对应的别名
        public String alias;

        //数据类型，使用 org.apache.pig.data.DataType 中定义的值
        public byte type;

        //如果本身是个 tuple，那么它是可以具有模式的。否则，这个字段一定为 null
        public Schema schema;

        /**
         * 适应任意数据类型的构造器
         * @param a 如果已知，则为别名。否则，保持为 null。
         * @param t 数据类型，使用 org.apache.pig.data.DataType 中定义的值。
         */
        public FieldSchema(String a, byte t) { ... }
    }
    /**
     * 创建一个包含多个字段的模式
     * @param fields 描述字段的字段模式列表。
     */
    public Schema(List<FieldSchema> fields) { ... }

    /**
     * 创建只有一个字段的模式
     * @param fieldSchema 将要加入的字段模式
     */
    public Schema(FieldSchema fieldSchema) { ... }

    /**
     * 通过别名找到对应的 FiledSchema 对象。
     * @param alias  用于查找的别名
     * @return FieldSchema 对象，或者 null，如果 tuple 中没有这个别名
     */
    public FieldSchema getField(String alias) throws FrontendException {
        //这里省略了一些错误检查的代码
```

```
        return mAliases.get(alias);
    }
    /**
     * 根据字段编号，查找对应的 FileldSchema 对象。
     *
     * @param fieldNum 字段编号
     * @return 对应的 FieldSchema 对象。
     */
    public FieldSchema getField(int fieldNum) throws FrontendException {
        //这里省略了一些错误检查的代码
        return mFields.get(fieldNum);
    }
}
```

正如之前所提到的，当 UDF 返回的是简单数据类型时，Pig 会通过反射机制推断出返回数据类型。当 UDF 返回的是 bag 或者 tuple 时，如果需要 Pig 理解 bag 或者 tuple 中的内容，那么用户就需要实现 **outputSchema**。

10.1.4 错误处理和处理过程信息报告

我们前面使用的例子已经给出了如何处理错误的一些提示。当用户的 UDF 碰到一个错误的时候，对于如何对其进行处理有几个选择。最通常的做法是抛出一个警告然后返回 null。这会告知 Pig 这个 UDF 执行失败了同时其输出应该是不可知的。[1] 我们可以看一个这种情况的例子，这时 **Pow** 这个函数发现溢出了：

```
for (int i = 0; i < exponent; i++) {
     long preresult = result;
     result *= base;
     if (preresult > result) {
          //一旦溢出给出警告，但是并不抛出异常
          warn("Overflow!", PigWarning.TOO_LARGE_FOR_INT);
          //返回 null 将会告诉 Pig 失败了但是期望仍然继续执行
          return null;
     }
}
```

warn 是 **EvalFunc** 中的一个方法，接受用户提供的警告描述信息和警告代码两个参数。**org.apache.pig.PigWarning** 类中提供了具体的警告代码，其中包含了几个用户自定义的返回码，用于描述那些没有对应的描述信息的情况。Pig 会收集这些警告信息并在任务结束的时候报告给用户。

提示警告并且返回 null 是很方便的，因为这样仍然允许任务继续执行。当用户处理数十亿行的记录时，并不希望因为这数十亿行记录中的一条记录因为有个字段预期是 int 型而实际上是 chararray 类型从而导致整个任务失败。只要数据量足够的大，存在一些

[1] 我们应该还记得在 Pig 中 null 表示这个值是未知的，而并非表示这个值是 0 或者是未设置值。

脏数据并不会影响到大多数的计算。

对于不能够忍受的错误，用户的 UDF 可以抛出一个异常。Pig 如果获取到一个异常，那么它会认为用户希望它停掉所有的处理过程，因此会导致这个 task 任务失败。Hadoop 会重启这个 task。默认情况下，如果某个特定的 task 任务失败了 3 次，Hadoop 就不会再重启它了。而且，它会终止所有其他的 task 任务并声明这个 job 任务是失败的。

当用户确定需要一个异常的时候，应当给出一个日志信息，这样之后可以通过查看 task 任务日志信息来推测异常发生的真实原因。**EvalFunc** 类有一个成员 **log**，其是 **org.apache.commons.logging.Log** 的一个实例。Hadoop 会将所有的日志记录到 task 所在机器的日志文件中，通过 JobTracker 的 UI 页面可以查看到这些日志信息。7.1.6 节“MapReduce 任务运行状态信息”小节中有详细介绍。用户也可以将一些日志信息记录到这些日志文件中，这样可以方便调试。

除了错误报告外，一些 UDF 还需要报告执行进度信息。Hadoop 会监听它的 task 任务并确保它们正在执行。如果 Hadoop 有 5 分钟没有得到 task 任务的报告信息，它就会断定这个 task 任务死亡了或者进入死循环了。如果这个 task 任务还在运行中，它将会终止该 task 任务并进行资源清理，然后会在其他地方重启这个 task 任务。Pig 会定期向 Hadoop 报告进度。然而，如果用户有一个 UDF 需要消耗大量计算资源而且对其单次调用都需要多于 5 分钟的时间，这时也应当报告执行进度。为了解决这个问题，类 **EvalFunc** 中提供了一个成员 **reporter**。通过至少每 5 分钟调用一次 **report.progress()** 或 **report.progress(String msg)**（其中 **msg** 部分是根据需要自定义输出的内容），用户的 UDF 就可以避免被认为是超时的。

10.1.5 构造器和将数据从前端传送到后端

我们到目前为止的所有讨论都是假定用户在开发 UDF 的时候是知道它所需要的所有信息的，通常情况下并非如此。假设有一个 UDF，它需要从 HDFS 中读取一张查询表。用户需要声明这个文件的文件名才可以使用这个 UDF。因此需要为 UDF 定义一个非缺省构造方法。

缺省情况下，类 **EvalFuncs** 具有一个无参构造方法，但是用户可以提供一个包含一个或多个 **String** 类型的参数。Pig Latin 中在使用 **define** 语句定义 UDF 的时候会调用到这个替代构造方法，5.4.2 节“define 命令和 UDF”有详细的介绍。

作为一个例子，我们将看一个新的 UDF，**MetroResolver**。这个 UDF 以一个城市名作为输入然后返回这个城市所属于的大都市区。例如，如果输入是帕萨迪纳市，那么返回的结果是洛杉矶。这时需要一个不同的查询表来确定输入的城市属于哪个大都市区。存放在 HDFS 中的这个查看表文件的文件名可以作为构造参数传入。这个 UDF 类的声明、成员以及构造方法如下所示：

```
// java/com/acme/marketing/MetroResolver.java
```

```
/**
 * 一个用于查找指定城市对应于哪个大都市区的 UDF
 */
public class MetroResolver extends EvalFunc<String> {

    String lookupFile;
    HashMap<String, String> lookup = null;

    /*
     * @param file – 包含城市和对应大都市区对应关系的文件。
     * 这个文件需要位于 UDF 执行时所在的文件系统中。
     *
     */
    public MetroResolver(String file) {
        //只存储文件名。不会加载查询表，因为我们可能在前端或后端都使用到
        lookupFile = file;
    }
}
```

Pig Latin 脚本中可以按照如下方式调用这个 UDF：

```
    register 'acme.jar';
    define MetroResolver
com.acme.marketing.MetroResolver('/user/you/cities/us');
    A = load 'input' as (city:chararray);
    B = foreach A generate city, MetroResolver(city);
    dump B;
```

每当 Pig 构造 **MetroResolver** 的时候都会将文件名*/user/you/cities/us* 传递给它。到这里，我们的 UDF 还没有完成，因为我们还没有构造好查询表。事实上，我们明确地将其设置为 null。在构造器中构造它是没有意义的，因为无论前端还是后端都会调用构造器。确实有一些黑魔法可以允许 UDF 来断定是否在前端或者后端进行调用，但是我无法对此给出建议，因为不同的发行版中它们的处理逻辑可能不同。最好是在一个方式中构造好查询表，这样我们就能很清楚地知道，只能在后端对其进行调用。

EvalFunc 中并没有提供一个后端在进行处理前需要预先执行的初始化方法。用户可以通过设置一个标记来判断在任务中是否已经初始化了。**MetroResolver** 的 **exec** 函数通过判断 **lookup** 是否为 null 来进行推断：

```
    public String exec(Tuple input) throws IOException {
        if (lookup == null) {
            //我们还没有进行初始化，那么现在进行初始化

            lookup = new HashMap<String, String>();
            //获取 HDFS FileSystem 类的一个实例，这样我们就可以从 HDFS
            //中读取文件了。我们需要获取一份配置文件信息
            //从 UDFContext 对象中可以获取到配置文件
```

```
            FileSystem fs = FileSystem.get(UDFContext.getUDFContext().
getJobConf());
            DataInputStream in = fs.open(new Path(lookupFile));
            String line;
            while ((line = in.readLine()) != null) {
               String[] toks = new String[2];
               toks = line.split(":", 2);
               lookup.put(toks[0], toks[1]);
            }
            in.close();
        }
        return lookup.get((String)input.get(0));
    }
```

这个初始化代码片段打开了文件并读取了文件内容。为了能打开文件，其需要先连接到HDFS中。这个过程通过**FileSytstem.get**方法实现了。该方法也因此需要一个**JonConf**对象，这个对象中存储了Hadoop的所有的job信息。**JobConf**对象可以通过**UDFContext**获得，关于 **UDFContext** 本书后面会有更详细的介绍。需要注意的是，只有在后端才可以通过这个方式获得 **JonConf**，因为在前端并不存在任何的任务配置信息。

一但连接上了 HDFS，我们就可以像读取其他文件一样读取这个文件。文件内容会被解析成两个字段并保存在哈希表中。所有的对 **exec** 方法的后续调用都将是对这个哈希表进行查找。

加载分布式内存

MetroResolver 这个 UDF 会频繁地从 HDFS 中打开并读取查询表文件。然而，成百上千的 map 任务同一时间打开 HDFS 上的同一个文件时，会对 NameNode 已经存储这个文件数据块的 DataNode 造成巨大的压力。为了避免这种情况，Hadoop 提供的解决方案就是分布式内存，它允许用户将这些 HDFS 文件事先加载到任务将要执行的那些节点的本地磁盘上。附录 B 中的“分布式缓存”部分会有详细介绍。

下面我们重新写一个使用分布式内存的 **MetroResolver**。从 0.9 版本开始，**EvalFunc** 可以在前端调用 **getCacheFiles** 方法。它会返回使用到的分布式内存中存放的文件列表。每个文件的格式是 *client_file#task_file*，其中 *client_file* 是用户客户端上这个文件所在的路径，*task_file* 是 task 节点机器上为这个文件起的别名。*task_file* 是用户的 UDF 在后端的工作目录的相对路径。用户应当将所有的文件放到工作目录下并且不能使用绝对路径。*task_file* 将是 task 节点上的一个本地文件，而且应当是可以使用 Java 标准文件工具方法进行读取的。这时不应该使用 HDFS 的 **FileSystem** 来读取：

```
// java/com/acme/marketing/MetroResolverV2.java
/**
 *一个用于查找指定城市对应于哪个大都市区的 UDF,这次使用分布式内存。
 */
public class MetroResolverV2 extends EvalFunc<String> {
```

```
    String lookupFile;
    HashMap<String, String> lookup = null;

    /*
     * @param file -包含城市和对应大都市区对应关系的文件。
     * 这个文件需要位于 UDF 执行时所在的文件系统中。
     * run.
     */
    public MetroResolverV2(String file) {
        //只存储文件名，不会加载查询表，因为我们可能在前端或后端都使用到
        lookupFile = file;
    }

    public String exec(Tuple input) throws IOException {
        if (lookup == null) {
            //现在进行初始化
            lookup = new HashMap<String, String>();

            //以本地文件的方式打开文件
            FileReader fr = new FileReader("./mrv2_lookup");
            BufferedReader d = new BufferedReader(fr);
            String line;
            while ((line = d.readLine()) != null) {
                String[] toks = new String[2];
                toks = line.split(":", 2);
                lookup.put(toks[0], toks[1]);
            }
            fr.close();
        }
        return lookup.get((String)input.get(0));
    }

    public List<String> getCacheFiles() {
        List<String> list = new ArrayList<String>(1);
        //我们之前已经将这个文件名上传到 HDFS 中。后面跟上
        //task 节点上的文件名
        list.add(lookupFile + "#mrv2_lookup");
        return list;
    }
}
```

UDFContext

如果用户知道脚本正在写的数据是什么，那么构造器参数是一种传递信息给用户 UDF 的方式。用户可以通过参数传入的方式来使用这个方式（6.5.1 节“参数传入”有说明），这样可以在脚本执行过程中传入数据。但是有一些用户需要从前端传递到后端的信息在脚本执行过程中是无法获知的，或者无法通过控制台命令行获取。一个例子就是从

环境中获取属性信息然后传递它们。

为了允许 UDF 能够从前端传递数据到后端，从 Pig 0.8 版本开始，Pig 提供了一个单例类，**UDFContext**。用户可以通过调用 **getUDFContext** 方法来获取一份引用。我们已经见过可以使用 **UDFContext** 来获取一份 **JobConf** 对象实例。从 0.9 版本开始，还可以使用 **UDFContext** 获取客户端的 **System** 属性信息然后将其传递到后端。用户在 UDF 中可以通过调用 **getClientSystemProperties** 方法来获取这些信息。

UDFContext 还提供了允许用户显式地传递属性对象给指定用户 UDF 的机制。用户因此既可以传递一个属性对象给同一个类的所有 UDF，也可以传递一个特定的对象给所有用户 UDF 实例。通过调用 **getUDFProperties(this.getClass())**方法，所有的 UDF 实例可以使用同一个属性对象。这个方法将返回一个 **Properties** 对象，它是 **UDFConText** 所保存的属性对象的一个引用。这个属性对象的任何改动，**UDFContext** 都将会捕获到并传递到后端。用户可以在 **outputSchema** 方法中调用这个 **getUDFProperties (this.getClass())**方法，前端一定会调用 **outputSchema** 这个方法。当需要使用到这个数据时，只需要在 **exec** 方法中再次调用同一个方法就可以了。当使用 **exec** 方法中这个对象时，需要注意的是对于返回的 **Properties** 对象所做的任何更改都不会传递给后端的其他 UDF 实例，除非恰巧同一个 task 任务中具有同一个 UDF 的另一个实例。这种机制使得可以从前端传递信息到后端，但是不能在后端实例间进行传递。

有时用户需要传递不同的数据给同一个 UDF 的不同实例。这里说的不同的实例是指 Pig Latin 脚本中的不同调用，而不是在各种各样的 map 和 reduce 任务中不同的实例化。为了满足这个需求，**UDFContext** 提供了 **getUDFProperties(Class,String[])**方法。一般通过 UDF 的构造参数传入一组字符串，这可使得 UDF 的每个实例相互区别。如果用户的 UDF 并没有定义构造参数，或者所有的参数都是相同的值，那么用户可以增加一个额外的参数来用于区分 UDF 的不同实例。

如果 UDF 能具有自己特有的属性文件，那么当用户要将不同的属性文件传递给不同的 UDF 时，或者如果用户期望修改一些 UDF 自身特有的属性而无需修改 Pig 的属性配置文件时，这是很有用的。下面我们来重写下第 6 章介绍过的那个股票分析 UDF：

```
// java/com/acme/financial/AnalyzeStockV2.java
/**
 * 这个 UDF 读取一支股票的信息数据包然后
 * 产生一个在 1 和 100 之间的浮点数,
 * 1 表示抛售, 100 表示购入。
 */
public class AnalyzeStockV2 extends EvalFunc<Float> {

    Random r = new Random();
    Properties myProperties = null;

    @Override
```

```
    public Float exec(Tuple input) throws IOException {
        if (myProperties == null) {
            // Retrieve our class-specific properties from UDFContext.
            myProperties =
                UDFContext.getUDFContext().getUDFProperties (this.
getClass());
        }

        //确保输入非 null 并且大小是正确的
        if (input == null || input.size() != 1) return null;

        DataBag b = (DataBag)input.get(0);
        for (Tuple t : b) {

            //在这里做一些分析工作，可以使用 myProperties 中的一些属性信息
            //来设计如何分析
        }
        return r.nextFloat()* 100;
    }
    @Override
    public Schema outputSchema(Schema input) {
        try {
            //读取我们的属性文件
            Properties prop = new Properties();
            prop.load(new FileInputStream("/tmp/stock.properties"));
            //获取当前 UDF 的专属的属性对象
            UDFContext context = UDFContext.getUDFContext();
            Properties udfProp = context.getUDFProperties(this.
getClass());
            //将我们的属性信息添加到本地 UDF 属性对象中
            //无需将其重新传给 UDFContext 对象
            for (Map.Entry<Object, Object> e : prop.entrySet()) {
                udfProp.setProperty((String)e.getKey(), (String)e.
getValue());
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }

        return new Schema(new Schema.FieldSchema(null, DataType.FLOAT));
    }
}
```

10.1.6 重载 UDF

根据 UDF 所处理的数据类型，有时用户需要做多个不同的 UDF 实现。例如，MIN(long) 应该返回一个 long 类型值，而 MIN（int）应该返回一个 int 型值。为达到这个效果，EvalFunc 提供了一个 getArgToFuncMaping 方法。如果这个方法返回的是 null，Pig 就会使用当前的 UDF。为了能基于输入的数据类型返回一系列的 UDF，这个函数返回一

组 FuncSpecs。FuncSpec 是一个描述一个 UDF 的 Pig 类。这些 FuncSpecs 中的每一个都描述了一组预期的输入数据类型和 UDF，作为一个 Java 类用于处理它们。Pig 的类型检查器会使用这个列表来确定将哪个 Java 类放置在执行数据流里（详细信息后面会有讨论）。Pig 的内置 UDF MIN 函数的 getArgToFuncMaping 方法如下所示：

```
    // src/org/apache/pig/builtin/MIN.java
    public List<FuncSpec> getArgToFuncMapping()
    throws FrontendException {
        List<FuncSpec> funcList = new ArrayList<FuncSpec>();

        //这个列表的第一个元素是这个类本身，当输入是 bytearray 时其可以进行处理
        //因为这里返回类名本身和一个模式，因为本函数期望输入是一个
        //包含多个 tuple 字段的 bag，其中每个 tuple 里只有一个 bytearray 类型元素
        // generateNestedSchema 是一个帮助方法，用于生成这些包含只有一个字段的 tuple
组成
        //的 bag 的模式
        funcList.add(new FuncSpec(this.getClass().getName(),
            Schema.generateNestedSchema(DataType.BAG, DataType.
BYTEARRAY)));

        //如果输入的模式显示 bag 中包含的 tuple 所包含的字段是 double 类型
        //那么我们使用类 DoubleMin 而不是类 MIN 来实现求最小值操作
        funcList.add(new FuncSpec(DoubleMin.class.getName(),
            Schema.generateNestedSchema(DataType.BAG, DataType.DOUBLE)));

        //以及如下等等
        funcList.add(new FuncSpec(FloatMin.class.getName(),
            Schema.generateNestedSchema(DataType.BAG, DataType.FLOAT)));

        funcList.add(new FuncSpec(IntMin.class.getName(),
            Schema.generateNestedSchema(DataType.BAG, DataType.INTEGER)));

        funcList.add(new FuncSpec(LongMin.class.getName(),
            Schema.generateNestedSchema(DataType.BAG, DataType.LONG)));

        funcList.add(new FuncSpec(StringMin.class.getName(),
            Schema.generateNestedSchema(DataType.BAG, DataType.
CHARARRAY)));

        return funcList;
    }
```

Pig 的类型检查器会通过一系列的步骤来确定哪一个 FuncSpec 是最匹配的，然后应该将哪个 Java 类放置在任务的执行数据流中。在每一步，如果发现一个匹配的，就会使用这个相匹配的项。如果在某一步发现有多个匹配的，那么就会抛出一个错误并给出所有符合匹配的项。如果在整个列表中没有发现匹配的项，也会抛出错误。我们可以通过一个例子来演示。假设我们有另外一个版本的 Pow UDF，这个版本称之为 PowV2。

它的输入要么是两个 long 类型值要么是两个 double 类型值。它的 getArgToFuncMapping 方法如下所示：

```
// java/com/acme/math/PowV2.java
public List<FuncSpec> getArgToFuncMapping() throws FrontendException {
    List<FuncSpec> funcList = new ArrayList<FuncSpec>();

    Schema s = new Schema();
    s.add(new Schema.FieldSchema(null, DataType.DOUBLE));
    s.add(new Schema.FieldSchema(null, DataType.DOUBLE));
    funcList.add(new FuncSpec(this.getClass().getName(), s));
    s = new Schema();
    s.add(new Schema.FieldSchema(null, DataType.LONG));
    s.add(new Schema.FieldSchema(null, DataType.LONG));
    funcList.add(new FuncSpec(LongPow.class.getName(), s));
    return funcList;
}
```

类型检查器在查找最优 UDF 的过程中，第 1 步是查找完全匹配的项，也就是说 UDF 声明的输入类型和 Pig Latin 脚本中传入的实际的数据类型是完全一致的。Pow(2.0,3.1415) 传入的是两个 double 类型数据，因此 Pig Latin 将会选择 PowV2.Pow(2L,3L)传入的是两个 long 类型数据，因此会使用 LongPow。

第 2 步是查找传递到 UDF 中的 bytearray 类型数据并判断是否通过增加类型转换过程从而获得相匹配的项。例如，Pig 会重写 Pow(x,y)为 Pow((long)x,2L)，其中 x 是 bytearray 类型，然后最终使用 LongPow。当所有的参数都是 bytearray 时这个规则可能会令 Pig 困惑，因为 bytearray 是可以将类型转换成所有类型的。Pow（x,y），其中 x 和 y 都是 bytearray 类型，将会产生如下错误信息：

```
Multiple matching functions for com.acme.math.PowV2 with input schema:
    ({double,double}, {long,long}). Please use an explicit cast.
```

第 3 步是查看是否可以通过隐式类型转换使得与提供的模式相匹配，将会使用隐式类型转换后最接近的那个。数值型隐式类型转换次序是从 int 到 long 到 float 到 double，这里说的“最近”是指最少转换次数后和列表中相匹配。因此 Pow(2,2)将会使用 LongPow，而 Pow（2.0，2）将会使用 PowV2。

第 4 步是查找第 2 步和第 3 步的组合，也就是 bytearray 类型转换加上隐式类型转换。例如 Pow（x,3.14f），其中 x 是 bytearray 类型，将会把 3.14f 转换为 double 型并将 x 转换为 double 型，因为最终会使用 PowV2。

如果通过这些步骤最终还是没有发现一个合适的方法，那么会失败并提示其无法判断使用哪个方法。例如，Pow（'hello', 2）将会产生如下错误信息：

```
Could not infer the matching function for com.acme.math.PowV2 as multiple
or none of
```

```
them fit. Please use an explicit cast.
```

10.1.7 运算函数的内存问题

UDF 中有一些操作可能需要消耗比实际所能提供的更多的内存。例如，用户可能需要构建一个一直计算输入值的累积和的 UDF。这可能会返回大量的值，因为对于每个输入，都需要返回对于这个输入的中间累积值。

当传输的数据大小超过阈值或者只有少量的内存剩余时，Pig 的 bag 数据会自动将溢出的时间写入磁盘。溢出写入磁盘的代价是昂贵的并且是应当尽量避免的。但是如果用户必须在一个 bag 中存储大量的数据，Pig 也是可以处理的。

bag 是 Pig 中唯一一个知道如何处理溢出数据的数据类型。tuple 和 map 类型的数据都需要载入内存中。特别大以致于无法载入内存中的 bag 还是可以在一个 tuple 或者 map 中被引用的，这时会认为是因为这些 tuple 或者 map 太大而无法载入内存。

10.2 代数运算接口

其实在很多地方我已经反复提及只要能使用一定要使用 Hadoop 的组合器，这样可以有显著的性能提升。它可以降低 reduce task 中的数据倾斜问题，也会减少 map 和 reduce task 任务间网络传递的数据量。附录 B 中“Combiner 阶段”部分会对组合器在什么时候用到它进行详细的介绍。

当用户处理的是多个数据集合，通常情况下用户会先进行聚合处理然后再对一个或者小数据集数据进行计算。有两类函数可以很好地和组合器相结合：可分布计算的和可进行代数计算的。如果可以通过如下方式处理而不会改变结果，那么就是可分布计算的函数。其一，将输入数据集分割成多个子集，对这些子集分别使用函数进行处理，之后对处理后的结果再次使用函数进行处理；其二，对原始数据集调用函数进行处理。一个典型的例子就是 **SUM**。如果一个函数可以分解成初始化函数、中间处理函数和最终处理函数（通常和初始化函数不同）的时候就可以称之为可进行代数计算的，其中初始化函数用于输入集合的子集，中间处理函数会对初始化函数的输出结果进行处理，最终处理函数对所有的中间处理函数的输出结果进行处理。COUNT 就是一个可进行代数计算的函数，初始函数进行计数，中间函数和最终结果函数进行累加。可分布计算函数是可代数计算函数的一种特殊情况，其初始化函数、中间处理函数和最终处理函数和原始函数是相同的。

EvalFunc 可以通过实现 Java 接口 Algebraic 来声明其是可进行代数计算的函数。Algebraic 为用户 UDF 提供了 3 个方法来声明 Java 类的初始化函数、中间处理函数和最终处理函数。这些类必须继承 EvalFunc：

```
// src/org/apache/pig/Algebraic.java
```

```
public interface Algebraic{

    /**
     * 获取初始化函数。
     * @return 名为 f_init 的函数。 f_init 必须是一个求值函数。
     */
    public String getInitial();

    /**
     * 获取中间处理函数。
     * @return 名为 f_intermed 的函数。f_intermed 必须是一个求值函数。
     */
    public String getIntermed();

    /**
     * 获取最终处理函数。
     * @return 名为 f_final 的函数。 f_final 必须是一个求值函数。
     * 参数标准和实现这个接口的求值函数相一致。
     */
    public String getFinal();

}
```

这些方法都会返回一个 Java 类的名称，其本身应当是实现了 EvalFunc 函数的。Pig 将会使用这些 UDF 来重写用户脚本的执行过程。假设有如下 Pig Latin 脚本：

```
input = load 'data' as (x, y);
grpd = group input by x;
cnt = foreach grpd generate group, COUNT(input);
store cnt into 'result';
```

这段脚本的执行数据流初始过程如下所示：

```
Map
    load
Reduce
    foreach(group, COUNT),store
```

如果重写使用组合器，那么就会如下所示：

```
Map
    load
    foreach(group, COUNT.Initial)
Combine
    foreach(group, COUNT.Intermediate)
Reduce
```

```
foreach(group, COUNT.Final),store
```

作为一个例子，我们来梳理下 UDF COUNT 的实现。它的代数计算函数如下所示：

```
// src/org/apache/pig/builtin/COUNT.java
public String getInitial() {
    return Initial.class.getName();
}

public String getIntermed() {
    return Intermediate.class.getName();
}

public String getFinal() {
    return Final.class.getName();
}
```

COUNT 中所有这些被引用的类都是内部静态类。Initial 类的实现如下所示：

```
// src/org/apache/pig/building/COUNT.java
static public class Initial extends EvalFunc<Tuple> {

    public Tuple exec(Tuple input) throws IOException {
        //既然 Initial 类只会在 map 过程中调用到，因此其输入是一个
        //只包含一个 tuple 的 bag。如果 bag 是有值的，那么计数
        //值应该一直是 1
        DataBag bag = (DataBag)input.get(0);
        Iterator it = bag.iterator();
        if (it.hasNext()) {
            Tuple t = (Tuple)it.next();
            if (t != null && t.size() > 0 && t.get(0) != null)
                return mTupleFactory.newTuple(Long.valueOf(1));
        }
        return mTupleFactory.newTuple(Long.valueOf(0));
    }
}
```

尽管初始化函数应该只接受输入中的一条记录，这条记录要和原始函数的模式匹配。所以，对于 COUNT,它应当是一个 bag。因此，这个初始化函数判断了这个 bag 中是否存在非 null 的记录。如果存在则返回 1,否则返回 0。初始化函数的返回结果是一个 tuple。这个 tuple 的内容完全由 UDF 的编写者决定。本例中，初始化函数返回的是一个包含一个 long 类型字段的 tuple。

COUNT 的中间处理函数进行累加过程的代码如下：

```
// src/org/apache/pig/builtin/COUNT.java
static public class Intermediate extends EvalFunc<Tuple> {

    public Tuple exec(Tuple input) throws IOException {
        try {
```

```
            return mTupleFactory.newTuple(sum(input));
        } catch (ExecException ee) {
            ...
        }
    }
}

static protected Long sum(Tuple input)
throws ExecException, NumberFormatException {
    DataBag values = (DataBag)input.get(0);
    long sum = 0;
    for (Iterator<Tuple> it = values.iterator(); it.hasNext();) {
        Tuple t = it.next();
        sum += (Long)t.get(0);
    }
    return sum;
}
```

中间处理函数的输入是初始化函数返回的包含很多 tuple 的 bag。中间处理函数可以被调用零次、一次或者多次。因此，它需要输出和它期望的输入 tuple 匹配的输出 tuple。框架本身会控制将那些 tuple 放置在 bag 中。COUNT 的中间处理函数返回的是包含一个 long 型字段的 tuple。因为这里我们想对前面的计数值进行累加操作，因此这个函数实现了 SUM 而不是 COUNT。

reducer 中会调用到最终处理函数，这个函数只能被调用一次。它的输入是初始化函数和中间处理函数返回的包含 tuple 的 bag。它的返回类型应当和原始 UDF 的返回类型一致，本例子中返回值类型应当是 long。对于 COUNT，这个操作和中间处理过程的操作是相同的，因为它是对中间处理过程累加值的再次求和：

```
// src/org/apache/pig/builtin/COUNT.java
static public class Final extends EvalFunc<Long> {
    public Long exec(Tuple input) throws IOException {
        try {
            return sum(input);
        } catch (Exception ee) {
            ...
        }
    }
}
```

实现代数运算并非意味着所有的实现都总是会被用到。只有处于同一个 foreach 语句中的所有 UDF 都是可以代数计算的时候，Pig 才会选择代数接口实现方法。这是因为我们测试后发现对于不能显著进行组合的数据使用组合器会降低任务执行效率，同时在 Hadoop 中无法通过路由的方式将一些数据传递给组合器（对于可以进行代数计算的函数）而将一些数据直接传递给 reducer（对于不可以进行代数计算的函数）。这意味着用户的 UDF 必须实现 exec 方法，即使实际期望其总是以代数计算模式使用的。额外的方

式是在必要的情况下为用户的 UDF 实现可代数计算的模式。

10.3 累加器接口

一些计算并非可以通过代数计算的方式完成，特别是那些需要在开始前其输入数据必须是排好序的函数都是非可代数计算的。但是这些方法中很多仍然是不需要一次全看到所有输出数据的。只要能够确保可以最终获取到所有的数据，它们也是可以用于数据子集的。这意味着 Pig 不需要一次性将所有的记录都读入内存中。与之相反，它可以读取所有记录的一个子集然后传递给 UDF。为了处理这种情况，Pig 提供了累加器接口。不是调用一次 UDF 来一次性读取 bag 中所有的输入集合，Pig 将会调用多次，每次只处理一部分记录。当所有的数据都传入后，这时会寻求一个结果。最终，会给 UDF 一个重置状态的机会之后才会将记录传递给下一组：

```
// src/org/apache/pig/Accumulator.java
public interface Accumulator <T> {
    /**
     * 传递 tuple 类型数据给这个 UDF。
     * @param b 仅包含一个 bag 类型字段的 tuple。这个 bag 字段将
     *包含本次迭代中传递给 UDF 的一组 tuple。*/
    public void accumulate(Tuple b) throws IOException;

    /**
     * 当当前键对应的所有 tuple 都已经传递给迭代器后再调用这个方法。
     * @return UDF 对于这个键的返回值。
     */
    public T getValue();

    /**
     * 在调用 getValue()方法后调用这一方法为处理下一个键做准备。
     */
    public void cleanup();
}
```

作为一个例子，我们来看看 COUNT 对累加器的实现：

```
// src/org/apache/pig/builtin/COUNT.java
private long intermediateCount = 0L;

public void accumulate(Tuple b) throws IOException {
    try {
        DataBag bag = (DataBag)b.get(0);
        Iterator it = bag.iterator();
        while (it.hasNext()){
            Tuple t = (Tuple)it.next();
            if (t != null && t.size() > 0 && t.get(0) != null) {
                intermediateCount += 1;
```

```
            }
        }
    } catch (Exception e) {
        ...
    }
}
public void cleanup() {
    intermediateCount = 0L;
}

public Long getValue() {
    return intermediateCount;
}
```

默认情况下，Pig 一次累加处理 20,000 行的记录。用户可以通过设置属性 pig.accumulative.batchsize 来修改这个值，当然可以在控制台命令行设置或者在脚本中使用 set 命令来设置。

正如早前所提及的，可以使用累加器的主要的一类是那些需要排好序的输入的方法，例如session分析。通常情况这样的UDF期望同一个组内的记录是按照时间戳排好序的。例如，假设用户具有从 Web 服务器采集来的日志数据，其中包含用户 ID、时间戳和用户访问的 URL 链接，用户需要对这份数据进行 session 分析：

```
logs = load 'serverlogs' as (id:chararray, ts: long, url: chararray);
byuser = group logs by id;
results = foreach byuser {
            sorted = order logs by ts;
            generate group, SessionAnalysis(sorted);
};
```

Pig 可以将排序过程转换成 Hadoop 的排序过程，这是在 Hadoop 的 shuffle 阶段完成的。然而，Pig 仍可以一次从 Hadoop 中只读取部分数据然后直接将其传递给 SessionAnalysis。这个重要的优化允许累加器 UDF 处理排好序的数据。

只要可以，Pig 都会选择使用基于累加器的 UDF 中对可代数计算的实现。这是因为累加器可以帮助避免数据写磁盘，但是其并没有减少网络带宽消耗也没有均衡 reducer 间的负荷。如果一个 foreach 语句中的所有 UDF 都实现了累加器并且至少有一个没有实现代数运算，那么 Pig 将会使用累加器。如果至少有一个没有使用累加器，Pig 就不会使用累加器。这是因为对于没有实现累加器的 UDF，Pig 必须将全部的 bag 载入内存中，因此累加器中就不再有数据了。

10.4 使用 Python 写 UDF

Pig 和 Hadoop 是通过 Java 语言实现的，因此 Java 也是编写 UDF 的原生语言，但是最

好不要因此而不得不使用 Java。对于只有几行代码的简单的 UDF，编码、编译，然后打成 JAR 包，再进行部署，这整个环节是个非常沉重的过程。为了允许用户使用脚本语言编写 UDF，我们在 Pig 0.8 版本中加入了使用 Python 编写 UDF 的功能。我们通过使用几百行的代码将任一种的脚本语言编译成可在 JVM 执行的过程。我们希望在将来能够支持更多的脚本语言。

Python UDF 由一个独立的函数组成，这个函数取代了 Java 函数中的 **exec** 方法。通过注解的方式可以指出它们的模式。求值函数的更高级的功能——例如重载、构造器参数以及代数运算和累计器接口目前都还没有提供。

Python UDF 是通过 Jython 框架来执行的。使用这个框架的好处是 Python UDF 可以编译成 Java 字节码同时只会增加微小的性能损耗。缺点是 Jython 只有 Python 2.5 版本兼容，因此编写 UDF 的开发者不能使用 Python 3 的特性。

本书 5.4.1 节中的“注册 Python UDF”部分介绍了如何在 Pig Latin 脚本中注册和定义 Python UDF。本节我们将着重介绍如何编写 UDF。下面我们一起来看看之前章节所使用的名为 **production** 的 UDF：

```
# production.py
@outputSchema("production:float")
def production(slugging_pct, onbase_pct):
    return slugging_pct + onbase_pct
```

这段代码无需多说大家就可以明白。注解@outputSchema 告诉 Pig 这个 UDF 将返回一个名字为“production”的 float 类型字段。这个输出结果模式注解可以指定任意的 Pig 数据类型。指定 tuple 和 bag 数据类型的语法和在 **load** 语句中声明一个字段是 tuple 或者 bag 数据类型是相同的，详细信息可以查看 4.2 节“模式”。

有时模式是变化的而且是无法静态表述的，这种情况下可以通过提供一个模式函数来定义模式。下面我们来编写一个对一个数求平方而且返回值类型不变的 UDF：

```
# square.py
@outputSchemaFunction("schema")
def square(num):
    return num * num


@schemaFunction("schema")
def schema(input):
    # Return whatever type we were handed
    return input
```

模式函数的输入格式与@outputSchema:*colname:type* 中指定的格式是一样的，它的输出预期也是相同的格式。

如果 Python 函数中既没有提供@outputSchema 也没有提供@outputSchemaFunction，那

么将会返回一个 bytearray 类型的值。因为并没有任何的加载函数是以这个数据类型为输入的，所以 Pig 无法进行将其类型转换为其他任何类型，也因此除了 **store** 或者 **dump** 操作符外对于其他操作符都是没有意义的。

为了能在 Java 和 Pythonh 间直接交互数据，Pig 需要定义一组类型的对应关系。表 10-1 描述了 Pig 中的数据类型和 Python 中的数据类型的对应关系。

表 10-1　　Pig 和 Python 间的数据类型对应表

Pig 中的数据类型	Python 中的数据类型
int	number
long	number
float	number
double	number
chararray	string
bytearray	string
map	dictionary
tuple	tuple
bag	list of tuples

Pig 中所有值为 null 的值将会转换成 Python 中的 **None** 对象。同样的，Python 所返回的 **None** 对象，在 Pig 中会将其映射为对应类型的 null 值。

Pig 不会为 Python 编写的 UDF 做的一个处理是自动引用依赖的模板。如果用户的 Python 文件引入了其他模板，那么用户需要将这些模板打成一个 JAR 包并且需要在 Pig 脚本中进行注册。[1]

最后一件事就是对性能的影响问题。使用 Python 而不是 Java 对性能有什么影响呢？当然这取决于用户的脚本，还有做的什么样的计算，以及数据本身的一些特点。同时因为 Python UDF 尚不支持像代数模型这样的高级功能，所以很难对其进行优化。鉴于所有这些警告，测试表明 Jython 函数需要更高的实例化消耗。完成这个过程所需要的时间大概是 Java 函数需要的时间的 1.2 倍。因为实例化消耗过大的原因，少量输入数据（10,000 行左右）下的测试结果显示需要消耗两倍的 Java 执行时间。这些测试使用的都是几乎没什么处理过程的简单函数，因此这并非能很好地说明 Jython 和 Java 键的性能比较情况，但是可以说明 Pig 使用 Jython 时的性能消耗情况。

[1] 允许 Pig 为用户的 Python 代码选择依赖的模板，并且将其作为任务的一部分，这个功能的代码已经提交到代码库中了。但是在写这章内容的时候，还没有发布，现在在 Pig 0.10.0 版本中已经解决了这个问题。详细信息请查看 PIG-1824。

10.5 书写过滤器函数

过滤器函数是返回布尔值的求值函数。Pig 并不支持将布尔作为一个成熟的数据类型，因此过滤器函数不能出现在像 **foreach** 这样将输出结果传递给另一个操作符的语句中。然而，过滤器函数可以用在 **filter** 语句中。假设有一个求“最近距离的”函数，给定两个邮政编码，根据这两个邮政编码间的距离是否在一个指定范围内来决定返回 true 还是 false：

```
/**
 *本 UDF 用于判断两个邮政编码间的距离是否在一个指定范围内。
 */
public class CloseEnough extends FilterFunc {

    int distance;
    Random r = new Random();

    /*
     * @param miles – 两个邮件编码区域间相隔距离为多少里
     *还是可以认为是相近的。
     */
    public CloseEnough(String miles) {
        //UDF 只能接受字符串，这里将类型转换为整数
        distance = Integer.valueOf(miles);
    }

    public Boolean exec(Tuple input) throws IOException {
        //期望是两个字符串
        String zip1 = (String)input.get(0);
        String zip2 = (String)input.get(1);
        //做一些邮政编码表查找操作
        return r.nextBoolean();
    }
}
```

第 11 章

编写加载函数和存储函数

现在我们来考虑下 Pig 中一些更加复杂和最重要的部分：数据输入和输出。操作大数据集本身就是 I/O 敏感的。Hadoop 海量并行和处理数据过程的改进减轻了这种瓶颈但是并没有完全解决它。因此使用高效的方法来加载和存储数据是十分重要的。Pig 为文本数据和 HBase 提供了内置的加载和存储函数，但是很多用户会发现为了处理他们的数据格式和存储机制往往需要自己写加载和存储函数。

正如求值函数一样，加载和存储函数的设计目标是使简单的事情简单，困难的事情变得更有可能完成。同样地，加载和存储函数需要对 Hadoop 的 InputFormat 和 OutputFormat 做一个简单的封装。目的是一旦为数据定义输入格式和输出格式，那么额外的工作至少需要实现 Pig 中对 tuple 的创建和存储过程。求值函数也是通过同样的方式实现的，更复杂的功能，例如模式设计和前推下推等优化，为了避免和根接口相冲突需要通过实现不同的接口来完成。Pig 在从 0.6 版本到 0.7 版本对加载和存储函数进行了完全重写。本章只会包含 0.7 版本以及之后版本中的接口。

加载和存储函数的另一个设计目标是不能假定输入资源和输出方式都是 HDFS。本书中的所有例子，A=load 'foo'; 这样的语句显示的 **foo** 是一个文件，但是不必太在意这是否情况属实。**foo** 是个资源定位的标记，使用它才使加载函数讲得通。它可以是一个 HDFS 文件，或者一个 HBase 表，一个数据库 JDBC 连接字符串，或者一个网页服务 URL。因为最常见的情况是从 HDFS 读取资源，因此很多默认的和帮助类的函数是基于这种情况提供的。

本章我们将逐步完成一个对 HDFS 上 JSON 格式数据进行处理的加载函数和存储函数，分别是 JsonLoader 和 JsonStorage。这两个函数的源码位于示例源码 *udfs/java/com/acme/io* 目录下。它们使用的是 Jackson JSON 库，在 Pig 发行版中可以找到 Jackson JSON 库。然而，这个 Jackson JAR 在 Pig 中并没有分发到后台，因此当在脚本中使用这些 UDF 时，除了需要将样例 acme JAR 文件注册进去之外，还需要将这

个 Jackson JAR 注册进去：

```
register 'acme.jar';
register 'src/pig/trunk/build/ivy/lib/Pig/jackson-core-asl-1.6.0.jar';
```

这些 UDF 是作为例子提供的，但是它们并没有包含加载和存储函数的所有功能。对于这些例子中没有包含的那些部分，我们通过其他一些已经存在的加载和存储函数来看。

11.1 加载函数

Pig 的加载函数是基于 Hadoop 的 **InputFormat** 创建的，Hadoop 使用这个类读取数据。**InputFormat** 提供了两个目的：它决定输入数据在 map 任务间将是怎样划分的，同时它也提供了一个 **RecordReader** 用于产生键-值对作为 map 任务的输入。加载函数将接受这些键−值对然后返回一个 Pig **Tuple**。

加载函数的基类是 **LoadFunc**。这是一个抽象类，也就是允许它提供帮助函数和默认的实现。很多加载函数只需要实现 **LoadFunc** 即可。

加载函数的运作分为 Pig 前端操作和后端操作。在前端，Pig 进行任务计划和优化，后面会讲到加载函数通过哪些方式来实现这个操作。在后端，加载函数从 **RecordReader** 获取每一条记录，将其转换为 tuple，然后再将它传递给 Pig 的 map 任务。加载函数还需要具有在前后端传递数据的能力，这样才能够维持状态。

11.1.1 前端执行计划函数

对于所有的加载函数，作为前端执行计划的一部分，Pig 需要做 3 件事情：其一，它需要知道使用什么样的输入格式来读取数据；其二，它需要确定加载函数执行数据所在的具体位置；其三，它需要知道加载函数返回的 bytearray 类型的数据如何进行类型转换。

决定输入格式

Pig 需要知道使用哪种输入格式来读取输入数据。它会调用 **getInputFormat** 方法来获取输入格式的一个实例。它获取的是一个实例而不是类本身，因此加载函数是可以控制这个实例的：即所有的类参数、构造函数参数等。我们所举的加载函数例子中，这个方法是非常简单的。它使用 **TextInputFormat**，这是一个读取 HDFS 文本文件数据的输入格式。

```
// JsonLoader.java
public InputFormat getInputFormat() throws IOException {
    return new TextInputFormat();
}
```

确定加载路径

Pig 通过 **setLocation** 方法将用户提供的加载路径传递给加载函数。因此，如果 Pig Latin 脚本中的加载操作是 **A = load 'input';**，“input”就是加载路径了。这个方法前端和后端都会调用，而且可能需要多次调用。因此需要确认这个方法不会因为多次调用而导致出现问题。加载函数需要将这个路径信息传送给数据格式定义方法。例如，**JsonLoader** 是通过一个帮助方法将文件名传递给 **FileInputFormat** 的（TextInputFormat 的一个超类）：

```
// JsonLoader.java
public void setLocation(String location, Job job) throws IOException {
    FileInputFormat.setInputPaths(job, location);
}
```

Hadoop 的 **Job** 会一直将这个位置信息传递下去，这是因为这也是数据格式通常存放它们的配置信息的地方。

因为输入格式会将位置信息存放在 **Job** 对象中，所以无论前端还是后端都会调用 **setLocation** 方法，正如前面所列举的例子中所显示的那样。对于 MapReduce 任务，因为通常只有一个输入，所以这是行的通的。而对于 Pig 任务，可能同一个输入格式会用于多个不同的输入（例如在 join 或者 union 操作中），输入路径的一个实例会重写 **Job** 对象中的另一个实例。为了应对这个问题，Pig 将输入路径存放在一个特有的输入参数上，然后在后端重新调用 **setLocation** 方法，这样可以保证输入格式在读取之前可以先将自己合理地配置好。

对于存储在 HDFS 上的文件，用户提供的路径可能是相对路径而不是绝对路径。为了处理这种情况，Pig 需要基于加载时所处的工作目录将相对路径转换为绝对路径。对于如下的 Pig Latin 脚本：

```
cd /user/joe;
input1 = load 'input';
cd /user/fred;
input2 = load 'input';
```

这两个加载过程应该加载的是两个不同的文件。但是 Pig 不能保证总是能够将一个相对路径转换成对应的正确的绝对路径，因为它并不知道输入是什么。输入可以是一个 HDFS 路径，也可以是一个数据库表名等等，因此将这些留给加载函数。在调用 **setLocation** 方法之前，Pig 将输入的路径字符串传递给 **relativeToAbsolutePath** 方法，在这个方法里进行一些必要的转换。因为大多数情况下是从 HDFS 中读取数据的，所以 **LoadFunc** 类中默认的实现是针对 HDFS 的。如果加载过程根本不需要这个转换过程，那么也应该重载这个方法并且返回路径字符串本身。

获得类型转换函数

一些 Pig 函数，例如 **PigStorage** 和 **HBaseStorage**，默认情况下无需知道字段的类型信

息就可以加载数据，然后将数据不做任何改变存储在 **DataByteArray** 对象中。之后，当 Pig 需要将数据转换为其他类型时，它并不知道如何去转换，因为它并不知道数据在字节数组中是如何描述的。因此，它需要在加载函数中提供一个方法来将字节数组转换为合适的数据类型。

Pig 通过调用加载函数中的 **getLoadCaster** 方法来确定使用哪组类型转换函数。这方面要么返回 **null**，表示加载函数不需要做任何的类型转换；要么返回 **LoadCaster** 接口的一个实现，在这个实现中进行必要的类型转换。我们将在 11.1.4 小节“对字节数组进行类型转换”来了解 **LoadCaster** 包含的方法。

我们的样例加载函数中这个方法返回的是 **null**，因为它具有存储数据的模式，已经提供了包含类型信息的数据，因此不需要再进行类型转换。数据中的所有字节数组都是不需要进行类型转换的二进制数据。

11.1.2 从前端调用传递信息到后端调用

与求值函数一样，加载函数可以使用 **UDFContext** 从前端调用传递信息到后端调用。关于 **UDFContext** 的详细信息，请查看 10.1.5 小节“UDFContext”。在求值函数和在加载函数中使用 **UDFContext** 的一个显著的区别在于确定函数特有的实例签名。在求值函数中，构造函数参数被推荐为完成这件事的方式。对于加载函数来说，输入路径通常是差异化因素。然而，**LoadFunc** 并不保证在使用 **UDFContext** 的地方会在其他方法之前调用 **setLocation** 方法。为了解决这个问题，提供了 **setUDFContextSignature** 方法。该方法提供了一个每一个实例都不相同的签名，并且可以在调用 **getUDFProperties** 方法时使用。这个方法可以保证在 **LoadFunc** 中无论前端还是后端都会在调用其他方法之前先调用到 **setLocation** 方法。UDF 可以存储这个签名并且在获取属性对象时使用这个签名：

```
// JsonLoader.java
private String udfcSignature = null;

public void setUDFContextSignature(String signature) {
    udfcSignature = signature;
}
```

setLocation 是加载函数中唯一的一个可以在前端进行调用的方法，它也因此是存储 **UDFContext** 所需信息的最好的成员。用户需要检查并确认所写入的数据是存在的而且是非 null 的，这样才能避免当后端调用 setLocation 方法的时候把值重写了。

11.1.3 后端数据读取

在后端，加载函数根据数据格式读取键–值对然后生成 Pig **Tuple**。

为数据读取做准备

在读取任何数据之前，Pig 提供了 **prepareToRead** 方法为加载函数提供了初始化自己的一个途径。在所有的 map 任务中都会调用这个方法并且会传递一个 **RecordReader** 的拷贝，这是加载函数之后在读取输入中的记录时需要使用到的。**RecordReader** 是 **InputFormat** 在从输入划分中读取记录时需要使用到的一个类。加载函数通过 **getInputFormat** 方法获得输入格式，Pig 通过对这个输入格式调用 **getRecordReader** 方法获得对应的记录读取器然后再传送给 **prepareToRead** 方法。Pig 同时会传递一个 **PigSplit** 实例，其中包含了加载函数中这一实例将要读取的输入划分对应的 Hadoop **InputSplit**。如果需要知道特有的划分信息，那么从这里就可以获取到。

在我们的样例加载函数中，除了存储记录读取器之外，还读取了前端存储在 **UDFContext** 中的模式文件，这样它就可以知道如何解析输入文件了。需要注意它是如何使用通过 **setUDFContextSignature** 方法设置的签名来访问特定的属性对象的。最后，它创建了一个 **JsonFactory** 对象用于对每行都产生一个解析器：

```
   // JsonLoader.java
   public void prepareToRead(RecordReader reader, PigSplit split)
   throws IOException {
       this.reader = reader;

       //从 UDFContext 对象中获取模式信息字符串
       UDFContext udfc = UDFContext.getUDFContext();
       Properties p =
          udfc.getUDFProperties(this.getClass(), new String[]
{udfcSignature});
       String strSchema = p.getProperty("pig.jsonloader.schema");
       if (strSchema == null) {
          throw new IOException("Could not find schema in UDF context");
       }

       //从属性对象中存储的模式信息字符串中解析出模式
       ResourceSchema schema =
          new ResourceSchema(Utils.getSchemaFromString(strSchema));
       fields = schema.getFields();


       jsonFactory = new JsonFactory();
   }
```

读取记录

现在我们已经到达加载函数的核心部分了，从它的记录读取器中读取数据然后将结果 tuple 返回给 Pig。Pig 会调用 **getNext**，然后将结果 tuple 放到它的处理过程数据流中。会不断执行这个过程直到 **getNext** 返回 **null**，返回 null 意味着对于这个划分的输入已经全部读取完成了。

Pig 并不会复制从这个方法得到的结果，而是直接将结果传递到数据流中以避免复制的

开销。这意味着该方法不能重用对象，因此需要为它读取的每条记录都要创建一个 tuple 并载入内容。从另一方面来说，记录读取器从一条记录到另一条记录可能会选择重用它们的键和值的对象，大多数的标准实现确实是这么做的。因此，在写一个设法更高效地将记录读取器中键和值直接封装到 tuple 中以避免复制之前，需要确定记录读取器是如何控制它的数据的。

关于为 Pig 构造 tuple 的时候创建适当的 Java 对象的更多信息，请查看 10.1.2 小节中的“与 Pig 值进行交互”部分。

我们的样例加载函数中对于 **getNext** 的实现是从 Hadoop 记录中读取值（键被忽略），构造一个 **JsonParser** 来解析它，解析字段后返回结果 tuple。如果存在解析错误，它不会抛出异常，而是会返回一个 null 字段的 tuple，其中的数据是不能解析的。这可以防止因为存在一些脏数据而导致整个任务失败。但是会提供给用户警告信息，这样用户就可以知道哪些记录被忽略了：

```
   // JsonLoader.java
   public Tuple getNext() throws IOException {
      Text val = null;
      try {
          //从记录读取器中读取下一个键-值对。 如果
          //读取结束了，则返回 null
          if (!reader.nextKeyValue()) return null;

          //获取当前值。没有使用的键
          val = (Text)reader.getCurrentValue();
      } catch (InterruptedException ie) {
          throw new IOException(ie);
      }
      //为这行输入创建解析器。这可能并非是
      //最高效的方法
      ByteArrayInputStream bais = new ByteArrayInputStream(val.getBytes());
      JsonParser p = jsonFactory.createJsonParser(bais);

      //创建返回结果存放的 tuple。创建的时候指定
      //了确切的字段个数，Tuple 对象对这种情况已经进行了优化
      Tuple t = tupleFactory.newTuple(fields.length);

      //读取起始对象标记。在本文件通篇中如果解析的结果和我们
      //预期的不一致，将会返回一个包含 null 值字段的 tuple，而不是
      //抛出异常。这样可以防止因少数的错误行数据导致这个任务失败
      if (p.nextToken() != JsonToken.START_OBJECT) {
          log.warn("Bad record, could not find start of record " +
val.toString());
          return t;
      }

      //读取记录中的每一个字段
```

```
    for (int i = 0; i < fields.length; i++) {
        t.set(i, readField(p, fields[i], i));
    }

    if (p.nextToken() != JsonToken.END_OBJECT) {
        log.warn("Bad record, could not find end of record " +
            val.toString());
        return t;
    }
    p.close();
    return t;
}

private Object readField(JsonParser p,
                         ResourceFieldSchema field,
                         int fieldnum) throws IOException {
    //读取下一个标记
    JsonToken tok = p.nextToken();
    if (tok == null) {
         log.warn("Early termination of record, expected " + fields.length
             + " fields bug found " + fieldnum);
         return null;
    }

    //检验这个值是否为 null
    if (tok == JsonToken.VALUE_NULL) return null;

    //按照期望的数据类型进行读取
    switch (field.getType()) {
    case DataType.INTEGER:
        //读取字段名称
        p.nextToken();
        return p.getValueAsInt();

    case DataType.LONG:
        p.nextToken();
        return p.getValueAsLong();

    case DataType.FLOAT:
        p.nextToken();
        return (float)p.getValueAsDouble();

    case DataType.DOUBLE:
        p.nextToken();
        return p.getValueAsDouble();

    case DataType.BYTEARRAY:
        p.nextToken();
        byte[] b = p.getBinaryValue();
```

```
        //使用 DBA 构造器来复制字节便于内存控制
        return new DataByteArray(b, 0, b.length);

    case DataType.CHARARRAY:
        p.nextToken();
        return p.getText();

    case DataType.MAP:
        //应该是 map 对象的起始处
        if (p.nextToken() != JsonToken.START_OBJECT) {
            log.warn("Bad map field, could not find start of object, field "
                + fieldnum);
            return null;
        }
        Map<String, String> m = new HashMap<String, String>();
        while (p.nextToken() != JsonToken.END_OBJECT) {
            String k = p.getCurrentName();
            String v = p.getText();
            m.put(k, v);
       }
        return m;

    case DataType.TUPLE:
        if (p.nextToken() != JsonToken.START_OBJECT) {
            log.warn("Bad tuple field, could not find start of object, "
                + "field " + fieldnum);
            return null;
        }

        ResourceSchema s = field.getSchema();
        ResourceFieldSchema[] fs = s.getFields();
        Tuple t = tupleFactory.newTuple(fs.length);

        for (int j = 0; j < fs.length; j++) {
            t.set(j, readField(p, fs[j], j));
        }

        if (p.nextToken() != JsonToken.END_OBJECT) {
            log.warn("Bad tuple field, could not find end of object, "
                + "field " + fieldnum);
            return null;
        }
        return t;

    case DataType.BAG:
        if (p.nextToken() != JsonToken.START_ARRAY) {
            log.warn("Bad bag field, could not find start of array, "
                + "field " + fieldnum);
            return null;
```

```
            }

            s = field.getSchema();
            fs = s.getFields();
            //向下获取到 tuple 对应的模式的下一层
            s = fs[0].getSchema();
            fs = s.getFields();

            DataBag bag = bagFactory.newDefaultBag();

            JsonToken innerTok;
            while ((innerTok = p.nextToken()) != JsonToken.END_ARRAY) {
                if (innerTok != JsonToken.START_OBJECT) {
                    log.warn("Bad bag tuple field, could not find start of "
                        + "object, field " + fieldnum);
                    return null;
                }
                t = tupleFactory.newTuple(fs.length);
                for (int j = 0; j < fs.length; j++) {
                    t.set(j, readField(p, fs[j], j));
                }

                if (p.nextToken() != JsonToken.END_OBJECT) {
                    log.warn("Bad bag tuple field, could not find end of "
                        + "object, field " + fieldnum);
                    return null;
                }
                bag.add(t);
            }
            return bag;

        default:
            throw new IOException("Unknown type in input schema: " +
                field.getType());
        }
    }
```

11.1.4 可扩展的加载函数接口

通过实现其他更多的接口，加载函数可以提供更复杂的功能。（实现哪些接口是可以根据需要来定的。）

加载元数据

很多数据存储机制可以将模式和数据存储在一起。Pig 并没有存储模式的能力，但是如果使用的存储可以存放模式，那将对用户是非常有帮助的。这样可以让使用者在写 Pig Latin 脚本时使用加载函数时不需要指定字段名称和字段类型。这是用户友好的而且出错较少，同时可以避免因为数据的模式发生改变而去修改脚本。

有一些数据存储也会对数据进行分区。如果 Pig 清楚这个分区，对于一个特定的脚本它可以只加载需要的分区数据。它除了要获得用户提供的路径字符串也要获得 Hadoop **Job** 这个对象，以防需要使用这个对象中的信息来打开模式。预期它将返回一个 **ResourceSchema**，其代表了将要返回的数据。**ResourceSchema** 与求值函数中使用到的 **Schema** 很相似。（详细信息请查看 10.1.3 小节“输入和输出模式”）然后，还有一个非常重要的区别，在 **ResouceFieldSchema** 中，和一个 bag 相联系的模式对象总是有一个 tuple 类型的字段。bag 中 tuple 的模式是通过这个 tuple 的 **ResourceFieldSchema** 来描述的。

我们的样例加载和存储函数将模式信息存储在 HDFS 中一个叫做 *_schema* 的支持文件[1] 中。我们对 **getSchema** 的实现会读取这个文件，然后将模式序列化到 **UDFContext** 中，这样从后端也是可以获得的：

```
   // JsonLoader.java
   public ResourceSchema getSchema(String location, Job job)
   throws IOException {
      //打开模式文件并读取模式
      //获取一个 HDFS 操作句柄
      FileSystem fs = FileSystem.get(job.getConfiguration());
      DataInputStream in = fs.open(new Path(location + "/_schema"));
      String line = in.readLine();
      in.close();

      //解析模式
      ResourceSchema s = new ResourceSchema(Utils.getSchemaFrom
String(line));
      if (s == null) {
          throw new IOException("Unable to parse schema found in file " +
              location + "/_schema");
      }

      //现在模式确定了，需要将其保存在
      // UDFContext 属性对象中，这样之后在后端就可以对之进行调用
      UDFContext udfc = UDFContext.getUDFContext();
      Properties p =
         udfc.getUDFProperties(this.getClass(), new String[]
{udfcSignature});
      p.setProperty("pig.jsonloader.schema", line);

      return s;
   }
```

一旦用户的加载函数实现了 **getSchema** 方法，那么使用加载函数的 **load** 语句就不再需要定义它们的模式以方便在脚本中使用相应的字段名称。例如，如果我们有这样的数

[1] 处在同一个目录下但本身并不是划分部分的文件。支持文件是以一个下划线字符开始的。MapReduce 的 FileInputFormat 在为一个任务读取输入文件的时候知道需要忽略这些文件。

据，其模式是 **user:chararray,age:int,gpa:double**，那么如下的 Pig Latin 脚本是可以编译通过并可以执行的：

```
register 'acme.jar';
register 'src/pig/trunk/build/ivy/lib/Pig/jackson-core-asl-1.6.0.jar';

A = load 'input' using com.acme.io.JsonLoader();
B = foreach A generate user;
dump B;
```

LoadMetadata 同时包含了一个 **getStatistics** 方法。Pig 还没有在任务计划中使用统计，这个方法是为未来使用的。

使用分区

有一些数据存储会对数据进行分区，这允许用户对于给定的任务只读取与之相关的分区。**LoadMetadata** 接口同样提供了处理数据中分区的方法。为了能让 Pig 只加载相关的分区，需要让它知道数据是如何分区的。Pig 通过 **getPartitionKeys** 方法来确定这项内容。如果 **getPartitionKeys** 方法返回的是 **null** 或者使用的加载函数没有实现 **LoadMetadata** 接口，Pig 将假定需要读取整个输入。

Pig 期望 **getPartitionKeys** 方法返回一个字符串数组，其中每个字符串都是一个字段名。这些字段是用于对数据进行分区的键。Pig 会扫描紧跟着 **load** 语句后面的那个语句来查找 **filter** 语句，并判断过滤语句中是否有一个或多个这些字段。如果发现这样的语句，那么此信息将传递给 **setPartitionFilter** 方法。如果 **filter** 语句包含分区和非分区字段并且是可以进行划分的 [1]，Pig 将会对之进行分割，然后只要与分区键相关的表达式就传递给 **setPartitionFilter** 方法。举个例子，假设有一张 HCatalog[2] 表 *web_server_logs* 有两个分区字段，**date** 和 **colo**:

```
logs = load 'web_server_logs' using HCatLoader();
cleaned = filter logs by date = '20110614' and NotABot(user_id);
...
```

Pig 将会调用 **getPartitionKeys** 方法，同时 **HCatLoader** 将返回两个键的名称，**date** 和 **colo**。Pig 将会发现 **date** 字段是包含在 **filter** 语句中的并且按照下面例子中的方式重写这个过滤器，通过 **setPartitionFilter** 方法将 **date = '20110614'**下推到 **HCatLoader** 加载函数中：

```
logs = load 'web_server_logs' using HCatLoader();
```

[1] 这意味着过滤器可以分割成两个过滤器——一个包含分区字段另一个包含非分区字段，并且产生相同的结果。如果过滤器中表达式的键是通过 and 连接的就可以进行这种划分，而如果是通过 or 连接的则不可以。

[2] HCatalog 是一个基于 Hadoop 的表管理服务，它囊括了 Pig 的加载和存储函数。关于 HCatalog 的更多信息请查 12.4 节“Hadoop 中的元数据”。

```
cleaned = filter logs by NotABot(user_id);
...
```

现在轮到 HCatalog 加载器来保证只返回表 *web_server_logs* 中 **date** 是 **20110614** 的数据。

有一个例外情况是如果分区字段是用于求值函数或者过滤函数时不会进行下推。Pig 假设加载器是不知道如何调用 UDF 的，因此 Pig 不会对这些表达式进行下推优化。

我们的样例加载器是处理文件数据的，因此它没有实现 **getPartitionKeys** 方法和 **setPartitionFilter** 方法。如果想查看一个实现了这些方法的例子，请通过如下链接查看 HCatalog 代码：*http://svn.apache.org/viewvc/incubator/hcatalog/trunk/src/java/org/apache/hcatalog/ pig/HCatLoader.java?view=markup*.

对字节数组进行类型转换

如果需要控制加载函数所加载的二进制数据如何转换为其他数据类型，那么需要实现 **LoadCaster** 接口。因为该接口包含了大量的方法，所以经常创建一个单独的类来实现它。这样也可以允许多个加载函数共享使用对 **LoadCaster** 的实现，因为 Java 本身不支持多重继承。

LoadCaster 接口包含了一组方法：**bytesToInteger**、**bytesToLong** 等。调用这些方法可以将一个 bytearray 类型转换为其他合适的数据类型。从 0.9 版本开始，提供了两个 **bytesToMap** 方法。我们应当实现那个以 **ResouceFieldSchema** 为输入参数的方法，另外一个方法是用于向下兼容的。**bytesToBag**、**bytesToTuple** 和 **bytesToMap** 方法接收一个描述了将要进行转换的字段信息的 **ResouceFieldSchema** 对象。对该对象调用 **getSchema** 方法将会返回一个描述这个 bag、tuple 或者 map 的模式，当然如果存在的话。如果 Pig 不知道这个对象的预期结构，那么 **getSchema** 方法将返回 null。需要牢记的是，bag 的模式只有一个 tuple 类型字段，然后依次需要一个描述这个 tuple 内容的模式。

默认情况下提供了一个加载类型转换器，**udf8StorageConverter**。它可以将 UTF8 格式编码的文本转换为 Pig 数据类型。对于数量转换是以一种直截了当的方式完成的。对于 map 类型使用方括号（[]）包围，其中的键值间使用哈希符（#）连接，键值对间使用逗号（,）分割。对于 tuple 类型使用圆括号（()）包围，其中字段通过逗号（,）进行分割。对于 bag 类型使用花括号（{}）包围，其中的 tuple 按照逗号（,）进行分割。不能够对这些特殊字符进行转义。

推测执行

一个 Pig Latin 脚本经常只需要读取输入中的一些字段。一些类型的存储格式是按照列存储数据而不是按照行存储数据（例如 Hive 中的 RCFile 存储格式）。对于这些类型的存储格式，只加载脚本中指定的那些列将会有更显著的执行效率。甚至对于面向行的存储格式，也是可以避免对一些没有使用到的列进行反序列化的。

作为优化过程的一部分，Pig 会分析 Pig Latin 脚本，然后确定脚本中每一步需要输入数据的哪些字段。通过这个信息可以强有力地移除那些不再使用的字段。如果加载函数实现了 **LoadPushDown** 接口，Pig 就可以更进一步将这个信息提供给加载函数。

一旦 Pig 知道了哪些字段是需要的，它就会将这些字段信息封装到一个 **RequiredFieldList** 对象中，然后将其传递给 **pushProjection**。加载函数的返回信息中会表明它是否能满足需求。它通过一个 **RequiredFieldResponse** 对象做回复，这个对象是个精心封装的布尔值。如果布尔值是 true，Pig 将会假定 **getNext** 方法只会返回需要的字段。如果布尔值是 false，Pig 就会假定 **getNext** 方法需要返回所有的字段，然后它再根据自身的推测去除掉不需要的字段。

描述需要那些字段的 **RequiredField** 类稍微有些复杂。除了允许用户指定某个字段是否是需要的以外，它还能够用于指定那个字段的哪个子字段是否是需要的。例如，对于 map 类型数据，可以按照需求列举指定键的数据。对于 tuple 和 bag 类型数据，可以按照需要列举指定的字段的数据。

实现了 **LoadPushDown** 接口的加载函数不应该修改通过 **getSchema** 方法返回的模式对象，应该总是完整输入对应的模式。Pig 会控制读取完整的模式，但是 **getNext** 方法返回的是指定的部分字段的数据。

我们的样例加载函数没有实现 **LoadPushDown** 接口。如果想查看一个实现了这个接口的加载函数的例子，请通过如下连接查看 **HCatLoader**: *http://svn.apache.org/viewvc/incubator/hcatalog/trunk/src/java/org/apache/hcatalog/pig/HCatLoader.java?view=markup*.

11.2 存储函数

Pig 的存储函数在许多方面与加载函数很相像，它是基于 Hadoop 的 **OutputFormat** 构建的。存储函数接受的是 Pig 的 **Tuple**，然后根据输出格式创建键–值对后写入存储中。

StoreFunc 是一个抽象类，它允许对一些方法提供默认情况下的实现。然而，一些函数是可以同时具有加载和存储功能的，例如 **PigStorage**。因为 Java 并不支持多重继承，因此提供了 **StoreFuncInterface** 接口。这些双重加载/存储函数可以实现这个接口而无需继承 **StoreFunc** 类。

存储函数操作在 Pig 中分为前端和后端两个部分。在前端 Pig 完成执行计划和优化过程。存储函数在这个时候可以验证是否提供了有效的模式并且设置存储路径。在后端，存储函数接受 Pig 提供的 tuple 类型数据，将其转换成键–值对，然后将其传递给 Hadoop 的一个 **RecordWriter** 对象。存储函数可以从前端调用，通过 **UDFContext** 传递信息到后端调用。

11.2.1 存储函数前端执行计划

存储函数在前端需要完成 3 项工作：

- 指定存储数据所使用的 OutputFomat；
- 检查将要存储的数据的模式；
- 记录数据将要存储到的位置。

确定输出格式

Pig 通过调用 **getOutputFormat** 方法获得存储函数存储数据时使用的输出格式的实例。这个方法返回的是一个实例而不是类名或者类本身。这也就允许存储函数控制这个类是如何实例化的。样例存储函数 **JsonStorage** 使用的是 **TextOutputFormat**，这是一个存储文本数据到 HDFS 中的输出格式。对照这个输出格式按 **TextOutputFormat** 所期望的，我们需要提供一个 **LongWritable** 类型的键和一个 **Text** 类型的值：

```
// JsonStorage.java
public OutputFormat getOutputFormat() throws IOException {
   return new TextOutputFormat<LongWritable, Text>();
}
```

设置输出路径

Pig 通过调用 **setStoreLocation** 方法将用户提供的路径字符串传递给存储函数。例如对于 Pig Latin 脚本 **store Z into 'output';**，“output”就是路径字符串。这个方法，前端和后端都是可以调用的，并且可以被多次调用，因此，要确保不能因为多次调用产生副作用而出现问题。存储函数需要将路径信息传递给输出格式。我们的样例存储函数使用的是 **FileOutputFormat** 的工具函数 **setOutputPath** 来完成这项工作：

```
// JsonStorage.java
public void setStoreLocation(String location, Job job) throws IOException
{
    FileOutputFormat.setOutputPath(job, new Path(location));
}
```

Hadoop 的 **Job** 对象也会传递给这个函数。大多数的输出格式是将路径信息存储在这个对象中的。

正如我们的样例存储函数所展示的，因为存储格式通常将路径信息存储在 job 对象中，Pig 在前端还是后端都可以调用 **setStoreLocation** 方法。这对于 MapReduce 任务来说是起作用的，因为它可以保证只使用一个输出格式。但是由于有 **split** 操作符，Pig 在一个任务中对于同一个存储函数可以有多个实例。如果期望一个存储函数有多个实例，只需要调用 **FileOutputFormat.setOutputPath**，无论哪个实例最后调用了它，那么将会覆盖其他的实例。Pig 通过保存特殊的输出格式信息，然后在后端重新调用

setStoreLocation 方法来避免这个问题，因为这样 Pig 可以合理地配置好输出格式。

对于 HDFS 文件，用户可能提供的是一个相对路径。Pig 需要根据当前执行存储操作所在的工作目录将其转换为绝对路径。为了完成这项工作，Pig 会在调用 **setStoreLocation** 方法之前对用户提供的路径字符串调用 **relToAbsPathForStoreLocation** 方法。这个方法在相对路径和绝对路径之间进行转换。对于向 HDFS 中写入数据的存储函数，**StoreFunc** 类中的默认实现会处理这个转换。如果所写的存储函数不是使用文件路径的（如 HBase），那么就需要重写该方法然后输入字符串本身。

检查模式

作为前端执行计划的一部分，Pig 对于存储函数提供了一个可以检查将要存储的数据的模式的机会。如果需要将数据存储到一个需要输出数据具有特定模式的系统（如 RDBMS），或者不能够存储某些数据类型，这就是一个需要进行那些检查的地方。需要注意的是，这个方法返回的是 void 而不是一个布尔值。因此如果用户察觉到这个模式有些问题，那么必须要抛出一个 **IOException** 异常。

我们的样例存储函数对于存储的模式没有任何限制。然而，它使用这个函数作为将模式序列化到 **UDFContext** 对象的一个方式，这样才能在写数据的时候在后端使用到：

```
// JsonStorage.java

public void checkSchema(ResourceSchema s) throws IOException {
    UDFContext udfc = UDFContext.getUDFContext();
    Properties p =
        udfc.getUDFProperties(this.getClass(), new
String[]{udfcSignature});
    p.setProperty("pig.jsonstorage.schema", s.toString());
}
```

11.2.2 存储函数和 UDFContext

存储函数使用 **UDFContext** 的方式与加载函数完全一致，但是有一个例外：存储函数的签名是通过 **setStoreFuncUDFContextSignature** 方法传递给存储函数的。关于加载函数如何使用 **UDFContext** 的讨论请查看 11.1.2 小节“从前端调用传递信息到后端调用”。我们的样例存储函数将这个签名存储为一个成员变量便于后期使用：

```
// JsonStorage.java
public void setStoreFuncUDFContextSignature(String signature) {
    udfcSignature = signature;
}
```

11.2.3 写数据

在后端处理过程中，存储函数先进行初始话，然后接受 Pig 中的 tuple 数据并将其转化

为键–值对写入存储中。

为写入做准备

在写入任何数据前 Pig 会在每个 map 或者 reduce 任务中调用存储函数中的 **prepareToWrite** 方法。写数据时这个调用会传递一个 **RecordWriter** 实例。**OutputFormat** 使用 **RecordWriter** 类来写入每条数据。基于通过存储函数的 **getOutputFormat** 方法返回的存储格式，然后调用 **getRecordWriter** 方法传递给存储函数使用哪种记录写入方式，Pig 就知道了使用何种记录写入方式的。存储函数需要保留这一引用才能在 **putNext** 方法中使用。

样例存储函数 **JsonStorage** 也会使用 prpareTowrite 方法来从 **UDFContext** 中读取模式信息。在存储数据的时候会用到这个模式，在 **putNext** 方法中会创建一个 **JsonFactory** 以便使用：

```
   // JsonStorage.java
   public void prepareToWrite(RecordWriter writer) throws IOException {
       //将记录写入器的引用进行存储，这样在之后写 tuple 的时候
       //进行使用
       this.writer = writer;

       //从 UDFContext 对象中获取模式字符串
       UDFContext udfc = UDFContext.getUDFContext();
       Properties p =
          udfc.getUDFProperties(this.getClass(), new String[]
{udfcSignature});
       String strSchema = p.getProperty("pig.jsonstorage.schema");
       if (strSchema == null) {
          throw new IOException("Could not find schema in UDF context");
       }

       //从属性对象中存储的字符串中解析出模式
       ResourceSchema schema =
          new ResourceSchema(Utils.getSchemaFromString(strSchema));
       fields = schema.getFields();

       //创建一个 Json 工厂实例
       jsonFactory = new JsonFactory();
       jsonFactory.configure(
          JsonGenerator.Feature.WRITE_NUMBERS_AS_STRINGS, false);
   }
```

写入记录

putNext 是存储函数类的核心方法。Pig 会对所有需要存储的 tuple 调用这个方法。存储函数需要能够接收这些 tuple，然后把它们转换成输出格式要求的键–值对数据。关于数据将存储到什么样的 Java 对象中以及如何从对象中提取出数据的详细信息，请参考

10.1.2 小节中的“与 Pig 值进行交互”部分内容。

JsonStorage 会对 tuple 内容按照 JSON 格式编码并将结果字符串写入 **TextOutputFormat** 的值字段中。键的字段保留为 null：

```
// JsonStorage.java
public void putNext(Tuple t) throws IOException {
    //创建一个 ByteArrayOutputStream 用于将 JSON 写入
    ByteArrayOutputStream baos = new ByteArrayOutputStream(BUF_SIZE);
    //创建生成器
    JsonGenerator json =
        jsonFactory.createJsonGenerator(baos, JsonEncoding.UTF8);

    //写入顶层 tuple 对象的开始部分
    json.writeStartObject();
    for (int i = 0; i < fields.length; i++) {
        writeField(json, fields[i], t.get(i));
    }
    json.writeEndObject();
    json.close();

    //提交 null 键和我们的字符串给 Hadoop
    try {
        writer.write(null, new Text(baos.toByteArray()));
    } catch (InterruptedException ie) {
        throw new IOException(ie);
    }
}

private void writeField(JsonGenerator json,
                        ResourceFieldSchema field,
                        Object d) throws IOException {

    //如果这个字段有缺失或者值为 null，则写入 null
    if (d == null) {
        json.writeNullField(field.getName());
        return;
    }

    //基于这个字段的数据类型，将其写出
    switch (field.getType()) {
    case DataType.INTEGER:
        json.writeNumberField(field.getName(), (Integer)d);
        return;

    case DataType.LONG:
        json.writeNumberField(field.getName(), (Long)d);
        return;
```

```
        case DataType.FLOAT:
            json.writeNumberField(field.getName(), (Float)d);
            return;

        case DataType.DOUBLE:
            json.writeNumberField(field.getName(), (Double)d);
            return;

        case DataType.BYTEARRAY:
            json.writeBinaryField(field.getName(), ((DataByteArray)d).
get());
            return;

        case DataType.CHARARRAY:
            json.writeStringField(field.getName(), (String)d);
            return;

        case DataType.MAP:
            json.writeFieldName(field.getName());
            json.writeStartObject();
            for (Map.Entry<String, Object> e : ((Map<String, Object>)d).
entrySet()) {
                json.writeStringField(e.getKey(), e.getValue().toString());
            }
            json.writeEndObject();
            return;

        case DataType.TUPLE:
            json.writeFieldName(field.getName());
            json.writeStartObject();

            ResourceSchema s = field.getSchema();
            if (s == null) {
                throw new IOException("Schemas must be fully specified to use "
                    + "this storage function. No schema found for field " +
                    field.getName());
            }
            ResourceFieldSchema[] fs = s.getFields();

            for (int j = 0; j < fs.length; j++) {
                writeField(json, fs[j], ((Tuple)d).get(j));
            }
            json.writeEndObject();
            return;

        case DataType.BAG:
            json.writeFieldName(field.getName());
            json.writeStartArray();
            s = field.getSchema();
```

```
            if (s == null) {
                throw new IOException("Schemas must be fully specified to use "
                   + "this storage function. No schema found for field " +
                   field.getName());
            }
            fs = s.getFields();
            if (fs.length != 1 || fs[0].getType() != DataType.TUPLE) {
               throw new IOException("Found a bag without a tuple "
                   + "inside!");
            }
            //向下获取到这个 tuple 模式的下一层
            s = fs[0].getSchema();
            if (s == null) {
               throw new IOException("Schemas must be fully specified to use "
                   + "this storage function. No schema found for field " +
                   field.getName());
            }
            fs = s.getFields();
            for (Tuple t : (DataBag)d) {
               json.writeStartObject();
               for (int j = 0; j < fs.length; j++) {
                   writeField(json, fs[j], t.get(j));
               }
               json.writeEndObject();
            }
            json.writeEndArray();
            return;
        }
    }
```

11.2.4 任务失败后数据的清理

如果任务启动开始执行后最终失败，那么存储函数可能需要清理一部分已经存储的结果数据。Pig 为存储函数提供了 **cleanupOnFailure** 方法来完成这项任务。它会传递给这个方法路径字符串信息和 job 对象，这样存储函数就可以知道需要清理什么了。如果数据存储在 HDFS 中，那么默认的实现是清除所有由这个存储函数产生的输出文件。如果用户将数据存放在其他地方而非 HDFS 中，那么就需要自己实现这个方法了。

11.2.5 存储元数据信息

如果用户的存储格式除了存储数据外还可以存储模式信息，那么用户的存储函数可以实现 **StoreMetadata** 接口。这个接口提供了一个 **storeSchema** 方法，其可以被 Pig 作为前端操作的一部分进行调用。Pig 会传递给 **storeSchema** 一个 **ResourceShcema** 对象、路径字符串信息和 job 对象，这样它才能连接到它的存储。**ResourceShcema** 与在 10.1.3 小节“输入和输出模式”描述的 **Schema** 类非常相似。但是有一个重大的区别，在 **ResourceFieldSchema** 中，与 bag 相关的模式对象通常只有一个字段，而且是 tuple 类

型的。bag 中这些 tuple 的模式通过 tuple 的 **ResourceFieldSchema** 来描述。

样例存储函数 **JsonStorage** 是通过一个位于数据所在文件夹下的名字叫做*_schema* 的支持文件来存储模式信息的。模式是以字符串的方式存储的，使用的是类中提供的 **toString** 方法：

```
//JsonStorage.java
public void storeSchema(ResourceSchema schema, String location, Job job)
throws IOException {
    //在相同的目录下存储包含模式的支持文件。MapReduce
    //在为 job 读取数据时会忽略以 “_” 开头的文件
    FileSystem fs = FileSystem.get(job.getConfiguration());
    DataOutputStream out = fs.create(new Path(location + “/_schema”));
    out.writeBytes(schema.toString);
    out.writeByte(‘\n’);
    out.close();
}
```

StoreMetadata 同样也有一个 **storeStatistics** 函数，但是 Pig 目前还没有使用到它。

第 12 章

Pig 和其他 Hadoop 社区的成员

12.1 Pig 和 Hive

Apache 的 Hive 项目在基于 Hadoop 之上提供了一个 SQL 层。它接受 SQL 查询命令然后将它们转化为 MapReduce 任务，这和 Pig 对 Pig Latin 脚本进行解析的过程有些许相似。它以表的形式存储数据，并且存储描述这些表的元数据信息，例如分区信息和模式。很多人把 Pig 和 Hive 看作是竞争者关系。因为两者都提供了一种不需要写 Java 代码而可以操作存放在 Hadoop 上的数据的方式，这是一种想当然的推断。然而，正如 1.1.2 节中的“查询语言和数据流语言的比较”中我们所谈到的，SQL 和 Pig Latin 各有各的优势和弱势。因为 Hive 提供的是类 SQL，那么它对于传统的数据分析是个更好的工具。大多数的分析师都很熟悉 SQL，同时商业智能工具更期望使用 SQL 的形式访问数据。而如果构建数据流或者对原始数据进行研究，那么 Pig Latin 会是个更好的选择。

12.2 Cascading

另一个基于 Hadoop 的数据处理框架是 *Cascading*，通过 http://www.cascading.org 链接用户可以查看到更多信息。在为用户提供构建基于 Hadoop 的数据流的功能这点上，Cascading 和 Pig 的目标是一致的。然而，它和 Pig 的实现方式差异非常大。Cascaing 的数据流是使用 Java 语言编写的，而不是像 Pig 那样提供一种新语言。它提供了一个操作符库方便用户将数据操作符连贯起来或者实现他们自己的操作符。这为用户提供了更多的控制力，也因此需要更底层的编码实现。

12.3 NoSQL 数据库

在过去的几年里涌现出了大量的 *NoSQL* 数据库。这些数据库或多或少地打破了传统关

系型数据的一些惯例规则。它们不指望数据是标准化的。相反的，一个独立程序所获取的数据是在一个大表中的，因此很少或不需要进行 join 连接操作。这些数据库中大部分并没有实现完整的 ACID 语义[1]。

与 MapReduce 一样，这些系统用于处理上吉级的数据。而与 MapReduce 不同的是，它们更专注于数据的随机读和写。MapReduce 以及基于 MapReduce 的其他技术（如 Pig）对于读取海量的数据是非常快的，而这些 NoSQL 注重在对小数据量级的数据查找时非常快。这个不同点并不意味着 Pig 不能和这些系统一起工作。用户经常需要分析存放在这些系统中的数据。同时，因为这些系统提供了很好的随机查询功能，对于一些特定的 join 连接操作也因为数据存放在这些系统中而获益不少。

有两种 NoSQL 数据库已经结合到 Pig 中了：HBase 和 Cassandra。

12.3.1 HBase

Apahce 项目 HBase 是一个使用 HDFS 存储数据的 NoSQL 数据库。HBase 以表的形式将数据展现给用户。在每张表中，每一行都有一个键。在 HBase 中是使用一个键或一组键，或一个块扫描来读取数据的。用户也可以按照键更新或者插入不同的行。除了键，HBase 中的行还包含列族，而且一张表中所有行使用相同的列族。每个列族可以包含很多列。对于一个指定的列族，并没有限制要求每一行和其他行一样必须具有相同的列。因此一个 HBase 表 T 可能有一个列族 F，表中所有的列必须都有这个列族，同时允许列族 F 中键 x 对应的行可以包含列 a、b、c，列族 F 中键 y 对应的行可以包含列 a、b、d。列值同时会包含一个版本号。HBase 会根据配置保留一个数量的版本，因此用户可以获取最新版的数据也可以获取之前版本的列值。HBase 中所有的键和列值都是字节数组。

Pig 提供了 HBaseStorage 类用于从 HBase 表中读取数据和写入数据。所有的这些读和写都是批处理操作过程。从 HBase 中批量读数据会比在 HDFS 中扫描数据要慢。然而，如果数据已经存在 HBase 中，直接读取会比先提取，再存放到 HDFS 中，最后再读取要快得多。

当从 HBase 中加载数据时，用户必须要告诉 Pig 从哪张表中读取以及读取哪些列族和哪些列。用户可以读取特定的一些列或者从 0.9 版本开始，读取所有列族。因为列族包含了一组列和列对应的值，所以它们必须要进行类型转换为 Pig 中的 map 类型。举个例子，假设我们有一个 HBase 表名称为 **users**，这张表中存放着用户信息和他们与其他用户的关系信息。它包含两个列族：**user_info** 和 **links**。这张表的关键是用户 ID。**user_info** 列族包含 **name**、**email** 等列。**link** 列族包含一列存放着这个用户关联的用户。列名是关联的用户 ID，这些列的值是关系的类别——朋友、亲属、同事等：

[1] 原子性、一致性、隔离性和持久性。请访问 http://en.wikipedia.org/wiki/ACID 链接了解关系型数据库中对这些属性的更多介绍。

```
user_links = load 'hbase://users'
               using org.apache.pig.backend.hadoop.hbase.HBaseStorage(
               'user_info:name, links:*', '-loadKey true -gt 10000')
               as (id, name:chararray, links:map[]);
```

在 HBase 中，列通过 ***column_famil:column*** 的方式来引用。在之前的例子中，**user_info:name** 表示的是列族 **user_info** 中的列 **name**。当用户想提取一整个列族的时候，用户指定的列族名称要加上星号。例如，**links.***。用户也可以获取一个列族中一部分的列。例如，**links:100***将返回一个所有以“100”开头的列组成的 map。这个 map 包含一个列族，以 HBase 列名作为键，以列值作为值。

用户可以通过“选项”字符串配置 **HBaseStorage**。这可用于控制是否加载键，加载哪些行和其他一些功能。所有的这些选项都放到一个字符串内，通过空格进行分隔。表 12-1 描述了这些选项。

表 12-1　　HBaseStorage 提供的选项

选　项	合法的值	默认的值	描　　述
loadKey	布尔值	false	如果设置为 true，那么键将作为输入中的第一个列
gt	行键	无	只会加载比给定的键值大的行
gte	行键	无	只会加载比给定的键值大或者相等的行
lt	行键	无	只会加载比给定的键值小的行
lte	行键	无	只会加载比给定的键值小或者相等的行
caching	整数	100	扫描器应该缓存的行数
limit	整数	无	每次从 HBase 中读取数据的行数的最大值
caster	Java 类名	Utf8StorageConverter	用于将 Pig 类型和 HBase 存储的二进制数据进行类型转换使用到的 Java 类。这个类必需实现 Pig 的 LoadCaster 和 StoreCaster 两个接口。当 HBase 中存储的数据是 UTF8 编码格式而且数值是以字符串类型存储时（而不是二进制数据），可以使用默认提供的 Utf8StorageConverter。HBaseBinaryCOnverter 使用了 Java 的 Byte.toInt、Byte.toString 等方法。不能使用这个转换器对 map 进行类型转换，因此无法读取整个列族

在我写这章的时候，Pig 还只能读取最晚版本的列值 [1]。对于如何提供最好的接口以及数据类型转换映射结构才能使 Pig 支持多版本数据读取还在讨论之中。这个功能在未来的某个时间点极有可能会增加进去。

[1] 在 Pig 0.11 版本中已经实现了这个功能。

HBaseStorage 同样可以将数据存到 HBase 中。当使用它存储数据的时候，正如在加载数据时一样，需要指定表名为输入地址字符串。构造参数和加载时也是类似的。第一个参数描述了 Pig 字段和 HBase 表之间的映射关系，和加载函数中一样使用 ***column_family:column*** 的语法结构表示。任何一种 Pig 值都可以映射为一个列。Pig 中的 map 可以映射为一个列族，可以通过 ***column_family：**** 的方式指定（同样，这也是在 0.9 及之后的版本中才有的特性）。这个参数并没有引用到行的键，但是这里是假定 Pig tuple 的第一个字段一定是这个值。用作存储函数时，作为可选项的第 2 个参数只有唯一的一个合法选项就是**-caster**。

假设处理结束后 Pig 数据具有模式为：**id:long,name:chararray,email:chararray, link:map。**存储到之前我们用到的那个 HBase 表中将会进行如下这样的操作：

```
//user_links 的模式是（id, name, email, links）
//注意参数中是如何省略 id(key)的
store user_links into 'hbase://users'
    using org.apache.pig.backend.hadoop.hbase.HBaseStorage(
     'user_info:name, user_info:email, links:*');
```

12.3.2 Cassandra

Apache 项目 *Cassandra* 是另一个用于大容量随机读写数据的易扩展的数据库。它和 HBase 的不同在于数据分布的方式。HBase 可以确保服务器之间数据的一致性，然而 Cassandra 有一个最终一致性模型，这也就意味着服务器间在某些时间段内对于相同的数据可能具有不同的值。关于 Cassandra 的更多信息，请看 Edben Hewitt(O'Reilly 出版社出版)所著的《Cassandra 编程指南》一书。

Cassandra 一开始就是支持 Pig 的，这也就意味着用户可以加载 Cassandra 的列族数据和存储数据到 Cassandra 列族中去。这与使用 Pig 的其他存储机制是一样的，如 HDFS。这包括用于输入分割的数据定位。

Pig 和 Cassandra 可以有很多方式共同使用。Pig 可以用于做传统的分析而 Cassandra 可用于实时操作。因为 Pig 和 MapReduce 可以在 Cassandra 上运行，因此可以不需要 Cassandra 和 HDFS 之间移动数据就可以使用。还是需要使用 HDFS 存放中间结果的，然而，Pig 也可以用于对 Cassandra 数据进行数据探索、数据研究、数据测试、数据验证和数据修正。它也可以用于填充存储以新的表或者列族的形式增加进来的数据。

Pygmalion 这个项目的目的就是消除当使用 Pig 处理存储在 Cassandra 中的数据时的开发量。它包含用于从结果中抽取列值和将数据进行整理以提供给 Cassandra，以及其他用途的众多 UDF。

为了使 Pig 和存储在 Cassandra 中的数据能达到平衡，Cassandra 集群需要将数据和 Hadoop task tracker 联系在一起。这样可以允许 Hadoop job tracker 将数据处理过程分配到数据所在的节点上。传统上，Cassandra 用于大数据量的写和实时的、随机获取的查

询。通过将集群按照负载类型进行划分可以在 Cassandra 上运行繁重的 Hadoop 分析任务而不会影响到实时查询的性能。一组节点用于处理分析批处理过程，另一组节点用于处理实时查询。Cassandra 的交叉数据中心冗余备份机制会对集群的这些部分进行数据复制，因此不需要手动进行数据复制，而用于分析的那个部分总是可以获得最新的数据。

12.4 Hadoop 中的元数据

Apache 的 *HCatalog* 项目为 Hadoop 提供了一个元数据和表的控制层。它允许 Hadoop 用户，不管他们使用的是 MapReduce、Pig、Hive 或者其他的工具，都可以像查看表中数据一样查看存储在 HDFS 上的数据。这些表是进行分区的而且有持久的模式。这个抽象的一个结果是，Pig 用户不需要关注一个字段位于哪里，需要使用哪个加载和存储函数，以及文件是否是压缩的。同时也可以让 Pig、MapReduce 和 Hive 的用户更好地分享数据，因为 HCatalog 对于这些工具提供了一个独立的模式和数据类型。数据类型模型是从 Hive 中获取的，与 Pig 的数据类型有所差别，但是加载和存储函数可以对数据类型进行映射转换。HCatalog 使用 Hive 的元数据存储元数据。关于 HCatalog 的完整详细信息，请查看 *http://incubator.apache.org/hcatalog* 链接。

HCatalog 中包含一个加载函数 **HCatLoader。HcatLoader** 函数的位置字符串是表名。它是对 **LoadMetadata** 的实现，因此用户不需要在 **load** 语句中指定模式，Pig 会从 **HCatLoader** 函数中获取到。同样的，因为它实现了这个接口，所以 Pig 也可以使用 HCatalog 的分区功能。如果用户在 **filter** 语句后描述了哪些分区用户期望在加载后立即读取，Pig 会将这个过滤条件加入到加载过程，因此 HCatalog 只会返回相关的分区。

HCatStorer 是 HCatalog 的存储函数。与加载函数一样，位置字符串表示用于存放数据的表。这个存储函数同时提供了一个构造参数用于指定存储的分区的键值对。在写本章内容的时候（0.1 版本），一次存储过程只能指定一个分区。在 0.2 版本中有计划需要向多个分区写数据 [1]。**HCatStorer** 要求将要存储的数据别名的模式要与表中已有数据的模式一致。

举个例子，我们假设有个简单的数据流从表 **web_server_logs** 中读取原始网页日志并做一些处理，然后将数据重新存放到 HCatalog 中指定的表 **processed_logs** 中。表 **web_server_log** 的模式是 (**userid:chararray,date:chararray,time:chararray,url:chararray**), 同时表 **processed_log** 的模式是（**userid:charrarray,user_ref:int,date:chararray,time:chararray,pageid:int,url:chararray**）。如下就是一个做此处理过程的 Pig Latin 脚本：

```
logs = load 'web_server_logs' using HCatalogLoader();
```

[1] 目前在 Hcatalog 0.2 版本中已经实现了这个功能。

```
-- 使用参数传入的方式这样就不需要重写脚本中
-- 过滤器传递给加载器的时间日期了
today = filter logs by date = '$DATE' and NotABot(user_id);
…
-- 输出的模式要与 processed_logs 的 HCatalog 模式完全一致，
-- 包括字段名要完全一致
output = foreach rslvd generate userid, user_ref, date, time , pageid,url;
store output into 'processed_logs' using HCatStorer('date=$DATE');
```

附录 A

内置的用户自定义函数和 Piggybank

本附录包含了 Pig 发行版中所有的 UDF，包括内置的 UDF 和 Piggybank 中用户贡献的 UDF。

内置 UDF

Pig 包中包含的众多 UDF 都是可以直接使用而不需要使用 **register** 或者 **define** 语句来定义的。这里面包括加载函数、存储函数、求值函数和过滤函数。

内置加载函数和存储函数

表 A-1 列举的是内置加载函数；表 A-2 列举的是存储函数。

表 A-1　　加载函数

函数名称	加载地址所表明的含义	构造函数参数	描　　述
HBaseStorage	HBase 表名	第一个参数是描述列族的字符串和与 Pig 字段对应的列 第 2 个参数选项，字符串类型（可选）	从 HBase 中加载数据（12.3.1 小节“HBase”中有详细介绍）
PigStorage	HDFS 文件路径	第 1 个参数是字段间的分隔符（可选的，默认为制表符）	从 HDFS 中加载 text 文件数据(请查 5.2.1 节“加载”)
TextLoader	HDFS 文件路径	没有参数	按行读取 text 文本文件，每一行作为一个有一个 chararay 类型字段的 tuple

表 A-2　　存储函数

函数名称	加载地址所表明的含义	构造函数参数	描　　述
HBaseStorage	HBase 表名	第 1 个参数是描述 Pig 字段对应的 HBase 列族和列的字符串	存储数据到 HBase 中（请查看 12.3.1 小节“HBase”）
PigStorage	HDFS 文件路径	第一个参数是字段间的分隔符（可选的，默认为制表符）	存储文本文件数据到 HDFS 中（请查看 5.2.1 节“加载”）

内置求值函数和过滤函数

求值函数可以分为模仿众多 Java 中数学函数的数学函数；接收数据包然后产生出一个值的聚合函数；操作或者产生复制数据类型的函数；chararrray 和 bytearray 处理函数；过滤函数和辅助函数。

下面会逐个讨论内置的求值函数和过滤函数。在下面这个列表中，为简明起见，包含指定类型的 tuple 的 bag 使用花括号内用圆括号包围数据类型的方式展现。例如，一个包含一个整数类型字段的 tuple 的 bag 可以表示为{(int)}。

内置数学函数

double ABS(double input)

参数：
　　input
返回值：
　　绝对值
起始版本：
　　0.8

double ACOS(double input)

参数：
　　input
返回值：
　　反余弦
起始版本：
　　0.8

double ASIN(double input)

参数：

input

返回值：

反正弦

起始版本：

0.8

double ATAN(double input)

参数：

input

返回值：

反正切

起始版本：

0.8

double CBRT(double input)

参数：

input

返回值：

立方根

起始版本：

0.8

double CEIL(double input)

参数：

input

返回值：

向上取整计算，它返回的是大于或等于函数的参数，并且与之最接近的整数。

起始版本：

0.8

double COS(double input)

参数：

input

返回值：

余弦

起始版本：

0.8

double COSH(double input)

参数：

input

返回值：

双曲余弦

起始版本：

0.8

double EXP(double input)

参数：

input

返回值：

以 e 为底的指数函数，返回 **e**（自然对数的底）的幂次方。

起始版本：

0.8

double FLOOR(double input)

参数：

input

返回值：

返回小于或者等于指定表达式的最大整数。

起始版本：

0.8

double LOG(double input)

参数：

input

返回值：

自然对数

起始版本：

0.8

double LOG10(double input)

参数：

input

返回值：

以 10 为底的对数函数

起始版本：

0.8

long ROUND(double input)

参数：

input

返回值：

对输入值进行四舍五入取值

起始版本：

0.8

double SIN(double input)

参数：

input

返回值：

正弦

起始版本：

0.8

double SINH(double input)

参数：

input

返回值：

双曲正弦

起始版本：

0.8

double SQRT(double *input*)

参数：

input

返回值：

平方根

起始版本：

0.8

double TAN(double *input*)

参数：

input

返回值：

正切

起始版本：

0.8

double TANH(double *input*)

参数：

input

返回值：

双曲正切

起始版本：

0.8

内置聚合函数

int AVG({(*int*)} *input*)

参数：

input

返回值：

input 中的所有值的平均值，null 值会被忽略。

起始版本：

0.2

long AVG({(long)} *input*)

参数：

input

返回值：

input 中的所有值的平均值，null 值会被忽略。

起始版本：

0.2

float AVG({(float)} *input*)

参数：

input

返回值：

input 中的所有值的平均值，null 值会被忽略。

起始版本：

0.2

double AVG({(double)} *input*)

参数：

input

返回值：

input 中的所有值的平均值，null 值会被忽略。

起始版本：

0.2

double AVG({(bytearray)} *input*)

参数：

input

返回值：

input 中的所有 bytearray 类型值的平均值，会被类型转换为 double 型，null 值会被忽略。

起始版本：

0.1

long COUNT

这个函数的作用与 SQL 中 COUNT（字段）的作用相同。

参数：

input

返回值：

input 中的记录的个数，包括 null 值。

起始版本：

0.1

long COUNT_STAR

这个函数的作用与 SQL 中 COUNT(*)的作用相同。

参数：

input

返回值：

input 中的所有记录的个数，包括 null 值。

起始版本：

0.4

int MAX({(int)} *input*)

参数：

input

返回值：

input 中的最大值，null 值会被忽略。

起始版本：

0.2

long MAX({(long)} *input*)

参数：

input

返回值：

input 中的最大值，null 值会被忽略。

起始版本：

0.2

float MAX({(float)} *input*)

参数：

input

返回值：

input 中的最大值，null 值会被忽略。

起始版本：

0.2

double MAX({(double)} *input*)

参数：

input

返回值：

input 中的最大值，null 值会被忽略。

起始版本：

0.2

chararray MAX

参数：

input

返回值：

input 中的最大值，null 值会被忽略。

起始版本：

0.2

double MAX({(bytearray)} *input*)

参数：

input

返回值：

input 中所有 bytearray 类型中值的最大值，会被类型转换为 double 型，null 值会被忽略。

起始版本：

0.1

int MIN({(int)} *input*)

参数：

input

返回值：

input 中的最小值，null 值会被忽略。

起始版本：

0.2

long MIN({(long)} *input*)

参数：

input

返回值：

input 中的最小值，null 值会被忽略。

起始版本：

0.2

float MIN({(float)} *input*)

参数：

input

返回值：

input 中的最小值，null 值会被忽略。

起始版本：

0.2

double MIN({(double)} *input*)

参数：

input

返回值：

input 中的最小值，null 值会被忽略。

起始版本：

0.2

chararray MIN

参数：

input

返回值：

input 中的最小值，null 值会被忽略。

起始版本：

0.2

double MIN({(bytearray)} *input*)

参数：

input

返回值：

input 中所有 bytearray 类型中值的最小值，会被类型转换为 double 型； null 值会被忽略。

起始版本：

0.1

long SUM({(int)} *input*)

参数：

input

返回值：

数据包中所有值的和；null 值会被忽略。

起始版本：

0.1

long SUM({(long)} *input*)

参数：

input

返回值：

数据包中所有值的和；null 值会被忽略。

起始版本：

0.2

double SUM({(float)} *input*)

参数：

input

返回值：

数据包中所有值的和；null 值会被忽略。

起始版本：

0.2

double SUM({(double)} *input*)

参数：

input

返回值：

数据包中所有值的和；null 值会被忽略。

起始版本：

0.2

double SUM({(bytearray)} *input*)

参数：

input

返回值：

数据包中所有 bytearray 类型值的和，会被类型转换为 double 型；null 值会被忽略。

起始版本：

0.1

内置 chararray 和 bytearray 处理 UDF

chararray CONCAT(chararray *c1*, chararray *c2*)

参数：

c1　c2

返回值：

c1 和 c2 连接

起始版本：

0.1

bytearray CONCAT(bytearray *b1*, bytearray *b2*)

参数：

b1　b2

返回值：

b1 和 b2 连接

起始版本：

0.1

int INDEXOF(chararray *source*, chararray *search*)

参数：

source：被查找的字符串

search：要查找的字符串

返回值：

source 字符串中第 1 个 seach 字符串所在的位置；如果字符串中没有要查找的字符串则返回−1。

起始版本：

0.8

int LAST_INDEX_OF(chararray *source*, chararray *search*)

参数：

source：被查找的字符串

search：要查找的字符串

返回值：

source 字符串中最后一个 seach 字符串所在的位置，如果字符串中没有要查找的字符串则返回−1。

起始版本：

0.8

chararray LCFIRST(chararray *input*)

参数：

input

返回值：

第 1 个字符转换为小写

起始版本：

0.8

chararray LOWER(chararray *input*)

参数：

input

返回值：

所有的输入字符转换为小写

起始版本：

0.8

chararray REGEX_EXTRACT(chararray *source*, chararray *regex*, int *n*)

参数：

source：被查找的字符串

regex：使用的正则表达式

n：第 n 个匹配的部分，从 0 开始

返回值：

source 中符合正则表达式的第 n 个部分；如果没有匹配的部分则返回 null。

起始版本：

0.8

(chararray) REGEX_EXTRACT_ALL(chararray *source*, chararray *regex*)

参数：

source：被查找的字符串

regex：使用的正则表达式

返回值：

source 中符合正则表达式的所有部分存放到 tuple 中；如果没有匹配的部分则返回 null。

起始版本：

0.8

chararray REPLACE(chararray *source*, chararray *toReplace*, chararray *newValue*)

参数：

source：被查找的字符串

toReplace：将被替换的字符串

newValue：用于替换的新的字符串

返回值：

source 中所有的 toReplace 字符串被替换为 newValue 字符串。

起始版本：

0.8

long SIZE(chararray *input*)

参数：

input

返回值：

input 中字符的个数

起始版本：

0.2

long SIZE(chararray *input*)

参数：

input

返回值：

input 中字符的个数

起始版本：

0.2

(chararray) STRSPLIT(chararray *source*)

按照空格分隔字符串

参数：

source：将进行分割的字符串

返回值：

source 分割后的使用包含一个字段的 tuple 存放各个部分。

起始版本：

0.8

(chararray) STRSPLIT(chararray *source*, chararray *regex*)

按照正则表达式对字符串进行分割

参数：

source：将进行分割的字符串

regex：使用的正则表达式

返回值：

source 分割后的使用包含一个字段的 tuple 存放各个部分。

起始版本：

0.8

(chararray) STRSPLIT(chararray *source*, chararray *regex*, int *max*splits)

按照正则表达式对字符串进行分割

参数：

source：将进行分割的字符串

regex：使用的正则表达式

max：分割的最大个数

返回值：

source 分割后的使用包含一个字段的 tuple 存放各个部分，如果有多于 max 个部分，那么只会有前面 max 个部分会保留。

起始版本：

0.8

chararray SUBSTRING(chararray *source*, int *start*, int *end*)

参数：

source：将进行分割的字符串

start：开始位置（闭区间），从 0 开始

end：结束位置（开区间），从 0 开始

返回值：

子字符串；如果任何输入值的长度小于 start 值，那么会报错。

起始版本：

0.8

{(chararray)} TOKENIZE(chararray *input*)

参数：

source：将进行分割的字符串

返回值：

按照空格对输入进行分割，每一个结果值都会放入一个单独的 tuple 中，最后所有的 tuple 放到 bag 中。

起始版本：

0.1

chararray TRIM(chararray *input*)

参数：

input

返回值：

去掉 input 前面和后面的所有空格

起始版本：

0.8

chararray UCFIRST(chararray *input*)

参数：

input

返回值：

将 input 的第 1 个字符转换成大写

起始版本：

0.8

chararray UPPER(chararray *input*)

参数：

input

返回值：

将 input 中的所有字符转换成大写

起始版本：

0.8

内置复杂数据类型 UDF

{(chararray, chararray, double)} COR({(double)} *b1*, {(double)} *b2*)

计算两个 double 数据包之间的相关系数

参数：

b1　b2

返回值：

第 1 个 chararray 是 b1 的别名，第 2 个 chararray 是 b2 的别名，double 是 b1 和 b2 间的相关系数。

起始版本：

0.8

{(chararray, chararray, double)} COV({(double)} *b1*, {(double)} *b2*)

计算两个 double 数据包之间的协方差

参数：

b1　b2

返回值：

第 1 个 chararry 是 b1 的别名，第 2 个 chararray 是 b2 的别名，double 是 b1 和 b2 间的协方差。

起始版本：

0.8

bag DIFF(bag *b1*, bag *b2*)

参数：

b1　b2

返回值：

在 b1 中而不在 b2 中的值，与在 b2 中而不在 b1 中的值的合集。

起始版本：

0.1

long SIZE(map *input*)

参数：

input

返回值：

input 中的键–值对个数

起始版本：

0.2

long SIZE(tuple *input*)

参数：

input

返回值：

input 中的字段个数

起始版本：

0.2

long SIZE(bag *input*)

参数：

input

返回值：

input 中的 tuple 个数

起始版本：

0.2

bag TOBAG(...)

参数：

变量

返回值：

如果所有的输入具有相同的模式，那么结果数据包就会有那个模式，否则它将会有 null 模式；如果参数是 tuple，所有的模式除了类型相同外字段名也必须相同。

起始版本：

0.8

map TOMAP(...)

参数：

变量

返回值：

输入的参数会形成键-值对存放到 map 中，所有的键都必须是 chararray 字符串类型；如果输入参数的个数是奇数的话则会报错。

起始版本：

0.9

bag TOP(int *numRecords*, int *field*, bag *source*)

参数：

numRecords *将要返回的记录个数*

field：将要进行排序的字段

source：被处理的数据包

返回值：

返回包含有 numRecords 个记录的数据包

起始版本：

0.8

tuple TOTUPLE(...)

参数：

变量

返回值：

作为参数传入的所有记录形成的数据包

起始版本：

0.8

内置过滤函数

boolean IsEmpty(bag)

参数：

input

返回值：

布尔值

起始版本：

0.1

boolean IsEmpty(tuple)

参数：

input

返回值：

布尔值

起始版本：

0.1

其他内置 UDF

double RANDOM()

返回值：

一个 0 和 1 之间的随机 double 值

起始版本：

0.4

Piggybank

Piggybank 是用户贡献的函数资源库。Piggybank 函数作为 Pig 发行版中的一个组成部分一起发布，但是它们并非是内置的。如果想使用里面的函数，用户必须先要注册 Piggybank 的 JAR 包，一般在 Pig 发行版的如下目录中可以看到：*contrib/piggybank/java/piggybank.jar*。

在我写这本书的时候，关于 Piggybank 还没有特定的网站或者文档介绍说明[1]。如果想知道里面有什么内容，我们可以去查看相关代码。在用户的发行版 ***contrib/piggybank*** 目录下可以查看到它所包含的函数。Piggybank 还没有包含任何的 Python 函数，但是因为它本身允许用户使用 Java 以外的其他语言编写函数，所以希望将来这能有所改变。

[1] https://cwiki.apache.org/confluence/display/PIG/PiggyBank.

附录 B

Hadoop 综述

本附录将对 Hadoop 进行一个简短的综述，重点关注那些 Pig 用户可能感兴趣的部分。如果想对 Hadoop 有深入的了解，请阅读 Tom White（O'Reilly 出版社出版）所著的《Hadoop:编程指南》一书。Hadoop 的两个主要组成部分是 MapReduce 和 HDFS。

MapReduce

MapReduce 是在 Hadoop 中执行任务的框架。它提供了一个简单但是强大的并行数据处理框架。

JobTracker 是 MapReduce 中任务的中央协调者。它控制着哪些任务在执行，这些任务获得了哪些资源等等。集群中的每个节点都会有一个 *TaskTracker* 负责执行 JobTracker 分配给它的 map 或 reduce 任务。

MapReduce 将它的输入视为一组记录。当从 HDFS 中读取时，一条记录通常就是一行文本文件。每条记录都有一个键和一个值。并不要求数据是按照键排序的，也不要求键必须是唯一的。同样的，MapReduce 会产生一组记录，每条记录也包含一个键和一个值。

MapReduce 操作着 job 中的数据。每个 job 有一个输入和一个输出 [1]。MapReduce 将每个 job 分割成一系列的任务。这些任务有两个主要的类型：map 任务和 reduce 任务。

Map 阶段

在 map 阶段，MapRedue 允许用户对单个数据集中的每条记录进行操作。这个阶段通常用于剔除那些不需要的字段，或对字段进行转换，或者在这里使用过滤器。一些特定

[1] 可能会违背这个规则，正如 Pig 和其他很多程序确实会违背。例如，这里的一个输入可以是一连串的多个输入文件，这些文件可以在任务进行过程中被打开或被写入或被读取。但是从概念上，每个 job 都有一个主输入和一个主输出。

的 join 和 group 过程也可以在 map 阶段完成（例如对于已经排好序的或者基于哈希的聚合所进行的 join）。这里并不要求对于每条输入记录一定要产生对应的一条输出记录。map 可以将一些不需要的记录移除或者将一条记录分解成多条记录。

每个 MapRedcue 任务都会指定一个 InputFormat。这个类用于决定 map 任务间是如何分割数据的并会提供一个 RecordReader。

为了指定 map 任务间是如何分割数据的，一个 InputFormat 会将输入数据分割成一组 InputSplit。每个 InputSplit 都被提供给独立的一个 map。除了包含读取什么样的信息之外，InputSplit 还会包含一个需要用于读取数据的集群节点的列表。通过这种方式，当数据存放在 HDFS 中时，MapReduce 就有能力对这些数据进行计算。

InputFormat 所提供的 RecordReader 会读取输入数据，然后产生将传送给 map 的 key-value 键值对。这个类控制数据是如何解压缩的（如果有需要），以及控制如何转换成 MapReduce 可以使用的 Java 数据类型。

Combiner 阶段

组合器允许应用程序在早期执行它们的 reducer 逻辑。当 map 阶段写输出的时候，它将被序列化并且被放入一个内存缓冲区。当这个缓冲区满了的时候，MapReduce 将会对缓冲区进行排序，之后如果程序提供了一个组合器的实现那么就会运行组合器。结果输出随后会被写入本地磁盘，在 shuffle 阶段会再次被读取然后传送给 reducer。MapReduce 如果侦测到不运行组合器效率会更高，那么它就会选择不运行组合器。

shuffle 阶段后，每个 reducer 都将会有每个 map 的一个输入。在开始处理前，reducer 需要将这些输入进行合并，同时合并太多的输入效率不会高。因此，如果输入的个数超过一个特定的值，数据在传送给 reducer 之前会被合并然后重新写入磁盘。在这个合并过程中，组合器用于尝试减少输入数据量的大小。请翻阅 Hadoop 文档查看关于如何以及何时可以触发这种预处理合并过程的更多信息。

因此组合阶段将会运行 0 次、1 次或者多次，组合器的输入和输出的键和值必须是相同的数据类型。

Shuffle 阶段

在 shuffle 阶段，MapReduce 在不同的 reducer 间划分数据。

MapReduce 使用一个叫做 **Partitioner** 的类在 shuffle 阶段将记录划分给 reducer。**Partitioner** 的一个实现将接受记录的键和值，以及 reduce 任务的所有编号，同时会返回这条记录应该被传送过去的 reduce 任务编号。默认情况下，MapReduce 会使用 **HashPartitioner**，这个实现会对键调用 **hashCode()**方法，然后返回 reduce 任务的编号取模后的值作为结果。通过过载默认的分割器，MapReduce 用户可以继承实现并使用他们自己的 Patitioner。关于 Partitioner 的更多信息请参考 Hadoop 文档。

数据在经过 map 阶段、combiner 阶段和 shuffle 阶段后进行了划分和排序，之后会抵达 reducer。默认情况下，数据是按照划分的键进行排序的。例如，如果一个用户按照用户 ID 对数据进行划分，那么在 reducer 中也将是按照用户 ID 进行排序。因此，MapReduce 使用排序的方式将相似的键分组到一起。除了按照划分的键进行排序，也可以指定增加其他的键进行排序。例如，用户可以选择按用户 ID 进行划分同时也按照时间戳进行排序。这个特性是很有用的，因为这样用户就不必对 reduce 阶段的数据实现他们自己的排序方法了。

Reduce 阶段

reduce 阶段的输入是 shuffle 阶段的键加上与这个键相关的记录。因为所有包含键所对应的值的记录现在都被聚集在一起了，所以就可以做一些 join 操作和像 count 那样的聚合操作。MapReduce 用户在 reduce 阶段显式地控制着并行操作。对于不需要 reduce 阶段的 MapReduce 任务可以设置 reduce 计数器为 0。这类任务被称为是 *map-only 的任务*。

输出阶段

reducer（或者只有 map 过程的 map）通过一个 OutputFormat 类写输出。OutputFormat 用于提供一个 RecodWriter 的实现类，该实现类会接收任务产生的键-值对然后将它们进行存储。这个过程包括序列化，还可能包括压缩，以及将它们写入 HDFS、HBase 等。OutputFormat 同时也用于提供 OutputCommitter，它被用于协助操作，例如执行失败后的数据清理，以及通知存储介质数据是可用的（如一个数据库事务）。

分布式缓存

有时一个 MapReduce job 内的所有或者一些任务需要访问一个文件或者一组文件。例如，当将一个大文件和一个小文件进行 join 连接操作的时候，一种处理方式是将小文件作为一个侧文件打开（也就是说，在 map 阶段直接打开而不是将其指定为 MapReduce job 的一个输入），将它加载到内存中，然后在 map 阶段进行 join 操作。当成千上万的 map 或者 reduce 任务同时尝试打开一个相同的 HDFS 文件的时候，这将极大地增加 NameNode 和存储这个文件的所有 DataNode 的压力。为了避免这种情况，MapReduce 提供了分布式缓存。分布式缓存允许用户指定，作为他们 MapReduce job 的一部分，任意的 HDFS 文件可以提供给每一个任务进行访问。这些文件会被复制到这些任务节点的本地磁盘上，同时作为任务的初始化过程的一部分。Map 或者 reduce 任务就可以以本地文件的方式读取它们。

故障处理

MapReduce 强大的部分可以体现在它可以很好地处理故障并且会为用户进行多次重新尝试执行。如果用户有一个调用了 10,000 个 map 任务的 MapReduce job（并非是一个不常见的情景），负载不均衡相当地高以致于在 job 执行过程中至少会导致一台机器不

可用。在设计 MapReduce 的时候就是基于故障是很常见的而且必须进行处理这个假定的，而不是直接将故障移除系统。当一个特定的 map 或者 reduce 任务失败了，MapReduce 会通过产生一个替补的任务来做这个工作的方式进行处理，这被称为推测执行。当一个任务错误次数（默认为 4 次）达到一定值的时候，MapReduce 就会放弃该任务并认为这个任务和 job 是失败了。

HDFS

Hadoop 分布式文件系统（HDFS）将文件存储到一个 Hadoop 集群的所有节点中。它将文件分割成大的数据块然后分发到不同的机器中，同时也会对每个数据块进行多份冗余来保证如果任意一台机器宕掉了，不会造成数据丢失或不可用。默认情况下会对每个数据块进行 3 份冗余，不过这个值是可以配置改变的。有一份冗余总是存放在用户执行写入操作的那个节点上。如果用户的 Hadoop 集群分布在多台机架上，那么 HDFS 会将一份数据块冗余存放在写入操作所在机器的所在机架上，另一份冗余放在不同的机架上。当一台机器宕掉或者磁盘坏掉或者数据块坏掉了，HDFS 会对这些丢失的数据块进行重新复制以确保数据维持在一个合适的冗余数。

HDFS 很明确地被设计成可以很好地支持 MapReduce。数据块很大，默认为 64MB。很多用户会把这个值设置得更高，如 128MB 甚至 256MB。MapReduce 的批处理模型很适合在大数据块中进行排序，在这个模型中假定每个 job 会读取一个文件中的所有记录。现代的磁盘连续读要比随机读快得多。因此对于大数据集，如果用户需要的不是很少的一些记录，连续读比随机读要好得多。数据的 3 种冗余策略，除了显然可以提高容错性以外，也利于 MapReduce，因为它给了 JobTracker 更多的选择去定位并分配 map 任务到具有这份冗余数据块的机器上去。

HDFS 提供了一个类似 POSIX 的用户交互界面并提供了标准的文件系统功能，例如文件权限控制、安全和分配额控制。

NameNode 是 HDFS 的大脑，它用于维护 HDFS 中文件的总清单，同时控制着文件名和数据块之间的映射。它知道每个数据块存放在什么地方，还要确保每个数据块按合适的备份数进行冗余备份。DataNode 是存储 HDFS 数据的机器，它们使用单独的文件存放数据块，每个 DataNode 都对应着一个 TaskTracker 用于推动对数据的计算。

作者介绍

Alan Gates 是将 Pig 从 Yahoo!的研究项目转化成一个成功的 Apache 开源项目的开发团队的创始成员之一。基于这个身份，他监督审查了这门语言的实现、编程接口和总体设计。他在众多的会议和用户组、大学和公司中对 Pig 进行了介绍。Alan 是 Apache 软件基金会的成员以及 Hortonwroks 的共同创始人之一。他是美国俄勒冈州立大学数学理学学士和福乐神学院的神学硕士。

书末说明

《Pig 编程指南》封面上的动物是家猪。虽然猪种群大多分布在非洲、亚洲和欧洲，但是现在家猪几乎存在于任何有人类居住的地方。事实上，一些猪通过特别培育可以适应各种各样的气候环境。例如，已经培育出毛皮厚重的猪品种可以适应寒冷的气候。人类将猪带到他们所去的各种地方的原因很明显：除了猪肉可以作为食物这最重要的一点以外，人类还可以使用猪的皮、骨头和猪毛制作各种各样的工具。

家猪直接起源于野猪，并且有证据显示历史上有 3 个独立的驯化事件。第 1 次事件发生在公元前 13,000 年的底格里斯河流域，第 2 次发生在中国，第 3 次发生在欧洲，尽管最后一次很可能发生在欧洲人从中东地区引入了家猪之后。虽然历史悠久，然而分类学家对家猪的确切分类并没有达成一致的认同。一些分类学家认为家猪和野猪都各属 pig 这个种群的子物种，而其他人则认为家猪和野猪各自是独立的物种。无论在哪种情况下，家猪具有好几百的品种，每一种都有其专用的特性。

也许因为它们历史悠久同时在人类社会中扮演了突出的角色，以及它们的社会性行为的趋势，家猪经常出现在电影、文学和其他文化媒体中。例子有童话故事《三只小猪的故事》，幼儿电视教育节目《芝麻街》里的粉红色猪小姐，动画片《波基猪》中的猪小弟。此外，目前家猪还被发现具有很好的智商和能力便于进行训练（这和对狗进行训练相同），也因此会被作为宠物来养。

封面上的图片来自于《多佛尔画报档案》。

异步社区
人民邮电出版社
www.epubit.com.cn

欢迎来到异步社区！

异步社区的来历

异步社区（www.epubit.com.cn）是人民邮电出版社旗下 IT 专业图书旗舰社区，于 2015 年 8 月上线运营。

异步社区依托于人民邮电出版社 20 余年的 IT 专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与 POD 按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。

社区里都有什么？

购买图书

我们出版的图书涵盖主流 IT 技术，在编程语言、Web 技术、数据科学等领域有众多经典畅销图书。社区现已上线图书 1000 余种，电子书 400 多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

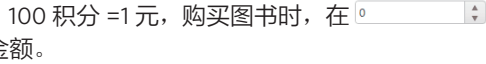
与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题，可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100 积分 =1 元，购买图书时，在 

里填入可使用的积分数值，即可扣减相应金额。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得 100 积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于 Markdown 的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握 IT 圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：

异步社区	微信服务号	微信订阅号	官方微博	QQ 群：368449889

社区网址：www.epubit.com.cn

官方微信：异步社区

官方微博：@ 人邮异步社区，@ 人民邮电出版社 - 信息技术分社

投稿 & 咨询：contact@epubit.com.cn