Hadoop 权威指南

大数据的存储与分析(第 4 版)

Hadoop: The Definitive Guide Storage and Analysis at Internet Scale

Tom White 著

王 海 华 东 刘 喻 吕粤海 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc. 授权清华大学出版社出版

清华大学出版社

北 京

内 容 简 介

本书结合理论和实践，由浅入深，全方位介绍了 Hadoop 这一高性能的海量数据处理和分析平台。全书 5 部分 24 章，第 I 部分介绍 Hadoop 基础知识，主题涉及 Hadoop、MapReduce、Hadoop 分布式文件系统、YARN、Hadoop 的 I/O 操作。第 II 部分介绍 MapReduce,主题包括 MapReduce 应用开发；MapReduce 的工作机制、MapReduce 的类型与格式、MapReduce 的特性。第III部分介绍 Hadoop 的运维，主题涉及构建 Hadoop 集群、管理 Hadoop。第IV部分介绍 Hadoop 相关开源项目，主题涉及 Avro、Parquet、Flume、Sqoop、Pig、Hive、Crunch、Spark、HBase、ZooKeeper。第 V 部分提供了三个案例，分别来自医疗卫生信息技术服务商塞纳(Cerner)、微软的人工智能项目 ADAM(一种大规模分布式深度学习框架)和开源项目 Cascading(一个新的针对 MapReduce 的数据处理 API)。

本书一本权威、全面的 Hadoop 参考与工具书，阐述了 Hadoop 生态圈的最新发展和应用，程序员可以从中探索海量数据集的存储和分析，管理员可以从中了解 Hadoop 集群的安装和运维。

Copyright © 2016 Tom White. All rights reserved.
Authorized Simplified Chinese translation edition, by O'Reilly Media, Inc., is published by Tsinghua University Press, 2017. Authorized translation of the original English edition, 2012 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.
本书之英文原版由 O'Reilly Media, Inc. 于 2016 年出版。
本书中文简体版由 O'Reilly Media,Inc.授权清华大学出版社 2017 年出版。此翻译版的出版和销售得到出版权和销售权的所有者——O'Reilly Media,Inc.的许可。

版权所有，未经书面许可，本书的任何部分和全部不得以任何形式复制。

北京市版权局著作权合同登记号　图字：**01-2015-2862**
本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：**010-62782989　13701121933**

图书在版编目(CIP)数据

Hadoop 权威指南/(美)汤姆·怀特(Tom White)著；王海，华东，刘喻，吕粤海译. —4 版. —北京：清华大学出版社，2017（2017.10 重印）

书名原文：Hadoop: The Definitive Guide

ISBN 978-7-302-46513-3

Ⅰ. ①H…　Ⅱ. ①汤…　②王…　③华…　④刘…　⑤吕…　Ⅲ. ①数据处理软件－指南　Ⅳ. ①TP274-62

中国版本图书馆 CIP 数据核字(2017)第 025689 号

责任编辑：文开琪
封面设计：Karen Montgomery　张　健
责任校对：周剑云
责任印制：刘海龙

出版发行：清华大学出版社
网　址：http://www.tup.com.cn，http://www.wqbook.com
地　址：北京清华大学学研大厦 A 座　　**邮　编**：100084
社总机：010-62770175　　**邮　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：三河市铭诚印务有限公司
经　销：全国新华书店
开　本：178mm×233mm　**印　张**：46　**插　页**：1　**字　数**：594 千字
版　次：2017 年 7 月第 4 版　　**印　次**：2017 年 10 月第3 次印刷
定　价：148.00 元

产品编号：063443-01

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站(GNN)；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，还是在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列(真希望当初我也想到了)非凡想法建立了数百万美元的业务。”

——*Business* 2.0

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是一位特立独行的人，他不光放眼于最长远、最广阔的视野并且切实地按照尤吉·贝拉的建议去做了：‘如果你在路上遇到岔路口，就选择走小路。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

推荐序一

Doug Cutting@加州院内小屋

Hadoop 起源于 Nutch 项目。我们几个人有一段时间一直在尝试构建一个开源的 Web 搜索引擎，但始终无法有效地将计算任务分配到多台计算机上，即使就只是屈指可数的几台。直到谷歌发表 GFS 和 MapReduce 的相关论文之后，我们的思路才清晰起来。他们设计的系统已经可以精准地解决我们在 Nutch 项目中面临的困境。于是，我们(两个半天工作制的人)开始尝试重建这些系统，并将其作为 Nutch 的一部分。

我们终于让 Nutch 可以在 20 台机器上平稳运行，但很快又意识一点：要想应对大规模的 Web 数据计算，还必须得让 Nutch 能在几千台机器上运行，不过这个工作远远不是两个半天工作制的开发人员能够搞定的。

差不多就在那个时候，雅虎也对这项技术产生了浓厚的兴趣并迅速组建了一个开发团队。我有幸成为其中一员。我们剥离出 Nutch 的分布式计算模块，将其称为“Hadoop”。在雅虎的帮助下，Hadoop 很快就能够真正处理海量的 Web 数据了。

从 2006 年起，Tom White 就开始为 Hadoop 做贡献。很早以前，我便通过他的一篇非常优秀的 Nutch 论文认识了他。在这篇论文中，他以一种优美的文风清晰地阐述复杂的思路。很快，我还得知他开发的软件一如他的文笔，优美易懂。

从一开始，Tom 对 Hadoop 所做的贡献就体现出他对用户和项目的关注。与大多数开源贡献者不同，Tom 并没有兴致勃勃地调整系统使其更符合自己个人的需要，而是尽可能地使其方便所有人使用。

最开始，Tom 专攻如何使 Hadoop 在亚马逊的 EC2 和 S3 服务上高效运行。随后，他转向解决更广泛的各种各样的难题，包括如何改进 MapReduce

API，如何增强网站特色，如何精心构思对象序列化框架，如此等等，不一而举。在所有这些工作中，Tom 都非常清晰、准确地阐明了自己的想法。在很短的时间里，Tom 就赢得大家的认可，拥有 Hadoop 提交者(committer)的权限并很快顺理成章地成为 Hadoop 项目管理委员会的成员。

现在的 Tom，是 Hadoop 开发社区中受人尊敬的资深成员。他精通 Hadoop 项目的若干个技术领域，但他更擅长于 Hadoop 的普及，使其更容易理解和使用。

基于我对 Tom 的这些了解，所以当我得知 Tom 打算写一本 Hadoop 的书之时，别提有多高兴了。是的，谁比他更有资格呢？！现在，你们有机会向这位年青的大师学习 Hadoop，不单单是技术，还有一些必知必会的常识，以及他化繁为简、通俗易懂的写作风格。

推荐序二

周立柱@清华园

在这本《Hadoop 权威指南(第 4 版)》即将出版之际，我十分高兴地再次向广大读者推荐这本书，并期待着它成为我国从事大数据系统研究与开发的科研人员、工程师的一本有价值的参考书。

迄今为止，Hadoop 的发展已经经历了两代，分别为 Hadoop 1.0 和 Hadoop 2.0。与《Hadoop 权威指南(第 3 版)》相比，第 4 版在重点介绍 Hadoop 2.0 的基础上，新增了对当前热门的 Hadoop 技术(如 YARN、Parquet、Flume、Crunch 和 Spark)的专门讲解，有助于 Hadoop 开发者更好地理解相关技术的背景、原理及使用。此外，第 4 版还引入了 Hadoop 在医疗健康领域和分子生物学领域的最新应用成果，并为此新增了相关的实例学习，这对广大 Hadoop 用户而言，具有更好的实践指导意义。

今天，Hadoop 开源项目已经成为研究大数据、开发大数据应用的重要平台，在我国已经形成一个庞大的 Hadoop 用户社群，他们对学习、掌握和提高 Hadoop 提出了很高的需求，《Hadoop 权威指南》系列版本的推出恰好可以满足这样的需要。该书从第 1 版发行以来，历次再版后的畅销也证明了它的用途和价值。

原著的内容组织得当，思路清晰，从原著第 4 版的大幅更新可以看出作者 Tom White 认真、严谨的态度以及对技术的尊重。几位译者在本书翻译过程中，也力求做到清晰、准确和忠实于原著，并为此付出了宝贵的时间和艰辛的劳动。

译者序

自 2006 年面世以来，Hadoop 技术发展迅猛，其技术生态圈也日益壮大，从最初只有 HDFS 和 MapReduce 两个组件，发展到当前的六十多个组件，覆盖了从数据存储、执行引擎到数据访问框架等各个层面。Hadoop 的本地化计算理念、弹性的多层级架构、高效的分布式计算框架，在提供了前所未有的计算能力的同时，也大大降低了计算成本，使其在大规模数据处理分析上的表现远远超过其他产品，不但被广泛应用于各行各业的数据分析和处理，更已成为各大企业数据平台的首选。

伴随着 Hadoop 的发展壮大，本书作者 Tom White 的《Hadoop 权威指南》从 2009 年初版发布至今，也在不断地修订、更新和进一步完善，目前已经推出了第 4 版。该书被业内誉为 Hadoop 圣经，见证了 Hadoop 发展过程中的每一次飞跃。该书的每一新版本都会将 Hadoop 的最新技术和最新应用实践分享给广大读者。

第 4 版完全围绕 Hadoop 2.0 展开描述和讨论。在这一版中，读者可以学习 MapReduce、HDFS 和 YARN 等基础组件的概念和使用步骤；学会设置和维护一个在 YARN 上运行 HDFS 和 MapReduce 的 Hadoop 集群；学习 Avro(用于数据序列化)和 Parquet 两种数据格式(用于嵌套数据)；学习使用数据注入工具，如 Flume(用于序列化数据)和 Sqoop(用于块数据传输)；深入了解高层数据处理工具如 Pig、Hive、Crunch 和 Spark 与 Hadoop 的配合使用；学习 HBase 分布式数据库和 ZooKeeper 分布式配置服务。相应的，作者专门分章介绍 YARN 及几个 Hadoop 相关联的项目如 Parquet、Flume、Crunch 和 Spark。通过第 4 版，读者可以学习到 Hadoop 的最新变化，以及 Hadoop 在健康医疗系统和基因数据处理中的新应用实践。本书非常适用于从事任意规模数据集分析的编程者，对于设置、运行并维护 Hadoop 集群的管理者来说也相当有指导意义。

作为经典书籍，这本书内容丰富、结构清晰，理论与实际结合紧密。2015年，Tom White根据Hadoop的新版本推出第4版，2016年，在Hadoop诞生十年之际，我们非常有幸能够承担本书的翻译工作。希望新版本能够给读者带来更高的技术含量，更好的阅读感受。

全书包含24章和4个附录。翻译和审校工作由王海教授组织完成。参加翻译工作的有刘喻(第1章到第5章)、华东(第6章到第11章)、吕粤海(第12章到第14章)、张娟(第15章到第17章)、米志超(第18章到第20章)、牛大伟(第21章到第23章)、董超(第24章及4个附录)。以下老师和同学也对本书的翻译和审校做出了贡献：于卫波、郭晓、李艾静、陈强、刘庆和徐晶。

为确保读者阅读的连贯性及与《Hadoop 权威指南》(第 3 版)的一致性，本书对于术语的翻译处理遵从实践做法；对于新出现的术语，从实践人员的实际使用情况考虑，采取翻译或保留英文名称两种处理方式，希望可以增加读者对新术语的理解。由于译者水平有限，译文中的不当之处也在所难免，真诚地希望广大读者批评与指正。

前言

数学科普作家马丁·加德纳(Martin Gardner)曾经在一次采访中谈到：

> “在我的世界里，只有微积分。这是我的专栏取得成功的奥秘。我花了很多时间才明白如何以大多数读者都能明白的方式将自己所知道的东西娓娓道来。”[①]

这也是我对 Hadoop 的诸多感受。它的内部工作机制非常复杂，是一个集分布式系统理论、实际工程和常识于一体的系统。而且，对门外汉而言，Hadoop 更像是“天外来客”。

但 Hadoop 其实并没有那么让人费解，抽丝剥茧，我们来看看它的“庐山真面目”。Hadoop 提供的用于处理大数据的工具都非常简单。如果说这些工具有一个共同的主题，那就是它们更抽象，为(有大量数据需要存储和分析却没有足够的时间、技能或者不想成为分布式系统专家的)程序员提供一套组件，使其能够利用 Hadoop 来构建一个处理数据的基础平台。

这样一个简单、通用的特性集，促使我在开始使用 Hadoop 时便明显感觉到 Hadoop 真的值得推广。但最开始的时候(2006 年初)，安装、配置和 Hadoop 应用编程是一门高深的艺术。之后，情况确实有所改善：文档增多了；示例增多了；碰到问题时，可以向大量活跃的邮件列表发邮件求助。对新手而言，最大的障碍是理解 Hadoop 有哪些能耐，它擅长什么，它如何使用。这些问题使我萌发了写作本书的动机。

Apache Hadoop 社区的发展来之不易。从本书的第 1 版发行以来，Hadoop 项目如雨后春笋般发展兴旺。“大数据”已成为大家耳熟能详的名词术

① 摘自“The science of fun”，网址为 *http://bit.ly/science_of_fun*。此文 2008 年 5 月 31 日发表于《卫报》。

语。[2]当前，软件在可用性、性能、可靠性、可扩展性和可管理性方面都实现了巨大的飞跃。在 Hadoop 平台上搭建和运行的应用增长迅猛。事实上，对任何一个人来说，跟踪这些发展动向都很困难。但为了让更多的人采用 Hadoop，我认为我们要让 Hadoop 更好用。这需要创建更多新的工具，集成更多的系统，创建新的增强型 API。我希望自己能够参与，同时也希望本书能够鼓励并吸引其他人也参与 Hadoop 项目。

说明

在文中讨论特定的 Java 类时，我常常会忽略包的名称以免啰嗦杂乱。如果想知道一个类在哪个包内，可以查阅 Hadoop 或相关项目的 Java API 文档(Apache Hadoop 主页 *http://hadoop.apache.org* 上有链接可以访问)。如果使用 IDE 编程，其自动补全机制(也称“自动完成机制”)能够帮助你找到你需要的东西。

与此类似，尽管偏离传统的编码规范，但如果要导入同一个包的多个类，程序可以使用星号通配符来节省空间(例如 `import org.apache.hadoop.io.*`)。

本书中的示例代码可以从本书网站下载，网址为 *http://www.hadoopbook.com/*。可以根据网页上的指示获取本书示例所用的数据集以及运行本书示例的详细说明、更新链接、额外的资源与我的博客。

第 4 版新增内容

第 4 版的主题是 Hadoop 2。Hadoop 2 系列发行版本是当前应用最活跃的系列，且包含 Hadoop 的最稳定的版本。

第 4 版新增的章节包括 YARN(第 4 章)、Parquet(第 13 章)、Flume(第 14 章)、Crunch(第 18 章)和 Spark(第 19 章)。此外，为了帮助读者更方便地阅读本书，第 1 章新增了一节“本书包含的内容”(参见 1.7 节)。

② 术语“大数据”在 2013 年被收入《牛津英语辞典》(Oxford English Dictionary)，网址为 *http://bit.ly/6_13_oed_update*。

第 4 版包括两个新的实例学习(第 22 章和第 23 章)：一个是关于 Hadoop 如何应用于医疗健康系统，另一个是关于将 Hadoop 技术如何应用于基因数据处理。旧版本中的实例学习可以在线查到，网址为 *http:/bit.ly/hadoop_tdg_prev*。

为了和 Hadoop 最新发行版本及其相关项目同步，第 4 版对原有章节进行了修订、更新和优化。

第 3 版新增内容

第 3 版概述 Apache Hadoop 1.x(以前的 0.20)系列发行版本，以及新近的 0.22 和 2.x(以前的 0.23)系列。除了少部分(文中有说明)例外，本书包含的所有范例都在这些版本上运行过。

第 3 版的大部分范例代码都使用了新的 MapReduce API。因为旧的 API 仍然应用很广，所以文中在讨论新的 API 时我们还会继续讨论它，使用旧 API 的对应范例代码可以到本书的配套网站下载。

Hadoop 2.0 最主要的变化是新增的 MapReduce 运行时 MapReduce 2，它建立在一个新的分布式资源管理系统之上，该系统称为 YARN。针对建立在 YARN 之上的 MapReduce，第 3 版增加了相关的介绍，包括它的工作机制(第 7 章)及如何运行(第 10 章)。

第 3 版还增加了更多对 MapReduce 的介绍，包括丰富的开发实践，比如用 Maven 打包 MapReduce 作业，设置用户的 Java 类路径，用 MRUnit 写测试等(这些内容都请参见第 6 章)。第 3 版还深入介绍了一些特性，如输出 committer 和分布式缓存(第 9 章)，任务内存监控(第 10 章)。第 3 版还新增了两小节内容，一节是关于如何写 MapReduce 作业来处理 Avro 数据(参见第 12 章)，另一节是关于如何在 Oozie 中运行一个简单的 MapReduce 工作流(参见第 6 章)。

关于 HDFS 的章节(第 3 章)，新增了对高可用性、联邦 HDFS、新的 WebHDFS 和 HttpFS 文件系统的介绍。

对 Pig，Hive，Sqoop 和 ZooKeeper 的相关介绍，第 3 版全部进行了相应的扩展，广泛介绍其最新发行版本中的新特性和变化。

此外，第 3 版还对第 2 版进行了彻底的更新、修订和优化。

第 2 版新增内容

《Hadoop 权威指南》(第 2 版)新增两章内容，分别介绍 Sqoop 和 Hive(第 15 章和第 17 章)，新增一个小节专门介绍 Avro(参见第 12 章)，补充了关于 Hadoop 新增安全特性的介绍(参见第 10 章)以及一个介绍如何使用 Hadoop 来分析海量网络图的新实例分析。

第 2 版继续介绍 Apache Hadoop 0.20 系列发行版本，因为当时最新、最稳定的发行版本。书中有时会提到一些最新发行版本中的一些新特性，但在首次介绍这些特性时，有说明具体的 Hadoop 版本号。

本书采用的约定

本书采用以下排版约定。

斜体

用于表明新的术语、URL、电子邮件地址、文件名和文件扩展名。

`等宽字体 Consolas`

用于程序清单，在正文段落中出现的程序元素(如变量或函数名)、数据库、数据类型、环境变量、语句和关键字也采用这样的字体。

`等宽字体 Consolas+加粗`

用于显示命令或应该由用户键入的其他文本。

`等宽字体 Consolas+斜体`

表明这里的文本需要替换为用户提供的值或其他由上下文确定的值。

这个图标表示通用的说明。

这个图标表示重要的指示或建议。

这个图标表示警告或需要注意的问题。

示例代码的使用

本书的补充材料（代码、示例及练习等）可以从本书网站(*http://www.hadoopbook.com*)或 GitHub(*https://github.com/tomwhite/hadoop-book/*)下载。

本书的目的是帮助读者完成工作。通常情况下，可以在你的程序或文档中使用本书中给出的代码。不必联系我们获得代码使用授权，除非你需要使用大量的代码。例如，在写程序的时候引用几段代码不需要向我们申请许可。但以光盘方式销售或重新发行 O'Reilly 书中的示例的确需要获得许可。引用本书或引用本书中的示例代码来回答问题也不需要申请许可。但是，如果要将本书中的大量范例代码加入你的产品文档，则需要申请许可。

我们欣赏你在引用时注明出处，但不强求。引用通常包括书名、作者、出版社和 ISBN，如“*Hadoop*: *The Definitive Guide*, Fourth Edition, by Tom White(O'Reilly).Copyright © 2015 Tom White，978-1-491-90163-2”。

如果觉得使用示例代码的情况不属于前面列出的合理使用或许可范围，请通过电子邮件联系我们，邮箱地址为 *permissions@oreilly.com*。

Safari Books Online

Safari Books Online(*www.safaribooksonline.com*)是一个按需定制的数字图书馆，以图书和视频的形式提供全球技术领域和经管领域内知名作者的专业作品。

专业技术人员、软件开发人员、网页设计人员、商务人员和创意专家将 Safari Books Online 用作自己开展研究、解决问题、学习和完成资格认证培训的重要来源。

Safari Books Online 为企业、政府部门、教育机构和个人提供广泛、灵活的计划和定价。

在这里，成员们通过一个可以全文检索的数据库中就能够访问数千种图书、培训视频和正式出版之前的书稿，这些内容提供商有 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 及其他上百家出版社。欢迎访问 Safari Books Online，了解更多详情。

联系我们

对于本书，如果有任何意见或疑问，请通过以下地址联系出版商：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室(100035)
奥莱利技术咨询(北京)有限公司

本书也有相关的网页，我们在上面列出了勘误表、范例以及其他一些信息。网址如下：

http://bit.ly/hadoop_tdg_4e(英文版)
http://www.oreilly.com.cn/book.php?bn=978-7-302-46513-3(中文版)

对本书做出评论或者询问技术问题，请发送 E-mail 至以下邮箱：

bookquestions@oreilly.com

如果希望获得关于本书、会议、资源中心和 O'Reilly 的更多信息，请访问以下网址：

http://www.oreilly.com
http://www.oreilly.com.cn

致谢

在本书写作期间，我仰赖于许多人的帮助，直接的或间接的。感谢 Hadoop 社区，我从中学到很多，这样的学习仍将继续。

特别感谢 Michael Stack 和 Jonathan Gray，HBase 这一章的内容就是他们写的。我还要感谢 Adrian Woodhead，Marc de Palol，Joydeep Sen Sarma，Ashish Thusoo，Andrzej Białecki，Stu Hood，Chris K. Wensel 和 Owen O'Malley，他们提供了学习实例。

感谢为草稿提出有用建议和改进建议的评审人：Raghu Angadi，Matt Biddulph，Christophe Bisciglia，Ryan Cox，Devaraj Das，Alex Dorman，Chris Douglas，Alan Gates，Lars George，Patrick Hunt，Aaron Kimball，Peter Krey，Hairong Kuang，Simon Maxen，Olga Natkovich，Benjamin Reed，Konstantin Shvachko，Allen Wittenauer，Matei Zaharia 和 Philip Zeyliger。Ajay Anand 组织本书的评审并使其顺利完成。Philip（“flip”）Komer 帮助我获得了 NCDC 气温数据，使本书示例很有特色。特别感谢 Owen O'Malley 和 Arun C. Murthy，他们为我清楚解释了 MapReduce 中 shuffle 的复杂过程。当然，如果有任何错误，得归咎于我。

对于第 2 版，我特别感谢 Jeff Bean，Doug Cutting，Glynn Durham，Alan Gates，Jeff Hammerbacher，Alex Kozlov，Ken Krugler，Jimmy Lin，Todd Lipcon，Sarah Sproehnle，Vinithra Varadharajan 和 Ian Wrigley，感谢他们仔细审阅本书，并提出宝贵的建议，同时也感谢对本书第 1 版提出勘误建议的读者。我也想感谢 Aaron Kimball 对 Sqoop 所做的贡献和 Philip（“flip”）Kromer 对图处理实例分析所做的贡献。

对于第 3 版，我想感谢 Alejandro Abdelnur，Eva Andreasson，Eli Collins，Doug Cutting，Patrick Hunt，Aaron Kimball，Aaron T. Myers，Brock Noland，Arvind Prabhakar，Ahmed Radwan 和 Tom Wheeler，感谢他们的反馈意见和建议。Rob Weltman 友善地对整本书提出了非常详细的反馈意见，这些意见和建议使得本书终稿的质量得以更上一层楼。此外，我还要向提交第 2 版勘误的所有读者表达最真挚的谢意。

对于第 4 版，我想感谢 Jodok Batlogg，Meghan Blanchette，Ryan Blue，Jarek Jarcec Cecho，Jules Damji，Dennis Dawson，Matthew Gast，Karthik Kambatla，Julien Le Dem，Brock Noland，Sandy Ryza，Akshai Sarma，Ben Spivey，Michael Stack，Kate Ting，Josh Walter，Josh Wills 和 Adrian Woodhead，感谢他们所有人非常宝贵的审阅反馈。Ryan Brush，Micah Whitacre 和 Matt Massie kindly 为第 4 版友情提供新的实例学习。再次感谢提交勘误的所有读者。

特别感谢 Doug Cutting 对我的鼓励、支持、友谊以及他为本书所写的序。

我还要感谢在本书写作期间以对话和邮件方式进行交流的其他人。

在本书第 1 版写到一半的时候，我加入了 Cloudera，我想感谢我的同事，他们为我提供了大量的帮助和支持，使我有充足的时间好好写书，并能及时交稿。

非常感谢我的编辑 Mike Loukides、Meghan Blanchette 及其 O'Reilly Media 的同事，他们在本书的准备阶段为我提供了很多帮助。Mike 和 Meghan 一直为我答疑解惑、审读我的初稿并帮助我如期完稿。

最后，写作是一项艰巨的任务，如果没有家人一如既往地支持，我是不可能完成这本的。我的妻子 Eliane，她不仅操持着整个家庭，还协助我，参与本书的审稿、编辑和跟进案例学习。还有我的女儿 Emilia 和 Lottie，她们一直都非常理解并支持我的工作，我期待有更多时间好好陪陪她们。

目录

第 I 部分　Hadoop 基础知识

第 1 章　初识 Hadoop 3

1.1　数据！数据！ 3
1.2　数据的存储与分析 5
1.3　查询所有数据 6
1.4　不仅仅是批处理 7
1.5　相较于其他系统的优势 8
1.5.1　关系型数据库管理系统 ... 8
1.5.2　网格计算 10
1.5.3　志愿计算 11
1.6　Apache Hadoop 发展简史 12
1.7　本书包含的内容 16

第 2 章　关于 MapReduce 19

2.1　气象数据集 19
2.2　使用 Unix 工具来分析数据..... 21
2.3　使用 Hadoop 来分析数据..... 22
2.3.1　map 和 reduce 23
2.3.2　Java MapReduce..... 24
2.4　横向扩展 31
2.4.1　数据流 31
2.4.2　combiner 函数..... 35
2.4.3　运行分布式的 MapReduce 作业 37
2.5　Hadoop Streaming 37
2.5.1　Ruby 版本..... 38
2.5.2　Python 版本 40

第 3 章　Hadoop 分布式文件系统.....42

3.1　HDFS 的设计..... 42
3.2　HDFS 的概念..... 44
3.2.1　数据块..... 44
3.2.2　namenode 和 datanode 45
3.2.3　块缓存 46
3.2.4　联邦 HDFS.....47
3.2.5　HDFS 的高可用性.....47
3.3　命令行接口 50
3.4　Hadoop 文件系统 52
3.5　Java 接口 56
3.5.1　从 Hadoop URL 读取数据 56
3.5.2　通过 FileSystem API 读取数据 58
3.5.3　写入数据.....61
3.5.4　目录 63
3.5.5　查询文件系统 63
3.5.6　删除数据.....68
3.6　数据流 68
3.6.1　剖析文件读取 68
3.6.2　剖析文件写入 71
3.6.3　一致模型.....74
3.7　通过 distcp 并行复制.....76

第 4 章　关于 YARN..... 78

4.1　剖析 YARN 应用运行机制 79
4.1.1　资源请求.....80
4.1.2　应用生命期 81
4.1.3　构建 YARN 应用 81
4.2　YARN 与 MapReduce 1 相比 82
4.3　YARN 中的调度 85
4.3.1　调度选项.....85
4.3.2　容量调度器配置.....87

4.3.3 公平调度器配置.....89
4.3.5 延迟调度.....93
4.3.5 主导资源公平性.....94
4.4 延伸阅读.....95
第 5 章 Hadoop 的 I/O 操作.....96
5.1 数据完整性.....96
5.1.1 HDFS 的数据完整性.....97
5.1.2 LocalFileSystem.....98
5.1.3 ChecksumFileSystem.....98
5.2 压缩.....99
5.2.1 codec.....100
5.2.2 压缩和输入分片.....105
5.2.3 在 MapReduce 中使用压缩.....106
5.3 序列化.....109
5.3.1 Writable 接口.....110
5.3.2 Writable 类.....112
5.3.3 实现定制的 Writable 集合.....121
5.3.4 序列化框架.....125
5.4 基于文件的数据结构.....127
5.4.1 关于 SequenceFile.....127
5.4.2 关于 MapFile.....135
5.4.3 其他文件格式和面向列的格式.....136

第 II 部分 关于 MapReduce

第 6 章 MapReduce 应用开发.....141
6.1 用于配置的 API.....142
6.1.1 资源合并.....143
6.1.2 变量扩展.....144
6.2 配置开发环境.....144
6.2.1 管理配置.....146
6.2.2 辅助类 GenericOptionsParser, Tool 和 ToolRunner.....149
6.3 用 MRUnit 来写单元测试.....152
6.3.1 关于 Mapper.....152
6.3.2 关于 Reducer.....156
6.4 本地运行测试数据.....156
6.4.1 在本地作业运行器上运行作业.....156
6.4.2 测试驱动程序.....158
6.5 在集群上运行.....160
6.5.1 打包作业.....160
6.5.2 启动作业.....162
6.5.3 MapReduce 的 Web 界面.....165
6.5.4 获取结果.....167
6.5.5 作业调试.....168
6.5.6 Hadoop 日志.....171
6.5.7 远程调试.....173
6.6 作业调优.....174
6.7 MapReduce 的工作流.....176
6.7.1 将问题分解成 MapReduce 作业.....177
6.7.2 关于 JobControl.....178
6.7.3 关于 Apache Oozie.....179
第 7 章 MapReduce 的工作机制.....184
7.1 剖析 MapReduce 作业运行机制.....184
7.1.1 作业的提交.....185
7.1.2 作业的初始化.....186
7.1.3 任务的分配.....187
7.1.4 任务的执行.....188
7.1.5 进度和状态的更新.....189
7.1.6 作业的完成.....191
7.2 失败.....191
7.2.1 任务运行失败.....191
7.2.2 application master 运行失败.....193
7.2.3 节点管理器运行失败.....193
7.2.4 资源管理器运行失败.....194
7.3 shuffle 和排序.....195
7.3.1 map 端.....195

7.3.2 reduce 端 197
7.3.3 配置调优 199
7.4 任务的执行 201
7.4.1 任务执行环境 201
7.4.2 推测执行 202
7.4.3 关于 OutputCommitters 204

第 8 章 MapReduce 的类型与格式 207

8.1 MapReduce 的类型 207
8.1.1 默认的 MapReduce 作业 212
8.1.2 默认的 Streaming 作业 216
8.2 输入格式 218
8.2.1 输入分片与记录 218
8.2.2 文本输入 229
8.2.3 二进制输入 233
8.2.4 多个输入 234
8.2.5 数据库输入(和输出).....235
8.3 输出格式 236
8.3.1 文本输出 236
8.3.2 二进制输出 237
8.3.3 多个输出 237
8.3.4 延迟输出 242
8.3.5 数据库输出 242

第 9 章 MapReduce 的特性 243

9.1 计数器 243
9.1.1 内置计数器 243
9.1.2 用户定义的 Java 计数器 248
9.1.3 用户定义的 Streaming 计数器 251
9.2 排序 252
9.2.1 准备 252
9.2.2 部分排序 253
9.2.3 全排序 255
9.2.4 辅助排序 259
9.3 连接 264
9.3.1 map 端连接 266
9.3.2 reduce 端连接 266
9.4 边数据分布 270
9.4.1 利用 JobConf 来配置作业 270
9.4.2 分布式缓存 270
9.5 MapReduce 库类 276

第Ⅲ部分 Hadoop 的操作

第 10 章 构建 Hadoop 集群 279

10.1 集群规范 280
10.1.1 集群规模 281
10.1.2 网络拓扑 282
10.2 集群的构建和安装 284
10.2.1 安装 Java 284
10.2.2 创建 Unix 用户账号 ...284
10.2.3 安装 Hadoop 284
10.2.4 SSH 配置 285
10.2.5 配置 Hadoop 286
10.2.6 格式化 HDFS 文件系统 286
10.2.7 启动和停止守护进程 286
10.2.8 创建用户目录 288
10.3 Hadoop 配置 288
10.3.1 配置管理 289
10.3.2 环境设置 290
10.3.3 Hadoop 守护进程的关键属性 293
10.3.4 Hadoop 守护进程的地址和端口 300
10.3.5 Hadoop 的其他属性 303
10.4 安全性 305
10.4.1 Kerberos 和 Hadoop 306
10.4.2 委托令牌 308
10.4.3 其他安全性改进 309

10.5 利用基准评测程序测试 Hadoop 集群 311
10.5.1 Hadoop 基准评测程序 311
10.5.2 用户作业 313

第 11 章 管理 Hadoop 314

11.1 HDFS 314
11.1.1 永久性数据结构 314
11.1.2 安全模式 320
11.1.3 日志审计 322
11.1.4 工具 322
11.2 监控 327
11.2.1 日志 327
11.2.2 度量和 JMX(Java 管理扩展) 328
11.3 维护 329
11.3.1 日常管理过程 329
11.3.2 委任和解除节点 331
11.3.3 升级 334

第Ⅳ部分 Hadoop 相关开源项目

第 12 章 关于 Avro 341

12.1 Avro 数据类型和模式 342
12.2 内存中的序列化和反序列化特定 API 347
12.3 Avro 数据文件 349
12.4 互操作性 351
12.4.1 Python API 351
12.4.2 Avro 工具集 352
12.5 模式解析 352
12.6 排列顺序 354
12.7 关于 Avro MapReduce 356
12.8 使用 Avro MapReduce 进行排序 359
12.9 其他语言的 Avro 362

第 13 章 关于 Parquet 363

13.1 数据模型 364
13.2 Parquet 文件格式 367
13.3 Parquet 的配置 368
13.4 Parquet 文件的读/写 369
13.4.1 Avro、Protocol Buffers 和 Thrift 371
13.4.2 投影模式和读取模式 373
13.5 Parquet MapReduce 374

第 14 章 关于 Flume 377

14.1 安装 Flume 378
14.2 示例 378
14.3 事务和可靠性 380
14.4 HDFS Sink 382
14.5 扇出 385
14.5.1 交付保证 386
14.5.2 复制和复用选择器 387
14.6 通过代理层分发 387
14.7 Sink 组 391
14.8 Flume 与应用程序的集成 395
14.9 组件编目 395
14.10 延伸阅读 397

第 15 章 关于 Sqoop 398

15.1 获取 Sqoop 398
15.2 Sqoop 连接器 400
15.3 一个导入的例子 401
15.4 生成代码 404
15.5 深入了解数据库导入 405
15.5.1 导入控制 407
15.5.2 导入和一致性 408
15.5.3 增量导入 408
15.5.4 直接模式导入 408
15.6 使用导入的数据 409
15.7 导入大对象 412

15.8 执行导出 414
15.9 深入了解导出功能.....416
15.9.1 导出与事务.....417
15.9.2 导出和 SequenceFile...418
15.10 延伸阅读 419
第 16 章 关于 Pig.....420
16.1 安装与运行 Pig.....421
16.1.1 执行类型 422
16.1.2 运行 Pig 程序 423
16.1.3 Grunt 424
16.1.4 Pig Latin 编辑器 424
16.2 示例.....425
16.3 与数据库进行比较.....428
16.4 PigLatin 429
16.4.1 结构.....430
16.4.2 语句.....431
16.4.3 表达式.....436
16.4.4 类型.....437
16.4.5 模式.....438
16.4.6 函数.....443
16.4.7 宏 445
16.5 用户自定义函数 446
16.5.1 过滤 UDF 447
16.5.2 计算 UDF 450
16.5.3 加载 UDF 452
16.6 数据处理操作.....455
16.6.1 数据的加载和存储.....455
16.6.2 数据的过滤.....455
16.6.3 数据的分组与连接.....458
16.6.4 数据的排序.....463
16.6.5 数据的组合和切分.....465
16.7 Pig 实战.....465
16.7.1 并行处理 465
16.7.2 匿名关系 466
16.7.3 参数代换 467
16.8 延伸阅读 468
第 17 章 关于 Hive.....469
17.1 安装 Hive 470
Hive 的 shell 环境.....471
17.2 示例 472
17.3 运行 Hive 473
17.3.1 配置 Hive 473
17.3.2 Hive 服务 476
17.3.3 Metastore.....478
17.4 Hive 与传统数据库相比 480
17.4.1 读时模式 vs.写时模式 480
17.4.2 更新、事务和索引 481
17.4.3 其他 SQL-on-Hadoop 技术 482
17.5 HiveQL.....483
17.5.1 数据类型.....484
17.5.2 操作与函数 487
17.6 表.....488
17.6.1 托管表和外部表.....488
17.6.2 分区和桶.....490
17.6.3 存储格式.....494
17.6.4 导入数据.....498
17.6.5 表的修改.....500
17.6.6 表的丢弃.....501
17.7 查询数据.....501
17.7.1 排序和聚集 501
17.7.2 MapReduce 脚本.....502
17.7.3 连接 503
17.7.4 子查询.....506
17.7.5 视图 507
17.8 用户定义函数.....508
17.8.1 写 UDF.....510
17.8.2 写 UDAF 512
17.9 延伸阅读.....516
第 18 章 关于 Crunch.....517
18.1 示例 518
18.2 Crunch 核心 API.....521
18.2.1 基本操作.....522
18.2.2 类型 527
18.2.3 源和目标.....530
18.2.4 函数 532
18.2.5 物化 535
18.3 管线执行.....537

18.3.1 运行管线 538
18.3.2 停止管线 539
18.3.3 查看 Crunch 计划 540
18.3.4 迭代算法 543
18.3.5 给管线设置检查点 544
18.4 Crunch 库 545
18.5 延伸阅读 547

第 19 章 关于 Spark 548

19.1 安装 Spark 549
19.2 示例 549
19.2.1 Spark 应用、作业、阶段和任务 551
19.2.2 Scala 独立应用 552
19.2.3 Java 示例 553
19.2.4 Python 示例 554
19.3 弹性分布式数据集 555
19.3.1 创建 555
19.3.2 转换和动作 557
19.3.3 持久化 561
19.3.4 序列化 563
19.4 共享变量 564
19.4.1 广播变量 564
19.4.2 累加器 565
19.5 剖析 Spark 作业运行机制 565
19.5.1 作业提交 566
19.5.2 DAG 的构建 566
19.5.3 任务调度 569
19.5.4 任务执行 570
19.6 执行器和集群管理器 570
19.7 延伸阅读 574

第 20 章 关于 HBase 575

20.1 HBase 基础 575
20.2 概念 576
20.2.1 数据模型的“旋风之旅” 576
20.2.2 实现 578
20.3 安装 581
20.4 客户端 584
20.4.1 Java 584
20.4.2 MapReduce 588
20.4.3 REST 和 Thrift 589
20.5 创建在线查询应用 589
20.5.1 模式设计 590
20.5.2 加载数据 591
20.5.3 在线查询 595
20.6 HBase 和 RDBMS 的比较 598
20.6.1 成功的服务 599
20.6.2 HBase 600
20.7 Praxis 601
20.7.1 HDFS 601
20.7.2 用户界面 602
20.7.3 度量 602
20.7.4 计数器 602
20.8 延伸阅读 602

第 21 章 关于 ZooKeeper 604

21.1 安装和运行 ZooKeeper 605
21.2 示例 607
21.2.1 ZooKeeper 中的组成员关系 608
21.2.2 创建组 608
21.2.3 加入组 611
21.2.4 列出组成员 612
21.2.5 删除组 614
21.3 ZooKeeper 服务 615
21.3.1 数据模型 615
21.3.2 操作 618
21.3.3 实现 622
21.3.4 一致性 624
21.3.5 会话 626
21.3.6 状态 628
21.4 使用 ZooKeeper 来构建应用 629
21.4.1 配置服务 629
21.4.2 可复原的 ZooKeeper 应用 633
21.4.3 锁服务 637

21.4.4 更多分布式数据结构和协议 639
21.5 生产环境中的 ZooKeeper 640
21.5.1 可恢复性和性能 641
21.5.2 配置 642
21.6 延伸阅读 643

第V部分 案例学习

第 22 章 医疗公司塞纳(Cerner)的可聚合数据 647

22.1 从多 CPU 到语义集成 647
22.2 进入 Apache Crunch.....648
22.3 建立全貌 649
22.4 集成健康医疗数据.....651
22.5 框架之上的可组合性.....654
22.6 下一步 655

第 23 章 生物数据科学：用软件拯救生命.....657

23.1 DNA 的结构 659
23.2 遗传密码：将 DNA 字符转译为蛋白质 660
22.3 将 DNA 想象成源代码.....661
23.4 人类基因组计划和参考基因组 663
22.5 DNA 测序和比对 664
23.6 ADAM，一个可扩展的基因组分析平台 666
23.7 使用 Avro 接口描述语言进行自然语言编程 666
23.8 使用 Parquet 进行面向列的存取.....668
23.9 一个简单例子：用 Spark 和 ADAM 做 *k*-mer 计数 669
23.10 从个性化广告到个性化医疗 672
23.11 联系我们 673

第 24 章 开源项目 Cascading.....674

24.1 字段、元组和管道 675
24.2 操作 678
24.3 Taps，Schemes 和 Flows 680
24.4 Cascading 实践应用.....681
24.5 灵活性 684
24.6 ShareThis 中的 Hadoop 和 Cascading 685
24.7 总结 689

附录 A 安装 Apache Hadoop.....691

附录 B 关于 CDH.....697

附录 C 准备 NCDC 气象数据 699

附录 D 新版和旧版 Java MapReduce API 702

第 I 部分
Hadoop 基础知识

第 1 章　初识 Hadoop
第 2 章　关于 MapReduce
第 3 章　Hadoop 分布式文件系统
第 4 章　关于 YARN
第 5 章　Hadoop 的 I/O 操作

第 1 章

初识 Hadoop

“在古时候，人们用牛来拉重物。当一头牛拉不动一根圆木时，人们从来没有考虑过要想方设法培育出一种更强壮的牛。同理，我们也不该想方设法打造什么超级计算机，而应该千方百计综合利用更多计算机来解决问题。”

——葛蕾丝·霍珀(Grace Hopper)[①]

1.1 数据！数据！

我们生活在这个数据大爆炸的时代，很难估算全球电子设备中存储的数据总共有多少。国际数据公司(IDC)曾经发布报告称，2013 年数字世界(digital universe)项目统计得出全球数据总量为 4.4ZB(zettabyte)并预测在 2020 年将达到 44 ZB。[②]1 ZB 等于 10^{21} 字节，等于 1000 EB(exabytes)，1 000 000 PB (petabytes)，等于大家更熟悉的 10 亿 TB(terrabytes)！这远远超过了全世界每人一块硬盘中所能保存的数据总量！

① 编注：全名 Grace Murray Hopper(1906—1992)，生于美国纽约州纽约市，取得瓦萨学院数学物理双学士学位以及耶鲁大学硕士博士学位。1943 年，志愿加入美国海军后备军团，1983 年被当时的里根总统特别任命为海军准将。她是一名计算机科学家，世界最早的一批程序员，也是最早的女性程序员。她是 Harvard Mark I 的第一个专职程序员，创造了现代第一个编译器 A-0 系统以及第一个高级商用计算机程序语言 COBOL，被誉为“COBOL 之母”。著名的计算机术语 debug(调试排错)开始于她受到从电脑中驱除飞蛾的启发，因而她也被冠以“debug 之母”的称号。她也是 Y2K 危机的创造者。她培养了很多编程语言专家，被誉为“不可思议的葛蕾丝”(Amazing Grace)。她有一句名言深入人心：“停在港口的船很安全，但那不是我们造船的目的(A ship in port is safe, but that is not what ships are built for)。”

② 这个统计数据引自研究报告“The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things”(充满机遇的数字世界：丰富的数据与物联网日益增长的价值)，网址为 *http://bit.ly/digital_universe*。

数据“洪流”有很多来源。以下面列出的为例：[①]

- 纽约证交所每天产生的交易数据大约在 4 TB 至 5 TB 之间；
- 脸谱网(Facebook)存储的照片超过 2400 亿张，并以每月至少 7PB 的速度增长；
- 家谱网站 Ancestry.com 存储的数据约为 10 PB；
- 互联网档案馆(The Internet Archive)存储的数据约为 18.5PB；
- 瑞士日内瓦附近的大型强子对撞机每年产生的数据约为 30 PB。

还有其他大量的数据。但是你可能会想它对自己又有哪些影响呢？地球人都知道，大部分数据都戒备森严，被锁存在一些大型互联网公司(如搜索引擎公司)或科学机构与金融机构中。大数据的出现会影响到小机构和个人吗？

我个人是这样认为的。以照片为例，我妻子的爷爷是一个骨灰级的摄影爱好者。他成年之后，一直都在拍照。他的整个相册，包括普通胶片、幻灯片以及 35mm 胶片，在扫描成高分辨率的图片之后，大约有 10 GB。相比之下，单单是 2008 年，我家用数码相机拍的照片就有 5 GB。对照爷爷的照片生成速度，我家是他老人家的 35 倍！并且，而且这个速度还在不断增长中，因为现在拍照片真的是越来越容易。

有一种情况更普遍，个人产生的数据正在快速增长。微软研究院的 MyLifeBits 项目 (*http://bit.ly/ms_mylifebits*)显示，在不久的将来，个人信息档案将日益普及。MyLifeBits 的一个实验是获取和保存个人的对外联系情况(包括电话、邮件和文件)，供日后存取。收集的数据中包括每分钟拍摄的照片等，数据量每月约为 1 GB。当存储成本急剧下降以至于可以存储音频和视频时，MyLifeBits 项目未来的存储数据量将是现在的很多倍。

保存个人成长过程中产生的所有数据似乎逐渐成为主流，但更重要的也许是，作为物联网一部分的机器设备产生的数据可能远远超过我们个人所产生的数据。机器日志、RFID 读卡器、传感器网络、车载 GPS 和零售交易数据等，所有这些都将产生巨量的数据。

① 所有的数据都是 2013 年或 2014 年的，更多信息可以参见 Tom Groenfeldt 的文章“At NYSE, The Data Deluge Overwhelms Traditional Databases”(*http://bit.ly/nyse_data_deluge*)；Rich Miller 的文章“Facebook Builds Exabyte Data Centers for Cold Storage”(*http://bit.ly/facebook_exabyte*)；Ancestry.com 网站的“*Company Facts*”(*http://corporate.ancestry.com/press/company-facts/*)；Archive.org 网站的“Petabox”(*https://archive.org/web/petabox.php*)；Worldwide LHC Computing Grid 项目的欢迎页面(*http://wlcg.web.cern.ch/*)。

在网上公开发布的数据也在逐年增加中。组织或企业，要想在未来取得成功，不仅需要管理好自己的数据，更需要从其他组织或企业的数据中获取有价值的信息。

这方面的先锋有 Amazon Web Services(*http://aws.amazon.com/public-data-sets/*) 和 Infochimps.org(*http://infochimps.org/*)，它们所发布的共享数据集，正在促进信息共享(information commons)，供所有人自由下载和分析（或者只需要一个合理的价格)。不同来源的信息在经过混搭和处理之后，会带来意外的效果和我们今天难以想象的应用。

以 Astrometry.net(*http://astrometry.net*)为例，主要查看和分析 Flickr 网站上天体测量兴趣小组所拍摄的星空照片。它对每一张照片进行分析并能辨别出它来自星空或其他天体(例如恒星和星系等)的哪一部分。虽然这项研究尚处于试验阶段，但也表明如果可用的数据足够多(在本例中，为加有标签的图片数据)，通过它们而产生的后续应用也许会超乎这些拍照片的人最初的想象（图片分析)。

有句话说得好："大数据胜于好算法。" 意思是说对于某些应用（譬如根据以往的偏好来推荐电影和音乐)，不论算法有多牛，基于小数据的推荐效果往往都不如基于大量可用数据的一般算法的推荐效果。[①]

现在，我们已经有了大量数据，这是个好消息。但不幸的是，我们必须想方设法好好地存储和分析这些数据。

1.2 数据的存储与分析

我们遇到的问题很简单：在硬盘存储容量多年来不断提升的同时，访问速度(硬盘数据读取速度)却没有与时俱进。1990 年，一个普通硬盘可以存储 1370 MB 数据，传输速度为 4.4 MB/s[②]，因此只需要 5 分钟就可以读完整个硬盘中的数据。20 年过去了，1 TB 的硬盘成为主流，但其数据传输速度约为 100 MB/s，读完整个硬盘中的数据至少得花 2.5 个小时。

① 引自 Anand Rajaraman 的一篇博客文章，标题为"More data usually beats better algorithms"(大数据胜过好算法)，网址为 *http://bit.ly/more_data*，该文介绍了"Netflix Challenge"(Negfix 挑战大赛)。Alon Halevy，Peter Norvig(谷歌研究主管)和 Fernando Pereira 在他们的一篇文章中也提出了类似的观点，标题为"The Unreasonable Effectiveness of Data"(数据的非理性效果)，网址为 *http://bit.ly/ unreasonable_effect*，发表于 *IEEE Intelligent Systems* 2009 年 3/4 月合刊。

② 这些规格对应的是希捷 ST-41600n 硬盘。

读完整个硬盘中的数据需要更长时间，写入数据就别提了。一个很简单的减少读取时间的办法是同时从多个硬盘上读数据。试想，如果我们有 100 个硬盘，每个硬盘存储 1%的数据，并行读取，那么不到两分钟就可以读完所有数据。

仅使用硬盘容量的 1%似乎很浪费。但是我们可以存储 100 个数据集，每个数据集 1 TB，并实现共享硬盘的读取。可以想象，用户肯定很乐于通过硬盘共享来缩短数据分析时间；并且，从统计角度来看，用户的分析工作都是在不同时间点进行的，所以彼此之间的干扰并不太大。

虽然如此，但要对多个硬盘中的数据并行进行读/写数据，还有更多问题要解决。

第一个需要解决的是硬件故障问题。一旦开始使用多个硬件，其中个别硬件就很有可能发生故障。为了避免数据丢失，最常见的做法是复制(replication)：系统保存数据的复本(replica)，一旦有系统发生故障，就可以使用另外保存的复本。例如，冗余硬盘阵列(RAID)就是按这个原理实现的，另外，Hadoop 的文件系统(Hadoop Distributed FileSystem, HDFS)也是一类，不过它采取的方法稍有不同，详见后文的描述。

第二个问题是大多数分析任务需要以某种方式结合大部分数据来共同完成分析，即从一个硬盘读取的数据可能需要与从另外 99 个硬盘中读取的数据结合使用。各种分布式系统允许结合不同来源的数据进行分析，但保证其正确性是一个非常大的挑战。MapReduce 提出一个编程模型，该模型抽象出这些硬盘读/写问题并将其转换为对一个数据集(由键-值对组成)的计算。后文将详细讨论这个模型，这样的计算由 map 和 reduce 两部分组成，而且只有这两部分提供对外的接口。与 HDFS 类似，MapReduce 自身也有很高的可靠性。

简而言之，Hadoop 为我们提供了一个可靠的且可扩展的存储和分析平台。此外，由于 Hadoop 运行在商用硬件上且是开源的，因而可以说 Hadoop 的使用成本是在可承受范围内的。

1.3 查询所有数据

MapReduce 看似采用了一种蛮力方法。每个查询需要处理整个数据集或至少一个数据集的绝大部分。但反过来想，这也正是它的能力。MapReduce 是一个批量查询处理器，能够在合理的时间范围内处理针对整个数据集的动态查询。它改变了我们对数据的传统看法，解放了以前只是保存在磁带和硬盘上的数据。它让我们

有机会对数据进行创新。以前需要很长时间处理才能获得结果的问题，到现在变得顷刻之间就迎刃而解，同时还可以引发新的问题和新的见解。

例如，Rackspace 公司的邮件部门 Mailtrust 就用 Hadoop 来处理邮件日志。他们写了一条特别的查询用于帮助找出用户的地理分布。他们是这么描述的："这些数据非常有用，我们每月运行一次 MapReduce 任务来帮助我们决定扩容时将新的邮件服务器放在哪些 Rackspace 数据中心。"

通过整合好几百 GB 的数据，用工具来分析这些数据，Rackspace 的工程师能够对以往没有注意到的数据有所理解，甚至还运用这些信息来改善现有的服务。

1.4 不仅仅是批处理

从 MapReduce 的所有长处来看，它基本上是一个批处理系统，并不适合交互式分析。你不可能执行一条查询并在几秒内或更短的时间内得到结果。典型情况下，执行查询需要几分钟或更多时间。因此，MapReduce 更适合那种没有用户在现场等待查询结果的离线使用场景。

然而，从最初的原型出现以来，Hadoop 的发展已经超越了批处理本身。实际上，名词"Hadoop"有时被用于指代一个更大的、多个项目组成的生态系统，而不仅仅是 HDFS 和 MapReduce。这些项目都属于分布式计算和大规模数据处理范畴。这些项目中有许多都是由 Apache 软件基金会管理，该基金会为开源软件项目社区提供支持，其中包括最初的 HTTP server 项目(基金会的名称也来源于这个项目)。

第一个提供在线访问的组件是 HBase，一种使用 HDFS 做底层存储的键值存储模型。HBase 不仅提供对单行的在线读/写访问，还提供对数据块读/写的批操作，这对于在 HBase 上构建应用来说是一种很好的解决方案。

Hadoop 2 中 YARN(Yet Another Resource Negotiator)的出现意味着 Hadoop 有了新处理模型。YARN 是一个集群资源管理系统，允许任何一个分布式程序(不仅仅是 MapReduce)基于 Hadoop 集群的数据而运行。

在过去的几年中，出现了许多不同的、能与 Hadoop 协同工作的处理模式。以下是一些例子。

Interactive SQL(交互式 SQL)

利用 MapReduce 进行分发并使用一个分布式查询引擎，使得在 Hadoop 上获得 SQL 查询低延迟响应的同时还能保持对大数据集规模的可扩展性。这个引擎使用指定的“总是开启(always on)”守护进程(如同 impala)或容器重用(如同 Tez 上的 Hive)。

Iterative processing(迭代处理)

许多算法，例如机器学习算法，自身具有迭代性，因此和那种每次迭代都从硬盘加载的方式相比，这种在内存中保存每次中间结果集的方式更加高效。MapReduce 的架构不允许这样，但如果使用 Spark 就会比较直接，它在使用数据集方面展现了一种高度探究的风格。

Stream processing(流处理)

流系统，例如 Storm，Spark Streaming 或 Samza 使得在无边界数据流上运行实时、分布式的计算，并向 Hadoop 存储系统或外部系统发布结果成为可能。

Search(搜索)

Solr 搜索平台能够在 Hadoop 集群上运行，当文档加入 HDFS 后就可对其进行索引，且根据 HDFS 中存储的索引为搜索查询提供服务。

无论 Hadoop 上出现了多少不同的处理框架，就批处理而言，MapReduce 仍然有着一席之地。MapReduce 提出的一些概念更具有通用性(例如，输入格式、数据集分片等)，因此最好是能够了解 MapReduce 的工作机制。

1.5 相较于其他系统的优势

Hadoop 不是历史上第一个用于数据存储和分析的分布式系统，但是 Hadoop 的一些特性将它和其他那些看上去类似的系统区别开来。接下来我们将对此予以介绍。

1.5.1 关系型数据库管理系统

为什么不能用配有大量硬盘的数据库来进行大规模数据分析？我们为什么需要

Hadoop?

这两个问题的答案来自于计算机硬盘的另一个发展趋势：寻址时间的提升远远不敌于传输速率的提升。寻址是将磁头移动到特定硬盘位置进行读/写操作的过程。它是导致硬盘操作延迟的主要原因，而传输速率取决于硬盘的带宽。

如果数据访问模式中包含大量的硬盘寻址，那么读取大量数据集就必然会花更长的时间(相较于流数据读取模式，流读取主要取决于传输速率)。另一方面，如果数据库系统只更新一小部分记录，那么传统的 B 树(关系型数据库中使用的一种数据结构，受限于寻址的速率)就更有优势。但数据库系统如果有大量数据更新时，B 树的效率就明显落后于 MapReduce，因为需要使用“排序/合并”(sort/merge)来重建数据库。

在许多情况下，可以将 MapReduce 视为关系型数据库管理系统的补充。两个系统之间的差异如表 1-1 所示。MapReduce 比较适合解决需要以批处理方式分析整个数据集的问题，尤其是一些特定目的的分析。RDBMS 适用于索引后数据集的点查询(point query)和更新，建立索引的数据库系统能够提供对小规模数据的低延迟数据检索和快速更新。MapReduce 适合一次写入、多次读取数据的应用，关系型数据库则更适合持续更新的数据集。[①]

表 1-1. 关系型数据库和 MapReduce 的比较

	传统的关系型数据库	MapReduce
数据大小	GB	PB
数据存取	交互式和批处理	批处理
更新	多次读/写	一次写入，多次读取
事务	ACID	无
结构	写时模式	读时模式
完整性	高	低
横向扩展	非线性的	线性的

① 2007 年 1 月，数据库理论专家 David J. DeWitt 和 Michael Stonebraker 发表的论文引发了一场激烈的口水大战，论文标题为“MapReduce: A major step backwards”(MapReduce：一个历史性的大倒退)，原文可参见 *http://bit_ly/step_backwards*。在文中，他们认为 MapReduce 不宜取代关系型数据库。许多评论认为这是一种错误的比较，详情可参见 Mark C. Chu-Carroll 的文章，标题为“Databasesare hammers; MapReduce is a screwdriver”(如果说数据库是锤子，MapReduce 则是螺丝刀)，原文可以参见 *http://bit_ly/dbs_are_hammers*。DeWitt 与 Stonebraker 随后以“MapReduce II”一文阐述了其他人的观点。

然而，关系型数据库和 Hadoop 系统之间的区别是模糊的。一方面，关系型数据库已经开始吸收 Hadoop 的一些思想，另一方面，诸如 Hive 这样的 Hadoop 系统不仅变得更具交互性(通过从 MapReduce 中脱离出来)，而且增加了索引和事务这样的特性，使其看上去更像传统的关系型数据库。

Hadoop 和关系型数据库的另一个区别在于它们所操作的数据集的结构化程度。**结构化数据**(structured data)是具有既定格式的实体化数据，如 XML 文档或满足特定预定义格式的数据库表。这是 RDBMS 包括的内容。另一方面，**半结构化数据**(semi-structured data)比较松散，虽然可能有格式，但经常被忽略，所以它只能作为对数据结构的一般性指导。例如电子表格，它在结构上是由单元格组成的网格，但是每个单元格内可以保存任何形式的数据。**非结构化数据**(unstructured data)没有什么特别的内部结构，例如纯文本或图像数据。Hadoop 对非结构化或半结构化数据非常有效，因为它是在处理数据时才对数据进行解释(即所谓的“读时模式”)。这种模式在提供灵活性的同时避免了 RDBMS 数据加载阶段带来的高开销，因为在 Hadoop 中仅仅是一个文件拷贝操作。

关系型数据往往是**规范的**(normalized)，以保持其数据的完整性且不含冗余。规范给 Hadoop 处理带来了问题，因为它使记录读取成为非本地操作，而 Hadoop 的核心假设之一偏偏就是可以进行(高速的)流读/写操作。

Web 服务器日志是典型的非规范化数据记录(例如，每次都需要记录客户端主机全名，这会导致同一客户端的全名可能多次出现)，这也是 Hadoop 非常适用于分析各种日志文件的原因之一。注意，Hadoop 也能够做连接(join)操作，只不过这种操作没有在关系型数据库中用的多。

MapReduce 以及 Hadoop 中其他的处理模型是可以随着数据规模线性伸缩的。对数据分区后，函数原语(如 map 和 reduce)能够在各个分区上并行工作。这意味着，如果输入的数据量是原来的两倍，那么作业的运行时间也需要两倍。但如果集群规模扩展为原来的两倍，那么作业的运行速度却仍然与原来一样快。SQL 查询一般不具备该特性。

1.5.2 网格计算

高性能计算(High Performance Computing，HPC)和**网格计算**(Grid Computing)组织多年以来一直在研究大规模数据处理，主要使用类似于消息传递接口(Message Passing Interface，MPI)的 API。从广义上讲，高性能计算采用的方法是将作业分

散到集群的各台机器上，这些机器访问存储区域网络(SAN)所组成的共享文件系统。这比较适用于计算密集型的作业，但如果节点需要访问的数据量更庞大(高达几百 GB，Hadoop 开始施展它的魔法)，很多计算节点就会因为网络带宽的瓶颈问题而不得不闲下来等数据。

Hadoop 尽量在计算节点上存储数据，以实现数据的本地快速访问。[1]**数据本地化**(data locality)特性是 Hadoop 数据处理的核心，并因此而获得良好的性能。意识到网络带宽是数据中心环境最珍贵的资源(到处复制数据很容易耗尽网络带宽)之后，Hadoop 通过显式网络拓扑结构来保留网络带宽。注意，这种排列方式并没有降低 Hadoop 对计算密集型数据进行分析的能力。

虽然 MPI 赋予程序员很大的控制权，但需要程序员显式处理数据流机制，包括用 C 语言构造底层的功能模块(例如套接字)和高层的数据分析算法。而 Hadoop 则在更高层次上执行任务，即程序员仅从数据模型(如 MapReduce 的键-值对)的角度考虑任务的执行，与此同时，数据流仍然是隐性的。

在大规模分布式计算环境下，协调各个进程的执行是一个很大的挑战。最困难的是合理处理系统的部分失效问题(在不知道一个远程进程是否挂了的情况下)同时还需要继续完成整个计算。有了 MapReduce 这样的分布式处理框架，程序员不必操心系统失效的问题，因为框架能够检测到失败的任务并重新在正常的机器上执行。正因为采用的是**无共享**(shared-nothing)框架，MapReduce 才能够呈现出这种特性，这意味着各个任务之间是彼此独立的。[2]因此，从程序员的角度来看，任务的执行顺序无关紧要。相比之下，MPI 程序必须显式管理自己的检查点和恢复机制，虽然赋予程序员的控制权加大了，但编程的难度也增加了。

1.5.3 志愿计算

第一次听说 Hadoop 和 MapReduce 的时候，人们经常会问这个问题："它们和 SETI@home 有什么不同？" SETI 全称为 Search for Extra-Terrestrial Intelligence(搜

① 1998 年图灵奖得主 Jim Gray 在 2003 年 3 月发表的"Distributed Computing Economics"(分布式计算经济学)一文中，率先提出这个结论：数据处理应该在离数据本身比较近的地方进行，因为这样有利于降低成本，尤其是网络带宽消费所造成的成本。原文网址为 *http://bit.ly/dist_comp_econ*。

② 这里讲得太简单了一点，因为 MapReduce 系统本身控制着 mapper 输出结果传给 reducer 的过程，所以在这种情况下，重新运行 reducer 比重新运行 mapper 更要小心，因为 reducer 需要获取必要的 mapper 输出结果，如果没有，必须再次运行对应的 mapper，重新生成输出结果。

索外星智慧生命)，项目名称为 SETI@home(*http://setiathome. berkeley.edu/*)。在该项目中，志愿者把自己计算机 CPU 的空闲时间贡献出来分析无线天文望远镜的数据，借此寻找外星智慧生命信号。SETI@home 是所有志愿计算项目中最有名的，其他还有“搜索大素数”(Great Internet Mersenne Prime Search)项目与 Folding@home 项目[①](了解蛋白质构成及其与疾病之间的关系)。

志愿计算项目将问题分成很多块，每一块称为一个**工作单元**(work unit)，发到世界各地的计算机上进行分析。例如，SETI@home 的工作单元是 0.35 MB 无线电望远镜数据，要对这等大小的数据量进行分析，一台普通计算机需要几个小时或几天时间才能完成。完成分析后，结果发送回服务器，客户端随后再获得另一个工作单元。为防止欺骗，每个工作单元要发送到 3 台不同的机器上执行，而且收到的结果中至少有两个相同才会被接受。

从表面上看，SETI@home 与 MapReduce 好像差不多(将问题分解为独立的小块，然后并行进行计算)，但事实上还是有很多明显的差异。SETI@home 问题是 CPU 高度密集的，比较适合在全球成千上万台计算机上运行[②]，因为计算所花的时间远远超过工作单元数据的传输时间。也就是说，志愿者贡献的是 CPU 周期，而不是网络带宽。

MapReduce 有三大设计目标：(1)为只需要短短几分钟或几个小时就可以完成的作业提供服务；(2)运行于同一个内部有高速网络连接的数据中心内；(3)数据中心内的计算机都是可靠的、专门的硬件。相比之下，SETI@home 则是在接入互联网的不可信的计算机上长时间运行，这些计算机的网络带宽不同，对数据本地化也没有要求。

1.6 Apache Hadoop 发展简史

Hadoop 是 Apache Lucene 创始人道格 · 卡丁(Doug Cutting)[③]创建的，Lucene 是一

① 编注：研究蛋白质折叠、误析、聚合及由此引起的相关疾病，如果蛋白质折叠不正确，会产生严重的后果，比如阿兹海默症、疯牛病及帕金森氏症。

② 2008 年 1 月，据报道称，SETI@home 每天使用 320 000 台计算机处理 300 GB 数据(这些机器大部分不专门用于 SETI@home，它们也在做其他一些处理)。原文可以参见 *http://bit.ly/new_seti_at_home_data*。

③ 编注：Lucene、Nutch、Hadoop 等项目的发起人，Apache 软件基金会主席，他将深奥的搜索技术做成了可以用的产品。他 1985 年毕业于斯坦福大学，在施乐当过实习生，也在这里做成了最早的产品级作品(施乐激光扫描仪有个操作系统的屏保)，也是他后来决定从事搜索技术的最初动机。1997 年，Lucene 诞生，2004 年 Nutch 问世，2006 年 Hadoop 诞生。

个应用广泛的文本搜索系统库。Hadoop 起源于开源网络搜索引擎 Apache Nutch，后者本身也是 Lucene 项目的一部分。

Hadoop 的得名

Hadoop 不是缩写，这个词是生造出来的。Hadoop 之父 Doug Cutting 是这样解释 Hadoop 来历的：

> “这个名字是我的孩子给他的毛绒象玩具取的。我的命名标准是好拼读，含义宽泛，不会被用于其他地方。小朋友是这方面的高手。Googol 就是他们起的。”

Hadoop 生态系统中的各项目所使用的名称也往往与其功能不相关，通常也以大象或其他动物为主题取名(例如 Pig)。较小一些的组件，名称通常都有较好的描述性(因此也更通俗)。这个原则很好，意味着我们可以望文知义，例如 namenode[①]，一看就知道它是用来管理文件系统命名空间(namespace)的。

从头打造一个网络搜索引擎是一个雄心勃勃的计划，不只是因为写爬虫程序很复杂，更因为必须有一个专职团队来实现，项目中包含许许多多需要随时修改的活动部件。同时，构建这样的系统代价非常高，据迈克·加法雷拉(Mike Cafarella)和 Doug Cutting 估计，一个支持 10 亿网页的索引系统，单单是硬件上的投入就高达 50 万美元，另外还有每月高达 3 万美元的运维费用。[②]不过，他们认为这个工作仍然值得投入，因为它开创的是一个优化搜索引擎算法的平台。

Nutch 项目开始于 2002 年，一个可以运行的网页爬取工具和搜索引擎系统很快面世。但后来，它的创造者认为这一架构的灵活性不够，不足以解决数十亿网页的搜索问题。一篇发表于 2003 年的论文为此提供了帮助，文中描述的是谷歌产品架构，该架构称为“谷歌分布式文件系统”(GFS)。[③]GFS 或类似的架构可以解决他们在

① 在本书中，我们使用小写的 namenode 来代表实体(泛称)，用驼峰体 `NameNode` 来表示对应的 Java 类实现。

② 参见 Mike Cafarella 和 Doug Cutting 在 2004 年 4 月发表在 ACM Queue 的文章，标题为“Building Nutch: Open Source Search”(开源搜索引擎 Nutch 的构建)，网址为 *http://bit.ly/building_nutch*。

③ 参见 Sanjay Ghemawat，Howard Gobioff 和 Shun-Tak Leung 在 2003 年 10 月发表的文章，标题为“The Google File System”(Google 文件系统)，网址为 *http://research.google.com/archive/gfs.html*。

网页爬取和索引过程中产生的超大文件的存储需求。特别关键的是，GFS 能够节省系统管理(如管理存储节点)所花的大量时间。在 2004 年，Nutch 的开发者开始着手做开源版本的实现，即 Nutch 分布式文件系统(NDFS)。

2004 年，谷歌发表论文向全世界介绍他们的 MapReduce 系统。[①]2005 年初，Nutch 的开发人员在 Nutch 上实现了一个 MapReduce 系统，到年中，Nutch 的所有主要算法均完成移植，用 MapReduce 和 NDFS 来运行。

Nutch 的 NDFS 和 MapReduce 实现不只适用于搜索领域。在 2006 年 2 月，开发人员将 NDFS 和 MapReduce 移出 Nutch 形成 Lucene 的一个子项目，命名为 Hadoop。大约在同一时间，Doug Cutting 加入雅虎，雅虎为此组织了专门的团队和资源，将 Hadoop 发展成能够以 Web 网络规模运行的系统(参见随后的补充材料)。在 2008 年 2 月，雅虎宣布，雅虎搜索引擎使用的索引是在一个拥有 1 万个内核的 Hadoop 集群上构建的。[②]

2008 年 1 月，Hadoop 已成为 Apache 的顶级项目，证明了它的成功、多样化和生命力。到目前为止，除雅虎之外，还有很多公司在用 Hadoop，例如 Last.fm、Facebook 和《纽约时报》等。

Hadoop 在雅虎

作者：Owen O'Melly[③]

构建互联网规模的搜索引擎离不开大量的数据，因此也离不开大量的机器来处理巨量的数据。雅虎搜索引擎(Yahoo！Search)有 4 个主要组成部分：*Crawler*，从网页服务器爬取网页；*WebMap*，构建一个已知网页的链接图；*Indexer*，为最佳页面构建一个反向索引；*Runtime*，处理用户的查询。WebMap 生成的链接图非常大，大约包括一万亿(10^{12})条边(每条边代表一个网页链接)和一千亿(10^{11})个节点(每个节点代表不同的网址)。创建并分析如此大的图需要大

① 参见 Jeffrey Dean 和 Sanjay Ghemawat 2004 年 12 月发表的文章，标题为“MapReduce: Simplified Data Processing on Large Clusters”(MapReduce: 大型集群的数据简化处理)，网址为 *http://research.google.com/archive/mapreduce.html*。

② 参见 2008 年 2 月 19 日发表的文章，标题为“Yahoo! Lauches World's Largest Hadoop ProductionApplications”(雅虎发布全球最大的 Hadoop 产品应用)，网址 *http://bit.ly/yahoo_hadoop*。

③ 编注：雅虎 Hadoop 团队重要成员之一，后来联合创办了 Hortonworks，公司名称来源于童书中一只叫 Horton 的大象。Hortonworks 在 IPO 前获五轮共 2.48 亿美元的融资，2014 年正式 IPO，市值 20 亿美元。国内类似的公司有星环科技和红象云腾，分别于主攻基于 Hadoop 的基础技术平台 TDH 和 CRH。

批计算机很多天长时间运行。到 2005 年初，WebMap 用的底层架构 *Dreadnaught* 需要重新设计，以便日后可以扩展到更多的节点。

Dreadnaught 从 20 个节点成功扩展到 600 个，但需要完全重新设计才能进一步扩大。Dreadnaught 与 MapReduce 在很多方面都很相似，但灵活性更强，结构也更松散。说具体点，一个 Dreadnaught 作业的每一个段(fragment，也称“分块”)都可以输送到下一阶段的各个片段继续执行，排序则是通过库函数来完成的。但实际情形是，大多数 WebMap 阶段是两两一对，对应于 MapReduce。因此，WebMap 应用不需要做大量重构操作就可以适应 MapReduce。

Eric Baldeschwieler(即 Eric14)组建了一个小团队，于是我们开始设计并在 GFS 和 MapReduce 之上用 C++来建立一个新框架的原型，并打算用它来取代 Dreadnaught。尽管我们的当务之急是需要一个新的 WebMap 框架，但更清楚的是，建立雅虎搜索引擎批处理平台的标准对我们更重要。使平台更通用以便支持其他用户，才能够更好地实现新平台的均衡性投资。

与此同时，我们也关注在 Hadoop(当时也是 Nutch 的一部分)及其进展情况。2006 年 1 月，雅虎聘请了 Doug Cutting。一个月后，我们决定放弃原型，转而采用 Hadoop。与我们的原型和设计相比，Hadoop 的优势在于它已经在 20 个节点上实际应用过(Nutch)。这样一来，我们便能在两个月内搭建一个研究集群并能够以很快的速度帮助我们的客户使用这个新的框架。另一个显著的优点是 Hadoop 已经开源，比较容易(尽管也不是想象的那么容易！)从雅虎法务部门获得许可对该开源系统进行进一步研究。因此，我们在 2006 年初建立了一个 200 节点的研究集群并暂时搁置 WebMap 计划，转而为研究用户提供 Hadoop 支持和优化服务。

《纽约时报》的案例广为流传，他们把存档报纸扫描之后得到 4 TB 的文件并用亚马逊的 EC2 云服务将文件存为 PDF 格式放到网上共享。[①]整个过程一共使用了 100 台计算机，所花的时间不到 24 小时。如果没有亚马逊的按小时付费模式(即允许《纽约时报》短期内访问大量机器)和 Hadoop 好用的并发编程模型珠联璧合，这个项目不太可能这么快就启动和完成。

① 参见 Derek Gottfrid 在 2007 年 11 月 1 日发表的文章，标题为“Self-service, Prorated Super Computing Fun!”(自助式比例分配超级计算的乐趣！)，网址为 *http://bit.ly/supercomputing_fun*。

2008 年 4 月，Hadoop 打破世界纪录，成为最快的 TB 级数据排序系统。运行于一个 910 节点的群集上，Hadoop 在 209 秒内(不到 3.5 分钟)完成了对完整的 1TB 数据的排序，击败了前一年的 297 秒冠军。[①]同年 11 月，谷歌在报告中声称，它的 MapReduce 对 1 TB 数据排序只用了 68 秒。[②]2009 年 4 月，有报道称雅虎有一个团队使用 Hadoop 对 1 TB 数据进行排序只花了 62 秒。[③]

从那以后，以更快的速度对更大规模的数据进行排序已成为一种趋势。在 2014 年 GraySort 基准排序大赛中，来自 Databricks 公司的一个团队获得并列第一。他们使用一个 207 个节点的 Spark 集群对 100TB 数据进行排序，只用了 1406 秒，处理速度为每分钟 4.27 TB。[④]

目前，Hadoop 被主流企业广泛使用。在工业界，Hadoop 已经是公认的大数据通用存储和分析平台，这一事实主要体现在大量直接使用或者间接包含 Hadoop 系统的产品如雨后春笋般大量涌现。一些大公司包括 EMC，IBM，Microsft 和 Oracle 以及一些专注于 Hadoop 的公司，如 Cloudera，Hortonworks 和 MapR 都可以提供商业化的 Hadoop 支持。

1.7 本书包含的内容

本书分为五大部分：第 I 部分～第Ⅲ部分讲解 Hadoop 核心，第Ⅳ部分主要讲述 Hadoop 生态系统中的相关项目，第Ⅴ部分包含 Hadoop 实例学习。读者可以按照章节顺序阅读，也可以根据自己的需求选择阅读顺序，如图 1-1 所示。

第 I 部分由 5 章内容组成，阐述的是 Hadoop 基础组件，应该在后续章节之前率先阅读。第 1 章是对 Hadoop 的宏观介绍。第 2 章简要介绍 MapReduce。第 3 章深入剖析 Hadoop 文件系统，特别是 HDFS。第 4 章讨论 Hadoop 集群资源管理系统

① 参见 Owen O'Malley 在 2008 年 5 月发表的文章，标题为“TeraByte Sort on Apache Hadoop”(Apache Hadoop 上的 TB 级数据排序)，网址为 *http://sortbenchmark.org/ YahooHadoop.pdf*。

② 参见 2008 年 11 月 21 日的文章，标题为“Sorting 1PB with MapReduce”(MapReduce 处理 1 PB 数据)，网址为 *http://bit.ly/sorting_1pb*。

③ 参见 Owen O'Malley 和 Arun C. Murthy 在 2009 年 4 月发表的文章，标题为“Winning a 60 Second Dash with a Yellow Elephant”(用小黄象赢得 60 秒冲刺)，网址为 *http://sortbenchmark.org/Yahoo2009.pdf*。

④ 参见 Databricks 联合创始人兼首席架构师辛湜(Reynold Xin)等人 2014 年 11 月发表的文章，标题为“GraySort on Apache Spark by Databricks”(基于 Databricks 之 Apache Spark 架构的 GraySort 排序)，网址为 *http://sortbenchmark.org/ApacheSpark2014.pdf*。

YARN。第 5 章讲述 Hadoop 的 I/O 构建模块：数据完整性、压缩、序列化及基于文件的数据结构。

图 1-1. 本书结构及阅读顺序

第Ⅱ部分包含 4 章内容，对 MapReduce 进行深度剖析。这些内容有助于对后续章节（如第Ⅳ部分的数据处理相关章节）的更好理解，但是可以在首次阅读时跳过。第 6 章全景呈现了 MapReduce 应用开发所涉及的具体步骤。第 7 章从用户的角度来看如何在 Hadoop 中实现 MapReduce。第 8 章主要包含 MapReduce 编程模型和 MapReduce 可以使用的各种数据格式。第 9 章是 MapReduce 高级主题，包括排序和数据连接。

第Ⅲ部分关注的是 Hadoop 管理，第 10 章和第 11 章主要描述如何设置和维护一个运行 HDFS 和 YARN(第二代 MapReduce 框架)的 Hadoop 集群。

第Ⅳ部分专门介绍在 Hadoop 上构建或相关的项目。每一章讲述一个项目，并且很大程度上独立于本部分其他章节，因此可按任何顺序阅读。

第Ⅳ部分前两章是关于数据格式的。第 12 章剖析 Avro，Hadoop 的一种跨语言数据序列化库。第 13 章描述 Parquet，一种有效的用于嵌套式数据的列式存储格式。

第 14 章～第 15 章讨论数据摄入，即如何将自己的数据输入到 Hadoop 系统中。第 14 章介绍 Flume，可以支持流数据的大批量摄入。第 15 章介绍 Sqoop，支持在结构化数据存储(如关系型数据库)和 HDFS 之间高效批量传输数据。

第 16 章～第 19 章从一个比 MapReduce 更高一级抽象的角度，描述数据处理这样一个共同议题。Pig(第 16 章)是一种用于开发大数据集的数据流语言。Hive(第 17 章)是一种数据仓库，用于管理 HDFS 中存储的数据并提供基于 SQL 的查询语言。Crunch(第 18 章)是一套高层次的 Java API，用于写可以运行在 MapReduce 或 Spark 上的数据处理管线程序(data processiong pipeline)。Spark(第 19 章)是一个面向大规模数据处理的集群计算框架，它提供一个有向无环图(DAG，directed acyclic graph)引擎，以及支持 Scala、Java 和 Python 语言的 API。

第 20 章介绍 HBase，一种使用 HDFS 作为底层存储的，分布式的面向列的实时数据库。第 21 章讲述 ZooKeeper，这是一种分布式高可用性的协调服务，提供用于构建分布式应用的原语集。

最后，第Ⅴ部分收集了一些实例，这些实例由以各种趣味方式使用 Hadoop 的用户所提供。

关于 Hadoop 的更多补充性资料，例如如何在自己的机器上安装 Hadoop，可以在附录中查看。

第 2 章

关于 MapReduce

MapReduce 是一种可用于数据处理的编程模型。该模型比较简单，但要想写出有用的程序却不太容易。Hadoop 可以运行各种语言版本的 MapReduce 程序。在本章中，我们将看到同一个程序的 Java、Ruby 和 Python 语言版本。最重要的是，MapReduce 程序本质上是并行运行的，因此可以将大规模的数据分析任务分发给任何一个拥有足够多机器的数据中心。MapReduce 的优势在于处理大规模数据集，所以这里先来看一个数据集。

2.1 气象数据集

在我们的例子里，要写一个挖掘气象数据的程序。分布在全球各地的很多气象传感器每隔一小时收集气象数据和收集大量日志数据，由于我们希望处理所有这些数据，而且这些数据是半结构化的且是按照记录方式存储的，因此非常适合用 MapReduce 来分析。

数据格式

我们使用的数据来自美国国家气候数据中心(National Climatic Data Center，简称 NCDC，*http://www.ncdc.noaa.gov/*)。这些数据按行并以 ASCII 格式存储，其中每一行是一条记录。该存储格式支持丰富的气象要素，其中许多要素可以选择性地列入收集范围或其数据所需的存储长度是可变的。为了简单起见，我们重点讨论一些基本要素(比如气温)，这些要素始终都有而且长度都是固定的。

范例 2-1 显示了一行采样数据，其中重要字段加了注释。这一行数据被分成很多

行以突出每个字段，但在实际文件中，这些字段合并成一行，没有任何分隔符。

范例 2-1. 国家气候数据中心数据记录的格式

```
0057
332130          # USAF weather station identifier
99999     # WBAN weather station identifier
19500101  # observation date
0300            # observation time
4
+51317    # latitude (degrees x 1000)
+028783   # longitude (degrees x 1000)
FM-12
+0171     # elevation (meters)
99999
V020
320             # wind direction (degrees)
1               # quality code
N
0072
1
00450     # sky ceiling height (meters)
1               # quality code
C
N
010000    # visibility distance (meters)
1               # quality code
N
9
-0128     # air temperature (degrees Celsius x 10)
1               # quality code
-0139           # dew point temperature (degrees Celsius x 10)
1               # quality code
10268     # atmospheric pressure (hectopascals x 10)
1               # quality code
```

数据文件按照日期和气象站进行组织。从 1901 年到 2001 年，每一年都有一个目录，每一个目录中包含各个气象站该年气象数据的打包文件及其说明文件。例如，1999 年对应文件夹下面就包含下面的记录：

```
% ls raw/1990 | head
010010-99999-1990.gz
010014-99999-1990.gz
010015-99999-1990.gz
010016-99999-1990.gz
010017-99999-1990.gz
010030-99999-1990.gz
010040-99999-1990.gz
010080-99999-1990.gz
010100-99999-1990.gz
```

```
010150-99999-1990.gz
```

气象台有成千上万个，所以整个数据集由大量的小文件组成。通常情况下，处理少量的大型文件更容易、更有效，因此，这些数据需要经过预处理，将每年的数据文件拼接成一个单独的文件。具体做法请参见附录 C。

2.2 使用 Unix 工具来分析数据

在这个数据集中，每年全球气温的最高记录是多少？我们先不使用 Hadoop 来解决这个问题，因为只有提供了性能基准和结果检查工具，才能和 Hadoop 进行有效的对比。

传统处理按行存储数据的工具是 *awk*。范例 2-2 是一个程序脚本，用于计算每年的最高气温。

范例 2-2. 该程序从 NCDC 气象记录中找出每年最高气温

```
#!/usr/bin/env bash
for year in all/*
do
  echo -ne `basename $year .gz`"\t"
  gunzip -c $year | \
    awk'{ temp = substr($0, 88, 5) + 0;
          q = substr($0, 93, 1);
          if ( temp!=9999 && q ~ /[01459]/ && temp > max) max = temp}
        END { print max }'
done
```

这个脚本循环遍历按年压缩的数据文件，首先显示年份，然后使用 *awk* 处理每一个文件。*awk* 从数据中提取两个字段：气温和质量代码。气温值加 0 后转换为整数。接着测试气温值是否有效(用 9999 替代 NCDC 数据集中的缺失的值)，通过质量代码来检测读取的数值是否有疑问或错误。如果数据读取正确，那么该值将与目前读取到的最大气温值进行比较，如果该值比原先的最大值大，就替换目前的最大值。处理完文件中所有的行后，再执行 END 块中的代码并在屏幕上输出最大气温值。

下面是某次运行结果的起始部分：

```
% ./max_temperature.sh
1901    317
1902    244
1903    289
1904    256
```

```
1905    283
...
```

由于源文件中的气温值被放大 10 倍，所以 1901 年的最高气温是 31.7℃(20 世纪初记录的气温数据比较少，所以这个结果也是可能的)。使用亚马逊的 EC2 High-CPU Extra Large Instance 运行这个程序，只需要 42 分钟就可以处理完一个世纪的气象数据，找出最高气温。

为了加快处理速度，我们需要并行处理程序来进行数据分析。从理论上讲，这很简单：我们可以使用计算机上所有可用的硬件线程(hardware thread)来处理，每个线程负责处理不同年份的数据。但这样做仍然存在一些问题。

首先，将任务划分成大小相同的作业通常并不是一件容易的事情。在我们这个例子中，不同年份数据文件的大小差异很大，所以有一部分线程会比其他线程更早结束运行。即使可以再为它们分配下一个作业，但总的运行时间仍然取决于处理最长文件所需要的时间。另一种更好的方法是将输入数据分成固定大小的块(chunk)，然后每块分到各个进程去执行，这样一来，即使有一些进程可以处理更多数据，我们也可以为它们分配更多的数据。

其次，合并各个独立进程的运行结果，可能还需要额外进行处理。在我们的例子中，每年的结果独立于其他年份，所以可能需要把所有结果拼接起来，然后再按年份进行排序。如果使用固定块大小的方法，则需要一种精巧的方法来合并结果。在这个例子中，某年的数据通常被分割成几个块，每个块独立处理。我们最终获得每个块的最高气温，所以最后一步找出最大值作为该年的最高气温，其他年份的数据都像这样处理。

最后，还是得受限于单台计算机的处理能力。即便开足马力，用上所有处理器，至少也得花 20 分钟，系统无法更快了。另外，某些数据集的增长可能会超出单台计算机的处理能力。一旦开始使用多台计算机，整个大环境中的其他因素就会互相影响，主要归类为协调性和可靠性两个方面。哪个进程负责运行整个作业？我们如何处理失败的进程？

因此，虽然并行处理也是可行的，但实际上也很麻烦。可以借助于 Hadoop 类似框架来解决这些问题。

2.3 使用 Hadoop 来分析数据

为了充分利用 Hadoop 提供的并行处理优势，我们需要将查询表示成 MapReduce

作业。完成某种本地端的小规模测试之后，就可以把作业部署到在集群上运行。

2.3.1 map 和 reduce

MapReduce 任务过程分为两个处理阶段：map 阶段和 reduce 阶段。每阶段都以键-值对作为输入和输出，其类型由程序员来选择。程序员还需要写两个函数：map 函数和 reduce 函数。

map 阶段的输入是 NCDC 原始数据。我们选择文本格式作为输入格式，将数据集的每一行作为文本输入。键是某一行起始位置相对于文件起始位置的偏移量，不过我们不需要这个信息，所以将其忽略。

我们的 map 函数很简单。由于我们只对年份和气温属性感兴趣，所以只需要取出这两个字段数据。在本例中，map 函数只是一个数据准备阶段，通过这种方式来准备数据，使 reduce 函数能够继续对它进行处理：即找出每年的最高气温。map 函数还是一个比较适合去除已损记录的地方：此处，我们筛掉缺失的、可疑的或错误的气温数据。

为了全面了解 map 的工作方式，我们考虑以下输入数据的示例数据(考虑到篇幅，去除了一些未使用的列，并用省略号表示)：

```
0067011990999991950051507004...9999999N9+00001+99999999999...
0043011990999991950051512004...9999999N9+00221+99999999999...
0043011990999991950051518004...9999999N9-00111+99999999999...
0043012650999991949032412004...0500001N9+01111+99999999999...
0043012650999991949032418004...0500001N9+00781+99999999999...
```

这些行以键-值对的方式作为 map 函数的输入：

```
(0, 0067011990999991950051507004...9999999N9+00001+99999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+99999999999...)
(212, 0043011990999991950051518004...9999999N9-00111+99999999999...)
(318, 0043012650999991949032412004...0500001N9+01111+99999999999...)
(424, 0043012650999991949032418004...0500001N9+00781+99999999999...)
```

键(key)是文件中的行偏移量，map 函数并不需要这个信息，所以将其忽略。map 函数的功能仅限于提取年份和气温信息(以粗体显示)，并将它们作为输出(气温值已用整数表示)：

```
(1950, 0)
(1950, 22)
(1950, -11)
(1949, 111)
(1949, 78)
```

map 函数的输出经由 MapReduce 框架处理后，最后发送到 reduce 函数。这个处理过程基于键来对键-值对进行排序和分组。因此，在这一示例中，reduce 函数看到的是如下输入：

```
(1949, [111, 78])
(1950, [0, 22, -11])
```

每一年份后紧跟着一系列气温数据。reduce 函数现在要做的是遍历整个列表并从中找出最大的读数：

```
(1949, 111)
(1950, 22)
```

这是最终输出结果，每一年的全球最高气温记录。

整个数据流如图 2-1 所示。在图的底部是 Unix 管线，用于模拟整个 MapReduce 的流程，部分内容将在讨论 Hadoop Streaming 时再次涉及。

图 2-1. MapReduce 的逻辑数据流

2.3.2 Java MapReduce

明白 MapReduce 程序的工作原理之后，下一步就是写代码实现它。我们需要三样东西：一个 map 函数、一个 reduce 函数和一些用来运行作业的代码。map 函数由 `Mapper` 类来表示，后者声明一个抽象的 `map()`方法。范例 2-3 显示了我们的 map 函数实现。

范例 2-3. 查找最高气温的 Mapper 类

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
```

```
public class MaxTemperatureMapper
  extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {

  private static final int MISSING = 9999;

  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    String line = value.toString();
    String year = line.substring(15, 19);
    int airTemperature;
    if (line.charAt(87) == '+') { // parseInt doesn't like leading plus signs
      airTemperature = Integer.parseInt(line.substring(88, 92));
    } else {
      airTemperature = Integer.parseInt(line.substring(87, 92));
    }
    String quality = line.substring(92, 93);
    if (airTemperature != MISSING && quality.matches("[01459]")) {
      context.write(new Text(year), new IntWritable(airTemperature));
    }
  }
}
```

这个 Mapper 类是一个泛型类型，它有四个形参类型，分别指定 map 函数的输入键、输入值、输出键和输出值的类型。就现在这个例子来说，输入键是一个长整数偏移量，输入值是一行文本，输出键是年份，输出值是气温(整数)。Hadoop 本身提供了一套可优化网络序列化传输的基本类型，而不直接使用 Java 内嵌的类型。这些类型都在 org.apache.hadoop.io 包中。这里使用 LongWritable 类型(相当于 Java 的 Long 类型)、Text 类型(相当于 Java 中的 String 类型)和 IntWritable 类型(相当于 Java 的 Integer 类型)。

map()方法的输入是一个键和一个值。我们首先将包含有一行输入的 Text 值转换成 Java 的 String 类型，之后用 substring()方法提取我们感兴趣的列。

map()方法还提供 Context 实例用于输出内容的写入。在这种情况下，我们将年份数据按 Text 对象进行读/写（因为我们把年份当作键），将气温值封装在 IntWritable 类型中。只有气温数据不缺并且所对应质量代码显示为正确的气温读数时，这些数据才会被写入输出记录中。

以类似方法用 Reducer 来定义 reduce 函数，如范例 2-4 所示。

范例 2-4. 查找最高气温的 Reducer 类

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
```

```
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class MaxTemperatureReducer
  extends Reducer<Text, IntWritable, Text, IntWritable> {

  @Override
  public void reduce(Text key, Iterable<IntWritable> values,
      Context context)
      throws IOException, InterruptedException {

    int maxValue = Integer.MIN_VALUE;
    for (IntWritable value : values) {
      maxValue = Math.max(maxValue, value.get());
    }
    context.write(key, new IntWritable(maxValue));
  }
}
```

同样，reduce 函数也有四个形式参数类型用于指定输入和输出类型。reduce 函数的输入类型必须匹配 map 函数的输出类型：即 `Text` 类型和 `IntWritable` 类型。在这种情况下，reduce 函数的输出类型也必须是 `Text` 和 `IntWritable` 类型，分别输出年份及其最高气温。这个最高气温是通过循环比较每个气温与当前所知最高气温所得到的。

第三部分代码负责运行 MapReduce 作业，参见范例 2-5。

范例 2-5. 这个应用程序在气象数据集中找出最高气温

```
import java.io.IOException;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.input.FileOutputFormat;
import org.apache.hadoop.mapredduce.input.FileOutputFormat

public class MaxTemperature {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperature <input path> <output path>");
      System.exit(-1);
    }

    Job job = new Job();
    job.setJarByClass(MaxTemperature.class);
    job.setJobName("Max temperature");

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

```
    job.setMapperClass(MaxTemperatureMapper.class);
    job.setReducerClass(MaxTemperatureReducer.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Job 对象指定作业执行规范。我们可以用它来控制整个作业的运行。我们在 Hadoop 集群上运行这个作业时，要把代码打包成一个 JAR 文件(Hadoop 在集群上发布这个文件)。不必明确指定 JAR 文件的名称，在 Job 对象的 setJarByClass()方法中传递一个类即可，Hadoop 利用这个类来查找包含它的 JAR 文件，进而找到相关的 JAR 文件。

构造 Job 对象之后，需要指定输入和输出数据的路径。调用 FileInputFormat 类的静态方法 addInputPath()来定义输入数据的路径，这个路径可以是单个的文件、一个目录(此时，将目录下所有文件当作输入)或符合特定文件模式的一系列文件。由函数名可知，可以多次调用 addInputPath()来实现多路径的输入。

调用 FileOutputFormat 类中的静态方法 setOutputPath()来指定输出路径(只能有一个输出路径)。这个方法指定的是 reduce 函数输出文件的写入目录。在运行作业前该目录是不应该存在的，否则 Hadoop 会报错并拒绝运行作业。这种预防措施的目的是防止数据丢失(长时间运行的作业如果结果被意外覆盖，肯定是非常恼人的)。

接着，通过 setMapperClass()和 setReducerClass()方法指定要用的 map 类型和 reduce 类型。

setOutputKeyClass()和 setOutputValueClass()方法控制 reduce 函数的输出类型，并且必须和 Reduce 类产生的相匹配。map 函数的输出类型默认情况下和 reduce 函数是相同的，因此如果 mapper 产生出和 reducer 相同的类型时(如同本例所示)，不需要单独设置。但是，如果不同，则必须通过 setMapOutputKeyClass()和 setMapOutputValueClass()方法来设置 map 函数的输出类型。

输入的类型通过输入格式来控制，我们的例子中没有设置，因为使用的是默认的 TextInputFormat(文本输入格式)。

在设置定义 map 和 reduce 函数的类之后，可以开始运行作业。Job 中的

waitForCompletion()方法提交作业并等待执行完成。该方法唯一的参数是一个标识，指示是否已生成详细输出。当标识为 true(成功)时，作业会把其进度信息写到控制台。

waitForCompletion()方法返回一个布尔值，表示执行的成(true)败(false)，这个布尔值被转换成程序的退出代码 0 或者 1。

本章及全书使用的 Java MapReduce API，被为“新 API”，取代了功能上等价的旧版本 API。附录 D 阐述了新旧 API 之间的区别，同时介绍了一些新旧 API 转换的小技巧。读者在附录 D 中也可以找到等价的、用旧 API 写的“查找最高气温”应用。

2.3.2.1 运行测试

写好 MapReduce 作业之后，通常要拿一个小型数据集进行测试以排除代码问题。首先，以独立(本机)模式安装 Hadoop，详细说明请参见附录 A。在这种模式下，Hadoop 在本地文件系统上运行作业程序。然后，使用本书网站上的指令安装和编译示例。

以前面讨过的 5 行采样数据为例来测试 MapReduce 作业(考虑到篇幅，这里对输出稍有修改，删除了一些行)：

```
% export HADOOP_CLASSPATH=hadoop-examples.jar
% hadoop MaxTemperature input/ncdc/sample.txt output

14/09/16 09:48:39 WARN util.NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
14/09/16 09:48:40 WARN mapreduce.JobSubmitter: Hadoop command-line option
parsing not performed. Implement the Tool interface and execute your
application with ToolRunner to remedy this.
14/09/16 09:48:40 INFO input.FileInputFormat: Total input paths to process : 1
14/09/16 09:48:40 INFO mapreduce.JobSubmitter: number of splits:1
14/09/16 09:48:40 INFO mapreduce.JobSubmitter: Submitting tokens for job:
job_local26392882_0001
14/09/16 09:48:40 INFO mapreduce.Job: The url to track the job: http://localhost:8080/
14/09/16 09:48:40 INFO mapreduce.Job: Running job: job_local26392882_0001
14/09/16 09:48:40 INFO mapred.LocalJobRunner: OutputCommitter set in config null
14/09/16 09:48:40 INFO mapred.LocalJobRunner: OutputCommitter is
org.apache.hadoop.mapreduce.lib.output.FileOutputCommitter
14/09/16 09:48:40 INFO mapred.LocalJobRunner: Waiting for map tasks
14/09/16 09:48:40 INFO mapred.LocalJobRunner: Starting task:
attempt_local26392882_0001_m_000000_0
14/09/16 09:48:40 INFO mapred.Task: Using ResourceCalculatorProcessTree : null
14/09/16 09:48:40 INFO mapred.LocalJobRunner:
14/09/16 09:48:40 INFO mapred.Task: Task:attempt_local26392882_0001_m_000000_0
is done. And is in the process of committing
```

```
14/09/16 09:48:40 INFO mapred.LocalJobRunner: map
14/09/16 09:48:40 INFO mapred.Task: Task 'attempt_local26392882_0001_m_000000_0' done.
14/09/16 09:48:40 INFO mapred.LocalJobRunner: Finishing task:
attempt_local26392882_0001_m_000000_0
14/09/16 09:48:40 INFO mapred.LocalJobRunner: map task executor complete.
14/09/16 09:48:40 INFO mapred.LocalJobRunner: Waiting for reduce tasks
14/09/16 09:48:40 INFO mapred.LocalJobRunner: Starting task:
attempt_local26392882_0001_r_000000_0
14/09/16 09:48:40 INFO mapred.Task: Using ResourceCalculatorProcessTree : null 14/09/16
09:48:40 INFO mapred.LocalJobRunner: 1 / 1 copied.
14/09/16 09:48:40 INFO mapred.Merger: Merging 1 sorted segments
14/09/16 09:48:40 INFO mapred.Merger: Down to the last merge-pass, with 1 segments left of
total size: 50 bytes
14/09/16 09:48:40 INFO mapred.Merger: Merging 1 sorted segments
14/09/16 09:48:40 INFO mapred.Merger: Down to the last merge-pass, with 1 segments left of
total size: 50 bytes
14/09/16 09:48:40 INFO mapred.LocalJobRunner: 1 / 1 copied.
14/09/16 09:48:40 INFO mapred.Task: Task:attempt_local26392882_0001_r_000000_0 is done. And
is in the process of committing
14/09/16 09:48:40 INFO mapred.LocalJobRunner: 1 / 1 copied.
14/09/16 09:48:40 INFO mapred.Task: Task attempt_local26392882_0001_r_000000_0 is allowed
to commit now
14/09/16 09:48:40 INFO output.FileOutputCommitter: Saved output of task
'attempt...local26392882_0001_r_000000_0' to file:/Users/tom/book-workspace/ hadoop-
book/output/_temporary/0/task_local26392882_0001_r_000000
14/09/16 09:48:40 INFO mapred.LocalJobRunner: reduce > reduce
14/09/16 09:48:40 INFO mapred.Task: Task 'attempt_local26392882_0001_r_000000_0' done.
14/09/16 09:48:40 INFO mapred.LocalJobRunner: Finishing task:
attempt_local26392882_0001_r_000000_0
14/09/16 09:48:40 INFO mapred.LocalJobRunner: reduce task executor complete. 14/09/16
09:48:41 INFO mapreduce.Job: Job job_local26392882_0001 running in uber mode : false
14/09/16 09:48:41 INFO mapreduce.Job: map 100% reduce 100%
14/09/16 09:48:41 INFO mapreduce.Job: Job job_local26392882_0001 completed successfully
14/09/16 09:48:41 INFO mapreduce.Job: Counters: 30
     File System Counters
          FILE: Number of bytes read=377168
          FILE: Number of bytes written=828464
          FILE: Number of read operations=0
          FILE: Number of large read operations=0
          FILE: Number of write operations=0
     Map-Reduce Framework
          Map input records=5
          Map output records=5
          Map output bytes=45
          Map output materialized bytes=61
          Input split bytes=129
          Combine input records=0
          Combine output records=0
          Reduce input groups=2
          Reduce shuffle bytes=61
          Reduce input records=5
          Reduce output records=2
          Spilled Records=10
          Shuffled Maps =1 Failed
```

```
        Shuffles=0
        Merged Map outputs=1
        GC time elapsed (ms)=39
        Total committed heap usage (bytes)=226754560
File Input Format Counters
        Bytes Read=529
File Output Format Counters
        Bytes Written=29
```

如果调用 hadoop 命令的第一个参数是类名，Hadoop 就会启动一个 JVM(Java 虚拟机)来运行这个类。该 Hadoop 命令将 Hadoop 库(及其依赖关系)添加到类路径中，同时也能获得 Hadoop 配置信息。为了将应用类添加到类路径中，我们定义了一个 HADOOP_CLASSPATH 环境变量，然后由 *Hadoop* 脚本来执行相关操作。

以本地(独立)模式运行时，本书中所有程序均假设已按照这种方式来设置 HADOOP_CLASSPATH。命令的运行需要在范例代码所在的文件夹下进行。

运行作业所得到的输出提供了一些有用的信息。例如，我们可以看到，这个作业有指定的标识，即 job_local26392882_0001，并且执行了一个 map 任务和一个 reduce 任务(使用 attempt_local26392882_ 0001_m_000000_0 和 attempt_local26392882_0001_r_000000_0 两个 ID)。在调试 MapReduce 作业时，知道作业 ID 和任务 ID 是非常有用的。

输出的最后一部分，以 Counters 为标题，显示 Hadoop 上运行的每个作业的一些统计信息。这些信息对检查数据是否按照预期进行处理非常有用。例如，我们查看系统输出的记录信息可知：5 个 map 输入记录产生 5 个 map 输出记录(由于 mapper 为每个合法的输入记录产生一个输出记录)，随后，分为两组的 5 个 reduce 输入记录(一组对应一个唯一的键)产生两个 reduce 输出记录。

输出数据写入 *output* 目录，其中每个 reducer 都有一个输出文件。我们例子中的作业只有一个 reducer，所以只能找到一个名为 *part-r-00000* 的文件：

```
% cat output/part-r-00000
1949    111
1950    22
```

这个结果和我们之前手动寻找的结果一样。我们把这个结果解释为 1949 年的最高气温记录为 11.1℃，而 1950 年为 2.2℃。

2.4 横向扩展

前面介绍了 MapReduce 针对少量输入数据是如何工作的，现在我们开始鸟瞰整个系统以及有大量输入时的数据流。为了简单起见，到目前为止，我们的例子都只是用了本地文件系统中的文件。然而，为了实现横向扩展(scaling out)，我们需要把数据存储在分布式文件系统中(典型的为 HDFS，将在第 3 章中介绍)。通过使用 Hadoop 资源管理系统 YARN，Hadoop 可以将 MapReduce 计算转移到存储有部分数据的各台机器上(参见第 4 章)。下面我们看看具体过程。

2.4.1 数据流

首先定义一些术语。MapReduce 作业(job) 是客户端需要执行的一个工作单元：它包括输入数据、MapReduce 程序和配置信息。Hadoop 将作业分成若干个任务(task)来执行，其中包括两类任务：map 任务和 reduce 任务。这些任务运行在集群的节点上，并通过 YARN 进行调度。如果一个任务失败，它将在另一个不同的节点上自动重新调度运行。

Hadoop 将 MapReduce 的输入数据划分成等长的小数据块，称为输入分片(input split)或简称“分片”。Hadoop 为每个分片构建一个 map 任务，并由该任务来运行用户自定义的 map 函数从而处理分片中的每条记录。

拥有许多分片，意味着处理每个分片所需要的时间少于处理整个输入数据所花的时间。因此，如果我们并行处理每个分片，且每个分片数据比较小，那么整个处理过程将获得更好的负载平衡，因为一台较快的计算机能够处理的数据分片比一台较慢的计算机更多，且成一定的比例。即使使用相同的机器，失败的进程或其他并发运行的作业能够实现满意的负载平衡，并且随着分片被切分得更细，负载平衡的质量会更高。

另一方面，如果分片切分得太小，那么管理分片的总时间和构建 map 任务的总时间将决定作业的整个执行时间。对于大多数作业来说，一个合理的分片大小趋向于 HDFS 的一个块的大小，默认是 128 MB，不过可以针对集群调整这个默认值(对所有新建的文件)，或在每个文件创建时指定。

Hadoop 在存储有输入数据(HDFS 中的数据)的节点上运行 map 任务，可以获得最佳性能，因为它无需使用宝贵的集群带宽资源。这就是所谓的“数据本地化优

化”(data locality optimization)。但是，有时对于一个 map 任务的输入分片来说，存储该分片的 HDFS 数据块复本的所有节点可能正在运行其他 map 任务，此时作业调度需要从某一数据块所在的机架中的一个节点上寻找一个空闲的 map 槽(slot)来运行该 map 任务分片。仅仅在非常偶然的情况下(该情况基本上不会发生)，会使用其他机架中的节点运行该 map 任务，这将导致机架与机架之间的网络传输。图 2-2 显示了这三种可能性。

现在我们应该清楚为什么最佳分片的大小应该与块大小相同：因为它是确保可以存储在单个节点上的最大输入块的大小。如果分片跨越两个数据块，那么对于任何一个 HDFS 节点，基本上都不可能同时存储这两个数据块，因此分片中的部分数据需要通过网络传输到 map 任务运行的节点。与使用本地数据运行整个 map 任务相比，这种方法显然效率更低。

map 任务将其输出写入本地硬盘，而非 HDFS。这是为什么？因为 map 的输出是中间结果：该中间结果由 reduce 任务处理后才产生最终输出结果，而且一旦作业完成，map 的输出结果就可以删除。因此，如果把它存储在 HDFS 中并实现备份，难免有些小题大做。如果运行 map 任务的节点在将 map 中间结果传送给 reduce 任务之前失败，Hadoop 将在另一个节点上重新运行这个 map 任务以再次构建 map 中间结果。

图 2-2. 本地数据(a)、本地机架(b)和跨机架(c)map 任务

reduce 任务并不具备数据本地化的优势，单个 reduce 任务的输入通常来自于所有 mapper 的输出。在本例中，我们仅有一个 reduce 任务，其输入是所有 map 任务的输出。因此，排过序的 map 输出需通过网络传输发送到运行 reduce 任务的节点。数据在 reduce 端合并，然后由用户定义的 reduce 函数处理。reduce 的输出通常存储在 HDFS 中以实现可靠存储。如第 3 章所述，对于 reduce 输出的每个 HDFS 块，第一个复本存储在本地节点上，其他复本出于可靠性考虑存储在其他机架的节点中。因此，将 reduce 的输出写入 HDFS 确实需要占用网络带宽，但这与正常的 HDFS 管线写入的消耗一样。

一个 reduce 任务的完整数据流如图 2-3 所示。虚线框表示节点，虚线箭头表示节点内部的数据传输，而实线箭头表示不同节点之间的数据传输。

reduce 任务的数量并非由输入数据的大小决定，相反是独立指定的。8.1.1 节将介绍如何为指定的作业选择 reduce 任务的数量。

如果有好多个 reduce 任务，每个 map 任务就会针对输出进行分区(partition)，即为每个 reduce 任务建一个分区。每个分区有许多键(及其对应的值)，但每个键对应的键-值对记录都在同一分区中。分区可由用户定义的分区函数控制，但通常用默认的 partitioner 通过哈希函数来分区，很高效。

图 2-3. 一个 reduce 任务的 MapReduce 数据流

一般情况下，多个 reduce 任务的数据流如图 2-4 所示。该图清楚地表明了为什么 map 任务和 reduce 任务之间的数据流称为 shuffle(混洗)，因为每个 reduce 任务的

输入都来自许多 map 任务。shuffle 一般比图中所示的更复杂，而且调整混洗参数对作业总执行时间的影响非常大，详情参见 7.3 节。

图 2-4. 多个 reduce 任务的数据流

最后，当数据处理可以完全并行(即无需混洗时)，可能会出现无 reduce 任务的情况(示例参见 8.2.2 节)。在这种情况下，唯一的非本地节点数据传输是 map 任务将结果写入 HDFS(参见图 2-5)。

图 2-5. 无 reduce 任务的 MapReduce 数据流

2.4.2 combiner 函数

集群上的可用带宽限制了 MapReduce 作业的数量，因此尽量避免 map 和 reduce 任务之间的数据传输是有利的。Hadoop 允许用户针对 map 任务的输出指定一个 combiner（就像 mapper 和 reducer 一样），combiner 函数的输出作为 reduce 函数的输入。由于 combiner 属于优化方案，所以 Hadoop 无法确定要对一个指定的 map 任务输出记录调用多少次 combiner (如果需要)。换而言之，不管调用 combiner 多少次，0 次、1 次或多次，reducer 的输出结果都是一样的。

combiner 的规则制约着可用的函数类型。这里最好用一个例子来说明。还是假设以前计算最高气温的例子，1950 年的读数由两个 map 任务处理(因为它们在不同的分片中)。假设第一个 map 的输出如下：

```
(1950, 0)
(1950, 20)
(1950, 10)
```

第二个 map 的输出如下：

```
(1950, 25)
(1950, 15)
```

reduce 函数被调用时，输入如下：

```
(1950, [0, 20, 10, 25, 15])
```

因为 25 为该列数据中最大的，所以它的输出如下：

```
(1950, 25)
```

我们可以像使用 reduce 函数那样，使用 combiner 找出每个 map 任务输出结果中的最高气温。如此一来，reduce 函数调用时将被传入以下数据：

```
(1950, [20, 25])
```

reduce 输出的结果和以前一样。更简单地说，我们可以通过下面的表达式来说明气温数值的函数调用：

```
max(0, 20, 10, 25, 15) = max(max(0, 20, 10), max(25, 15)) = max(20, 25) = 25
```

并非所有函数都具有该属性。[1]例如，如果我们计算平均气温，就不能用求平均函数 mean 作为我们的 combiner 函数，因为

```
mean(0, 20, 10, 25, 15) = 14
```

但是又有

```
mean(mean(0, 20, 10), mean(25, 15)) = mean(10, 20) = 15
```

combiner 函数不能取代 reduce 函数。为什么呢？我们仍然需要 reduce 函数来处理不同 map 输出中具有相同键的记录。但 combiner 函数能帮助减少 mapper 和 reducer 之间的数据传输量，因此，单纯就这点而言，在 MapReduce 作业中是否使用 combiner 函数还是值得斟酌的。

指定一个 combiner

让我们回到 Java MapReduce 程序，combiner 是通过 Reducer 类来定义的，并且在这个例子中，它的实现与 MaxTemperatureReducer 中的 reduce 函数相同。唯一的改动是在 Job 中设置 combiner 类(参见范例 2-6)。

范例 2-6. 用 combiner 函数快速找出最高气温

```
public class MaxTemperatureWithCombiner {
  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperatureWithCombiner <input path> " +
          "<output path>");
      System.exit(-1);
    }

    Job job = new Job();
    job.setJarByClass(MaxTemperatureWithCombiner.class);
    job.setJobName("Max temperature");

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
```

① 有此属性的函数叫 commutative 和 associative。有时也有文章将它们称为 distributive，比如在 Gray 等人 1995 年发表的论文“Data Cube: A Relational Aggregation Operatior Generalizing Group by, Cross-Tab, and Sub-Totals”中。

```
    job.setReducerClass(MaxTemperatureReducer.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

2.4.3 运行分布式的 MapReduce 作业

这个程序用不着修改便可以在一个完整的数据集上直接运行。这是 MapReduce 的优势：它可以根据数据量的大小和硬件规模进行扩展。这里有一个运行结果：在一个 10 节点 EC2 集群运行 High-CPU Extra Large Instance，程序执行时间只花了短短 6 分钟。[①]

我们将在第 6 章分析在集群上运行程序的机制。

2.5 Hadoop Streaming

Hadoop 提供了 MapReduce 的 API，允许你使用非 Java 的其他语言来写自己的 map 和 reduce 函数。*Hadoop Streaming* 使用 Unix 标准流作为 Hadoop 和应用程序之间的接口，所以我们可以使用任何编程语言通过标准输入/输出来写 MapReduce 程序。[②]

Streaming 天生适合用于文本处理。map 的输入数据通过标准输入流传递给 map 函数，并且是一行一行地传输，最后将结果行写到标准输出。map 输出的键-值对是以一个制表符分隔的行，reduce 函数的输入格式与之相同(通过制表符来分隔的键-值对)并通过标准输入流进行传输。reduce 函数从标准输入流中读取输入行，该输入已由 Hadoop 框架根据键排过序，最后将结果写入标准输出。

下面使用 Streaming 来重写按年份查找最高气温的 MapReduce 程序。

① 这比在单台机器上通过 awk 串行运行快 7 倍。性能没有得到线性增长的主要原因是输入数据并不是均匀分块的。为了方便起见，数据已经按照年份压缩(gzip)，导致后续年份的文件比较大，因为这些年份的天气记录更多。

② 对于 C++程序员而言，Hadoop Pipes 来代替 Streaming。它用套接字(socket)与运行 C++语言写的 map 或 reduce 函数的进程通信。

2.5.1 Ruby 版本

范例 2-7 显示了用 Ruby 编写的 map 函数。

范例 2-7. 用 Ruby 编写查找最高气温的 map 函数

```
#!/usr/bin/env ruby
STDIN.each_line do |line|
  val = line
  year, temp, q = val[15,4], val[87,5], val[92,1]
  puts "#{year}\t#{temp}" if (temp != "+9999" && q =~ /[01459]/)
end
```

程序通过程序块读取 STDIN(一个 IO 类型的全局常量)中的每一行来迭代执行标准输入中的每一行。该程序块从输入的每一行中取出相关字段，如果气温有效，就将年份以及气温以制表符\t 隔开写为标准输出(使用 puts)。

值得一提的是 Streaming 和 Java MapReduce API 之间的设计差异。Java API 控制的 map 函数一次只处理一条记录。针对输入数据中的每一条记录，该框架均需调用 Mapper 的 map()方法来处理。然而在 Streaming 中，map 程序可以自己决定如何处理输入数据，例如，它可以轻松读取并同时处理若干行，因为它受读操作的控制。

用户的 Java map 实现的是“推”记录方式，但它仍然可以同时处理多行，具体做法是通过 mapper 中实例变量将之前读取的多行汇聚在一起。[1]在这种情况下，需要实现 cleanup()方法，以便知道何时读到最后一条记录，进而完成对最后一组记录行的处理。

由于这个脚本只能在标准输入和输出上运行，所以最简单的方式是通过 Unix 管道进行测试，而不使用 Hadoop：

```
% cat input/ncdc/sample.txt | ch02-mr-intro/src/main/ruby/max_temperature_map.rb
1950    +0000
1950    +0022
1950    -0011
1949    +0111
1949    +0078
```

范例 2-8 显示的 reduce 函数更复杂一些。

① 另一种方法是，在新版 MapReduce API 中使用“拉”的方式来处理。详情可以参见附录 D。

范例 2-8. 用 Ruby 编写的查找最高气温的 reduce 函数

```
#!/usr/bin/env ruby

last_key, max_val = nil, -1000000
STDIN.each_line do |line|
  key, val = line.split("\t")
  if last_key && last_key != key
    puts "#{last_key}\t#{max_val}"
    last_key, max_val = key, val.to_i
  else
    last_key, max_val = key, [max_val, val.to_i].max
  end
end
puts "#{last_key}\t#{max_val}" if last_key
```

同样，程序遍历标准输入中的行，但在我们处理每个键组时，要存储一些状态。在这种情况下，键是年份，我们存储最后一个看到的键和迄今为止见到的该键对应的最高气温。MapReduce 框架保证了键的有序性，我们由此可知，如果读到一个键与前一个键不同，就需要开始处理一个新的键组。相比之下，Java API 系统提供一个针对每个键组的迭代器，而在 Streaming 中，需要在程序中找出键组的边界。

我们从每行取出键和值，然后如果正好完成一个键组的处理(`last_key & last_key != key`)，就针对该键组写入该键及其最高气温，用一个制表符来进行分隔，最后开始处理新键组时我们需要重置最高气温值。如果尚未完成对一个键组的处理，那么就只更新当前键的最高气温。

程序的最后一行确保处理完输入的最后一个键组之后，会有一行输出。

现在可以用 Unix 管线来模拟整个 MapReduce 管线，该管线与图 2-1 中显示的 Unix 管线是相同的：

```
% cat input/ncdc/sample.txt | \
  ch02-mr-intro/src/main/ruby/max_temperature_map.rb | \
  sort | ch02-mr-intro/src/main/ruby/max_temperature_reduce.rb
1949    111
1950    22
```

输出结果和 Java 程序的输出一样，所以下一步是通过 Hadoop 运行它。

`hadoop` 命令不支持 Streaming，因此，我们需要在指定 Streaming JAR 文件流与 `jar` 选项时指定。Streaming 程序的选项指定了输入和输出路径以及 map 和 reduce 脚本。如下所示：

```
% hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
  -input input/ncdc/sample.txt \
  -output output \
  -mapper ch02-mr-intro/src/main/ruby/max_temperature_map.rb \
  -reducer ch02-mr-intro/src/main/ruby/max_temperature_reduce.rb
```

在一个集群上运行一个庞大的数据集时，我们应该使用-combiner 选项来设置 combiner。

```
% hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
  -files ch02-mr-intro/src/main/ruby/max_temperature_map.rb,\
ch02-mr-intro/src/main/ruby/max_temperature_reduce.rb \
  -input input/ncdc/all \
  -output output \
  -mapper ch02-mr-intro/src/main/ruby/max_temperature_map.rb \
  -combiner ch02-mr-intro/src/main/ruby/max_temperature_reduce.rb \
  -reducer ch02-mr-intro/src/main/ruby/max_temperature_reduce.rb
```

还需要注意-files 选项的使用，在集群上运行 Streaming 程序时，我们会使用这个选项，从而将脚本传输到集群。

2.5.2 Python 版本

Streaming 支持任何可以从标准输入读/写到标准输出中的编程语言，因此对于更熟悉 Python 的读者，下面提供了同一个例子的 Python 版本。[①]map 脚本参见范例 2-9，reduce 脚本参见范例 2-10。

范例 2-9. 用于查找最高气温的 map 函数(python 版)

```
#!/usr/bin/env python

import re
import sys

for line in sys.stdin:
  val = line.strip()
  (year, temp, q) = (val[15:19], val[87:92], val[92:93])
  if (temp != "+9999" and re.match("[01459]", q)):
    print "%s\t%s" % (year, temp)
```

范例 2-10. 用于查找最高气温的 reduce 函数(python 版)

```
#!/usr/bin/env python

import sys

(last_key, max_val) = (None, -sys.maxint)
```

① 作为 Streaming 的替代方案，Python 程序员可以考虑 Dumbo(网址为 *http://klbostee.github.io/dumbo/*)，它能使 Streaming MapReduce 接口更像 Python，更好用。

```
for line in sys.stdin:
  (key, val) = line.strip().split("\t")
  if last_key and last_key != key:
    print "%s\t%s" % (last_key, max_val)
    (last_key, max_val) = (key, int(val))
  else:
    (last_key, max_val) = (key, max(max_val, int(val)))
if last_key:
  print "%s\t%s" % (last_key, max_val)
```

我们可以像测试 Ruby 程序那样测试程序并运行作业。例如，可以像下面这样运行测试：

```
% cat input/ncdc/sample.txt | \
  ch02-mr-intro/src/main/python/max_temperature_map.py | \
  sort | ch02-mr-intro/src/main/python/max_temperature_reduce.py
1949    111
1950    22
```

第 3 章

Hadoop 分布式文件系统

当数据集的大小超过一台独立的物理计算机的存储能力时，就有必要对它进行分区(partition)并存储到若干台单独的计算机上。管理网络中跨多台计算机存储的文件系统称为分布式文件系统(distributed filesystem)。该系统架构于网络之上，势必会引入网络编程的复杂性，因此分布式文件系统比普通磁盘文件系统更为复杂。例如，使文件系统能够容忍节点故障且不丢失任何数据，就是一个极大的挑战。

Hadoop 自带一个称为 HDFS 的分布式文件系统，即 Hadoop Distributed Filesystem。在非正式文档或旧文档以及配置文件中，有时也简称为 DFS，它们是一回事儿。HDFS 是 Hadoop 的旗舰级文件系统，也是本章的重点，但实际上 Hadoop 是一个综合性的文件系统抽象，因此接下来我们将了解将 Hadoop 与其他存储系统集成的途径，例如本地文件系统和 Amazon S3 系统。

3.1 HDFS 的设计

HDFS 以流式数据访问模式来存储超大文件，运行于商用硬件集群上。[①]让我们仔细看看下面的描述。

- **超大文件** “超大文件”在这里指具有几百 MB、几百 GB 甚至几百 TB

① Robert Chansler 等人在文章“The Hadoop Distributed File System”(Hadoop 分布系统)中详细叙述了 HDFS 的架构(*http://www.aosabook.org/en/hdfs.html*)，该文章内容可以参见 Amy Brown 和 Greg Wilson 等人的文章“The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks”。

大小的文件。目前已经有存储 PB 级数据的 Hadoop 集群了。[1]

- **流式数据访问** HDFS 的构建思路是这样的：一次写入、多次读取是最高效的访问模式。数据集通常由数据源生成或从数据源复制而来，接着长时间在此数据集上进行各种分析。每次分析都将涉及该数据集的大部分数据甚至全部，因此读取整个数据集的时间延迟比读取第一条记录的时间延迟更重要。

- **商用硬件** Hadoop 并不需要运行在昂贵且高可靠的硬件上。它是设计运行在商用硬件(在各种零售店都能买到的普通硬件[2])的集群上的，因此至少对于庞大的集群来说，节点故障的几率还是非常高的。HDFS 遇到上述故障时，被设计成能够继续运行且不让用户察觉到明显的中断。

  同样，那些不适合在 HDFS 上运行的应用也值得研究。目前 HDFS 对某些应用领域并不适合，不过以后可能会有所改进。

- **低时间延迟的数据访问** 要求低时间延迟数据访问的应用，例如几十毫秒范围，不适合在 HDFS 上运行。记住，HDFS 是为高数据吞吐量应用优化的，这可能会以提高时间延迟为代价。目前，对于低延迟的访问需求，HBase(参见第 20 章)是更好的选择。

- **大量的小文件** 由于 namenode 将文件系统的元数据存储在内存中，因此该文件系统所能存储的文件总数受限于 namenode 的内存容量。根据经验，每个文件、目录和数据块的存储信息大约占 150 字节。因此，举例来说，如果有一百万个文件，且每个文件占一个数据块，那至少需要 300 MB 的内存。尽管存储上百万个文件是可行的，但是存储数十亿个文件就超出了当前硬件的能力。[3]

- **多用户写入，任意修改文件** HDFS 中的文件写入只支持单个写入者，而且写操作总是以“只添加”方式在文件末尾写数据。它不支持多个写入者的操作，也不支持在文件的任意位置进行修改。可能以后会支持这些操

① 参见 Konstantin V. Shvachko 和 Arun C. Murthy 在 2008 年 9 月 30 日发表的文章，标题为“Scaling Hadoop to 4000 nodes at Yahoo!”(Yahoo 将 Hadoop 扩展应用到 4000 个节点)，详情可以访问 *http://bit.ly/scaling_hadoop*。

② 详见第 10 章的典型计算机规格。

③ 对于 HDFS 可扩展性限制的阐述，请参见 Konstantin V. Shvachko 在 2010 年 4 月发表的论文，标题为“HDFS Scalability: The limits to growth”(HDFS 的扩展性：增长的极限)，网址为 *http://bit.ly/limits_to_growth*。

作，但它们相对比较低效。

3.2 HDFS 的概念

3.2.1 数据块

每个磁盘都有默认的数据块大小，这是磁盘进行数据读/写的最小单位。构建于单个磁盘之上的文件系统通过磁盘块来管理该文件系统中的块，该文件系统块的大小可以是磁盘块的整数倍。文件系统块一般为几千字节，而磁盘块一般为 512 字节。这些信息(文件系统块大小)对于需要读/写文件的文件系统用户来说是透明的。尽管如此，系统仍然提供了一些工具(如 *df* 和 *fsck*)来维护文件系统，由它们对文件系统中的块进行操作。

HDFS 同样也有块(block)的概念，但是大得多，默认为 128 MB。与单一磁盘上的文件系统相似，HDFS 上的文件也被划分为块大小的多个分块(chunk)，作为独立的存储单元。但与面向单一磁盘的文件系统不同的是，HDFS 中小于一个块大小的文件不会占据整个块的空间(例如，当一个 1MB 的文件存储在一个 128 MB 的块中时，文件只使用 1 MB 的磁盘空间，而不是 128 MB)。如果没有特殊指出，本书中提到的“块”特指 HDFS 中的块。

HDFS 中的块为什么这么大？

HDFS 的块比磁盘的块大，其目的是为了最小化寻址开销。如果块足够大，从磁盘传输数据的时间会明显大于定位这个块开始位置所需的时间。因而，传输一个由多个块组成的大文件的时间取决于磁盘传输速率。

我们来做一个速算，如果寻址时间约为 10 ms，传输速率为 100 MB/s，为了使寻址时间仅占传输时间的 1%，我们要将块大小设置约为 100 MB。默认的块大小实际为 128 MB，但是很多情况下 HDFS 安装时使用更大的块。以后随着新一代磁盘驱动器传输速率的提升，块的大小会被设置得更大。

但是这个参数也不会设置得过大。MapReduce 中的 map 任务通常一次只处理一个块中的数据，因此如果任务数太少(少于集群中的节点数量)，作业的运行速度就会比较慢。

对分布式文件系统中的块进行抽象会带来很多好处。第一个最明显的好处是，一个文件的大小可以大于网络中任意一个磁盘的容量。文件的所有块并不需要存储在同一个磁盘上，因此它们可以利用集群上的任意一个磁盘进行存储。事实上，尽管不常见，但对于整个 HDFS 集群而言，也可以仅存储一个文件，该文件的块占满集群中所有的磁盘。

第二个好处是，使用抽象块而非整个文件作为存储单元，大大简化了存储子系统的设计。简化是所有系统的目标，但是这对于故障种类繁多的分布式系统来说尤为重要。将存储子系统的处理对象设置为块，可简化存储管理(由于块的大小是固定的，因此计算单个磁盘能存储多少个块就相对容易)。同时也消除了对元数据的顾虑(块只是要存储的大块数据，而文件的元数据，如权限信息，并不需要与块一同存储，这样一来，其他系统就可以单独管理这些元数据)。

不仅如此，块还非常适合用于数据备份进而提供数据容错能力和提高可用性。将每个块复制到少数几个物理上相互独立的机器上(默认为 3 个)，可以确保在块、磁盘或机器发生故障后数据不会丢失。如果发现一个块不可用，系统会从其他地方读取另一个复本，而这个过程对用户是透明的。一个因损坏或机器故障而丢失的块可以从其他候选地点复制到另一台可以正常运行的机器上，以保证复本的数量回到正常水平(参见 5.1 节对数据完整性的讨论，进一步了解如何应对数据损坏)。同样，有些应用程序可能选择为一些常用的文件块设置更高的复本数量进而分散集群中的读取负载。

与磁盘文件系统相似，HDFS 中 `fsck` 指令可以显示块信息。例如，执行以下命令将列出文件系统中各个文件由哪些块构成，详情可以参见 11.1.4 节对文件系统检查(fsck)的讨论：

```
% hdfs fsck / -files -blocks
```

3.2.2 namenode 和 datanode

HDFS 集群有两类节点以管理节点-工作节点模式运行，即一个 namenode(管理节点)和多个 datanode(工作节点)。namenode 管理文件系统的命名空间。它维护着文件系统树及整棵树内所有的文件和目录。这些信息以两个文件形式永久保存在本地磁盘上：命名空间镜像文件和编辑日志文件。namenode 也记录着每个文件中各个块所在的数据节点信息，但它并不永久保存块的位置信息，因为这些信息会在系统启动时根据数据节点信息重建。

客户端(client)代表用户通过与 namenode 和 datanode 交互来访问整个文件系统。客户端提供一个类似于 POSIX(可移植操作系统界面)的文件系统接口，因此用户在编程时无需知道 namenode 和 datanode 也可实现其功能。

datanode 是文件系统的工作节点。它们根据需要存储并检索数据块(受客户端或 namenode 调度)，并且定期向 namenode 发送它们所存储的块的列表。

没有 namenode，文件系统将无法使用。事实上，如果运行 namenode 服务的机器毁坏，文件系统上所有的文件将会丢失，因为我们不知道如何根据 datanode 的块重建文件。因此，对 namenode 实现容错非常重要，Hadoop 为此提供两种机制。

第一种机制是备份那些组成文件系统元数据持久状态的文件。Hadoop 可以通过配置使 namenode 在多个文件系统上保存元数据的持久状态。这些写操作是实时同步的，且是原子操作。一般的配置是，将持久状态写入本地磁盘的同时，写入一个远程挂载的网络文件系统(NFS)。

另一种可行的方法是运行一个辅助 namenode，但它不能被用作 namenode。这个辅助 namenode 的重要作用是定期合并编辑日志与命名空间镜像，以防止编辑日志过大。这个辅助 namenode 一般在另一台单独的物理计算机上运行，因为它需要占用大量 CPU 时间，并且需要与 namenode 一样多的内存来执行合并操作。它会保存合并后的命名空间镜像的副本，并在 namenode 发生故障时启用。但是，辅助 namenode 保存的状态总是滞后于主节点，所以在主节点全部失效时，难免会丢失部分数据。在这种情况下，一般把存储在 NFS 上的 namenode 元数据复制到辅助 namenode 并作为新的主 namenode 运行。(注意，也可以运行热备份 namenode 代替运行辅助 namenode，具体参见 3.2.5 节对 HDFS 高可用性的讨论。)

关于文件系统镜像和编辑日志的更多讨论，请参见 11.1.1 节。

3.2.3 块缓存

通常 datanode 从磁盘中读取块，但对于访问频繁的文件，其对应的块可能被显式地缓存在 datanode 的内存中，以堆外块缓存(off-heap *block cache*)的形式存在。默认情况下，一个块仅缓存在一个 datanode 的内存中，当然可以针每个文件配置 datanode 的数量。作业调度器(用于 MapReduce、Spark 和其他框架的)通过在缓存块的 datanode 上运行任务，可以利用块缓存的优势提高读操作的性能。例如，连接(join)操作中使用的一个小的查询表就是块缓存的一个很好的候选。

用户或应用通过在缓存池(*cache pool*)中增加一个 *cache directive* 来告诉 namenode 需要缓存哪些文件及存多久。缓存池是一个用于管理缓存权限和资源使用的管理性分组。

3.2.4 联邦 HDFS

namenode 在内存中保存文件系统中每个文件和每个数据块的引用关系，这意味着对于一个拥有大量文件的超大集群来说，内存将成为限制系统横向扩展的瓶颈(参见 10.3.2 节)。在 2.x 发行版本系列中引入的联邦 HDFS 允许系统通过添加 namenode 实现扩展，其中每个 namenode 管理文件系统命名空间中的一部分。例如，一个 namenode 可能管理*/user* 目录下的所有文件，而另一个 namenode 可能管理*/share* 目录下的所有文件。

在联邦环境下，每个 namenode 维护一个命名空间卷(namespace volume)，由命名空间的元数据和一个数据块池(block pool)组成，数据块池包含该命名空间下文件的所有数据块。命名空间卷之间是相互独立的，两两之间并不相互通信，甚至其中一个 namenode 的失效也不会影响由其他 namenode 维护的命名空间的可用性。数据块池不再进行切分，因此集群中的 datanode 需要注册到每个 namenode，并且存储着来自多个数据块池中的数据块。

要想访问联邦 HDFS 集群，客户端需要使用客户端挂载数据表将文件路径映射到 namenode。该功能可以通过 `ViewFileSystem` 和 `viewfs://URI` 进行配置和管理。

3.2.5 HDFS 的高可用性

通过联合使用在多个文件系统中备份 namenode 的元数据和通过备用 namenode 创建监测点能防止数据丢失，但是依旧无法实现文件系统的高可用性。namenode 依旧存在单点失效(SPOF, single point of failure)的问题。如果 namenode 失效了，那么所有的客户端，包括 MapReduce 作业，均无法读、写或列举(list)文件，因为 namenode 是唯一存储元数据与文件到数据块映射的地方。在这一情况下，Hadoop 系统无法提供服务直到有新的 namenode 上线。

在这样的情况下，要想从一个失效的 namenode 恢复，系统管理员得启动一个拥有文件系统元数据副本的新的 namenode，并配置 datanode 和客户端以便使用这个新的 namenode。新的 namenode 直到满足以下情形才能响应服务：(1)将命名空间的

映像导入内存中；(2)重演编辑日志；(3)接收到足够多的来自 datanode 的数据块报告并退出安全模式。对于一个大型并拥有大量文件和数据块的集群，namenode 的冷启动需要 30 分钟，甚至更长时间。

系统恢复时间太长，也会影响到日常维护。事实上，预期外的 namenode 失效出现概率很低，所以在现实中，计划内的系统失效时间实际更为重要。

Hadoop2 针对上述问题增加了对 HDFS 高可用性(HA)的支持。在这一实现中，配置了一对活动-备用(active-standby) namenode。当活动 namenode 失效，备用 namenode 就会接管它的任务并开始服务于来自客户端的请求，不会有任何明显中断。实现这一目标需要在架构上做如下修改。

- namenode 之间需要通过高可用共享存储实现编辑日志的共享。当备用 namenode 接管工作之后，它将通读共享编辑日志直至末尾，以实现与活动 namenode 的状态同步，并继续读取由活动 namenode 写入的新条目。
- datanode 需要同时向两个 namenode 发送数据块处理报告，因为数据块的映射信息存储在 namenode 的内存中，而非磁盘。
- 客户端需要使用特定的机制来处理 namenode 的失效问题，这一机制对用户是透明的。
- 辅助 namenode 的角色被备用 namenode 所包含，备用 namenode 为活动的 namenode 命名空间设置周期性检查点。

可以从两种高可用性共享存储做出选择：NFS 过滤器或群体日志管理器(QJM，*quorum journal manager*)。QJM 是一个专用的 HDFS 实现，为提供一个高可用的编辑日志而设计，被推荐用于大多数 HDFS 部署中。QJM 以一组日志节点(journal node)的形式运行，每一次编辑必须写入多数日志节点。典型的，有三个 journal 节点，所以系统能够忍受其中任何一个的丢失。这种安排与 ZooKeeper 的工作方式类似，当然必须认识到，QJM 的实现并没使用 ZooKeeper。(然而，值得注意的是，HDFS HA 在选取活动的 namenode 时确实使用了 ZooKeeper 技术，详情参见下一章。)

在活动 namenode 失效之后，备用 namenode 能够快速(几十秒的时间)实现任务接管，因为最新的状态存储在内存中：包括最新的编辑日志条目和最新的数据块映射信息。实际观察到的失效时间略长一点(需要 1 分钟左右)，这是因为系统需要保守确定活动 namenode 是否真的失效了。

在活动 namenode 失效且备用 namenode 也失效的情况下，当然这类情况发生的概率非常低，管理员依旧可以声明一个备用 namenode 并实现冷启动。这类情况并不会比非高可用(non-HA)的情况更差，并且从操作的角度讲这是一个进步，因为上述处理已是一个标准的处理过程并植入 Hadoop 中。

故障切换与规避

系统中有一个称为故障转移控制器(failover controller)的新实体，管理着将活动 namenode 转移为备用 namenode 的转换过程。有多种故障转移控制器，但默认的一种是使用了 ZooKeeper 来确保有且仅有一个活动 namenode。每一个 namenode 运行着一个轻量级的故障转移控制器，其工作就是监视宿主 namenode 是否失效(通过一个简单的心跳机制实现)并在 namenode 失效时进行故障切换。

管理员也可以手动发起故障转移，例如在进行日常维护时。这称为“平稳的故障转移”(*graceful failover*)，因为故障转移控制器可以组织两个 namenode 有序地切换角色。

但在非平稳故障转移的情况下，无法确切知道失效 namenode 是否已经停止运行。例如，在网速非常慢或者网络被分割的情况下，同样也可能激发故障转移，但是先前的活动 namenode 依然运行着并且依旧是活动 namenode。高可用实现做了更进一步的优化，以确保先前活动的 namenode 不会执行危害系统并导致系统崩溃的操作，该方法称为“规避”(fencing)。

同一时间 QJM 仅允许一个 namenode 向编辑日志中写入数据。然而，对于先前的活动 namenode 而言，仍有可能响应并处理客户过时的读请求，因此，设置一个 SSH 规避命令用于杀死 namenode 的进程是一个好主意。当使用 NFS 过滤器实现共享编辑日志时，由于不可能同一时间只允许一个 namenode 写入数据(这也是为什么推荐 QJM 的原因)，因此需要更有力的规避方法。规避机制包括：撤销 namenode 访问共享存储目录的权限（通常使用供应商指定的 NFS 命令）、通过远程管理命令屏蔽相应的网络端口。诉诸的最后手段是，先前活动 namenode 可以通过一个相当形象的称为“一枪爆头”*STONITH*，shoot the other node in the head)的技术进行规避，该方法主要通过一个特定的供电单元对相应主机进行断电操作。

客户端的故障转移通过客户端类库实现透明处理。最简单的实现是通过客户端的配置文件实现故障转移的控制。HDFS URI 使用一个逻辑主机名，该主机名映射到一对 namenode 地址(在配置文件中设置)，客户端类库会访问每一个 namenode 地址直至处理完成。

3.3 命令行接口

现在我们通过命令行交互来进一步认识 HDFS。HDFS 还有很多其他接口，但命令行是最简单的，同时也是许多开发者最熟悉的。

参照附录 A 中伪分布模式下设置 Hadoop 的说明，我们先在一台机器上运行 HDFS。稍后介绍如何在集群上运行 HDFS，以提供可扩展性与容错性。

在我们设置伪分布配置时，有两个属性项需要进一步解释。第一项是 `fs.defaultFS`，设置为 `hdfs://localhost/`，用于设置 Hadoop 的默认文件系统。[①]文件系统是由 URI 指定的，这里我们已使用 `hdfs` URI 来配置 HDFS 为 Hadoop 的默认文件系统。HDFS 的守护程序通过该属性项来确定 HDFS namenode 的主机及端口。我们将在 localhost 默认的 HDFS 端口 8020 上运行 namenode。这样一来，HDFS 客户端可以通过该属性得知 namenode 在哪里运行进而连接到它。

第二个属性 `dfs.replication`，我们设为 1，这样一来，HDFS 就不会按默认设置将文件系统块复本设为 3。在单独一个 datanode 上运行时，HDFS 无法将块复制到 3 个 datanode 上，所以会持续给出块复本不足的警告。设置这个属性之后，上述问题就不会再出现了。

文件系统的基本操作

至此，文件系统已经可以使用了，我们可以执行所有常用的文件系统操作，例如，读取文件，新建目录，移动文件，删除数据，列出目录，等等。可以输入 `hadoop fs -help` 命令获取每个命令的详细帮助文件。

首先从本地文件系统将一个文件复制到 HDFS：

```
% hadoop fs -copyFromLocal input/docs/quangle.txt \ hdfs://localhost/user/tom/quangle.txt
```

该命令调用 Hadoop 文件系统的 shell 命令 `fs`，后者提供了一系列子命令，在这个例子中，我们执行的是`-copyFromLocal`。本地文件 *quangle.txt* 被复制到运行在 localhost 上的 HDFS 实例中，路径为*/user/tom/quangle.txt*。事实上，我们可以简

① Hadoop 1 中，该属性的名称为 `fs.default.name`。Hadoop2 中使用了许多新的属性名称，旧名称均不再使用(详情参见 6.2.2 节)。本书使用的均为新的属性名称。

化命令格式以省略主机的 URI 并使用默认设置，即省略 hdfs://localhost，因为该项已在 *core-site.xml* 中指定。

```
% hadoop fs -copyFromLocal input/docs/quangle.txt /user/tom/quangle.txt
```

我们也可以使用相对路径，并将文件复制到 HDFS 的 home 目录中，本例中为 */user/tom*：

```
% hadoop fs -copyFromLocal input/docs/quangle.txt quangle.txt
```

我们把文件复制回本地文件系统，并检查是否一致：

```
% hadoop fs -copyToLocal quangle.txt quangle.copy.txt
% md5 input/docs/quangle.txt quangle.copy.txt
 MD5 (input/docs/quangle.txt) = e7891a2627cf263a079fb0f18256ffb2
 MD5 (quangle.copy.txt) = e7891a2627cf263a079fb0f18256ffb2
```

MD5 键值相同，表明这个文件在 HDFS 之旅中得以幸存并保存完整。

最后，看一下 HDFS 文件列表。我们新建一个目录，看它在列表中怎么显示：

```
% hadoop fs -mkdir books
% hadoop fs -ls .
Found 2 items
drwxr-xr-x - tom supergroup 0 2014-10-04 13:22 books
-rw-r--r-- 1 tom supergroup 119 2014-10-04 13:21 quangle.txt
```

返回的结果信息与 Unix 命令 ls -l 的输出结果非常相似，仅有细微差别。第 1 列显示的是文件模式。第 2 列是这个文件的备份数(这在传统 Unix 文件系统是没有的)。由于我们在整个文件系统范围内设置的默认复本数为 1，所以这里显示的也都是 1。这一列的开头目录为空，因为本例中没有使用复本的概念，目录作为元数据保存在 namenode 中，而非 datanode 中。第 3 列和第 4 列显示文件的所属用户和组别。第 5 列是文件的大小，以字节为单位，目录为 0。第 6 列和第 7 列是文件的最后修改日期与时间。最后，第 8 列是文件或目录的名称。

HDFS 中的文件访问权限

针对文件和目录，HDFS 的权限模式与 POSIX 的权限模式非常相似。

一共提供三类权限模式：只读权限(r)、写入权限(w)和可执行权限(x)。读取文件或列出目录内容时需要只读权限。写入一个文件或是在一个目录上新建及删除文件或目录，需要写入权限。对于文件而言，可执行权限可以忽略，因为你

不能在 HDFS 中执行文件(与 POSIX 不同)，但在访问一个目录的子项时需要该权限。

每个文件和目录都有所属用户(owner)、所属组别(group)及模式(mode)。这个模式是由所属用户的权限、组内成员的权限及其他用户的权限组成的。

在默认情况下，Hadoop 运行时安全措施处于停用模式，意味着客户端身份是没有经过认证的。由于客户端是远程的，一个客户端可以在远程系统上通过创建和任一个合法用户同名的账号来进行访问。当然，如果安全设施处于启用模式，这些都是不可能的(详情见 10.4 节关于安全性的有关论述)。无论怎样，为防止用户或自动工具及程序意外修改或删除文件系统的重要部分，启用权限控制还是很重要的(这也是默认的配置，参见 `dfs.permissions.enabled` 属性)

如果启用权限检查，就会检查所属用户权限，以确认客户端的用户名与所属用户是否匹配，另外也将检查所属组别权限，以确认该客户端是否是该用户组的成员；若不符，则检查其他权限。

这里有一个超级用户(super-user)的概念，超级用户是 namenode 进程的标识。对于超级用户，系统不会执行任何权限检查。

3.4 Hadoop 文件系统

Hadoop 有一个抽象的文件系统概念，HDFS 只是其中的一个实现。Java 抽象类 `org.apache.hadoop.fs.FileSystem` 定义了 Hadoop 中一个文件系统的客户端接口，并且该抽象类有几个具体实现，其中和 Hadoop 紧密相关的见表 3-1。

表 3-1. Hadoop 文件系统

文件系统	URI 方案	Java 实现(都在 org.apache.hadoop 包中)	描述
Local	file	fs.LocalFileSystem	使用客户端校验和的本地磁盘文件系统。使用 RawLocalFileSystem 表示无校验和的本地磁盘文件系统。详情参见 5.1.2 节
HDFS	hdfs	hdfs.DistributedFileSystem	Hadoop 的分布式文件系统。将 HDFS 设计成与 MapReduce 结合使用，可以实现高性能

续表

文件系统	URI 方案	Java 实现(都在 org.apache.hadoop 包中)	描述
WebHDFS	Webhdfs	Hdfs.web.WebHdfsFileSystem	基于 HTTP 的文件系统，提供对 HDFS 的认证读/写访问。详情参见 3.4 节相关内容
Secure WebHDFS	swebhdfs	hdfs.web.SWebHdfsFileSystem	WebHDFS 的 HTTPS 版本
HAR	har	fs.HarFileSystem	一个构建在其他文件系统之上用于文件存档的文件系统。Hadoop 存档文件系统通常用于将 HDFS 中的多个文件打包成一个存档文件，以减少 namenode 内存的使用。使用 hadoop 的 achive 命令来创建 HAR 文件
View	viewfs	viewfs.ViewFileSystem	针对其他 Hadoop 文件系统的客户端挂载表。通常用于为联邦 namenode 创建挂载点，详情参见 3.2.4 节
FTP	ftp	fs.ftp.FTPFileSystem	由 FTP 服务器支持的文件系统
S3	S3a	fs.s3a.S3AFileSystem	由 Amazon S3 支持的文件系统。替代老版本的 s3n(S3 原生)实现
Azure	wasb	fs.azure.NativeAzureFileSystem	由 Microsoft Azure 支持的文件系统
Swift	swift	fs.swift.snative.SwiftNativeFileSystem	由 OpenStack Swift 支持的文件系统

Hadoop 对文件系统提供了许多接口，它一般使用 URI 方案来选取合适的文件系统实例进行交互。举例来说，我们在前一小节中遇到的文件系统命令行解释器可以操作所有的 Hadoop 文件系统命令。要想列出本地文件系统根目录下的文件，可以输入以下命令：

```
% hadoop fs -ls file:///
```

尽管运行的 MapReduce 程序可以访问任何文件系统(有时也很方便)，但在处理大数据集时，建议你还是选择一个有数据本地优化的分布式文件系统，如 HDFS(参见 2.4 节)。

接口

Hadoop 是用 Java 写的，通过 Java API 可以调用大部分 Hadoop 文件系统的交互操作。例如，文件系统的命令解释器就是一个 Java 应用，它使用 Java 的 FileSystem 类来提供文件系统操作。其他一些文件系统接口也将在本小节中做简

单介绍。这些接口通常与 HDFS 一同使用，因为 Hadoop 中的其他文件系统一般都有访问基本文件系统的工具(对于 FTP，有 FTP 客户端；对于 S3，有 S3 工具，等等)，但它们大多数都能用于任何 Hadoop 文件系统。

1. HTTP

Hadoop 以 Java API 的形式提供文件系统访问接口，非 Java 开发的应用访问 HDFS 会很不方便。由 WebHDFS 协议提供的 HTTP REST API 则使得其他语言开发的应用能够更方便地与 HDFS 交互。注意，HTTP 接口比原生的 Java 客户端要慢，所以不到万不得已，尽量不要用它来传输特大数据。

通过 HTTP 来访问 HDFS 有两种方法：直接访问，HDFS 守护进程直接服务于来自客户端的 HTTP 请求；通过代理(一个或多个)访问，客户端通常使用 `DistributedFileSystem` API 访问 HDFS。这两种方法如图 3-1 所示。两者都使用了 WebHDFS 协议。

图 3-1. 通过 HTTP 直接访问 HDFS 或者通过多个 HDFS 代理访问 HDFS

在第一种情况中，namenode 和 datanode 内嵌的 web 服务器作为 WebHDFS 的端节点运行。(由于 `dfs.webhdfs.enabled` 被设置为 `true`，WebHDFS 默认是启用状

态。)文件元数据操作由 namenode 管理，文件读(写)操作首先被发往 namenode，由 namenode 发送一个 HTTP 重定向至某个客户端，指示以流方式传输文件数据的目的或源 datanode。

第二种方法依靠一个或者多个独立代理服务器通过 HTTP 访问 HDFS。(由于代理服务是无状态的，因此可以运行在标准的负载均衡器之后。)所有到集群的网络通信都需要经过代理，因此客户端从来不直接访问 namenode 或 datanode。使用代理服务器后可以使用更严格的防火墙策略和带宽限制策略。通常情况下都通过代理服务器，实现在不同数据中心中部署的 Hadoop 集群之间的数据传输，或从外部网络访问云端运行的 Hadoop 集群。

HttpFS 代理提供和 WebHDFS 相同的 HTTP(和 HTTPS)接口，这样客户端能够通过 `webhdfs(swebhdfs)` URI 访问这两类接口。HttpFS 代理的启动独立于 namenode 和 datanode 的守护进程，使用 *httpfs.sh* 脚本，默认在一个不同的端口上监听(端口号 14000)。

2. C 语言

Hadoop 提供了一个名为 *libhdfs* 的 C 语言库，该语言库是 Java `FileSystem` 接口类的一个镜像(它被写成访问 HDFS 的 C 语言库，但其实它可以访问任何一个 Hadoop 文件系统)。它使用 Java 原生接口(JNI, Java Native Interface)调用 Java 文件系统客户端。同样还有一个 *libwebhdfs* 库，该库使用了前述章节描述的 WebHDFS 接口。

这个 C 语言 API 与 Java 的 API 非常相似，但它的开发滞后于 Java API，因此目前一些新的特性可能还不支持。可以在 Apache Hapdoop 二进制压缩发行包的 *include* 目录中找到头文件 *hdfs.h*。

Apache Hapdoop 二进制压缩包自带预先编译好的 *libhdfs* 二进制编码，支持 64 位 Linux。但对于其他平台，需要按照源代码树顶层的 *BUILDING.txt* 指南自行编译。

3. NFS

使用 Hadoop 的 NFSv3 网关将 HDFS 挂载为本地客户端的文件系统是可行的。然后你可以使用 Unix 实用程序(如 `ls` 和 `cat`) 与该文件系统交互，上传文件，通过任意一种编程语言调用 POSIX 库来访问文件系统。由于 HDFS 仅能以追加模式写文件，因此可以往文件末尾添加数据，但不能随机修改文件。

关于如何配置和运行 NFS 网关，以及如何从客户端连接网关，可以参考 Hadoop 相关文档资料。

4. FUSE

用户空间文件系统(FUSE, Filesystem in Userspace，)允许将用户空间实现的文件系统作为 Unix 文件系统进行集成。通过使用 Hadoop 的 Fuse-DFS 功能模块，HDFS(或任何一个 Hadoop 文件系统)均可以作为一个标准的本地文件系统进行挂载。Fuse-DFS 是用 C 语言实现的，使用 *libhdfs* 作为访问 HDFS 的接口。在写操作时，Hadoop NFS 网关对于挂载 HDFS 来说是更健壮的解决方案，相比 Fuse-DFS 而言应优先选择。

3.5 Java 接口

在本小节中，我们要深入探索 Hadoop 的 `Filesystem` 类：它是与 Hadoop 的某一文件系统进行交互的 API。[①]虽然我们主要聚焦于 HDFS 实例，即 `DistributedFileSystem`，但总体来说，还是应该集成 `FileSystem` 抽象类，并编写代码，使其在不同文件系统中可移植。这对测试你编写的程序非常重要，例如，你可以使用本地文件系统中的存储数据快速进行测试。

3.5.1 从 Hadoop URL 读取数据

要从 Hadoop 文件系统读取文件，最简单的方法是使用 `java.net.URL` 对象打开数据流，从中读取数据。具体格式如下：

```
InputStream in = null;
 try {
     in = new URL("hdfs://host/path").openStream();
     // process in
 } finally {
     IOUtils.closeStream(in);
 }
```

让 Java 程序能够识别 Hadoop 的 `hdfs URL` 方案还需要一些额外的工作。这里采用的方法是通过 `FsUrlStreamHandlerFactory` 实例调用 `java.net.URL` 对象的

① 在 Hadoop 2 及后续版本中，新增一个名为 `FileContext` 的文件系统接口，该接口能够更好地处理多文件系统问题(例如，单个 `FileContext` 接口能够解决多文件系统方案)，并且该接口更简明，更一致。然而，`FileSystem` 仍然在广泛使用中。

setURLStreamHandlerFactory()方法。每个 Java 虚拟机只能调用一次这个方法，因此通常在静态方法中调用。这个限制意味着如果程序的其他组件(如不受你控制的第三方组件)已经声明一个 URLStreamHandlerFactory 实例，你将无法使用这种方法从 Hadoop 中读取数据。下一节将讨论另一种备选方法。

范例 3-1 展示的程序以标准输出方式显示 Hadoop 文件系统中的文件，类似于 Unix 中的 cat 命令。

范例 3-1. 通过 URLStreamHandler 实例以标准输出方式显示 Hadoop 文件系统的文件

```
public class URLCat {

  static {
    URL.setURLStreamHandlerFactory(new FsUrlStreamHandlerFactory());
  }

  public static void main(String[] args) throws Exception {
    InputStream in = null;
    try {
      in = new URL(args[0]).openStream();
      IOUtils.copyBytes(in, System.out, 4096, false);
    } finally {
      IOUtils.closeStream(in);
    }
  }
}
```

我们可以调用 Hadoop 中简洁的 IOUtils 类，并在 finally 子句中关闭数据流，同时也可以在输入流和输出流之间复制数据(本例中为 System.out)。copyBytes 方法的最后两个参数，第一个设置用于复制的缓冲区大小，第二个设置复制结束后是否关闭数据流。这里我们选择自行关闭输入流，因而 System.out 不必关闭输入流。

下面是一个运行示例：[1]

```
% export HADOOP_CLASSPATH=hadoop-examples.jar
% hadoop URLCat hdfs://localhost/user/tom/quangle.txt
 On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

① 这段文字来自爱德华·李尔(Edward Lear)的诗歌“The Quangle Wangle's Hat”。

3.5.2 通过 FileSystem API 读取数据

正如前一小节所解释的，有时根本不可能在应用中设置 URLStreamHandlerFactory 实例。在这种情况下，我们需要用 FileSystem API 来打开一个文件的输入流。

Hadoop 文件系统中通过 Hadoop Path 对象(而非 java.io.File 对象，因为它的语义与本地文件系统联系太紧密)来代表文件。可以将路径视为一个 Hadoop 文件系统 URI，如 hdfs://localhost/user/tom/quangle.txt。

FileSystem 是一个通用的文件系统 API，所以第一步是检索我们需要使用的文件系统实例，这里是 HDFS。获取 FileSystem 实例有下面这几个静态工厂方法：

```
public static FileSystem get(Configuration conf) throws IOException
Public static FileSystem get(URI uri, Configuration conf) throws IOException
public static FileSystem get(URI uri, Configuration conf, String user)
  throws IOException
```

Configuration 对象封装了客户端或服务器的配置，通过设置配置文件读取类路径来实现(如 *etc/hadoop/core-site.xml*)。第一个方法返回的是默认文件系统(在 *core-site.xml* 中指定的，如果没有指定，则使用默认的本地文件系统)。第二个方法通过给定的 URI 方案和权限来确定要使用的文件系统，如果给定 URI 中没有指定方案，则返回默认文件系统。第三，作为给定用户来访问文件系统，对安全来说是至关重要。详情可以参见 10.4 节。

在某些情况下，你可能希望获取本地文件系统的运行实例，此时你可以使用的 getLocal()方法很方便地获取。

```
public static LocalFileSystem getLocal(Configuration conf) throws IOException
```

有了 FileSystem 实例之后，我们调用 open()函数来获取文件的输入流：

```
Public FSDataInputStream open(Path f) throws IOException
Public abstract FSDataInputStream open(Path f, int bufferSize) throws IOException
```

第一个方法使用默认的缓冲区大小 4 KB。

最后，我们重写范例 3-1，得到范例 3-2。

范例 3-2. 直接使用 FileSystem 以标准输出格式显示 Hadoop 文件系统中的文件

```
public class FileSystemCat {
  public static void main(String[] args) throws Exception {
    String uri = args[0];
```

```
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);
    InputStream in = null;
    try {
      in = fs.open(new Path(uri));
      IOUtils.copyBytes(in, System.out, 4096, false);
    } finally {
      IOUtils.closeStream(in);
    }
  }
}
```

程序运行结果如下：

```
% hadoop FileSystemCat hdfs://localhost/user/tom/quangle.txt
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

FSDataInputStream 对象

实际上，FileSystem 对象中的 open()方法返回的是 FSDataInputStream 对象，而不是标准的 java.io 类对象。这个类是继承了 java.io.DataInputStream 的一个特殊类，并支持随机访问，由此可以从流的任意位置读取数据。

```
package org.apache.hadoop.fs;

public class FSDataInputStream extends DataInputStream
     implements Seekable, PositionedReadable {
     // implementation elided
 }
```

Seekable 接口支持在文件中找到指定位置，并提供一个查询当前位置相对于文件起始位置偏移量(getPos())的查询方法：

```
public interface Seekable {
  void seek(long pos) throws IOException;
  long getPos() throws IOException;
}
```

调用 seek()来定位大于文件长度的位置会引发 IOException 异常。与 java.io.InputStream 的 skip()不同，seek()可以移到文件中任意一个绝对位置，skip()则只能相对于当前位置定位到另一个新位置。

范例 3-3 是对范例 3-2 的简单扩展，它将一个文件写入标准输出两次：在一次写完

之后，定位到文件的起始位置再次以流方式读取该文件并输出。

范例 3-3. 使用 seek()方法，将 Hadoop 文件系统中的一个文件在标准输出上显示两次

```
public class FileSystemDoubleCat {

  public static void main(String[] args) throws Exception {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);
    FSDataInputStream in = null;
    try {
      in = fs.open(new Path(uri));
      IOUtils.copyBytes(in, System.out, 4096, false);
      in.seek(0); // go back to the start of the file
      IOUtils.copyBytes(in, System.out, 4096, false);
    } finally {
      IOUtils.closeStream(in);
    }
  }
}
```

在一个小文件上运行的结果如下：

```
% hadoop FileSystemDoubleCat hdfs://localhost/user/tom/quangle.txt
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

FSDataInputStream 类也实现了 PositionedReadable 接口，从一个指定偏移量处读取文件的一部分：

```
public interface PositionedReadable {

  public int read(long position, byte[] buffer, int offset, int length)
        throws IOException;

  public void readFully(long position, byte[] buffer, int offset, int length)
  throws IOException;

  public void readFully(long position, byte[] buffer) throws IOException;
}
```

read()方法从文件的指定 position 处读取至多为 length 字节的数据并存入缓冲区 buffer 的指定偏离量 offset 处。返回值是实际读到的字节数：调用者需要检查这个值，它有可能小于指定的 length 长度。readFully()方法将指定

length 长度的字节数数据读取到 buffer 中(或在只接受 buffer 字节数组的版本中，读取 buffer.length 长度字节数据)，除非已经读到文件末尾，这种情况下将抛出 EOFException 异常。

所有这些方法会保留文件当前偏移量，并且是线程安全的(FSDataInputStrean 并不是为并发访问设计的，因此最好为此新建多个实例)，因此它们提供了在读取文件的主体时，访问文件其他部分(可能是元数据)的便利方法。

最后务必牢记，seek()方法是一个相对高开销的操作，需要慎重使用。建议用流数据来构建应用的访问模式(比如使用 MapReduce)，而非执行大量 seek()方法。

3.5.3 写入数据

FileSystem 类有一系列新建文件的方法。最简单的方法是给准备建的文件指定一个 Path 对象，然后返回一个用于写入数据的输出流：

```
public FSDataOutputStream create(Path f) throws IOException
```

此方法有多个重载版本，允许我们指定是否需要强制覆盖现有的文件、文件备份数量、写入文件时所用缓冲区大小、文件块大小以及文件权限。

create()方法能够为需要写入且当前不存在的文件创建父目录。尽管这样很方便，但有时并不希望这样。如果希望父目录不存在就导致文件写入失败，则应该先调用 exists()方法检查父目录是否存在。另一种方案是使用 FileContext，允许你可以控制是否创建父目录。

还有一个重载方法 Progressable 用于传递回调接口，如此一来，可以把数据写入 datanode 的进度通知给应用：

```
package org.apache.hadoop.util;

public interface Progressable {
  public void progress();
}
```

另一种新建文件的方法是使用 append()方法在一个现有文件末尾追加数据(还有其他一些重载版本)：

```
public FSDataOutputStream append(Path f) throws IOException
```

这样的追加操作允许一个 writer 打开文件后在访问该文件的最后偏移量处追加数据。有了这个 API，某些应用可以创建无边界文件，例如，应用可以在关闭日志

文件之后继续追加日志。该追加操作是可选的，并非所有 Hadoop 文件系统都实现了该操作。例如，HDFS 支持追加，但 S3 文件系统就不支持。

范例 3-4 显示了如何将本地文件复制到 Hadoop 文件系统。每次 Hadoop 调用 progress()方法时，也就是每次将 64 KB 数据包写入 datanode 管线后，打印一个时间点来显示整个运行过程。注意，这个操作并不是通过 API 实现的，因此 Hadoop 后续版本能否执行该操作，取决于该版本是否修改过上述操作。API 只是让你知道“正在发生什么事情”。

范例 3-4. 将本地文件复制到 Hadoop 文件系统

```
public class FileCopyWithProgress {
  public static void main(String[] args) throws Exception {
    String localSrc = args[0];
    String dst = args[1];
    InputStream in = new BufferedInputStream(new FileInputStream(localSrc));

    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(dst), conf);
    OutputStream out = fs.create(new Path(dst), new Progressable() {
      public void progress() {
        System.out.print(".");
      }
    });

    IOUtils.copyBytes(in, out, 4096, true);
  }
}
```

典型应用如下：

```
% hadoop FileCopyWithProgress input/docs/1400-8.txt
hdfs://localhost/user/tom/1400-8.txt
 ................
```

目前，其他 Hadoop 文件系统写入文件时均不调用 progress()方法。后面几章将展示进度对 MapReduce 应用的重要性。

FSDataOutputStream 对象

FileSystem 实例的 create()方法返回 FSDataOutputStream 对象，与 FSDataInputStream 类相似，它也有一个查询文件当前位置的方法：

```
package org.apache.hadoop.fs;
public class FSDataOutputStream extends DataOutputStream implements Syncable {
  public long getPos() throws IOException {
```

```
    // implementation elided
  }

  // implementation elided
}
```

但与 FSDataInputStream 类不同的是，FSDataOutputStream 类不允许在文件中定位。这是因为 HDFS 只允许对一个已打开的文件顺序写入，或在现有文件的末尾追加数据。换句话说，它不支持在除文件末尾之外的其他位置进行写入，因此，写入时定位就没有什么意义。

3.5.4 目录

Filesystem 实例提供了创建目录的方法：

```
public boolean mkdirs(Path f) throws IOException
```

这个方法可以一次性新建所有必要但还没有的父目录，就像 java.io.File 类的 mkdirs()方法。如果目录(以及所有父目录)都已经创建成功，则返回 true。

通常，你不需要显式创建一个目录，因为调用 create()方法写入文件时会自动创建父目录。

3.5.5 查询文件系统

1. 文件元数据：FileStatus

任何文件系统的一个重要特征都是提供其目录结构浏览和检索它所存文件和目录相关信息的功能。FileStatus 类封装了文件系统中文件和目录的元数据，包括文件长度、块大小、复本、修改时间、所有者以及权限信息。

FileSystem 的 getFileStatus()方法用于获取文件或目录的 FileStatus 对象。范例 3-5 显示了它的用法。

范例 3-5. 展示文件状态信息

```
public class ShowFileStatusTest {

  private MiniDFSCluster cluster; // use an in-process HDFS cluster for testing
  private FileSystem fs;

  @Before
  public void setUp() throws IOException {
    Configuration conf = new Configuration();
    if (System.getProperty("test.build.data") == null) {
      System.setProperty("test.build.data", "/tmp");
```

```
    }
    cluster = new MiniDFSCluster.Builder(conf).build();
    fs = cluster.getFileSystem();
    OutputStream out = fs.create(new Path("/dir/file"));
    out.write("content".getBytes("UTF-8"));
    out.close();
  }

  @After
  public void tearDown() throws IOException {
    if (fs != null) { fs.close(); }
    if (cluster != null) { cluster.shutdown(); }
  }

  @Test(expected = FileNotFoundException.class)
  public void throwsFileNotFoundForNonExistentFile() throws IOException {
    fs.getFileStatus(new Path("no-such-file"));
  }

  @Test
  public void fileStatusForFile() throws IOException {
    Path file = new Path("/dir/file");
    FileStatus stat = fs.getFileStatus(file);
    assertThat(stat.getPath().toUri().getPath(), is("/dir/file"));
    assertThat(stat.isDirectory(), is(false));
    assertThat(stat.getLen(), is(7L));
    assertThat(stat.getModificationTime(),
        is(lessThanOrEqualTo(System.currentTimeMillis())));
    assertThat(stat.getReplication(), is((short) 1));
    assertThat(stat.getBlockSize(), is(128 * 1024 * 1024L));
    assertThat(stat.getOwner(), is(System.getProperty("user.name")));
    assertThat(stat.getGroup(), is("supergroup"));
    assertThat(stat.getPermission().toString(), is("rw-r--r--"));
  }

  @Test
  public void fileStatusForDirectory() throws IOException {
    Path dir = new Path("/dir");
    FileStatus stat = fs.getFileStatus(dir);
    assertThat(stat.getPath().toUri().getPath(), is("/dir"));
    assertThat(stat.isDirectory(), is(true));
    assertThat(stat.getLen(), is(0L));
    assertThat(stat.getModificationTime(),
        is(lessThanOrEqualTo(System.currentTimeMillis())));
    assertThat(stat.getReplication(), is((short) 0));
    assertThat(stat.getBlockSize(), is(0L));
    assertThat(stat.getOwner(),is(System.getProperty("user.name")));
    assertThat(stat.getGroup(), is("supergroup"));
    assertThat(stat.getPermission().toString(), is("rwxr-xr-x"));
  }

}
```

如果文件或目录均不存在，会抛出一个 FileNotFoundException 异常。但如果只是想检查文件或目录是否存在，那么调用 exists()方法会更方便：

```
public boolean exists(Path f) throws IOException
```

2. 列出文件

查找一个文件或目录相关的信息很实用，但通常还需要能够列出目录中的内容。这就是 FileSystem 的 listStatus()方法的功能：

```
public FileStatus[] listStatus(Path f) throws IOException
public FileStatus[] listStatus(Path f, PathFilter filter) throws IOException
public FileStatus[] listStatus(Path[] files) throws IOException
public FileStatus[] listStatus(Path[] files, PathFilter filter) throws IOException
```

当传入的参数是一个文件时，它会简单转变成以数组方式返回长度为 1 的 FileStatus 对象。当传入参数是一个目录时，则返回 0 或多个 FileStatus 对象，表示此目录中包含的文件和目录。

它的重载方法允许使用 PathFilter 来限制匹配的文件和目录，可以参见 3.5.5 节提供的例子。最后，如果指定一组路径，其执行结果相当于依次轮流传递每条路径并对其调用 listStatus()方法，再将 FileStatus 对象数组累积存入同一数组中，但该方法更为方便。在从文件系统树的不同分支构建输入文件列表时，这是很有用的。范例 3-6 简单显示了这个方法。注意 Hadoop 的 FileUtil 中 stat2Paths()方法的使用，它将一个 FileStatus 对象数组转换为一个 Path 对象数组。

范例 3-6. 显示 Hadoop 文件系统中一组路径的文件信息

```
public class ListStatus {

  public static void main(String[] args) throws Exception {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);

    Path[] paths = new Path[args.length];
    for (int i = 0; i < paths.length; i++) {
      paths[i] = new Path(args[i]);
    }
    FileStatus[] status = fs.listStatus(paths);
    Path[] listedPaths = FileUtil.stat2Paths(status);
    for (Path p : listedPaths) {
      System.out.println(p);
    }
  }
}
```

我们可以用这个程序来显示一组路径集目录列表的并集：

```
% hadoop ListStatus hdfs://localhost/ hdfs://localhost/user/tom
hdfs://localhost/user
hdfs://localhost/user/tom/books
hdfs://localhost/user/tom/quangle.txt
```

3. 文件模式

在单个操作中处理一批文件是一个很常见的需求。例如，一个用于处理日志的 MapReduce 作业可能需要分析一个月内包含在大量目录中的日志文件。在一个表达式中使用通配符来匹配多个文件是比较方便的，无需列举每个文件和目录来指定输入，该操作称为“通配”(globbing)。Hadoop 为执行通配提供了两个 FileSystem 方法：

```
public FileStatus[] globStatus(Path pathPattern) throws IOException
public FileStatus[] globStatus(Path pathPattern, PathFilter filter)
    throws IOException
```

globStatus()方法返回路径格式与指定模式匹配的所有 FileStatus 对象组成的数组，并按路径排序。PathFilter 命令作为可选项可以进一步对匹配结果进行限制。

Hadoop 支持的通配符与 Unix bash shell 支持的相同(参见表 3-2)。

表 3-2. 通配符及其含义

通配符	名称	匹配
*	星号	匹配 0 或多个字符
?	问号	匹配单一字符
[ab]	字符类	匹配{a,b}集合中的一个字符
[^ab]	非字符类	匹配非{a,b}集合中的一个字符
[a-b]	字符范围	匹配一个在{a,b}范围内的字符(包括 ab)，a 在字典顺序上要小于或等于 b
[^a-b]	非字符范围	匹配一个不在{a,b}范围内的字符(包括 ab)，a 在字典顺序上要小于或等于 b
{a,b}	或选择	匹配包含 a 或 b 中的一个的表达式
\c	转义字符	匹配元字符 c

假设有日志文件存储在按日期分层组织的目录结构中。如此一来，2007 年最后一天的日志文件就会保存在名为*/2007/12/31* 的目录中。假设整个文件列表如下所示：

```
/
├── 2007/
│     └─12/
│         ├─30/
│         └─31/
└── 2008/
      └─01/
        ├─01/
        └─02/
```

一些文件通配符及其扩展如下所示。

通配符	扩展
/*	/2007/2008
/*/*	/2007/12/2008/01
/*/12/*	/2007/12/30/2007/12/31
/200?	/2007/2008
/200[78]	/2007/2008
/200[7-8]	/2007/2008
/200[^01234569]	/2007/2008
/*/*/{31,01}	/2007/12/31/2008/01/01
/*/*/3{0,1}	/2007/12/30/2007/12/31
/*/{12/31,01/01}	/2007/12/31/2008/01/01

4. PathFilter 对象

通配符模式并不总能够精确地描述我们想要访问的文件集。比如，使用通配格式排除一个特定的文件就不太可能。FileSystem 中的 listStatus()和 globStatus()方法提供了可选的 PathFilter 对象，以编程方式控制通配符：

```
package org.apache.hadoop.fs;

public interface PathFilter {
   boolean accept(Path path);
}
```

PathFilter 与 java.io.FileFilter 一样，是 Path 对象而不是 File 对象。

范例 3-7 显示了 PathFilter 用于排除匹配正则表达式的路径。

范例 3-7. PathFilter 用于排除匹配正则表达式的路径

```
public class RegexExcludePathFilter implements PathFilter {

  private final String regex;

  public RegexExcludePathFilter(String regex) {
    this.regex = regex;
  }

  public boolean accept(Path path) {
    return !path.toString().matches(regex);
  }
}
```

这个过滤器只传递不匹配于正则表达式的文件。在通配符选出一组需要包含的初始文件之后，过滤器可优化其结果。如下示例将扩展到/2007/12/30：

```
fs.globStatus(new Path("/2007/*/*"), new RegexExcludeFilter("^.*/2007/12/31$"))
```

以 Path 为代表，过滤器只能作用于文件名。不能针对文件的属性(例如创建时间)来构建过滤器。但是，过滤器却能实现通配符模式和正则表达式都无法完成的匹配任务。例如，如果将文件存储在按照日期排列的目录结构中(如前一节中讲述的那样)，则可以写一个 Pathfilter 选出给定时间范围内的文件。

3.5.6 删除数据

使用 FileSystem 的 delete()方法可以永久性删除文件或目录。

```
public boolean delete(Path f, boolean recursive) throws IOException
```

如果 f 是一个文件或空目录，那么 recursive 的值就会被忽略。只有在 recrusive 值为 true 时，非空目录及其内容才会被删除(否则会抛出 IOException 异常)。

3.6 数据流

3.6.1 剖析文件读取

为了了解客户端及与之交互的 HDFS、namenode 和 datanode 之间的数据流是什么样的，我们可参考图 3-2，该图显示了在读取文件时事件的发生顺序。

客户端通过调用 FileSyste 对象的 open()方法来打开希望读取的文件，对于

HDFS 来说，这个对象是 DistributedFileSystem 的一个实例(图 3-2 中的步骤 1)。DistributedFileSystem 通过使用远程过程调用(RPC)来调用 namenode，以确定文件起始块的位置(步骤 2)。对于每一个块，namenode 返回存有该块副本的 datanode 地址。此外，这些 datanode 根据它们与客户端的距离来排序(根据集群的网络拓扑；参见 3.6.1 节的补充材料)。如果该客户端本身就是一个 datanode (比如，在一个 MapReduce 任务中)，那么该客户端将会从保存有相应数据块复本的本地 datanode 读取数据(参见图 2-2 及 10.3.5 节)。

图 3-2. 客户端读取 HDFS 中的数据

DistributedFileSystem 类返回一个 FSDataInputStream 对象(一个支持文件定位的输入流)给客户端以便读取数据。FSDataInputStream 类转而封装 DFSInputStream 对象，该对象管理着 datanode 和 namenode 的 I/O。

接着，客户端对这个输入流调用 read()方法(步骤 3)。存储着文件起始几个块的 datanode 地址的 DFSInputStream 随即连接距离最近的文件中第一个块所在的 datanode。通过对数据流反复调用 read()方法，可以将数据从 datanode 传输到客户端(步骤 4)。到达块的末端时，DFSInputStream 关闭与该 datanode 的连接，然后寻找下一个块的最佳 datanode(步骤 5)。所有这些对于客户端都是透明的，在客户看来它一直在读取一个连续的流。

客户端从流中读取数据时，块是按照打开 DFSInputStream 与 datanode 新建连接的顺序读取的。它也会根据需要询问 namenode 来检索下一批数据块的 datanode 的位置。一旦客户端完成读取，就对 FSDataInputStream 调用 close()方法(步骤 6)。

在读取数据的时候，如果 DFSInputStream 在与 datanode 通信时遇到错误，会尝试从这个块的另外一个最邻近 datanode 读取数据。它也记住那个故障 datanode，以保证以后不会反复读取该节点上后续的块。DFSInputStream 也会通过校验和确认从 datanode 发来的数据是否完整。如果发现有损坏的块，DFSInputStream 会试图从其他 datanode 读取其复本，也会将被损坏的块通知给 namenode。

这个设计的一个重点是，客户端可以直接连接到 datanode 检索数据，且 namenode 告知客户端每个块所在的最佳 datanode。由于数据流分散在集群中的所有 datanode，所以这种设计能使 HDFS 扩展到大量的并发客户端。同时，namenode 只需要响应块位置的请求(这些信息存储在内存中，因而非常高效)，无需响应数据请求，否则随着客户端数量的增长，namenode 会很快成为瓶颈。

网络拓扑与 Hadoop

在本地网络中，两个节点被称为“彼此近邻”是什么意思？在海量数据处理中，其主要限制因素是节点之间数据的传输速率——带宽很稀缺。这里的想法是将两个节点间的带宽作为距离的衡量标准。

不用衡量节点之间的带宽，实际上很难实现(它需要一个稳定的集群，并且在集群中两两节点对数量是节点数量的平方)，adoop 为此采用一个简单的方法：把网络看作一棵树，两个节点间的距离是它们到最近共同祖先的距离总和。该树中的层次是没有预先设定的，但是相对于数据中心、机架和正在运行的节点，通常可以设定等级。具体想法是针对以下每个场景，可用带宽依次递减：

- 同一节点上的进程
- 同一机架上的不同节点
- 同一数据中心中不同机架上的节点
- 不同数据中心中的节点[①]

例如，假设有数据中心 *d1* 机架 *r1* 中的节点 *n1*。该节点可以表示为*/d1/r1/n1*。利用这种标记，这里给出四种距离描述：

① 在本书写作期间，Hadoop 仍然不适合跨数据中心运行。

- *distance(/d1/r1/n1, /d1/r1/n1)*=0(同一节点上的进程)
- *distance(/d1/r1/n1, /d1/r1/n2)*=2(同一机架上的不同节点)
- *distance(/d1/r1/n1, /d1/r2/n3)*=4(同一数据中心中不同机架上的节点)
- *distance(/d1/r1/n1, /d2/r3/n4)*=6(不同数据中心中的节点)

示意图参见图 3-3(数学爱好者会注意到，这是一个测量距离的例子)。

最后，我们必须意识到 Hadoop 无法自动发现你的网络拓扑结构。它需要一些帮助(我们将在 10.1.2 节的“网络拓扑”中讨论如何配置网络拓扑)。不过在默认情况下，假设网络是扁平化的只有一层，或换句话说，所有节点都在同一数据中心的同一机架上。规模小的集群可能如此，不需要进一步配置。

图 3-3. Hadoop 中的网络距离

3.6.2 剖析文件写入

接下来我们看看文件是如何写入 HDFS 的。尽管比较详细，但对于理解数据流还是很有用的，因为它清楚地说明了 HDFS 的一致模型。

我们要考虑的情况是如何新建一个文件，把数据写入该文件，最后关闭该文件。如图 3-4 所示。

客户端通过对 `DistributedFileSystem` 对象调用 `create()`来新建文件(图 3-4 中的步骤 1)。`DistributedFileSystem` 对 namenode 创建一个 RPC 调用，在文件

系统的命名空间中新建一个文件，此时该文件中还没有相应的数据块(步骤 2)。namenode 执行各种不同的检查以确保这个文件不存在以及客户端有新建该文件的权限。如果这些检查均通过，namenode 就会为创建新文件记录一条记录；否则，文件创建失败并向客户端抛出一个 IOException 异常。DistributedFileSystem 向客户端返回一个 FSDataOutputStream 对象，由此客户端可以开始写入数据。就像读取事件一样，FSDataOutputStream 封装一个 DFSoutPutstream 对象，该对象负责处理 datanode 和 namenode 之间的通信。

图 3-4. 客户端将数据写入 HDFS

在客户端写入数据时(步骤 3)，DFSOutputStream 将它分成一个个的数据包，并写入内部队列，称为“数据队列”(data queue)。DataStreamer 处理数据队列，它的责任是挑选出适合存储数据复本的一组 datanode，并据此来要求 namenode 分配新的数据块。这一组 datanode 构成一个管线——我们假设复本数为 3，所以管线中有 3 个节点。DataStreamer 将数据包流式传输到管线中第 1 个 datanode，该 datanode 存储数据包并将它发送到管线中的第 2 个 datanode。同样，第 2 个 datanode 存储该数据包并且发送给管线中的第 3 个(也是最后一个)datanode (步骤 4)。

DFSOutputStream 也维护着一个内部数据包队列来等待 datanode 的收到确认回执，称为“确认队列”(ack queue)。收到管道中所有 datanode 确认信息后，该数

据包才会从确认队列删除(步骤 5)。

如果任何 datanode 在数据写入期间发生故障，则执行以下操作(对写入数据的客户端是透明的)。首先关闭管线，确认把队列中的所有数据包都添加回数据队列的最前端，以确保故障节点下游的 datanode 不会漏掉任何一个数据包。为存储在另一正常 datanode 的当前数据块指定一个新的标识，并将该标识传送给 namenode，以便故障 datanode 在恢复后可以删除存储的部分数据块。从管线中删除故障 datanode，基于两个正常 datanode 构建一条新管线。余下的数据块写入管线中正常的 datanode。namenode 注意到块复本量不足时，会在另一个节点上创建一个新的复本。后续的数据块继续正常接受处理。

在一个块被写入期间可能会有多个 datanode 同时发生故障，但非常少见。只要写入了 `dfs.namenode.replication.min` 的复本数(默认为 1)，写操作就会成功，并且这个块可以在集群中异步复制，直到达到其目标复本数(`dfs.replication` 的默认值为 3)。

客户端完成数据的写入后，对数据流调用 `close()`方法(步骤 6)。该操作将剩余的所有数据包写入 datanode 管线，并在联系到 namenode 告知其文件写入完成之前，等待确认(步骤 7)。namenode 已经知道文件由哪些块组成(因为 `Datastreamer` 请求分配数据块)，所以它在返回成功前只需要等待数据块进行最小量的复制。

复本怎么放

namenode 如何选择在哪个 datanode 存储**复本**(replica)？这里需要对可靠性、写入带宽和读取带宽进行权衡。例如，把所有复本都存储在一个节点损失的写入带宽最小(因为复制管线都是在同一节点上运行)，但这并不提供真实的冗余(如果节点发生故障，那么该块中的数据会丢失)。同时，同一机架上服务器间的读取带宽是很高的。另一个极端，把复本放在不同的数据中心可以最大限度地提高冗余，但带宽的损耗非常大。即使在同一数据中心(到目前为止，所有 Hadoop 集群均运行在同一数据中心内)，也有多种可能的数据布局策略。

Hadoop 的默认布局策略是在运行客户端的节点上放第 1 个复本 (如果客户端运行在集群之外，就随机选择一个节点，不过系统会避免挑选那些存储太满或太忙的节点)。第 2 个复本放在与第一个不同且随机另外选择的机架中节点上(离架)。第 3 个复本与第 2 个复本放在同一个机架上，且随机选择另一个节点。其他复本放在集群中随机选择的节点上，不过系统会尽量避免在同一个的机架上放太多复本。

一旦选定复本的放置位置，就根据网络拓扑创建一个管线。如果复本数为 3，则有图 3-5 所示的管线。

图 3-5. 一个典型的复本管线

总的来说，这一方法不仅提供很好的稳定性(数据块存储在两个机架中)并实现很好的负载均衡，包括写入带宽(写入操作只需要遍历一个交换机)、读取性能(可以从两个机架中选择读取)和集群中块的均匀分布(客户端只在本地机架上写入一个块)。

3.6.3 一致模型

文件系统的一致模型(coherency model)描述了文件读/写的数据可见性。HDFS 为性能牺牲了一些 POSIX 要求，因此一些操作与你期望的可能不同。

新建一个文件之后，它能在文件系统的命名空间中立即可见，如下所示：

```
Path p = new Path("p");
Fs.create(p);
assertThat(fs.exists(p),is(true));
```

但是，写入文件的内容并不保证能立即可见，即使数据流已经刷新并存储。所以文件长度显示为 0：

```
Path p = new Path("p");
OutputStream out = fs.create(p);
out.write("content".getBytes("UTF-8"));
out.flush();
assertThat(fs.getFileStatus(p).getLen(),is(0L));
```

当写入的数据超过一个块后，第一个数据块对新的 reader 就是可见的。之后的块

也不例外。总之，当前正在写入的块对其他 reader 不可见。

HDFS 提供了一种强行将所有缓存刷新到 datanode 中的手段，即对 FSDataOutputStream 调用 hflush()方法。当 hflush ()方法返回成功后，对所有新的 reader 而言，HDFS 能保证文件中到目前为止写入的数据均到达所有 datanode 的写入管道并且对所有新的 reader 均可见：

```
Path p = new Path("p");
FSDataOutputStream out = fs.create(p);
out.write("content".getBytes("UTF-8"));
out.hflush();
assertThat(fs.getFileStatus(p).getLen(), is(((long) "content".length())));
```

注意，hflush()不保证 datanode 已经将数据写到磁盘上，仅确保数据在 datanode 的内存中(因此，如果数据中心断电，数据会丢失)。为确保数据写入到磁盘上，可以用 hsync()替代[①]。

hsync()操作类似于 POSIX 中的 fsync()系统调用，该调用提交的是一个文件描述符的缓冲数据。例如，利用标准 Java API 数据写入本地文件，我们能够在刷新数据流且同步之后看到文件内容：

```
FileOutputStream out = new FileOutputStream(localFile);
out.write("content".getBytes("UTF-8"));
out.flush(); // flush to operating system
out.getFD().sync(); // sync to disk
assertThat(localFile.length(), is(((long) "content".length())));
```

在 HDFS 中关闭文件其实隐含了执行 hflush()方法：

```
Path p = new Path("p");
OutputStream out = fs.create(p);
out.write("content".getBytes("UTF-8"));
out.close();
assertThat(fs.getFileStatus(p).getLen(), is(((long) "content".length())));
```

对应用设计的重要性

这个一致模型和设计应用程序的具体方法息息相关。如果不调用 hflush()或 hsync()方法，就要准备好在客户端或系统发生故障时可能会丢失数据块。对很多应用来说，这是不可接受的，所以需要在适当的地方调用 hflush()方法，例如

① 在 Hadoop 1.x 中，hflush()被称为 sync()，hsync()不存在。

在写入一定的记录或字节之后。尽管 hflush()操作被设计成尽量减少 HDFS 负载，但它有许多额外的开销(hsync()的开销更大)，所以在数据鲁棒性和吞吐量之间就会有所取舍。怎样权衡与具体的应用相关，通过度量应用程序以不同频率调用 hflush()或 hsync()时呈现出的性能，最终选择一个合适的调用频率。

3.7 通过 distcp 并行复制

前面着重介绍单线程访问的 HDFS 访问模型。例如，通过指定文件通配符，可以对一组文件进行处理，但是为了提高性能，需要写一个程序来并行处理这些文件。Hadoop 自带一个有用程序 *distcp*，该程序可以并行从 Hadoop 文件系统中复制大量数据，也可以将大量数据复制到 Hadoop 中。

Distcp 的一种用法是替代 hadoop fs -cp。例如，我们可以将文件复制到另一个文件中：[①]

```
% hadoop distcp file1 file2
```

也可以复制目录：

```
% hadoop distcp dir1 dir2
```

如果 *dir2* 不存在，将新建 *dir2*，目录 *dir1* 的内容全部复制到 *dir2* 下。可以指定多个源路径，所有源路径下的内容都将被复制到目标路径下。

如果 *dir2* 已经存在，那么目录 *dir1* 将被复制到 *dir2* 下，形成目录结构 *dir2/dir1*。如果这不是你所需的，你可以补充使用-overwrite 选项，在保持同样的目录结构的同时强制覆盖原有文件。你也可以使用-update 选项，仅更新发生变化的文件。用一个示例可以更好解释这个过程。如果我们修改了 *dir1* 子树中一个文件，我们能够通过运行以下命令将修改同步到 *dir2* 中：

```
% hadoop distcp -update dir1 dir2
```

如果不确定 *discp* 操作的效果，最好先在一个小的测试目录树下试运行。

① 即使对于单个文件复制，由于 hadoop fs -cp 通过运行命令的客户端进行文件复制，因此更倾向于使用 *distcp* 变种复制大文件。

distcp 是作为一个 MapReduce 作业来实现的，该复制作业是通过集群中并行运行的 map 来完成。这里没有 reducer。每个文件通过一个 map 进行复制，并且 *distcp* 试图为每一个 map 分配大致相等的数据来执行，即把文件划分为大致相等的块。默认情况下，将近 20 个 map 被使用，但是可以通过为 *distcp* 指定-m 参数来修改 map 的数目。

关于 distcp 的一个常见使用实例是在两个 HDFS 集群间传送数据。例如，以下命令在第二个集群上为第一个集群/*foo* 目录创建了一个备份：

```
% hadoop distcp -update -delete -p hdfs://namenode1/foo hdfs://namenode2/foo
```

-delete 选项使得 *distcp* 可以删除目标路径中任意没在源路径中出现的文件或目录，-P 选项意味着文件状态属性如权限、块大小和复本数被保留。当你运行不带参数的 *distcp* 时，能够看到准确的用法。

如果两个集群运行的是 HDFS 的不兼容版本，你可以将 webhdfs 协议用于它们之间的 *distcp*：

```
% hadoop distcp webhdfs://namenode1:50070/foo webhdfs://namenode2:50070/foo
```

另一个变种是使用 HttpFs 代理作为 *distcp* 源或目标(又一次使用了 webhdfs 协议)，这样具有设置防火墙和控制带宽的优点，详情参见 3.4 节对 HTTP 的讨论。

保持 HDFS 集群的均衡

向 HDFS 复制数据时，考虑集群的均衡性是相当重要的。当文件块在集群中均匀分布时，HDFS 能达到最佳工作状态，因此你想确保 *distcp* 不会破坏这点。例如，如果将-m 选项指定为 1，即由一个 map 来执行复制作业，它的意思是不考虑速度变慢和未充分利用集群资源每个块的第一个复本将存储到运行 map 的节点上(直到磁盘被填满)。第二和第三个复本将分散在集群中，但这一个节点是不均衡的。将 map 的数量设定为多于集群中节点的数量，可以避免这个问题。鉴于此，最好首先使用默认的每个节点 20 个 map 来运行 *distcp* 命令。

然而，这也并不总能阻止集群的不均衡。也许想限制 map 的数量以便另外一些节点可以运行其他作业。若是这样，可以用均衡器(balancer)工具(参见 11.1.4 节对均衡器的讨论)，进而改善集群中块分布的均匀程度。

第 4 章

关于 YARN

Apache YARN(Yet Another Resource Negotiator 的缩写)是 Hadoop 的集群资源管理系统。YARN 被引入 Hadoop 2，最初是为了改善 MapReduce 的实现，但它具有足够的通用性，同样可以支持其他的分布式计算模式。

YARN 提供请求和使用集群资源的 API，但这些 API 很少直接用于用户代码。相反，用户代码中用的是分布式计算框架提供的更高层 API，这些 API 建立在 YARN 之上且向用户隐藏了资源管理细节。图 4-1 对此进行了描述,一些分布式计算框架(MapReduce，Spark 等等)作为 YARN 应用运行在集群计算层(YARN)和集群存储层(HDFS 和 HBase)上。

图 4-1. YARN 应用

还有一层应用是建立在图 4-1 所示的框架之上。如 Pig，Hive 和 Crunch 都是运行在 MapReduce，Spark 或 Tez(或三个都可)之上的处理框架，它们不和 YARN 直接打交道。

本章将介绍 YARN 的特性，为理解后续第Ⅳ部分关于 Hadoop 分布式处理框架的相关内容提供基础。

4.1 剖析 YARN 应用运行机制

YARN 通过两类长期运行的守护进程提供自己的核心服务：管理集群上资源使用的资源管理器(*resource manager*)、运行在集群中所有节点上且能够启动和监控容器(*container*)的节点管理器(*node manager*)。容器用于执行特定应用程序的进程，每个容器都有资源限制(内存、CPU 等)。一个容器可以是一个 Unix 进程，也可以是一个 Linux cgroup，取决于 YARN 的配置(详见 10.3.3 节)。图 4-2 描述了 YARN 是如何运行一个应用的。

图 4-2. YARN 应用的运行机制

为了在 YARN 上运行一个应用，首先，客户端联系资源管理器，要求它运行一个 *application master* 进程(图 4-2 中的步骤 1)。然后，资源管理器找到一个能够在容

器中启动 *application master* 的节点管理器(步骤 2a 和 2b)[①]。准确地说，application master 一旦运行起来后能做些什么依赖于应用本身。有可能是在所处的容器中简单地运行一个计算，并将结果返回给客户端；或是向资源管理器请求更多的容器(步骤 3)，以用于运行一个分布式计算(步骤 4a 和 4b)。后者是 MapReduce YARN 应用所做的事情，将在 7.1 节中详细介绍。

注意，从图 4-2 可看出，YARN 本身不会为应用的各部分(客户端、master 和进程)彼此间通信提供任何手段。大多数重要的 YARN 应用使用某种形式的远程通信机制(例如 Hadoop 的 RPC 层)来向客户端传递状态更新和返回结果，但是这些通信机制都是专属于各应用的。

4.1.1 资源请求

YARN 有一个灵活的资源请求模型。当请求多个容器时，可以指定每个容器需要的计算机资源数量(内存和 CPU)，还可以指定对容器的本地限制要求。

本地化对于确保分布式数据处理算法高效使用集群带宽[②]非常重要，因此，YARN 允许一个应用为所申请的容器指定本地限制。本地限制可用于申请位于指定节点或机架，或集群中任何位置(机架外)的容器。

有时本地限制无法被满足，这种情况下要么不分配资源，或者可选择放松限制。例如，一个节点由于已经运行了别的容器而无法再启动新的容器，这时如果有应用请求该节点，则 YARN 将尝试在同一机架中的其他节点上启动一个容器，如果还不行，则会尝试集群中的任何一个节点。

通常情况下，当启动一个容器用于处理 HDFS 数据块(为了在 MapReduce 中运行一个 map 任务)时，应用将会向这样的节点申请容器：存储该数据块三个复本的节点，或是存储这些复本的机架中的一个节点。如果都申请失败，则申请集群中的任意节点。

YARN 应用可以在运行中的任意时刻提出资源申请。例如，可以在最开始提出所有的请求，或者为了满足不断变化的应用需要，采取更为动态的方式在需要更多资源时提出请求。

① 客户端也有可能启动 application master,可能在集群外，或在与客户端相同的 JVM 中。这被称作“非托管的 application master”(*unmanaged application master*)。

② 关于该议题的详细讨论，可以参见 2.4 节和 3.6 节。

Spark 采用了上述第一种方式，在集群上启动固定数量的执行器(详见 19.6 节)。另一方面，MapReduce 则分两步走，在最开始时申请 map 任务容器， reduce 任务容器的启用则放在后期。同样，如果任何任务出现失败，将会另外申请容器以重新运行失败的任务。

4.1.2 应用生命期

YARN 应用的生命期差异性很大:有几秒的短期应用，也有连续运行几天甚至几个月的长期应用。与其关注应用运行多长时间，不如按照应用到用户运行的作业之间的映射关系对应用进行分类更有意义。最简单的模型是一个用户作业对应一个应用，这也是 MapReduce 采取的方式。

第二种模型是，作业的每个工作流或每个用户对话(可能并无关联性)对应一个应用。这种方法要比第一种情况效率更高，因为容器可以在作业之间重用，并且有可能缓存作业之间的中间数据。Spark 采取的是这种模型。

第三种模型是，多个用户共享一个长期运行的应用。这种应用通常是作为一种协调者的角色在运行。例如，Apache Slider(网址为 *http://slider.incubator.apache.org/*)有一个长期运行的 application master，主要用于启动集群上的其他应用。Impala(详见 17.4.3 节)也使用这种模型提供了一个代理应用，Impala 守护进程通过该代理请求集群资源。由于避免了启动新 application master 带来的开销，一个总是开启(always on) 的 application master 意味着用户将获得非常低延迟的查询响应。[①]

4.1.3 构建 YARN 应用

从无到有编写一个 YARN 应用是一件相当复杂的事，但在很多情况下不必这样。有很多现成的应用，在符合要求的情况下通常可以直接使用。例如，如果你有兴趣运行一个作业的有向无环图，那么 Spark 或 Tez 就很合适；如果对流处理有兴趣，Spark、Samza 或 Storm 能提供帮助。[②]

许多项目都简化了构建 YARN 应用的过程。先前提到的 Apache Slider，使得在 YARN 上运行现有的分布式应用成为可能。对于一个应用(如 HBase)来说，用户可

① 低延迟 application master 代码可以参见 Llama 项目，网址为 *http://cloudera. github.io/llama/*。
② 所有这些项目都是 Apache Software 基金会支持的。

以独立于其他用户在集群上运行自己的实例，则意味着不同的用户能够运行同一应用的不同版本。Slider 提供了控制手段，可以修改应用运行所在的节点的数量，也可以暂停和恢复应用的运行。

Apache Twill(*http://twill.incubator.apache.org/*)与 Slider 类似，但额外提供了一个简单的编程模型，用于开发 YARN 上的分布式应用。Twill 允许将集群进程定义为 `Java Runnable` 的扩展，然后在集群上的 YARN 容器中运行它们。Twill 同样为实时日志(来自于 runnable 类的日志事件以流的方式返回客户端)和命令消息提供支持。

当遇到以上方法都不适用的情况时，例如一个应用有复杂的调度需求，那么作为 YARN 项目自身一部分的 *distributed shell* 应用为如何写 YARN 应用做了一个示范。该应用演示了如何使用 YARN 客户端 API 来处理客户端或 application master 与 YARN 守护进程之间的通信。

4.2 YARN 与 MapReduce 1 相比

有时用“MapReduce 1”来指代 Hadoop 初始版本(版本 1 及更早期版本)中的 MapReduce 分布式执行框架，以区别于使用了 YARN(Hadoop 2 及以后的版本)的 MapReduce 2。

新旧版本 MapReduce API 之间的区别不同于 MapReduce 1 执行框架和 MapReduce 2 执行框架的区别。API 具有面向用户的、客户端的特性，并且决定着用户怎样写 MapReduce 程序(见附录 D)；而执行机制只是运行 MapReduce 程序的不同途径而已。新旧版本 API 在 MapReduce 1 和 2 运行的四种组合都是支持的。

MapReduce 1 中，有两类守护进程控制着作业执行过程：一个 *jobtracker* 及一个或多个 *tasktracker*。jobtracker 通过调度 tasktracker 上运行的任务来协调所有运行在系统上的作业。tasktracker 在运行任务的同时将运行进度报告发送给 jobtracker，jobtracker 由此记录每项作业任务的整体进度情况。如果其中一个任务失败，jobtracker 可以在另一个 tasktracker 节点上重新调度该任务。

MapReduce 1 中，jobtracker 同时负责作业调度(将任务与 tasktracker 匹配)和任务进度监控(跟踪任务、重启失败或迟缓的任务；记录任务流水，如维护计数器的计数)。相比之下，YARN 中，这些职责是由不同的实体担负的:它们分别是资源管理

器和 application master(每个 MapReduce 作业一个)。jobtracker 也负责存储已完成作业的作业历史，但是也可以运行一个作业历史服务器作为一个独立的守护进程来取代 jobtracker。在 YARN 中，与之等价的角色是时间轴服务器(timeline server)，它主要用于存储应用历史。[①]

YARN 中与 tasktracker 等价的角色是节点管理器。表 4-1 中对两者之间的映射关系进行了总结。

表 4-1. MapReduce 1 和 Yarn 在组成上的比较

MapReduce 1	YARN
Jobtracker	资源管理器、application master、时间轴服务器
Tasktracker	节点管理器
Slot	容器

YARN 的很多设计是为了解决 MapReduce 1 的局限性。使用 YARN 的好处包括以下几方面。

可扩展性(Scalability)

与 MapReduce 1 相比，YARN 可以在更大规模的集群上运行。当节点数达到 4000，任务数达到 40000[②]时，MapReduce 1 会遇到可扩展性瓶颈，瓶颈源自于 jobtracker 必须同时管理作业和任务这样一个事实。YARN 利用其资源管理器和 application master 分离的架构优点克服了这个局限性，可以扩展到面向将近 10000 个节点和 100000 个任务。

与 jobtracker 相比，一个应用的每个实例(这里指一个 MapReduce 作业)都对应一个专门的 application master，该管理进程(master)在应用的持续期间运行。这个模型实际上与初始的 Google MapReduce 论文中所描述的更接近。论文中描述了如何启动一个管理(master)进程以协调运行在一系列工作 (worker)上的 map 和 reduce 任务。

① Hadoop 2.5.1 中，YARN 时间轴服务器依旧不存储 MapReduce 作业历史，因此仍需要 MapReduce 作业历史服务器守护进程(详见 10.2 节)。

② 参见 Arun C. Murthy 于 2011 年 2 月 14 日发表的“下一代 Apache Hadoop MaReduce”，网址为 *http://bit_ly/next_gen_mapreduce*。

可用性(Availability)

当服务守护进程失败时，通过为另一个守护进程复制接管工作所需的状态以便其继续提供服务，从而可以获得高可用性(HA, High availability)。然而，jobtracker 内存中大量快速变化的复杂状态(例如，每个任务状态每几秒会更新一次)使得改进 jobtracker 服务获得高可用性非常困难。

由于 YARN 中 jobtracker 在资源管理器和 application master 之间进行了职责划分，高可用的服务随之成为一个分而治之问题：先为资源管理器提供高可用性，再为 YARN 应用(针对每个应用)提供高可用性。实际上，对于资源管理器和 application master，Hadoop 2 都支持 MapReduce 作业的高可用性。YARN 中的失败恢复将在 7.2 节中详细讨论。

利用率(Utilization)

MapReduce 1 中，每个 tasktracker 都配置有若干固定长度的 slot，这些 slot 是静态分配的，在配置的时候就被划分为 map slot 和 reduce slot。一个 map slot 仅能用于运行一个 map 任务，同样，一个 reduce slot 仅能用于运行一个 reduce 任务。

YARN 中，一个节点管理器管理一个资源池，而不是指定的固定数目的 slot。YARN 上运行的 MapReduce 不会出现由于集群中仅有 map slot 可用导致 reduce 任务必须等待的情况，而 MapReduce 1 则会有这样的问题。如果能够获得运行任务的资源，那么应用就会正常进行。

更进一步，YARN 中的资源是精细化管理的，这样一个应用能够按需请求资源，而不是请求一个不可分割的、对于特定的任务而言可能会太大(浪费资源)或太小(可能会导致失败)的 slot。

多租户(Multitenancy)

在某种程度上，可以说 YARN 的最大优点在于向 MapReduce 以外的其他类型的分布式应用开放了 Hadoop。MapReduce 仅仅是许多 YARN 应用中的一个。

用户甚至可以在同一个 YARN 集群上运行不同版本的 MapReduce，这使得升级 MapReduce 的过程更好管理。(注意，MapReduce 的一些部件，例如作业历史服务器和 shuffle 句柄，和 YARN 自身一样，仍然需要在集群范围内升级。)

由于 Hadoop 2 应用广泛且是最新的稳定版本，本书余下的章节中，除非特意声

明，MapReduce 一般指 MapReduce 2。第 7 章详细探讨了 YARN 中 MapReduce 的工作机制。

4.3 YARN 中的调度

理想情况下，YARN 应用发出的资源请求应该立刻给予满足。然而现实中资源是有限的，在一个繁忙的集群上，一个应用经常需要等待才能得到所需的资源。YARN 调度器的工作就是根据既定策略为应用分配资源。调度通常是一个难题，并且没有一个所谓“最好”的策略，这也是为什么 YARN 提供了多种调度器和可配置策略供我们选择的原因。接下来我们将探讨这个问题。

4.3.1 调度选项

YARN 中有三种调度器可用：FIFO 调度器(FIFO Scheduler)，容量调度器(Capacity Scheduler)和公平调度器(Fair Scheduler)。FIFO 调度器将应用放置在一个队列中，然后按照提交的顺序(先进先出)运行应用。首先为队列中第一个应用的请求分配资源，第一个应用的请求被满足后再依次为队列中下一个应用服务。

FIFO 调度器的优点是，简单易懂，不需要任何配置，但是不适合共享集群。大的应用会占用集群中的所有资源，所以每个应用必须等待直到轮到自己运行。在一个共享集群中，更适合使用容量调度器或公平调度器。这两种调度器都允许长时间运行的作业能及时完成，同时也允许正在进行较小临时查询的用户能够在合理时间内得到返回结果。

图 4-3 描述了调度器之间的差异性，由图中可以看出，当使用 FIFO 调度器(i)时，小作业一直被阻塞，直至大作业完成。

使用容量调度器时(图 4-3 中的 ii)，一个独立的专门队列保证小作业一提交就可以启动，由于队列容量是为那个队列中的作业所保留的，因此这种策略是以整个集群的利用率为代价的。这意味着与使用 FIFO 调度器相比，大作业执行的时间要长。

图 4-3. 用 FIFO 调度器(i)、容量调度器(ii)和公平调度器(iii)运行大小作业时集群的利用率

使用公平调度器时(图 4-3 中的 iii)，不需要预留一定量的资源，因为调度器会在所有运行的作业之间动态平衡资源。第一个(大)作业启动时，它也是唯一运行的作业，因而获得集群中所有的资源。当第二个(小)作业启动时，它被分配到集群的一半资源，这样每个作业都能公平共享资源。

注意，从第二个作业的启动到获得公平共享资源之间会有时间滞后，因为它必须等待第一个作业使用的容器用完并释放出资源。当小作业结束且不再申请资源后，大作业将回去再次使用全部的集群资源。最终的效果是：既得到了较高的集群利用率，又能保证小作业能及时完成。

图 4-3 对比了三种调度器的基本操作。在接下来的两个小节中，我们将讨论容量调度器和公平调度器的高级配置选项。

4.3.2 容量调度器配置

容量调度器允许多个组织共享一个 Hadoop 集群，每个组织可以分配到全部集群资源的一部分。每个组织被配置一个专门的队列，每个队列被配置为可以使用一定的集群资源。队列可以进一步按层次划分，这样每个组织内的不同用户能够共享该组织队列所分配的资源。在一个队列内，使用 FIFO 调度策略对应用进行调度。

正如图 4-3 所示，单个作业使用的资源不会超过其队列容量。然而，如果队列中有多个作业，并且队列资源不够用了呢？这时如果仍有可用的空闲资源，那么容量调度器可能会将空余的资源分配给队列中的作业，哪怕这会超出队列容量。[①]这称为“弹性队列”(*queue elasticity*)。

正常操作时，容量调度器不会通过强行中止来抢占容器(container)[②]。因此，如果一个队列一开始资源够用，然后随着需求增长，资源开始不够用时，那么这个队列就只能等着其他队列释放容器资源。缓解这种情况的方法是，为队列设置一个最大容量限制，这样这个队列就不会过多侵占其他队列的容量了。当然，这样做是以牺牲队列弹性为代价的，因此需要在不断尝试和失败中找到一个合理的折中。

① 如果属性 `yarn.scheduler.capacity.<queue-path>.user-limit-factor` 设置为大于 1(默认值)，那么一个作业可以使用超过其队列容量的资源。

② 然而，容量调度器可以执行“工作保留”式的抢占，此时资源管理器会要求应用返回容器以平衡容量资源。

假设一个队列的层次结构如下：

```
root
├─prod
└─dev
    ├─ eng
    └─ science
```

范例 4-1 是一个基于上述队列层次的容量调度器配置文件，文件名为 *capacity-scheduler.xml*。在 root 队列下定义两个队列：prod 和 dev，分别占 40%和 60%的容量。需要注意的是，对特定队列进行配置时，是通过以下形式的配置属性 yarn.scheduler.capacity.<*queue-path*>.<*sub-property*>进行设置的，其中，<*queue-path*>表示队列的层次路径(用圆点隔开)，例如 root.prod。

范例 4-1. 容量调度器的基本配置文件

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>yarn.scheduler.capacity.root.queues</name>
    <value>prod,dev</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.queues</name>
    <value>eng,science</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.prod.capacity</name>
    <value>40</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.capacity</name>
    <value>60</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.maximum-capacity</name>
    <value>75</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.eng.capacity</name>
    <value>50</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.science.capacity</name>
    <value>50</value>
  </property>
</configuration>
```

可以看到，dev 队列进一步被划分成 eng 和 science 两个容量相等的队列。由于 dev 队列的最大容量被设置为 75%，因此即使 prod 队列空闲，dev 队列也不会占

用全部集群资源。换而言之，prod 队列能即刻使用的可用资源比例总是能达到25%。由于没有对其他队列设置最大容量限制，eng 或 science 中的作业可能会占用 dev 队列的所有容量(将近 75%的集群资源)，而 prod 队列实际则可能会占用全部集群资源。

除了可以配置队列层次和容量，还有些设置是用来控制单个用户或应用能被分配到的最大资源数量、同时运行的应用数量及队列的 ACL 认证等。关于容量调度器配置的更多内容，请参见 *http://bit_ly/capacity_schedular*。

队列放置

将应用放置在哪个队列中，取决于应用本身。例如，在 MapReduce 中，可以通过设置属性 mapreduce.job.queuename 来指定要用的队列。如果队列不存在，则在提交时会发送错误。如果不指定队列，那么应用将被放在一个名为“default”的默认队列中。

对于容量调度器，队列名应该是队列层次名的最后一部分，完整的队列层次名是不会被识别的。例如，对于上述配置范例，prod 和 eng 是合法的队列名，但 root.dev.eng 和 dev.eng 作为队列名是无效的。

4.3.3 公平调度器配置

公平调度器旨在为所有运行的应用公平分配资源。图 4-3 展示了同一个队列中的应用是如何实现资源公平共享的。然而公平共享实际也可以在多个队列间工作，后续会对此进行分析。

术语 *queue* 和 *pool* 在公平调度器的上下文中会交替使用。

接下来解释资源是如何在队列之间公平共享的。想象两个用户 *A* 和 *B*，分别拥有自己的队列(参见图 4-4)。*A* 启动一个作业，在 *B* 没有需求时 *A* 会分配到全部可用资源；当 *A* 的作业仍在运行时 *B* 启动一个作业，一段时间后，按照我们先前看到的方式，每个作业都用到了一半的集群资源。这时，如果 *B* 启动第二个作业且其他作业仍在运行，那么第二个作业将和 *B* 的其他作业(这里是第一个)共享资源，因此 *B* 的每个作业将占用四分之一的集群资源，而 *A* 仍继续占用一半的集群资源。最终的结果就是资源在用户之间实现了公平共享。

图 4-4. 用户队列间的公平共享

1. 启用公平调度器

公平调度器的使用由属性 yarn.resourcemanager.scheduler.class 的设置所决定。默认是使用容量调度器(尽管在一些 Hadoop 分布式项目，如 CDH 中是默认使用公平调度器)，如果要使用公平调度器，需要将 *yarn-site.xml* 文件中的 yarn.resourcemanager.scheduler.class 设置为公平调度器的完全限定名：org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.FairScheduler。

2. 队列配置

通过一个名为 *fair-scheduler.xml* 的分配文件对公平调度器进行配置，该文件位于类路径下。(可以通过设置属性 `yarn.scheduler.fair.allocation.file` 来修改文件名)。当没有该分配文件时，公平调度器的工作策略同先前所描述的一样：每个应用放置在一个以用户名命名的队列中，队列是在用户提交第一个应用时动态创建的。

通过分配文件可以为每个队列进行配置。这样可以对容量调度器支持的层次队列进行配置。例如，可以像为容量调度器所做的那样，使用范例 4-2 所示的分配文件定义 `prod` 和 `dev`。

范例 4-2. 公平调度器的分配文件

```
<?xml version="1.0"?>
<allocations>
```

```
<defaultQueueSchedulingPolicy>fair</defaultQueueSchedulingPolicy>

<queue name="prod">
  <weight>40</weight>
  <schedulingPolicy>fifo</schedulingPolicy>
</queue>

<queue name="dev">
  <weight>60</weight>
  <queue name="eng" />
  <queue name="science" />
</queue>

  <queuePlacementPolicy>
    <rule name="specified" create="false" />
    <rule name="primaryGroup" create="false" />
    <rule name="default" queue="dev.eng" />
  </queuePlacementPolicy>
</allocations>
```

队列的层次使用嵌套 queue 元素来定义。所有的队列都是 root 队列的孩子，即使实际上并没有嵌套进 root queue 元素里。这里把 dev 队列又划分成 eng 和 science 两个队列。

队列中有权重元素，用于公平共享计算。在这个例子中，当集群资源按照 40:60 的比例分配给 prod 和 dev 时，集群分配被认为是公平的。eng 和 science 队列没有指定权重，因此它们会被平均分配。权重并不是百分百，例子中是为了简单起见使用了相加之和为 100 的两个数。也可以为 prod 和 dev 队列分别指定 2 和 3 的权重，在效果上是一样的。

当设置权重时，记住要考虑默认队列和动态创建的队列(例如以用户名命名的队列)。虽然没有在分配文件中为它们指定权重，但它们仍有值为 1 的权重。

每个队列可以有不同的调度策略。队列的默认调度策略可以通过顶层元素 defaultQueueSchedulingPolicy 进行设置，如果省略，默认使用公平调度。尽管名称是“公平”，公平调度器也支持队列级别的 FIFO(fifo)策略，以及 Dominant Resource Fairness(drf)策略，本章稍后会对此策略进行解释。

队列的调度策略可以被该队列的 schedulingPolicy 元素指定的策略覆盖。在上述例子中，由于我们希望每个生产性作业能够顺序运行且在最短可能的时间内结束，所以 prod 队列使用了 FIFO 调度策略。值得注意的是，在 prod 和 dev 队列之间、eng 和 science 队列之间及内部划分资源仍然使用了公平调度。

尽管上述的分配文件中没有展示，每个队列仍可配置最大和最小资源数量，及最大可运行的应用的数量(更多细节可以访问 *http://bit_ly/fair_schedular*)。最小资源数量不是一个硬性的限制，但是调度器常用它对资源分配进行优先排序。如果两个队列的资源都低于它们的公平共享额度，那么远低于最小资源数量的那个队列优先被分配资源。最小资源数量也会用于接下来将介绍的抢占行为。

3. 队列放置

公平调度器使用一个基于规则的系统来确定应用应该放到哪个队列。在范例 4-2 中，queuePlacementPolicy 元素包含了一个规则列表，每条规则会被依次尝试直到匹配成功。第一条规则，specified，表示把应用放进所指明的队列中，如果没有指明，或如果所指明的队列不存在，则规则不匹配，继续尝试下一条规则。primaryGroup 规则会试着把应用放在以用户的主 Unix 组名命名的队列中，如果没有这样的队列，则继续尝试下一条规则而不是创建队列。Default 规则是一条兜底规则，当前述规则都不匹配时，将启用该条规则，把应用放进 dev.eng 队列中。

当然，可以完全省略 queuePlacementPolicy 元素，此时队列放置默认遵从如下规则：

```
<queuePlacementPolicy>
  <rule name="specified" />
  <rule name="user" />
</queuePlacementPolicy>
```

换而言之，除非明确定义队列，否则必要时会以用户名为队列名创建队列。

另一个简单的队列放置策略是，将所有的应用放进同一个队列(default)中。这样可以在应用之间公平共享资源，而不是在用户之间共享。策略定义等价于以下规则：

```
<queuePlacementPolicy>
  <rule name="default" />
</queuePlacementPolicy>
```

不使用分配文件也可以设置以上策略，通过将属性 yarn.scheduler.fair.user-as-default-queue 设为 false，应用就会被放入 default 队列，而不是各个用户的队列。另外，将属性 yarn.scheduler.fair.allow-undeclared-pools 设置为 false，用户便不能随意创建队列了。

4. 抢占

在一个繁忙的集群中，当作业被提交给一个空队列时，作业不会立刻启动，直到集群上已经运行的作业释放了资源。为了使作业从提交到执行所需的时间可预测，公平调度器支持“抢占”(*preemption*)功能。

所谓抢占，就是允许调度器终止那些占用资源超过了其公平共享份额的队列的容器，这些容器资源释放后可以分配给资源数量低于应得份额的队列。注意，抢占会降低整个集群的效率，因为被终止的 containers 需要重新执行。

通过将 `yarn.scheduler.fair.preemption` 设置为 `true`，可以全面启用抢占功能。有两个相关的抢占超时设置：一个用于最小共享(*minimum share preemption timeout*)，另一个用于公平共享(*fair share preemption timeout*)，两者设定时间均为秒级。默认情况下，两个超时参数均不设置。所以为了允许抢占容器，需要至少设置其中一个超时参数。

如果队列在 *minimum share preemption timeout* 指定的时间内未获得被承诺的最小共享资源，调度器就会抢占其他容器。可以通过分配文件中的顶层元素 `defaultMinSharePreemptionTimeout` 为所有队列设置默认的超时时间，还可以通过设置每个队列的 `minSharePreemptionTimeout` 元素来为单个队列指定超时时间。

类似，如果队列在 *fair share preemption timeout* 指定的时间内获得的资源仍然低于其公平共享份额的一半，那么调度器就会抢占其他容器。可以通过分配文件中的顶层元素 `defaultFairSharePreemptionTimeout` 为所有队列设置默认的超时时间，还可以通过设置每个队列的 `fairSharePreemptionTimeout` 元素来为单个队列指定超时时间。通过设置 `defaultFairSharePreemptionThreshold` 和 `fairSharePreemptionThreshold` (针对每个队列)可以修改超时阈值，默认值是 0.5。

4.3.5 延迟调度

所有的 YARN 调度器都试图以本地请求为重。在一个繁忙的集群上，如果一个应用请求某个节点，那么极有可能此时有其他容器正在该节点上运行。显而易见的处理是，立刻放宽本地性需求，在同一机架中分配一个容器。然而，通过实践发现，此时如果等待一小段时间(不超过几秒)，能够戏剧性的增加在所请求的节点上

分配到一个容器的机会，从而可以提高集群的效率。这个特性称之为延迟调度(*delay scheduling*)。容量调度器和公平调度器都支持延迟调度。

YARN 中的每个节点管理器周期性的(默认每秒一次)向资源管理器发送心跳请求。心跳中携带了节点管理器中正运行的容器、新容器可用的资源等信息，这样对于一个计划运行一个容器的应用而言，每个心跳就是一个潜在的调度机会(*scheduling opportunity*)。

当使用延迟调度时，调度器不会简单的使用它收到的第一个调度机会，而是等待设定的最大数目的调度机会发生，然后才放松本地性限制并接收下一个调度机会。

对于容量调度器，可以通过设置 yarn.scheduler.capacity.node-locality-delay 来配置延迟调度。设置为正整数，表示调度器在放松节点限制、改为匹配同一机架上的其他节点前，准备错过的调度机会的数量。

公平调度器也使用调度机会的数量来决定延迟时间，尽管是使用集群规模的比例来表示这个值。例如将 `yarn.scheduler.fair.locality. threshold.node` 设置为 0.5，表示调度器在接受同一机架中的其他节点之间，将一直等待直到集群中的一半节点都已经给过调度机会。还有个相关的属性 `yarn.scheduler.fair.locality.threshold.rack`，表示在接受另一个机架替代所申请的机架之前需要等待的时长阈值。

4.3.5 主导资源公平性

对于单一类型资源，如内存的调度，容量或公平性的概念很容易确定。例如两个用户正在运行应用，可以通过度量每个应用使用的内存来比较两个应用。然而，当有多种资源类型需要调度时，事情就会变得复杂。例如，如果一个用户的应用对 CPU 的需求量很大，但对内存的需求量很少；而另一个用户需要很少的 CPU，但对内存需求量很大，那么如何比较这两个应用呢?

YARN 中调度器解决这个问题的思路是，观察每个用户的主导资源，并将其作为对集群资源使用的一个度量。这个方法称为“主导资源公平性”(Dominant Resource Fairness,DRF)[①]。这个思想用一个简单的例子就可以很好的给予解释。

① 对 DRF 的详细介绍，可以参见 Ghodsi 在 2011 年发表的论文，标题为“Dominant Resource Fairness: Fair Allocation of Multiple Resource Types”，网址为 *http://bit_ly/fair_allocation*。

想象一个总共有 100 个 CPU 和 10TB 的集群 。应用 A 请求的每份容器资源为 2 个 CPU 和 300GB 内存，应用 B 请求的每份容器资源为 6 个 CPU 和 100GB 内存。A 请求的资源在集群资源中占比分别为 2%和 3%，由于内存占比(3%)大于 CPU 占比(2%)，所以内存是 A 的主导资源。B 请求的资源在集群资源中占比分别为 6%和 1%，所以 CPU 是 B 的主导资源。由于 B 申请的资源是 A 的两倍(6% vs3%)，所以在公平调度下，B 将分到一半的容器数。

默认情况下不用 DRF，因此在资源计算期间，只需要考虑内存，不必考虑 CPU。对容量调度器进行配置后，可以使用 DRF，将 *capacity-scheduler.xml* 文件中的 `org.apache.hadoop.yarn.util.resource.DominantResourceCalculator` 设为 `yarn.scheduler.capacity.resource-calculator` 即可。

公平调度器若要使用 DRF，通过将分配文件中的顶层元素 `defaultQueueSchedulingPolicy` 设为 `drf` 即可。

4.4 延伸阅读

本章简要介绍了 YARN。如果希望了解更多细节，请参见 Arun C. Murthy 等人所著的 *Apache Hadoop YARN* 一书，网址为 *http://yarn-book.com/*。

第 5 章

Hadoop 的 I/O 操作

Hadoop 自带一套原子操作用于数据 I/O 操作。其中有一些技术比 Hadoop 本身更常用，如数据完整性保持和压缩，但在处理多达好几个 TB 的数据集时，特别值得关注。其他一些则是 Hadoop 工具或 API，它们所形成的构建模块可用于开发分布式系统，比如序列化框架和在盘(on-disk)数据结构。

5.1 数据完整性

Hadoop 用户肯定都希望系统在存储和处理数据时不会丢失或损坏任何数据。尽管磁盘或网络上的每个 I/O 操作不太可能将错误引入自己正在读/写的数据中，但是如果系统中需要处理的数据量大到 Hadoop 的处理极限时，数据被损坏的概率还是很高的。

检测数据是否损坏的常见措施是，在数据第一次引入系统时计算校验和(checksum)并在数据通过一个不可靠的通道进行传输时再次计算校验和，这样就能发现数据是否损坏。如果计算所得的新校验和与原来的校验和不匹配，我们就认为数据已损坏。但该技术并不能修复数据——它只能检测出数据错误。(这正是不使用低端硬件的原因。具体说来，一定要使用 ECC 内存。)注意，校验和也是可能损坏的，不只是数据，但由于校验和比数据小得多，所以损坏的可能性非常小。

常用的错误检测码是 CRC-32(32 位循环冗余校验)，任何大小的数据输入均计算得到一个 32 位的整数校验和。Hadoop ChecksumFileSystem 使用 CRC-32 计算校验和，HDFS 用于校验和计算的则是一个更有效的变体 CRC-32C。

5.1.1 HDFS 的数据完整性

HDFS 会对写入的所有数据计算校验和，并在读取数据时验证校验和。它针对每个由 `dfs.bytes-per-checksum` 指定字节的数据计算校验和。默认情况下为 512 个字节，由于 CRC-32 校验和是 4 个字节，所以存储校验和的额外开销低于 1%。

datanode 负责在收到数据后存储该数据及其校验和之前对数据进行验证。它在收到客户端的数据或复制其他 datanode 的数据时执行这个操作。正在写数据的客户端将数据及其校验和发送到由一系列 datanode 组成的管线(详见第 3 章)，管线中最后一个 datanode 负责验证校验和。如果 datanode 检测到错误，客户端便会收到一个 `IOException` 异常的一个子类，对于该异常应以应用程序特定的方式来处理，比如重试这个操作。

客户端从 datanode 读取数据时，也会验证校验和，将它们与 datanode 中存储的校验和进行比较。每个 datanode 均持久保存有一个用于验证的校验和日志(persistent log of checksum verification)，所以它知道每个数据块的最后一次验证时间。客户端成功验证一个数据块后，会告诉这个 datanode，datanode 由此更新日志。保存这些统计信息对于检测损坏的磁盘很有价值。

不只是客户端在读取数据块时会验证校验和，每个 datanode 也会在一个后台线程中运行一个 `DataBlockScanner`，从而定期验证存储在这个 datanode 上的所有数据块。该项措施是解决物理存储媒体上位损坏的有力措施。11.1.4 节将详细描述如何访问扫描报告。

由于 HDFS 存储着每个数据块的复本(replica)，因此它可以通过数据复本来修复损坏的数据块，进而得到一个新的、完好无损的复本。基本思路是，客户端在读取数据块时，如果检测到错误，首先向 namenode 报告已损坏的数据块及其正在尝试读操作的这个 datanode，再抛出 `ChecksumException` 异常。namenode 将这个数据块复本标记为已损坏，这样它不再将客户端处理请求直接发送到这个节点，或尝试将这个复本复制到另一个 datanode。之后，它安排这个数据块的一个复本复制到另一个 datanode，如此一来，数据块的复本因子(replication factor)又回到期望水平。此后，已损坏的数据块复本便被删除。

在使用 `open()`方法读取文件之前，将 `false` 值传递给 `FileSystem` 对象的 `setVerifyChecksum()`方法，即可以禁用校验和验证。如果在命令解释器中使用带`-get` 选项的`-ignoreCrc` 命令或者使用等价的`-copyToLocal` 命令，也可以达到相同的效果。如果有一个已损坏的文件需要检查并决定如何处理，这个特性是

非常有用的。例如，也许你希望在删除该文件之前尝试看看是否能够恢复部分数据。

可以用 hadoop 的命令 `fs -checksum` 来检查一个文件的校验和。这可用于在 HDFS 中检查两个文件是否具有相同内容，*distcp* 命令也具有类似的功能。详情可以参见 3.7 节。

5.1.2 LocalFileSystem

Hadoop 的 `LocalFileSystem` 执行客户端的校验和验证。这意味着在你写入一个名为 *filename* 的文件时，文件系统客户端会明确在包含每个文件块校验和的同一个目录内新建一个*.filename.crc* 隐藏文件。文件块的大小由属性 `file.bytes-per-checksum` 控制，默认为 512 个字节。文件块的大小作为元数据存储在*.crc* 文件中，所以即使文件块大小的设置已经发生变化，仍然可以正确读回文件。在读取文件时需要验证校验和，并且如果检测到错误，`LocalFileSystem` 还会抛出一个 `ChecksumException` 异常。

校验和的计算代价是相当低的(在 Java 中，它们是用本地代码实现的)，一般只是增加少许额外的读/写文件时间。对大多数应用来说，付出这样的额外开销以保证数据完整性是可以接受的。此外，我们也可以禁用校验和计算，特别是在底层文件系统本身就支持校验和的时候。在这种情况下，使用 `RawLocalFileSystem` 替代 `LocalFileSystem`。要想在一个应用中实现全局校验和验证，需要将 `fs.file.impl` 属性设置为 `org.apache.hadoop. fs.RawLocalFileSystem` 进而实现对文件 URI 的重新映射。还有一个可选方案可以直接新建一个 `RawLocalFileSystem` 实例。如果想针对一些读操作禁用校验和，这个方案非常有用。示例如下：

```
Configuration conf = ...
FileSystem fs = new RawLocalFileSystem();
fs.initialize(null, conf);
```

5.1.3 ChecksumFileSystem

`LocalFileSystem` 通过 `ChecksumFileSystem` 来完成自己的任务，有了这个类，向其他文件系统(无校验和系统)加入校验和就非常简单，因为 `ChecksumFileSystem` 类继承自 `FileSystem` 类。一般用法如下：

```
FileSystem rawFs = ...
```

```
FileSystem checksummedFs = new ChecksumFileSystem(rawFs);
```

底层文件系统称为“源”(raw)文件系统，可以使用 ChecksumFileSystem 实例的 getRawFileSystem()方法获取它。ChecksumFileSystem 类还有其他一些与校验和有关的有用方法，比如 getChecksumFile()可以获得任意一个文件的校验和文件路径。请参考文档了解其他方法。

如果 ChecksumFileSystem 类在读取文件时检测到错误，会调用自己的 reportChecksumFailure()方法。默认实现为空方法，但 LocalFileSystem 类会将这个出错的文件及其校验和移到同一存储设备上一个名为 *bad_files* 的边际文件夹(side directory)中。管理员应该定期检查这些坏文件并采取相应的行动。

5.2 压缩

文件压缩有两大好处：减少存储文件所需要的磁盘空间，并加速数据在网络和磁盘上的传输。这两大好处在处理大量数据时相当重要，所以我们值得仔细考虑在 Hadoop 中文件压缩的用法。

有很多种不同的压缩格式、工具和算法，它们各有千秋。表 5-1 列出了与 Hadoop 结合使用的常见压缩方法。

表 5-1. 压缩格式总结

压缩格式	工具	算法	文件扩展名	是否可切分
DEFLATE[①]	无	DEFLATE	.deflate	否
gzip	gzip	DEFLATE	.gz	否
bzip2	bzip2	bzip2	.bz2	是
LZO	lzop	LZO	.lzo	否[②]
LZ4	无	LZ4	.lz4	否
Snappy	无	Snappy	.snappy	否

① DEFLATE 是一个标准压缩算法，该算法的标准实现是 zlib。没有可用于生成 DEFLATE 文件的常用命令行工具，因为通常都用 gzip 格式。注意，gzip 文件格式只是在 DEFLATE 格式上增加了文件头和一个文件尾。*.deflate* 文件扩展名是 Hadoop 约定的。

② 但是如果 LZO 文件已经在预处理过程中被索引了，那么 LZO 文件是可切分的。详情参见 5.2.2 节。

所有压缩算法都需要权衡空间/时间：压缩和解压缩速度更快，其代价通常是只能节省少量的空间。表 5-1 列出的所有压缩工具都提供 9 个不同的选项来控制压缩时必须考虑的权衡：选项-1 为优化压缩速度，-9 为优化压缩空间。例如，下述命令通过最快的压缩方法创建一个名为*file.gz* 的压缩文件：

```
%gzip -1 file
```

不同压缩工具有不同的压缩特性。gzip 是一个通用的压缩工具，在空间/时间性能的权衡中，居于其他两个压缩方法之间。bzip2 的压缩能力强于 gzip，但压缩速度更慢一点。尽管 bzip2 的解压速度比压缩速度快，但仍比其他压缩格式要慢一些。另一方面，LZO、LZ4 和 Snappy 均优化压缩速度，其速度比 gzip 快一个数量级，但压缩效率稍逊一筹。Snappy 和 LZ4 的解压缩速度比 LZO 高出很多。[①]

表 5-1 中的“是否可切分”列表示对应的压缩算法是否支持切分(splitable)，也就是说，是否可以搜索数据流的任意位置并进一步往下读取数据。可切分压缩格式尤其适合 MapReduce，更多讨论，可以参见 5.2.2 节。

5.2.1 codec

Codec 是压缩-解压缩算法的一种实现。在 Hadoop 中，一个对 `CompressionCodec` 接口的实现代表一个 codec。例如，**`GzipCodec`** 包装了 gzip 的压缩和解压缩算法。表 5-2 列举了 Hadoop 实现的 codec。

表 5-2. Hadoop 的压缩 codec

压缩格式	HadoopCompressionCodec
DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec
bzip2	org.apache.hadoop.io.compress.BZip2Codec
LZO	com.hadoop.compression.lzo.LzopCodec
LZ4	org.apache.hadoop.io.compress.Lz4Codec
Snappy	org.apache.hadoop.io.compress.SnappyCodec

LZO 代码库拥有 GPL 许可，因而可能没有包含在 Apache 的发行版本中，因此，Hadoop 的 codec 需要单独从 Google(*http://code.google.com/p/hadoop-gpl-compression*)

① 对于综合性压缩测试集，可以参考 *jvm-compressor-benchmark*，这里针对 JVM 兼容类库(包括一些原始类库)提供了相当不错的介绍。

或 GitHub(*http://github.com/kevinweil/hadoop-lzo*)下载，该代码库包含有修正的软件错误及其他一些工具。LzopCodec 与 *lzop* 工具兼容，LzopCodec 基本上是 LZO 格式的但包含额外的文件头，因此这通常就是你想要的。也有针对纯 LZO 格式的 LzoCodec，并使用*.lzo_deflate* 作为文件扩展名(类似于 DEFLATE，是 gzip 格式但不包含文件头)。

1. 通过 CompressionCodec 对数据流进行压缩和解压缩

CompressionCodec 包含两个函数，可以轻松用于压缩和解压缩数据。如果要对写入输出数据流的数据进行压缩，可用 createOutputStream (OutputStream out)方法在底层的数据流中对需要以压缩格式写入在此之前尚未压缩的数据新建一个 CompressionOutputStream 对象。相反，对输入数据流中读取的数据进行解压缩的时候，则调用 createInputStream(InputStream in)获取 CompressionInputStream，可以通过该方法从底层数据流读取解压缩后的数据。

CompressionOutputStream 和 CompressionInputStream，类似于 java.util.zip.DeflaterOutputStream 和 java.util.zip.DeflaterInputStream，只不过前两者能够重置其底层的压缩或解压缩方法，对于某些将部分数据流(section of data stream)压缩为单独数据块(block)的应用，例如 SequenceFile(详情参见 5.4.1 节对 SequenceFile 类的讨论)，这个能力是非常重要的。

范例 5-1 显示了如何用 API 来压缩从标准输入中读取的数据并将其写到标准输出。

范例 5-1. 该程序压缩从标准输入读取的数据，然后将其写到标准输出

```
public class StreamCompressor {

  public static void main(String[] args) throws Exception {
    String codecClassname = args[0];
    Class<?> codecClass = Class.forName(codecClassname);
     Configuration conf = new Configuration();
    CompressionCodec codec = (CompressionCodec)
      ReflectionUtils.newInstance(codecClass, conf);

 CompressionOutputStream out = codec.createOutputStream(System.out);
    IOUtils.copyBytes(System.in, out, 4096, false);
    out.finish();
  }
}
```

该应用希望将符合 CompressionCodec 实现的完全合格名称作为第一个命令行参

数。我们使用 ReflectionUtils 新建一个 codec 实例，并由此获得在 System.out 上支持压缩的一个包裹方法。然后，对 IOUtils 对象调用 copyBytes()方法将输入数据复制到输出，（输出由 CompressionOutputStream 对象压缩）。最后，我们对 CompressionOutputStream 对象调用 finish()方法，要求压缩方法完成到压缩数据流的写操作，但不关闭这个数据流。我们可以用下面这行命令做一个测试，通过 GzipCodec 的 StreamCompressor 对象对字符串“Text”进行压缩，然后使用 *gunzip* 从标准输入中对它进行读取并解压缩操作：

```
% echo "Text" | hadoop StreamCompressor org.apache.hadoop.io. compress.GzipCodec \
  | gunzip
Text
```

2. 通过 CompressionCodecFactory 推断 CompressionCodec

在读取一个压缩文件时，通常可以通过文件扩展名推断需要使用哪个 codec。如果文件以.*gz* 结尾，则可以用 GzipCodec 来读取，如此等等。前面的表 5-1 为每一种压缩格式列举了文件扩展名。

通过使用其 getCodec()方法，CompressionCodecFactory 提供了一种可以将文件扩展名映射到一个 CompressionCodec 的方法，该方法取文件的 Path 对象作为参数。范例 5-2 所示的应用便使用这个特性来对文件进行解压缩。

范例 5-2. 该应用根据文件扩展名选取 codec 解压缩文件

```
public class FileDecompressor {

  public static void main(String[] args) throws Exception {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);

    Path inputPath = new Path(uri);
    CompressionCodecFactory factory = new CompressionCodecFactory(conf);
    CompressionCodec codec = factory.getCodec(inputPath);
    if (codec == null) {
      System.err.println("No codec found for " + uri);
      System.exit(1);
    }

    String outputUri =
      CompressionCodecFactory.removeSuffix(uri, codec.getDefaultExtension());

    InputStream in = null;
    OutputStream out = null;
    try {
      in = codec.createInputStream(fs.open(inputPath));
      out = fs.create(new Path(outputUri));
      IOUtils.copyBytes(in, out, conf);
    } finally {
```

```
      IOUtils.closeStream(in);
      IOUtils.closeStream(out);
    }
  }
}
```

一旦找到对应的 codec，便去除文件扩展名形成输出文件名，这是通过 CompressionCodecFactory 对象的静态方法 removeSuffix()来实现的。按照这种方法，一个名为 *file.gz* 的文件可以通过调用该程序解压为名为 *file* 的文件：

```
% hadoop FileDecompressor file.gz
```

CompressionCodecFactory 加载表 5-2 中除 LZO 之外的所有 codec，同样也加载 io.compression.codecs 配置属性(参见表 5-3)列表中的所有 codec。在默认情况下，该属性列表是空的，你可能只有在你拥有一个希望注册的定制 codec(例如外部管理的 LZO codec)时才需要加以修改。每个 codec 都知道自己默认的文件扩展名，因此 CompressionCodecFactory 可通过搜索注册的 codec 找到匹配指定文件扩展名的 codec(如果有的话)。

表 5-3. 压缩 codec 的属性

属性名称	类型	默认值	描述
io.compression.codecs	逗号分隔的类名		用于压缩/解压缩的额外的 CompressionCodec 类的列表

3. 原生类库

为了提高性能，最好使用“原生”(native)类库来实现压缩和解压缩。例如，在一个测试中，使用原生 gzip 类库可以减少约一半的解压缩时间和约 10%的压缩时间(与内置的 Java 实现相比)。表 5-4 说明了每种压缩格式是否有 Java 实现和原生类库实现。所有的格式都有原生类库实现，但是并非所有格式都有 Java 实现(如 LZO)。

表 5-4. 压缩代码库的实现

压缩格式	是否有 Java 实现	是否有原生实现
DEFLATE	是	是
gzip	是	是
bzip2	是	否
LZO	否	是
LZ4	否	是
Snappy	否	是

Apache Hadoop 二进制压缩包本身包含有为 64 位 Linux 构建的原生压缩二进制代码，称为 *libhadoop.so*。对于其他平台，需要自己根据位于源文件树最顶层的 *BUILDING.txt* 指令编译代码库。

可以通过 Java 系统的 `java.library.path` 属性指定原生代码库。*etc/hadoop* 文件夹中的 *hadoop* 脚本可以帮你设置该属性，但如果不用这个脚本，则需要在应用中手动设置该属性。

默认情况下，Hadoop 会根据自身运行的平台搜索原生代码库，如果找到相应的代码库就会自动加载。这意味着，你无需为了使用原生代码库而修改任何设置。但是，在某些情况下，例如调试一个压缩相关问题时，可能需要禁用原生代码库。将属性 `io.native.lib.available` 的值设置成 `false` 即可，这可确保使用内置的 Java 代码库(如果有的话)。

4. CodecPool

如果使用的是原生代码库并且需要在应用中执行大量压缩和解压缩操作，可以考虑使用 `CodecPool`，它支持反复使用压缩和解压缩，以分摊创建这些对象的开销。

范例 5-3 中的代码显示了 API 函数，不过在这个程序中，它只新建了一个 `Compressor`，并不需要使用压缩/解压缩池。

范例 5-3. 使用压缩池对读取自标准输入的数据进行压缩，然后将其写到标准输出

```
public class PooledStreamCompressor {

  public static void main(String[] args) throws Exception {
    String codecClassname = args[0];
    Class<?> codecClass = Class.forName(codecClassname);
    Configuration conf = new Configuration();
    CompressionCodec codec = (CompressionCodec)
      ReflectionUtils.newInstance(codecClass, conf);
    Compressor compressor = null;
    try {
      compressor = CodecPool.getCompressor(codec);
      CompressionOutputStream out =
        codec.createOutputStream(System.out, compressor);
      IOUtils.copyBytes(System.in, out, 4096, false);
      out.finish();
    } finally {
      CodecPool.returnCompressor(compressor);
    }
  }
}
```

在 codec 的重载方法 createOutputStream()中，对于指定的 CompressionCodec，我们从池中获取一个 Compressor 实例。通过使用 finally 数据块，我们在不同的数据流之间来回复制数据，即使出现 IOException 异常，也可以确保 compressor 可以返回池中。

5.2.2 压缩和输入分片

在考虑如何压缩将由 MapReduce 处理的数据时，理解这些压缩格式是否支持切分(splitting)是非常重要的。以一个存储在 HDFS 文件系统中且压缩前大小为 1 GB 的文件为例。如果 HDFS 的块大小设置为 128 MB，那么该文件将被存储在 8 个块中，把这个文件作为输入数据的 MapReduce 作业，将创建 8 个输入分片，其中每个分片作为一个单独的 map 任务的输入被独立处理。

现在想象一下，文件是经过 gzip 压缩的，且压缩后文件大小为 1 GB。与以前一样，HDFS 将这个文件保存为 8 个数据块。但是，将每个数据块单独作为一个输入分片是无法实现工作的，因为无法实现从 gzip 压缩数据流的任意位置读取数据，所以让 map 任务独立于其他任务进行数据读取是行不通的。gzip 格式使用 DEFLATE 算法来存储压缩后的数据，而 DEFLATE 算法将数据存储在一系列连续的压缩块中。问题在于每个块的起始位置并没有以任何形式标记，所以读取时无法从数据流的任意当前位置前进到下一块的起始位置读取下一个数据块，从而实现与整个数据流的同步。由于上述原因，gzip 并不支持文件切分。

在这种情况下，MapReduce 会采用正确的做法，它不会尝试切分 gzip 压缩文件，因为它知道输入是 gzip 压缩文件(通过文件扩展名看出)且 gzip 不支持切分。这是可行的，但牺牲了数据的本地性：一个 map 任务处理 8 个 HDFS 块，而其中大多数块并没有存储在执行该 map 任务的节点上。而且，map 任务数越少，作业的粒度就较大，因而运行的时间可能会更长。

在前面假设的例子中，如果文件是通过 LZO 压缩的，我们会面临相同的问题，因为这个压缩格式也不支持数据读取和数据流同步。但是，在预处理 LZO 文件的时候使用包含在 Hadoop LZO 库文件中的索引工具是可能的，你可以在 5.2.1 所列的 Google 和 GitHub 网站上获得该类库。该工具构建了切分点索引，如果使用恰当的 MapReduce 输入格式可有效实现文件的可切分特性。

另一方面，bzip2 文件提供不同数据块之间的同步标识(pi 的 48 位近似值)，因而它

支持切分。可以参见表 5-1，了解每个压缩格式是否支持切分。

应该使用哪种压缩格式？

Hadoop 应用处理的数据集非常大，因此需要借助于压缩。使用哪种压缩格式与待处理的文件的大小、格式和所使用的工具相关。下面有一些建议，大致是按照效率从高到低排列的。

- 使用容器文件格式，例如顺序文件(见 5.4.1 节)、Avro 数据文件(参见 12.3 节)、ORCFiles(见 5.4.3 节)或者 Parquet 文件(参见 13.2 节)，所有这些文件格式同时支持压缩和切分。通常最好与一个快速压缩工具联合使用，例如 LZO，LZ4，或者 Snappy。
- 使用支持切分的压缩格式，例如 bzip2(尽管 bzip2 非常慢)，或者使用通过索引实现切分的压缩格式，例如 LZO。
- 在应用中将文件切分成块，并使用任意一种压缩格式为每个数据块建立压缩文件(不论它是否支持切分)。这种情况下，需要合理选择数据块的大小，以确保压缩后数据块的大小近似于 HDFS 块的大小。
- 存储未经压缩的文件。

对大文件来说，不要使用不支持切分整个文件的压缩格式，因为会失去数据的本地特性，进而造成 MapReduce 应用效率低下。

5.2.3 在 MapReduce 中使用压缩

前面讲到通过 CompressionCodecFactory 来推断 CompressionCodec 时指出，如果输入文件是压缩的，那么在根据文件扩展名推断出相应的 codec 后，MapReduce 会在读取文件时自动解压缩文件。

要想压缩 MapReduce 作业的输出，应在作业配置过程中将 `mapreduce.output.fileoutputformat.compress` 属性设为 `true`，将 `mapre-duce.output.fileoutputformat.compress.codec` 属性设置为打算使用的压缩 codec 的类名。另一种方案是在 `FileOutputFormat` 中使用更便捷的方法设置这些属性，如范例 5-4 所示。

范例 5-4. 对查找最高气温作业所产生输出进行压缩

```
public class MaxTemperatureWithCompression {

  public static void main(String[] args) throws IOException {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperatureWithCompression <input path> " +
       "<output path>");
      System.exit(-1);
}

      Job job = new Job();
      job.setJarByClass(MaxTemperature.class);
      FileInputFormat.addInputPath(job, new Path(args[0]));
      FileOutputFormat.setOutputPath(job, new Path(args[1]));

      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(IntWritable.class);

      FileOutputFormat.setCompressOutput(job, true);
      FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);

      job.setMapperClass(MaxTemperatureMapper.class);
      job.setCombinerClass(MaxTemperatureReducer.class);
      job.setReducerClass(MaxTemperatureReducer.class);

      System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

我们按照如下指令对压缩后的输入运行程序(输出数据不必使用相同的压缩格式进行压缩，尽管本例中不是这样)：

```
% hadoop MaxTemperatureWithCompression input/ncdc/sample.txt.gz output
```

最终输出的每个部分都是经过压缩的。在这里，只有一部分结果：

```
% gunzip -c output/part-r-00000.gz
1949    111
1950    22
```

如果为输出生成顺序文件(sequence file)，可以设置 mapreduce.output.fileoutputformat.compress.type 属性来控制限制使用压缩格式。默认值是 RECORD，即针对每条记录进行压缩。如果将其改为 BLOCK，将针对一组记录进行压缩，这是推荐的压缩策略，因为它的压缩效率更高(参见 5.4.1 节)。

SequenceFileOutputFormat 类另外还有一个静态方法 putCompressionType()，可以用来便捷地设置该属性。

表 5-5 归纳概述了用于设置 MpaReduce 作业输出的压缩格式的配置属性。如果你的 MapReduce 驱动使用 Tool 接口(参见 6.2.2 节)，则可以通过命令行将这些属性传递给程序，这比通过程序代码来修改压缩属性更加简便。

表 5-5. MapReduce 的压缩属性

属性名称	类型	默认值	描述
mapreduce.output.fileoutputformat.compress	boolean	false	是否压缩输出
mapreduce.output.fileoutputformat.compress.codec	类名称	org.apache.hadoop.io.compress.DefaultCodec	map 输出所用的压缩 codec
mapreduce.output.fileoutputformat.compress.type	String	RECORD	顺序文件输出可以使用的压缩类型：NONE、RECORD 或者 BLOCK

对 map 任务输出进行压缩

尽管 MapReduce 应用读/写的是未经压缩的数据，但如果对 map 阶段的中间输入进行压缩，也可以获得不少好处。由于 map 任务的输出需要写到磁盘并通过网络传输到 reducer 节点，所以通过使用 LZO、LZ4 或者 Snappy 这样的快速压缩方式，是可以获得性能提升的，因为需要传输的数据减少了。启用 map 任务输出压缩和设置压缩格式的配置属性如表 5-6 所示。

表 5-6. map 任务输出的压缩属性

属性名称	类型	默认值	描述
mapreduce.map.output.compress	boolean	false	是否对 map 任务输出进行压缩
mapreduce.map.output.compress.codec	Class	org.apache.hadoop.io.compress.DefaultCodec	map 输出所用的压缩 codec

下面是在作业中启用 map 任务输出 gzip 压缩格式的代码(使用新 API)：

```
Configuration conf = new Configuration();
conf.setBoolean(Job.MAP_OUTPUT_COMPRESS, true);
conf.setClass(Job.MAP_OUTPUT_COMPRESS_CODEC, GzipCodec.class,
    CompressionCodec.class);
Job job = new Job(conf);
```

在旧的 API 中(参见附录 D)，JobConf 对象中可以通过更便捷的方法实现该功能：

```
conf.setCompressMapOutput(true);
conf.setMapOutputCompressorClass(GzipCodec.class);
```

5.3 序列化

序列化(serialization)是指将结构化对象转化为字节流以便在网络上传输或写到磁盘进行永久存储的过程。**反序列化**(deserialization)是指将字节流转回结构化对象的逆过程。

序列化用于分布式数据处理的两大领域：进程间通信和永久存储。

在 Hadoop 中，系统中多个节点上进程间的通信是通过“远程过程调用”(RPC，remote procedure call)实现的。RPC 协议将消息序列化成二进制流后发送到远程节点，远程节点接着将二进制流反序列化为原始消息。通常情况下，RPC 序列化格式如下。

- **紧凑**
  紧凑格式能充分利用网络带宽(数据中心中最稀缺的资源)。
- **快速**
  进程间通信形成了分布式系统的骨架，所以需要尽量减少序列化和反序列化的性能开销，这是最基本的。
- **可扩展**
  为了满足新的需求，协议不断变化。所以在控制客户端和服务器的过程中，需要直接引进相应的协议。例如，需要能够在方法调用的过程中增添新的参数，并且新的服务器需要能够接受来自老客户端的老格式的消息(无新增的参数)。
- **支持互操作**
  对于某些系统来说，希望能支持以不同语言写的客户端与服务器交互，所以需要设计需要一种特定的格式来满足这一需求。

表面看来，序列化框架对选择用于数据持久存储的数据格式应该会有不同的要求。毕竟，RPC 的存活时间不到 1 秒钟，持久存储的数据却可能在写到磁盘若干

年后才会被读取。但结果是，RPC 序列化格式的四大理想属性对持久存储格式而言也很重要。我们希望存储格式比较紧凑(进而高效使用存储空间)、快速(读/写数据的额外开销比较小)、可扩展(可以透明地读取老格式的数据)且可以互操作(以可以使用不同的语言读/写永久存储的数据)。

Hadoop 使用的是自己的序列化格式 Writable，它绝对紧凑、速度快，但不太容易用 Java 以外的语言进行扩展或使用。因为 Writable 是 Hadoop 的核心(大多数 MapReduce 程序都会为键和值类型使用它)，所以在接下来的三个小节中，我们要进行深入探讨，然后再介绍 Hadoop 支持的其他序列化框架。Avro(一个克服了 Writable 部分不足的序列化系统)将在第 12 章中讨论。

5.3.1 Writable 接口

Writable 接口定义了两个方法：一个将其状态写入 DataOutput 二进制流，另一个从 DataInput 二进制流读取状态：

```
package org.apache.hadoop.io;

import java.io.DataOutput;
import java.io.DataInput;
import java.io.IOException;

public interface Writable {
  void write(DataOutput out) throws IOException;
  void readFields(DataInput in) throws IOException;
}
```

让我们通过一个特殊的 Writable 类来看看它的具体用途。我们将使用 IntWritable 来封装 Java int 类型。我们可以新建一个对象并使用 set()方法来设置它的值:

```
IntWritable writable = new IntWritable();
writable.set(163);
```

也可以通过使用一个整数值作为输入参数的构造函数来新建一个对象：

```
IntWritable writable = new IntWritable(163);
```

为了检查 IntWritable 的序列化形式，我们在 java.io.DataOutputStream (java.io.DataOutput 的一个实现)中加入一个帮助函数来封装 java.io.ByteArrayOutputSteam，以便在序列化流中捕捉字节：

```
public static byte[] serialize(Writable writable) throws IOException {
  ByteArrayOutputStream out = new ByteArrayOutputStream();
  DataOutputStream dataOut = new DataOutputStream(out);
```

```
    writable.write(dataOut);
    dataOut.close();
    return out.toByteArray();
  }
```

一个整数占用 4 个字节(因为我们使用 JUnit4 进行声明)：

```
byte[] bytes = serialize(writable);
assertThat(bytes.length, is(4));
```

每个字节是按照大端顺序写入的(按照 java.io.DataOutput 接口中的声明，最重要的字节先写入流)，并且通过 Hadoop 的 StringUtils，我们可以看到这些字节的十六进制表示：

```
assertThat(StringUtils.byteToHexString(bytes), is("000000a3"));
```

让我们试试反序列化。我们再次新建一个辅助方法，从一个字节数组中读取一个 Writable 对象：

```
public static byte[] deserialize(Writable writable, byte[] bytes)
    throws IOException {
  ByteArrayInputStream in = new ByteArrayInputStream(bytes);
  DataInputStream dataIn = new DataInputStream(in);
  writable.readFields(dataIn);
  dataIn.close();
  return bytes;
}
```

我们构建了一个新的、空值的 IntWritable 对象，然后调用 deserialize()方法从我们刚写的输出数据中读取数据。最后，我们看到该值(通过 get()方法获取)是原始的数值 163：

```
IntWritable newWritable = new IntWritable();
deserialize(newWritable, bytes);
assertThat(newWritable.get(), is(163));
```

WritableComparable 接口和 comparator

IntWritable 实现原始的 WritableComparable 接口，该接口继承自 Writable 和 java.lang.Comparable 接口：

```
package org.apache.hadoop.io;

public interface WritableComparable<T> extends Writable, Comparable<T> {
}
```

对 MapReduce 来说，类型比较非常重要，因为中间有个基于键的排序阶段。Hadoop 提供的一个优化接口是继承自 Java Comparator 的 RawComparator

接口：

```
package org.apache.hadoop.io;
import java.util.Comparator;
public interface RawComparator<T> extends Comparator<T> {
  public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2);

}
```

该接口允许其实现直接比较数据流中的记录，无须先把数据流反序列化为对象，这样便避免了新建对象的额外开销。例如，我们根据 IntWritable 接口实现的 comparator 实现原始的 compare()方法，该方法可以从每个字节数组 b1 和 b2 中读取给定起始位置(s1 和 s2)以及长度(l1 和 l2)的一个整数进而直接进行比较。

WritableComparator 是对继承自 WritableComparable 类的 RawComparator 类的一个通用实现。它提供两个主要功能。第一，它提供了对原始 compare()方法的一个默认实现，该方法能够反序列化将在流中进行比较的对象，并调用对象的 compare()方法。第二，它充当的是 RawComparator 实例的工厂(已注册 Writable 的实现)。例如，为了获得 IntWritable 的 comparator，我们直接如下调用：

```
RawComparator<IntWritable> comparator = WritableComparator.get (IntWritable.class);
```

这个 comparator 可以用于比较两个 IntWritable 对象：

```
IntWritable w1 = new IntWritable(163);
IntWritable w2 = new IntWritable(67);
assertThat(comparator.compare(w1, w2), greaterThan(0));
```

或其序列化表示：

```
byte[] b1 = serialize(w1);
byte[] b2 = serialize(w2);
assertThat(comparator.compare(b1, 0, b1.length, b2, 0, b2.length),
greaterThan(0));
```

5.3.2 Writable 类

Hadoop 自带的 org.apache.hadoop.io 包中有广泛的 Writable 类可供选择。它们的层次结构如图 5-1 所示。

图 5-1. Writable 类的层次结构

1. Java 基本类型的 Writable 封装器

Writable 类对所有 Java 基本类型(参见表 5-7)提供封装，char 类型除外(可以存储在 IntWritable 中)。所有的封装包含 get()和 set()两个方法用于读取或存储封装的值。

表 5-7. Java 基本类型的 Writable 类

Java 基本类型	Writable 实现	序列化大小(字节)
boolean	BooleanWritable	1
byte	ByteWritable	1
Short	ShortWritable	2
int	IntWritable	4
	VintWritable	1~5
float	FloatWritable	4
long	LongWritable	8
	VlongWritable	1～9
double	DoubleWritable	8

对整数进行编码时，有两种选择，即定长格式(IntWritale 和 LongWritable)和变长格式(VIntWritable 和 VLongWritable)。需要编码的数值如果相当小(在-127 和 127 之间，包括-127 和 127)，变长格式就是只用一个字节进行编码；否则，使用第一个字节来表示数值的正负和后跟多少个字节。例如，值 163 需要两个字节：

```
byte[] data = serialize(new VIntWritable(163));
assertThat(StringUtils.byteToHexString(data), is("8fa3"));
```

如何在定长格式和变长格式之间进行选择呢？定长格式编码很适合数值在整个值域空间中分布非常均匀的情况，例如使用精心设计的哈希函数。然而，大多数数值变量的分布都不均匀，一般而言变长格式会更节省空间。变长编码的另一个优点是可以在 VIntWritable 和 VLongWritable 转换，因为它们的编码实际上是一致的。所以选择变长格式之后，便有增长的空间，不必一开始就用 8 字节的 long 表示。

2. Text 类型

Text 是针对 UTF-8 序列的 Writable 类。一般可以认为它是 java.lang.String 的

Writable 等价。

Text 类使用整型(通过边长编码的方式)来存储字符串编码中所需的字节数，因此该最大值为 2 GB。另外，Text 使用标准 UTF-8 编码，这使得能够更简便地与其他理解 UTF-8 编码的工具进行交互操作。

索引 由于着重使用标准的 UTF-8 编码，因此 Text 类和 Java String 类之间存在一定的差别。对 Text 类的索引是根据编码后字节序列中的位置实现的，并非字符串中的 Unicode 字符，也不是 Java char 的编码单元(如 String)。对于 ASCII 字符串，这三个索引位置的概念是一致的。charAt()方法的用法如下例所示：

```
Text t = new Text("hadoop");
assertThat(t.getLength(), is(6));
assertThat(t.getBytes().length, is(6));

assertThat(t.charAt(2), is((int) 'd'));
assertThat("Out of bounds", t.charAt(100), is(-1));
```

注意：harAt()方法返回的是一个表示 Unicode 编码位置的 int 类型值，而 String 返回一个 char 类型值。Text 还有一个 find()方法，该方法类似于 String 的 indexOf()方法：

```
Text t = new Text("hadoop");
assertThat("Find a substring", t.find("do"), is(2));
assertThat("Finds first 'o'", t.find("o"), is(3));
assertThat("Finds 'o' from position 4 or later", t.find("o", 4), is(4));
assertThat("No match", t.find("pig"), is(-1));
```

Unicode 一旦开始使用需要多个字节来编码的字符时，Text 和 String 之间的区别就昭然若揭了。考虑表 5-8 显示的 Unicode 字符。[①]

所有字符(除了表中最后一个字符 U+10400)，都可以使用单个 Java char 类型来表示。U+10400 是一个候补字符，并且需要两个 Java char 来表示，称为“字符代理对”(surrogate pair)。范例 5-5 中的测试显示了处理一个字符串(表 5-8 中的由 4 个字符组成的字符串)时 String 和 Text 之间的差别。

① 本例基于 Norbert Lindenberg 和 Masayoshi Okutsu 发表于 2004 年 5 月的文章“Supplementary Character in the Java Platform”(Java 平台中的增补字符)，网址为 *http://bit.ly/java_supp_characters*，中文版网址为 *http://m.blog.csdn.net/article/details?id=7345527*。

表 5-8. Unicode 字符

Unicode 编码点	U+0041	U+00DF	U+6771	U+10400
名称	拉丁大写字母 A	拉丁小写字母 SHARP S	无(统一表示的汉字)	DESERET CAPITAL LETTER LONG I
UTF-8 编码单元	41	c39f	e69db1	F0909080
Java 表示	\u0041	\u00DF	\u6771	\uuD801\uDC00

范例 5-5. 验证 String 类和 Text 类的差异性的测试

```
public class StringTextComparisonTest {

  @Test
  public void string() throws UnsupportedEncodingException {

    String s = "\u0041\u00DF\u6771\uD801\uDC00";
    assertThat(s.length(), is(5));
    assertThat(s.getBytes("UTF-8").length, is(10));

    assertThat(s.indexOf("\u0041"), is(0));
    assertThat(s.indexOf("\u00DF"), is(1));
    assertThat(s.indexOf("\u6771"), is(2));
    assertThat(s.indexOf("\uD801\uDC00"), is(3));

    assertThat(s.charAt(0), is('\u0041'));
    assertThat(s.charAt(1), is('\u00DF'));
    assertThat(s.charAt(2), is('\u6771'));
    assertThat(s.charAt(3), is('\uD801'));
    assertThat(s.charAt(4), is('\uDC00'));

    assertThat(s.codePointAt(0), is(0x0041));
    assertThat(s.codePointAt(1), is(0x00DF));
    assertThat(s.codePointAt(2), is(0x6771));
    assertThat(s.codePointAt(3), is(0x10400));
  }

  @Test
  public void text() {
    Text t = new Text("\u0041\u00DF\u6771\uD801\uDC00");
    assertThat(t.getLength(), is(10));
    assertThat(t.find("\u0041"), is(0));
    assertThat(t.find("\u00DF"), is(1));
    assertThat(t.find("\u6771"), is(3));
    assertThat(t.find("\uD801\uDC00"), is(6));

    assertThat(t.charAt(0), is(0x0041));
    assertThat(t.charAt(1), is(0x00DF));
    assertThat(t.charAt(3), is(0x6771));
```

```
    assertThat(t.charAt(6), is(0x10400));
  }
}
```

这个测试证实 String 的长度是其所含 char 编码单元的个数(5，由该字符串的前三个字符和最后的一个代理对组成)，但 Text 对象的长度却是其 UTF-8 编码的字节数(10=1+2+3+4)。相似的，String 类的 indexOf()方法返回 char 编码单元中的索引位置，Text 类的 find()方法则返回字节偏移量。

当代理对不能代表整个 Unicode 字符时，String 类中的 charAt()方法会根据指定的索引位置返回 char 编码单元。根据 char 编码单元索引位置，需要 codePointAt()方法来获取表示成 int 类型的单个 Unicode 字符。事实上，Text 类中的 charAt()方法与 String 中的 codePointAt()更加相似(相较名称而言)。唯一的区别是通过字节的偏移量进行索引。

迭代 利用字节偏移量实现的位置索引，对 Text 类中的 Unicode 字符进行迭代是非常复杂的，因为无法通过简单地增加索引值来实现该迭代。同时迭代的语法有些模糊(参见范例 5-6)：将 Text 对象转换为 java.nio.ByteBuffer 对象，然后利用缓冲区对 Text 对象反复调用 bytesToCodePoint()静态方法。该方法能够获取下一代码的位置，并返回相应的 int 值，最后更新缓冲区中的位置。当 bytesToCodePoint()返回-1 时，则检测到字符串的末尾。

范例 5-6. 遍历 Text 对象中的字符

```
public class TextIterator {
    public static void main(String[] args) {
      Text t = new Text("\u0041\u00DF\u6771\uD801\uDC00");

    ByteBuffer buf = ByteBuffer.wrap(t.getBytes(), 0, t.getLength());
    int cp;
    while(buf.hasRemaining() && (cp = Text.bytesToCodePoint(buf))!=-1){
      System.out.println(Integer.toHexString(cp));
    }
  }
}
```

运行这个程序，打印出字符串中四个字符的编码点(code point)：

```
% hadoop TextIterator
41
df
6771
10400
```

可变性 与 String 相比，Text 的另一个区别在于它是可变的(与所有 Hadoop 的

Writable 接口实现相似，NullWritable 除外，它是单实例对象)。可以通过调用其中一个 set()方法来重用 Text 实例。例如：

```
Text t = new Text("hadoop");
t.set("pig");
assertThat(t.getLength(), is(3));
assertThat(t.getBytes().length, is(3));
```

在某些情况下，getBytes()方法返回的字节数组可能比 getLength()函数返回的长度更长：

```
Text t = new Text("hadoop");
t.set(new Text("pig"));
assertThat(t.getLength(), is(3));
assertThat("Byte length not shortened", t.getBytes().length, is(6));
```

以上代码说明了在调用 getBytes()之前为什么始终都要调用 getLength()方法，因为可以由此知道字节数组中多少字符是有效的。

对 String 重新排序 Text 类并不像 java.lang.String 类那样有丰富的字符串操作 API。所以，在多数情况下需要将 Text 对象转换成 String 对象。这一转换通常通过调用 toString()方法来实现：

```
assertThat(new Text("hadoop").toString(), is("hadoop"));
```

3. BytesWritable

BytesWritable 是对二进制数据数组的封装。它的序列化格式为一个指定所含数据字节数的整数域(4 字节)，后跟数据内容本身。例如，长度为 2 的字节数组包含数值 3 和 5，序列化形式为一个 4 字节的整数(00000002)和该数组中的两个字节(03 和 05)：

```
BytesWritable b = new BytesWritable(new byte[] { 3, 5 });
byte[] bytes = serialize(b);
assertThat(StringUtils.byteToHexString(bytes), is("000000020305"));
```

BytesWritable 是可变的，其值可以通过 set()方法进行修改。和 Text 相似，BytesWritable 类的 getBytes()方法返回的字节数组长度(容量)可能无法体现 BytesWritable 所存储数据的实际大小。可以通过 getLength()方法来确定 BytesWritable 的大小。示例如下：

```
b.setCapacity(11);
assertThat(b.getLength(), is(2));
assertThat(b.getBytes().length, is(11));
```

4. NullWritable

NullWritable 是 Writable 的特殊类型，它的序列化长度为 0。它并不从数据流中读取数据，也不写入数据。它充当占位符；例如，在 MapReduce 中，如果不需要使用键或值的序列化地址，就可以将键或值声明为 NullWritable，这样可以高效存储常量空值。如果希望存储一系列数值，与键-值对相对，NullWritable 也可以用作在 SequenceFile 中的键。它是一个不可变的单实例类型，通过调用 NullWritable.get()方法可以获取这个实例。

5. ObjectWritable 和 GenericWritable

ObjectWritable 是对 Java 基本类型(String，enum，Writable，null 或这些类型组成的数组)的一个通用封装。它在 Hadoop RPC 中用于对方法的参数和返回类型进行封装和解封装。

当一个字段中包含多个类型时，ObjectWritable 非常有用：例如，如果 SequenceFile 中的值包含多个类型，就可以将值类型声明为 ObjectWritable，并将每个类型封装在一个 ObjectWritable 中。作为一个通用的机制，每次序列化都写封装类型的名称，这非常浪费空间。如果封装的类型数量比较少并且能够提前知道，那么可以通过使用静态类型的数组，并使用对序列化后的类型的引用加入位置索引来提高性能。GenericWritable 类采取的就是这种方式，所以你得在继承的子类中指定支持什么类型。

6. Writable 集合类

org.apache.hadoop.io 软件包中共有 6 个 Writable 集合类，分别是 ArrayWritable、ArrayPrimitiveWritable、TwoDArrayWritable、MapWritable、SortedMapWritable 以及 EnumMapWritable。

ArrayWritable 和 TwoDArrayWritable 是对 Writable 的数组和两维数组(数组的数组)的实现。ArrayWritable 或 TwoDArrayWritable 中所有元素必须是同一类的实例(在构造函数中指定)，如下所示：

```
ArrayWritable writable = new ArrayWritable(Text.class);
```

如果 Writable 根据类型来定义，例如 SequenceFile 的键或值，或更多时候作为 MapReduce 的输入，则需要继承 ArrayWritable(或相应的 TwoDArray

Writable 类)并设置静态类型。示例如下：

```
public class TextArrayWritable extends ArrayWritable {
  public TextArrayWritable() {
    super(Text.class);
  }
}
```

ArrayWritable 和 TwoDArrayWritable 都有 get()、set()和 toArray()方法。toArray()方法用于新建该数组(或二维数组)的一个“浅拷贝”(shallow copy)。

ArrayPrimitiveWritable 是对 Java 基本数组类型的一个封装。调用 set()方法时，可以识别相应组件类型，因此无需通过继承该类来设置类型。

MapWritable 和 SortedMapWritable 分别实现了 java.util.Map<Writable, Writable>和 java.util.SortedMap<WritableComparable, Writable>。每个键/值字段使用的类型是相应字段序列化形式的一部分。类型存储为单个字节(充当类型数组的索引)。在 org.apache.hadoop.io 包中，数组经常与标准类型结合使用，而定制的 Writable 类型也通常结合使用，但对于非标准类型，则需要在包头指明所使用的数组类型。根据实现，MapWritable 类和 SortedMapWritable 类通过正 byte 值来指示定制的类型，所以在 MapWritable 和 SortedMapWritable 实例中最多可以使用 127 个不同的非标准 Wirtable 类。下面显示使用了不同键值类型的 MapWritable 实例：

```
MapWritable src = new MapWritable();
src.put(new IntWritable(1), new Text("cat"));
src.put(new VIntWritable(2), new LongWritable(163));

MapWritable dest = new MapWritable();
WritableUtils.cloneInto(dest, src);
assertThat((Text) dest.get(new IntWritable(1)), is(new Text("cat")));
assertThat((LongWritable) dest.get(new VIntWritable(2)),
is(new LongWritable(163)));
```

显然，可以通过 Writable 集合类来实现集合和列表。可以使用 MapWritable 类型(或针对排序集合，使用 SortedMapWritable 类型)来枚举集合中的元素，用 NullWritable 类型枚举值。对集合的枚举类型可采用 EnumSetWritable。对于单类型的 Writable 列表，使用 ArrayWritable 就足够了，但如果需要把不同的 Writable 类型存储在单个列表中，可以用 GenericWritable 将元素封装在一个 ArrayWritable 中。另一个可选方案是，可以借鉴 MapWritable 的思路写一个通用的 ListWritable。

5.3.3 实现定制的 Writable 集合

Hadoop 有一套非常有用的 Writable 实现可以满足大部分需求，但在有些情况下，我们需要根据自己的需求构造一个新的实现。有了定制的 Writable 类型，就可以完全控制二进制表示和排序顺序。由于 Writable 是 MapReduce 数据路径的核心，所以调整二进制表示能对性能产生显著效果。虽然 Hadoop 自带的 Writable 实现已经过很好的性能调优，但如果希望将结构调整得更好，更好的做法往往是新建一个 Writable 类型(而不是组合打包的类型)。

如果你正考虑写一个定制的 Writable，值得尝试另一种序列化框架，例如 Avro，允许你以声明方式定义定制的类型。详情可以参见 5.3.4 节有关序列化框架的内容及第 12 章。

为了演示如何新建一个定制的 Writable，我们写一个表示一对字符串的实现，名为 TextPair。范例 5-7 展示了最基本的实现。

范例 5-7. 存储一对 Text 对象的 Writable 实现

```
import java.io.*;

import org.apache.hadoop.io.*;

public class TextPair implements WritableComparable<TextPair> {

  private Text first;
  private Text second;

  public TextPair() {
    set(new Text(), new Text());
  }

  public TextPair(String first, String second) {
    set(new Text(first), new Text(second));
  }

  public TextPair(Text first, Text second) {
    set(first, second);
  }

  public void set(Text first, Text second) {
    this.first = first;
    this.second = second;
  }

  public Text getFirst() {
    return first;
```

```
  }

  public Text getSecond() {
    return second;
  }

  @Override
  public void write(DataOutput out) throws IOException {
    first.write(out);
    second.write(out);
  }

@Override
  public void readFields(DataInput in) throws IOException {
    first.readFields(in);
    second.readFields(in);
  }

  @Override
  public int hashCode() {
    return first.hashCode() * 163 + second.hashCode();
  }

  @Override
  public boolean equals(Object o) {
    if (o instanceof TextPair) {
      TextPair tp = (TextPair) o;
      return first.equals(tp.first) && second.equals(tp.second);
    }
    return false;
  }

@Override
  public String toString() {
    return first + "\t" + second;
  }

  @Override
  public int compareTo(TextPair tp) {
    int cmp = first.compareTo(tp.first);
    if (cmp != 0) {
      return cmp;
    }
    return second.compareTo(tp.second);
  }
}
```

这个定制 Writable 实现的第一部分非常直观：包括两个 Text 实例变量(first 和 second)和相关的构造函数，以及 setter 和 getter(即设置函数和提取函数)。所有 Writable 实现都必须有一个默认的构造函数以便 MapReduce 框架可以对它们进行实例化，然后再调用 readFields()函数查看(填充)各个字段的值。

Writable 实例是可变的并且通常可以重用，所以应该尽量避免在 write()或 readFields()方法中分配对象。

通过让 Text 对象自我表示，TextPair 类的 write()方法依次将每个 Text 对象序列化到输出流中。类似的，通过每个 Text 对象的表示，readFields()方法对来自输入流的字节进行反序列化。DataOutput 和 DataInput 接口有一套丰富的方法可以用于对 Java 基本类型进行序列化和反序列化，所以，在通常情况下，你可以完全控制 Writable 对象在线上传输/交换(的数据)的格式(数据传输格式)。

就像针对 Java 语言构造的任何值对象那样，需要重写 java.lang.Object 中的 hashCode()、equals()和 toString()方法。HashPartitioner (MapReduce 中的默认分区类)通常用 hashCode()方法来选择 reduce 分区，所以应该确保有一个比较好的哈希函数来保证每个 reduce 分区的大小相似。

即便计划结合使用 TextOutputFormat 和定制的 Writable，也得自己动手实现 toString()方法。TextOutputFormat 对键和值调用 toString()方法，将键和值转换为相应的输出表示。针对 TextPair，我们将原始的 Text 对象作为字符串写到输出，各个字符串之间要用制表符来分隔。

TextPair 是 WritableComparable 的一个实现，所以它提供了 compareTo()方法，该方法可以强制数据排序：先按第一个字符排序，如果第一个字符相同，则按照第二个字符排序。注意，除了可存储的 Text 对象数目， TextPair 不同于 TextArrayWritable(前一小节中已经提到)，因为 TextArrayWritable 只继承 Writable，并没有继承WritableComparable。

1. 为提高速度实现一个 RawComparator

范例 5-7 中的 TextPair 代码可以按照其描述的基本方式运行；但我们也可以进一步优化。按照 5.3.1 节的说明，当 TextPair 被用作 MapReduce 中的键时，需要将数据流反序列化为对象，然后再调用 compareTo()方法进行比较。那么有没有可能看看它们的序列化表示就可以比较两个 TextPair 对象呢？

事实证明，我们可以这样做，因为 TextPair 是两个 Text 对象连接而成的，而 Text 对象的二进制表示是一个长度可变的整数，包含字符串之 UTF-8 表示的字节数以及 UTF-8 字节本身。诀窍在于读取该对象的起始长度，由此得知第一个 Text 对象的字节表示有多长；然后将该长度传给 Text 对象的 RawComparator 方法，最后通过计算第一个字符串和第二个字符串恰当的偏移量，这样便可以实现对象

的比较。详细过程参见范例 5-8 (注意，这段代码已嵌入 TextPair 类中)。

范例 5-8. 用于比较 TextPair 字节表示的 RawComparator

```
public static class Comparator extends WritableComparator {
      private static final Text.Comparator TEXT_COMPARATOR = new Text.Comparator();
  public Comparator() {
    super(TextPair.class);
  }

  @Override
  public int compare(byte[] b1, int s1, int l1,
                     byte[] b2, int s2, int l2) {
    try {
      int firstL1 = WritableUtils.decodeVIntSize(b1[s1]) + readVInt(b1, s1);
      int firstL2 = WritableUtils.decodeVIntSize(b2[s2]) + readVInt(b2, s2);
      int cmp = TEXT_COMPARATOR.compare(b1, s1, firstL1, b2, s2, firstL2);
      if (cmp != 0) {
        return cmp;
      }
      return TEXT_COMPARATOR.compare(b1, s1 + firstL1, l1 - firstL1,
                                     b2, s2 + firstL2, l2 - firstL2);
    } catch (IOException e) {
      throw new IllegalArgumentException(e);
    }
  }
}

static {
  WritableComparator.define(TextPair.class, new Comparator());
}
```

事实上，我们采取的做法是继承 WritableComparable 类，而非实现 RawComparator 接口，因为它提供了一些比较好用的方法和默认实现。这段代码最本质的部分是计算 firstL1 和 firstL2，这两个参数表示每个字节流中第一个 Text 字段的长度。两者分别由变长整数的长度(由 WritableUtils 的 decodeVIntSize()方法返回)和编码值(由 readVInt()方法返回)组成。

2. 定制的 comparator

从 TextPair 可以看出，编写原始的 comparator 需要谨慎，因为必须要处理字节级别的细节。如果真的需要自己写 comparator，有必要参考 org.apache.hadoop.io 包中对 Writable 接口的实现。WritableUtils 提供的方法也比较好用。

如果可能，定制的 comparator 也应该继承自 RawComparator。这些 comparator 定义的排列顺序不同于默认 comparator 定义的自然排列顺序。范例 5-9 显示了一个针对 TextPair 类型的comparator，称为FirstCompartator，它只考虑 TextPair 对象的第一

个字符串。注意，我们重载了针对该类对象的 compare()方法，使两个compare()方法有相同的语法。

范例 5-9. 定制的 RawComparator 用于比较 TextPair 对象字节表示的第一个字段

```
public static class FirstComparator extends WritableComparator {
    private static final Text.Comparator TEXT_COMPARATOR = new Text.Comparator();
    public FirstComparator() {
      super(TextPair.class);
    }

    @Override
    public int compare(byte[] b1, int s1, int l1,
                       byte[] b2, int s2, int l2) {
      try {
        int firstL1 = WritableUtils.decodeVIntSize(b1[s1]) + readVInt (b1, s1);
        int firstL2 = WritableUtils.decodeVIntSize(b2[s2]) + readVInt (b2, s2);
        return TEXT_COMPARATOR.compare(b1, s1, firstL1, b2, s2, firstL2);
      } catch (IOException e) {
        throw new IllegalArgumentException(e);
      }
    }

    @Override
    public int compare(WritableComparable a, WritableComparable b) {
      if (a instanceof TextPair && b instanceof TextPair) {
        return ((TextPair) a).first.compareTo(((TextPair) b).first);
      }
        return super.compare(a, b);
    }
  }
```

第 9 章在介绍 MapReduce 的连接操作和辅助排序(参见 9.3 节)的时候，将使用这个 comparator。

5.3.4 序列化框架

尽管大多数 MapReduce 程序使用的都是 Writable 类型的键和值，但这并不是 MapReduce API 强制要求使用的。事实上，可以使用任何类型，只要能有一种机制对每个类型进行类型与二进制表示的来回转换就可以。

为了支持这一机制，Hadoop 有一个针对可替换序列化框架(serialization framework)的 API。序列化框架用一个 Serialization 实现(包含在 org.apache.hadoop.io.serializer 包中)来表示。例如，WritableSerialization 类是对 Writable 类型的 Serialization 实现。

Serialization 对象定义了从类型到 Serializer 实例(将对象转换为字节流)和 Deserializer 实例(将字节流转换为对象)的映射方式。

为了注册 Serialization 实现，需要将 io.serizalizations 属性设置为一个由逗号分隔的类名列表。它的默认值包括 org.apache.hadoop.io.serializer.WritableSerialization 和 Avro 指定(Specific)序列化及 Reflect(自反)序列化类(详见 12.1 节)，这意味着只有 Writable 对象和 Avro 对象才可以在外部序列化和反序列化。

Hadoop 包含一个名为 JavaSerialization 的类，该类使用 Java Object Serialization。尽管它方便了我们在 MapReduce 程序中使用标准的 Java 类型，如 Integer 或 String，但不如 Writable 高效，所以不建议使用(参见以下的补充内容)。

序列化 IDL

还有许多其他序列化框架从不同的角度来解决该问题：不通过代码来定义类型，而是使用“接口定义语言”(IDL，Interface Description Language)以不依赖于具体语言的方式进行声明。由此，系统能够为其他语言生成类型，这种形式能有效提高互操作能力。它们一般还会定义版本控制方案(使类型的演化直观易懂)。

两个比较流行的序列化框架 Apache Thrift(*http://thrift.apache.org/*)和 Google 的 Protocol Buffers (*http://code.google.com/p/protobuf/*)，常常用作二进制数据的永久存储格式。MapReduce 格式对该类的支持有限，[①] 但在 Hadoop 内部，部分组件仍使用上述两个序列化框架来实现 RPC 和数据交换。

Avro 是一个基于 IDL 的序列化框架，非常适用于 Hadoop 的大规模数据处理。我们将在第 12 章讨论 Avro。

> **为什么不用 Java Object Serialization?**
>
> Java 有自己的序列化机制，称为“Java Object Serialization”(通常简称为“Java Serialization”)，该机制与编程语言紧密相关，所以我们很自然会问为什么不在 Hadoop 中使用该机制。针对这个问题，Doug Cutting 是这样解释的：“为什么

① Twitter 的大象鸟项目(*http://github.com/kevinweil/elephant-bird*)包含一些工具，用于在 Hadoop 中与 Thrif 和 Protocol Buffers 结合使用。

开始设计 Hadoop 的时候我不用 Java Serialization？因为它看起来太复杂，而我认为需要有一个至精至简的机制，可以用于精确控制对象的读和写，这个机制将是 Hadoop 的核心。使用 Java Serialization 虽然可以获得一些控制权，但用起来非常纠结。

不用 RMI(Remote Method Invocation 远程方法调用)也出于类似的考虑。高效、高性能的进程间通信是 Hadoop 的关键。我觉得我们需要精确控制连接、延迟和缓冲的处理方式，RMI 对此无能为力。”

问题在于 Java Serialization 不满足先前列出的序列化格式标准：精简、快速、可扩展、支持互操作。

5.4 基于文件的数据结构

对于某些应用，我们需要一种特殊的数据结构来存储自己的数据。对于基于 MapReduce 的数据处理，将每个二进制数据大对象(blob)单独放在各自的文件中不能实现可扩展性，所以，Hadoop 为此开发了很多更高层次的容器。

5.4.1 关于 SequenceFile

考虑日志文件，其中每一行文本代表一条日志记录。纯文本不合适记录二进制类型的数据。在这种情况下，Hadoop 的 `SequenceFile` 类非常合适，为二进制键-值对提供了一个持久数据结构。将它作为日志文件的存储格式时，你可以自己选择键(比如 `LongWritable` 类型所表示的时间戳)，以及值可以是 `Writable` 类型(用于表示日志记录的数量)。

`SequenceFiles` 也可以作为小文件的容器。HDFS 和 MapReduce 是针对大文件优化的，所以通过 `SequenceFile` 类型将小文件包装起来，可以获得更高效率的存储和处理。在 8.2.1 节中，我们讲到将整个文件作为一条记录处理时，提供了一个程序，它将若干个小文件打包成一个 `SequenceFile` 类[①]。

① 无独有偶，Stuart Sierra 的博客文章“A Million Little Files”中也包含将 tar 文件转为 `SequenceFile` 的代码，参见 *http://stuartsierra.com/2008/ 04/24/a-million-little-files*。

1. SequenceFile 的写操作

通过 createWriter()静态方法可以创建 SequenceFile 对象，并返回 SequenceFile.Writer 实例。该静态方法有多个重载版本，但都需要指定待写入的数据流(FSDataOutputStream 或 FileSystem 对象和 Path 对象)，Configuration 对象，以及键和值的类型。另外，可选参数包括压缩类型以及相应的 codec，Progressable 回调函数用于通知写入的进度，以及在 SequenceFile 头文件中存储的 Metadata 实例。

存储在 SequenceFile 中的键和值并不一定需要是 Writable 类型。只要能被 Serialization 序列化和反序列化，任何类型都可以。

一旦拥有 SequenceFile.Writer 实例，就可以通过 append()方法在文件末尾附加键-值对。写完后，可以调用 close()方法(SequenceFile.Writer 实现了 java.io.Closeable 接口)。

范例 5-10 显示了一小段代码，它使用刚才描述的 API 将键-值对写入一个 SequenceFile。

范例 5-10. 写入 SequenceFile 对象

```
public class SequenceFileWriteDemo {
  private static final String[] DATA = {
  "One, two, buckle my shoe",
  "Three, four, shut the door",
  "Five, six, pick up sticks",
  "Seven, eight, lay them straight",
  "Nine, ten, a big fat hen"
  };

  public static void main(String[] args) throws IOException {
  String uri = args[0];
  Configuration conf = new Configuration();
  FileSystem fs = FileSystem.get(URI.create(uri), conf);
  Path path = new Path(uri);

  IntWritable key = new IntWritable();
  Text value = new Text();
  SequenceFile.Writer writer = null;
  try {
    writer = SequenceFile.createWriter(fs, conf, path,
        key.getClass(), value.getClass());
    for (int i = 0; i < 100; i++) {
      key.set(100 - i);
      value.set(DATA[i % DATA.length]);
      System.out.printf("[%s]\t%s\t%s\n", writer.getLength(), key, value);
```

```
      writer.append(key, value);
    }
  } finally {
    IOUtils.closeStream(writer);
  }
 }
}
```

顺序文件中存储的键-值对，键是从 100 到 1 降序排列的整数，表示为 IntWritable 对象，值是 Text 对象。在将每条记录追加到 SequenceFile.Writer 实例末尾之前，我们调用 getLength()方法来获取文件的当前位置。(在下一小节中，如果不按顺序读取文件，则使用这一信息作为记录的边界。)我们把这个位置信息和键-值对输出到控制台。结果如下所示：

```
% hadoop SequenceFileWriteDemo numbers.seq
[128]   100     One, two, buckle my shoe
[173]   99      Three, four, shut the door
[220]   98      Five, six, pick up sticks
[264]   97      Seven, eight, lay them straight
[314]   96      Nine, ten, a big fat hen
[359]   95      One, two, buckle my shoe
[404]   94      Three, four, shut the door
[451]   93      Five, six, pick up sticks
[495]   92      Seven, eight, lay them straight
[545]   91      Nine, ten, a big fat hen
...
[1976]  60      One, two, buckle my shoe
[2021]  59      Three, four, shut the door
[2088]  58      Five, six, pick up sticks
[2132]  57      Seven, eight, lay them straight
[2182]  56      Nine, ten, a big fat hen
...
[4557]  5       One, two, buckle my shoe
[4602]  4       Three, four, shut the door
[4649]  3       Five, six, pick up sticks
[4693]  2       Seven, eight, lay them straight
[4743]  1       Nine, ten, a big fat hen
```

2. SequenceFile 的读操作

从头到尾读取顺序文件不外乎创建 SequenceFile.Reader 实例后反复调用 next()方法迭代读取记录。读取的是哪条记录与你使用的序列化框架相关。如果使用的是 Writable 类型，那么通过键和值作为参数的 next()方法可以将数据流中的下一条键-值对读入变量中：

```
public boolean next(Writable key, Writable val)
```

如果键-值对成功读取，则返回 true，如果已读到文件末尾，则返回 false。

对于其他非 Writable 类型的序列化框架(比如 Apache Thrift)，则应该使用下面两个方法：

```
public Object next(Object key) throws IOException
public Object getCurrentValue(Object val) throws IOException
```

在这种情况下，需要确保 io.serializations 属性已经设置了你想使用的序列化框架，详情参见 5.3.4 节。

如果 next()方法返回的是非 null 对象，则可以从数据流中读取键、值对，并且可以通过 getCurrentValue()方法读取该值。否则，如果 next()返回 null 值，则表示已经读到文件末尾。

范例 5-11 中的程序显示了如何读取包含 Writable 类型键、值对的顺序文件。注意如何通过调用 getKeyClass()方法和 getValueClass()方法进而发现 SequenceFile 中所使用的类型，然后通过 ReflectionUtils 对象生成常见键和值的实例。通过这个技术，该程序可用于处理有 Writable 类型键、值对的任意一个顺序文件。

范例 5-11. 读取 SequenceFile

```
public class SequenceFileReadDemo {

  public static void main(String[] args) throws IOException {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);
Path path = new Path(uri);

    SequenceFile.Reader reader = null;
    try {
      reader = new SequenceFile.Reader(fs, path, conf);
      Writable key = (Writable)
        ReflectionUtils.newInstance(reader.getKeyClass(), conf);
      Writable value = (Writable)
        ReflectionUtils.newInstance(reader.getValueClass(), conf);
      long position = reader.getPosition();
      while (reader.next(key, value)) {
        String syncSeen = reader.syncSeen() ? "*" : "";
        System.out.printf("[%s%s]\t%s\t%s\n", position, syncSeen, key, value);
        position = reader.getPosition(); // beginning of next record
      }
    } finally {
      IOUtils.closeStream(reader);
}
  }
}
```

该程序的另一个特性是能够显示顺序文件中同步点的位置信息。所谓同步点，是指数据读取迷路(lost)后能够再一次与记录边界同步的数据流中的某个位置，例如，在数据流中由于搜索而跑到任意位置后可采取此动作。同步点是由 SequenceFile.Writer 记录的，后者在顺序文件写入过程中插入一个特殊项以便每隔几个记录便有一个同步标识。这样的特殊项非常小，因而只造成很小的存储开销，不到 1%。同步点始终位于记录的边界处。

运行范例 5-11 的程序后，会显示星号表示的顺序文件中的同步点。第一同步点位于 2021 处(第二个位于 4075 处，但本例中并没有显示出来)：

```
% hadoop SequenceFileReadDemo numbers.seq
[128]   100     One, two, buckle my shoe
[173]   99      Three, four, shut the door
[220]   98      Five, six, pick up sticks
[264]   97      Seven, eight, lay them straight
[314]   96      Nine, ten, a big fat hen
[359]   95      One, two, buckle my shoe
[404]   94      Three, four, shut the door
[451]   93      Five, six, pick up sticks
[495]   92      Seven, eight, lay them straight
[545]   91      Nine, ten, a big fat hen
[590]   90      One, two, buckle my shoe
...
[1976]  60      One, two, buckle my shoe
[2021*] 59      Three, four, shut the door
[2088]  58      Five, six, pick up sticks
[2132]  57      Seven, eight, lay them straight
[2182]  56      Nine, ten, a big fat hen
...
[4557]  5       One, two, buckle my shoe
[4602]  4       Three, four, shut the door
[4649]  3       Five, six, pick up sticks
[4693]  2       Seven, eight, lay them straight
[4743]  1       Nine, ten, a big fat hen
```

在顺序文件中搜索给定位置有两种方法。第一种是调用 seek()方法，该方法将读指针指向文件中指定的位置。例如，可以按如下方式搜查记录边界：

```
reader.seek(359);
assertThat(reader.next(key, value), is(true));
assertThat(((IntWritable) key).get(), is(95));
```

但如果给定位置不是记录边界，调用 next()方法时就会出错：

```
reader.seek(360);
reader.next(key, value); // fails with IOException
```

第二种方法通过同步点查找记录边界。SequenceFile.Reader 对象的 sync(long

position)方法可以将读取位置定位到 position 之后的下一个同步点。如果 position 之后没有同步了，那么当前读取位置将指向文件末尾。这样，我们对数据流中的任意位置调用 sync()方法(不一定是一个记录的边界)而且可以重新定位到下一个同步点并继续向后读取：

```
reader.sync(360);
assertThat(reader.getPosition(), is(2021L));
assertThat(reader.next(key, value), is(true));
assertThat(((IntWritable) key).get(), is(59));
```

SequenceFile.Writer 对象有一个 sync()方法，该方法可以在数据流的当前位置插入一个同步点。不要把它和 Syncable 接口中定义的 hsync()方法混为一谈，后者用于底层设备缓冲区的同步。详情可以参见 3.6.3 节。

可以将加入同步点的顺序文件作为 MapReduce 的输入，因为该类顺序文件允许切分，由此该文件的不同部分可以由独立的 map 任务单独处理。参见 8.2.3 节对 SequenceFileInputFormat 的详细介绍。

1. 通过命令行接口显示 SequenceFile

hadoop fs 命令有一个-text 选项可以以文本形式显示顺序文件。该选项可以查看文件的代码，由此检测出文件的类型并将其转换成相应的文本。该选项可以识别 gzip 压缩文件、顺序文件和 Avro 数据文件；否则，便假设输入为纯文本文件。

对于顺序文件，如果键和值是有具体含义的字符串表示，那么这个命令就非常有用(通过 toString()方法定义)。同样，如果有自己定义的键或值的类，则需要确保它们在 Hadoop 类路径目录下。

对前一小节中创建的顺序文件执行这个命令，我们得到如下输出：

```
% hadoop fs -text numbers.seq | head
100     One, two, buckle my shoe
99      Three, four, shut the door
98      Five, six, pick up sticks
97      Seven, eight, lay them straight
96      Nine, ten, a big fat hen
95      One, two, buckle my shoe
94      Three, four, shut the door
93      Five, six, pick up sticks
92      Seven, eight, lay them straight
91      Nine, ten, a big fat hen
```

2. SequenceFile 的排序和合并

MapReduce 是对多个顺序文件进行排序(或合并)最有效的方法。MapReduce 本身是并行的，并且可由你指定要使用多少个 reducer(该数决定着输出分区数)。例如，通过指定一个 reducer，可以得到一个输出文件。我们可以使用 Hadoop 发行版自带的例子，通过指定键和值的类型来将输入和输出指定为顺序文件：

```
% hadoop jar \
 $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-*.jar \
 sort -r 1 \
 -inFormat org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat \
 -outFormat org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat \
 -outKey org.apache.hadoop.io.IntWritable \
 -outValue org.apache.hadoop.io.Text \
 numbers.seq sorted
% hadoop fs -text sorted/part-r-00000 | head
1       Nine, ten, a big fat hen
2       Seven, eight, lay them straight
3       Five, six, pick up sticks
4       Three, four, shut the door
5       One, two, buckle my shoe
6       Nine, ten, a big fat hen
7       Seven, eight, lay them straight
8       Five, six, pick up sticks
9       Three, four, shut the door
10      One, two, buckle my shoe
```

更多详情可以参见 9.2 节。

除了通过 MapReduce 实现排序/归并，还有一种方法是使用 `SequenceFile.Sorter` 类中的 `sort()`方法和 `merge()`方法。它们比 MapReduce 更早出现，比 MapReduce 更底层(例如，为了实现并行，需要手动对数据进行分区)，所以对顺序文件进行排序合并时采用 MapReduce 是更佳的选择。

3. SequenceFile 的格式

顺序文件由文件头和随后的一条或多条记录组成(参见图 5-2)。顺序文件的前三个字节为 SEQ(顺序文件代码)，紧随其后的一个字节表示顺序文件的版本号。文件头还包括其他字段，例如键和值类的名称、数据压缩细节、用户定义的元数据以及同步标识。[①]如前所述，同步标识用于在读取文件时能够从任意位置开始识别记录边界。每个文件都有一个随机生成的同步标识，其值存储在文件头中。同步标识

① 这些字段的格式细节可参见 `SequenceFile` 的文档(*http://bit.ly/sequence_file_docs*)和源码。

位于顺序文件中的记录与记录之间。同步标识的额外存储开销要求小于 1%，所以没有必要在每条记录末尾添加该标识(特别是比较短的记录)。

图 5-2. 压缩前和压缩后的顺序文件的内部结构

记录的内部结构取决于是否启用压缩。如果已经启用压缩，则结构取决于是记录压缩还是数据块压缩。

如果没有启用压缩(默认情况)，那么每条记录则由记录长度(字节数)、键长度、键和值组成。长度字段为 4 字节长的整数，遵循 `java.io.DataOutput` 类中 `writeInt()` 方法的协定。为写入顺序文件的类定义 `Serialization` 类，通过它来实现键和值的序列化。

记录压缩格式与无压缩情况基本相同，只不过值是用文件头中定义的 codec 压缩的。注意，键没有被压缩。

如图 5-3 所示，块压缩(block compression)是指一次性压缩多条记录，因为它可以利用记录间的相似性进行压缩，所以相较于单条记录压缩方法，该方法的压缩效率更高。可以不断向数据块中压缩记录，直到块的字节数不小于 `io.seqfile.compress.blocksize` 属性中设置的字节数：默认为 1 MB。每一个新块的开始处都需要插入同步标识。数据块的格式如下：首先是一个指示数据块中字节数的字段；紧接着是 4 个压缩字段(键长度、键、值长度和值)。

图 5-3. 采用块压缩方式之后，顺序文件的内部结构

5.4.2 关于 MapFile

MapFile 是已经排过序的 SequenceFile，它有索引，所以可以按键查找。索引自身就是一个 SequenceFile，包含了 map 中的一小部分键（默认情况下，是每隔 128 个键）。由于索引能够加载进内存，因此可以提供对主数据文件的快速查找。主数据文件则是另一个 SequenceFile，包含了所有的 map 条目，这些条目都按照键顺序进行了排序。

MapFile 提供了一个用于读写的、与 SequenceFile 非常类似的接口。需要注意的是，当使用 MapFile.Writer 进行写操作时，map 条目必须顺序添加，否则会抛出 IOException 异常。

1. MapFile 的变种

Hadoop 在通用的键-值对 MapFile 接口上提供了一些变种。

- SetFile 是一个特殊的 MapFile，用于存储 Writable 键的集合。键必须按照排好的顺序添加。
- ArrayFile 也是一个 MapFile 变种，该变种中的键是一个整型，用于表示数组中元素的索引，而值是一个 Writable 值。
- BloomMapFile 也是一个 MapFile 变种，该变种提供了 get()方法的一个高性能实现，对稀疏文件特别有用。该实现使用一个动态的布隆过滤器来检测某个给定的键是否在 map 文件中。这个测试非常快，因为是在内存中完成的，但是该测试结果出现假阳性的概率大于零。仅当测试通过时(键存在)，常规的 get()方法才会被调用。

5.4.3 其他文件格式和面向列的格式

顺序文件和 map 文件是 Hadoop 中最早的、但并不是仅有的二进制文件格式，事实上，对于新项目而言，有更好的二进制文件格式可供选择。

Avro 数据文件(详见 12.3 节)在某些方面类似顺序文件，是面向大规模数据处理而设计的(紧凑且可切分)。但是 Avro 数据文件又是可移植的，它们可以跨越不同的编程语言使用。Avro 数据文件中存储的对象使用模式来描述，而不是像 `Writable` 对象的实现那样使用 Java 代码(例如顺序文件就是这样的情况，这样的弊端是过于以 Java 为中心)。Avro 数据文件被 Hadoop 生态系统的各组件广为支持，因此它们被默认为是对二进制格式的一种比较好的选择。

顺序文件、map 文件和 Avro 数据文件都是面向行的格式，意味着每一行的值在文件中是连续存储的。在面向列的格式中，文件中的行(或等价的，Hive 中的一张表)被分割成行的分片，然后每个分片以面向列的形式存储：首先存储每行第 1 列的值，然后是每行第 2 列的值，如此以往。该过程如图 5-4 所示。

图 5-4. 面向行的和面向列的存储

面向列的存储布局可以使一个查询跳过那些不必访问的列。让我们考虑一个只需要处理图 5-4 中表的第 2 列的查询。在像顺序文件这样面向行的存储中，即使是只需要读取第二列，整个数据行(存储在顺序文件的一条记录中)将被加载进内存。虽说“延迟反序列化”(lazy deserialization)策略通过只反序列化那些被访问的列字段能节省一些处理开销，但这仍然不能避免从磁盘上读入一个数据行所有字节而付出的开销。

如果使用面向列的存储，只需要把文件中第 2 列所对应的那部分(图中高亮部分)读入内存。一般来说，面向列的存储格式对于那些只访问表中一小部分列的查询比较有效。相反，面向行的存储格式适合同时处理一行中很多列的情况。

由于必须在内存中缓存行的分片、而不是单独的一行，因此面向列的存储格式需要更多的内存用于读写。并且，当出现写操作时(通过 flush 或 sync 操作)，这种缓存通常不太可能被控制，因此，面向列的格式不适合流的写操作，这是因为，如果 writer 处理失败的话，当前的文件无法恢复。另一方面，对于面向行的存储格式，如顺序文件和 Avro 数据文件，可以一直读取到 writer 失败后的最后的同步点。由于这个原因，Flume(详见第 14 章)使用了面向行的存储格式。

Hadoop 中的第一个面向列的文件格式是 Hive 的 *RCFile*(Record Columnar File)，它已经被 Hive 的 *ORCFile*(Optimized Record Columnar File)及 *Parquet* 取代(详见第 13 章)。*Parquet* 是一个基于 Google Dremel 的通用的面向列的文件格式，被 Hadoop 组件广为支持。Avro 也有一个面向列的文件格式，称为 *Trevni*。

第 II 部分
关于 MapReduce

第 6 章　MapReduce 应用开发
第 7 章　MapReduce 的工作机制
第 8 章　MapReduce 的类型与格式
第 9 章　MapReduce 的特性

第 6 章

MapReduce 应用开发

在第 2 章中，我们介绍了 MapReduce 模型。本章中，我们从实现层面介绍在 Hadoop 中开发 MapReduce 应用程序。

MapReduce 编程遵循一个特定的流程。首先写 map 函数和 reduce 函数，最好使用单元测试来确保函数的运行符合预期。然后，写一个驱动程序来运行作业，看这个驱动程序是否可以正确运行，可以先从本地 IDE 中用一个小的数据集来运行它。如果驱动程序不能正确运行，就用本地 IDE 调试器来找出问题根源。根据这些调试信息，可以通过扩展单元测试来覆盖这一测试用例，从而改进 mapper 或 reducer，使其能正确处理类似输入。

一旦程序按预期通过小型数据集的测试，就可以考虑把它放到集群上运行了。当运行程序对整个数据集进行测试的时候，可能会暴露更多的问题，这些问题可以像前面一样修复，即通过扩展测试用例及修改 mapper 或 reducer 函数的方式来应对新情况。在集群中调试程序很具有挑战性，我们来看一些常用的技术使其变得更简单一些。

程序可以正确运行之后，如果想进行一些优化调整，首先需要执行一些标准检查，借此加快 MapReduce 程序的运行速度，然后再做任务剖析(task profiling)。分布式程序的分析并不简单，Hadoop 提供了钩子(hook)来辅助这个分析过程。

然而在开始写 MapReduce 程序之前，仍然需要设置和配置开发环境。为此，我们需要先学习如何配置 Hadoop。

6.1 用于配置的 API

Hadoop 中的组件是通过 Hadoop 自己的配置 API 来配置的。一个 Configuration 类的实例(可以在 org.apache.hadoop.conf 包中找到)代表配置属性及其取值的一个集合。每个属性由一个 String 来命名，而值的类型可以是多种类型之一，包括 Java 基本类型(如 boolean、int、long 和 float)、其他有用的类型(如 String、Class 和 java.io.File)及 String 集合。

Configuration 从资源(即使用简单结构定义名值对的 XML 文件)中读取其属性值。参见范例 6-1。

范例 6-1. 一个简单的配置文件 configuration-1.xml

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>color</name>
    <value>yellow</value>
    <description>Color</description>
  </property>

  <property>
    <name>size</name>
    <value>10</value>
    <description>Size</description>
  </property>

  <property>
    <name>weight</name>
    <value>heavy</value>
    <final>true</final>
    <description>Weight</description>
  </property>

  <property>
    <name>size-weight</name>
    <value>${size},${weight}</value>
    <description>Size and weight</description>
  </property>
</configuration>
```

假定一个 Configuration 位于 *configuration-1.xml* 文件中，我们可以通过如下代码访问其属性：

```
Configuration conf = new Configuration();
conf.addResource("configuration-1.xml");
```

```
assertThat(conf.get("color"), is("yellow"));
assertThat(conf.getInt("size", 0), is(10));
assertThat(conf.get("breadth", "wide"), is("wide"));
```

有这样几点需要注意：XML 文件中不保存类型信息；取而代之的是属性在被读取的时候，可以被解释为指定的类型；此外，get()方法允许为 XML 文件中没有定义的属性指定默认值，正如这一代码中最后一行的 breadth 属性一样。

6.1.1 资源合并

使用多个资源文件来定义一个 Configuration 时，事情变得有趣了。在 Hadoop 中，这用于分离(*core-default.xml* 文件内部定义的)系统默认属性与(*core-site.xml* 文件中定义的)位置相关(site-specific)的覆盖属性。范例 6-2 中的文件定义了 size 属性和 weight 属性。

范例 6-2. 第二个配置文件 configuration-2.xml

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>size</name>
    <value>12</value>
  </property>

  <property>
    <name>weight</name>
    <value>light</value>
  </property>
</configuration>
```

资源文件按顺序添加到 Configuration：

```
Configuration conf = new Configuration();
conf.addResource("configuration-1.xml");
conf.addResource("configuration-2.xml");
```

后来添加到资源文件的属性会覆盖(override)之前定义的属性。所以，size 属性的取值来自于第二个配置文件 *configuration-2.xml*：

```
assertThat(conf.getInt("size", 0), is(12));
```

不过，被标记为 final 的属性不能被后面的定义所覆盖。在第一个配置文件中，weight 属性的 final 状态是 true，因此，第二个配置文件中的覆盖设置失败，weight 取值仍然是第一个配置文件中的 heavy：

```
assertThat(conf.get ("weight"), is("heavy"));
```

试图覆盖 final 属性通常意味着配置错误，所以最后会弹出警告消息来帮助进行故障诊断。一般来说，管理员将守护进程站点中的属性标记为 final，表明他们不希望用户在客户端的配置文件或作业提交参数(job submission parameter)中有任何改动。

6.1.2 变量扩展

配置属性可以用其他属性或系统属性进行定义。例如，在第一个配置文件中的 size-weight 属性可以定义为${size}和${weight}，而且这些属性是用配置文件中的值来扩展的：

```
assertThat(conf.get("size-weight"), is("12,heavy"));
```

系统属性的优先级高于资源文件中定义的属性：

```
System.setProperty("size", "14");
assertThat(conf.get("size-weight"), is("14,heavy"));
```

该特性特别适用于在命令行方式下用 JVM 参数-D*property=value* 来覆盖属性。

注意，虽然配置属性可以通过系统属性来定义，但除非系统属性使用配置属性重新定义，否则，它们是无法通过配置 API 进行访问的。因此：

```
System.setProperty("length", "2");
assertThat(conf.get("length"), is((String) null));
```

6.2 配置开发环境

首先新建一个项目，以便编译 MapReduce 程序并通过命令行或在自己的 IDE 中以本地(独立，standalone)模式运行它们。在范例 6-3 中的 Maven POM 项目对象模型(Project Object Model)说明了编译和测试 Map-Reduce 程序时需要的依赖项(dependency)。

范例 6-3. 编译和测试 MapReduce 应用的 Maven POM

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.hadoopbook</groupId>
  <artifactId>hadoop-book-mr-dev</artifactId>
  <version>4.0</version>
  <properties>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<hadoop.version>2.5.1</hadoop.version>
```

```
</properties>
<dependencies>
  <!-- Hadoop main client artifact -->
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-client</artifactId>
    <version>${hadoop.version}</version>
  </dependency>
  <!-- Unit test artifacts -->
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.mrunit</groupId>
    <artifactId>mrunit</artifactId>
    <version>1.1.0</version>
<classifier>hadoop2</classifier>
    <scope>test</scope>
</dependency>
<!-- Hadoop test artifact for running mini clusters -->
  <dependency>
    <groupId>org.apache.hadoop</groupId>
<artifactId>hadoop-minicluster</artifactId>
<version>${hadoop.version}</version>
<scope>test</scope>
</dependency>
  </dependencies>
  <build>
    <finalName>hadoop-examples</finalName>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.5</version>
        <configuration>
          <outputDirectory>${basedir}</outputDirectory>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

依赖关系是 POM 中有趣的一部分。(只要你使用此处定义的依赖关系，就可以直接使用其他的构建工具，例如 Gradle 或者 Ant with Ivy。)要想构建 MapReduce 作业，你只需要有 `hadoop-client` 依赖关系，它包含了和 HDFS 及 MapReduce 交互所需要的所有 Hadoop client-side 类。当运行单元测试时，我们要使用 junit 类；当写 MapReduce 测试用例时，我们使用 `mrunit` 类。`hadoop-minicluster` 库中包含了“mini-”集群，这有助于在一个单 JVM 中运行 Hadoop 集群进行测试。

很多 IDE 可以直接读 Maven POM，因此你只需要在包含 *pom.xml* 文件的目录中指向这些 Maven POM，就可以开始写代码。也可以使用 Maven 为 IDE 生成配置文件。例如，如下创建 Eclipse 配置文件以便将项目导入 Eclipse：

```
% mvn eclipse:eclipse -DdownloadSources=true -DdownloadJavadocs=true
```

6.2.1 管理配置

开发 Hadoop 应用时，经常需要在本地运行和集群运行之间进行切换。事实上，可能在几个集群上工作，也可能在本地“伪分布式”集群上测试。伪分布式集群是其守护进程运行在本机的集群，这种运行模式的配置请参见附录 A。

应对这些变化的一种方法是使 Hadoop 配置文件包含每个集群的连接设置，并且在运行 Hadoop 应用或工具时指定使用哪一个连接设置。最好的做法是，把这些文件放在 Hadoop 安装目录树之外，以便于轻松地在 Hadoop 不同版本之间进行切换，从而避免重复或丢失设置信息。

为了方便本书的介绍，我们假设目录 *conf* 包含三个配置文件：*hadoop-local.xml*，*hadoop-localhost.xml* 和 *hadoop-cluster.xml*(这些文件在本书的范例代码里)。注意，文件名没有特殊要求，这样命名只是为了方便打包配置的设置。(将此与附录 A 的表 A-1 进行对比，后者存放的是对应服务器端的配置信息。)

针对默认的文件系统和用于运行 MapReduce 作业的本地(指在 JVM 中的)框架，*hadoop-local.xml* 包含默认的 Hadoop 配置：

```
<?xml version="1.0"?>
<configuration>

  <property>
    <name>fs.defaultFS</name>
    <value>file:///</value>
  </property>

  <property>
```

```
    <name>mapreduce.framework.name</name>
    <value>local</value>
  </property>
</configuration>
```

hadoop-localhost.xml 文件中的设置指向本地主机上运行的 namenode 和 YARN 资源管理器：

```
<?xml version="1.0"?>
<configuration>

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

  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>

  <property>
    <name>yarn.resourcemanager.address</name>
    <value>localhost:8032</value>
  </property>

</configuration>
```

最后，*hadoop-cluster.xml* 文件包含集群上 namenode 和 YARN 资源管理器地址的详细信息(事实上，我们会以集群的名称来命名这个文件，而不是这里显示的那样用 cluster 泛指)：

```
<?xml version="1.0"?>
<configuration>

  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://namenode/</value>
  </property>

  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>

  <property>
    <name>yarn.resourcemanager.address</name>
    <value>resourcemanager:8032</value>
  </property>

</configuration>
```

还可以根据需要为这些文件添加其他配置信息。

设置用户标识

在 HDFS 中，可以通过在客户端系统上运行 `whoami` 命令来确定 Hadoop 用户标识(identity)。类似，组名(group name)来自 `groups` 命令的输出。

如果 Hadoop 用户标识不同于客户机上的用户账号，可以通过设置 `HADOOP_USER_NAME` 环境变量来显式设定 Hadoop 用户名。还可以通过 `hadoop.user.group.static.mapping.overrides` 配置属性来覆盖用户组映射关系。例如，`dr.who=;preston=directors, inventors`，表示用户 `dr.who` 不在组中，而用户 `preston` 在 `directors` 和 `inventors` 组中。

可以通过设置 `hadoop.http.staticuser.user` 属性来设置 Hadoop 网络接口运行时的用户标识。在默认情况下，这个用户标识是 `dr.who`，但不是超级用户，因此，不能通过该网络接口访问系统文件。

注意，在默认情况下，系统没有认证机制。10.4 节介绍了如何在 Hadoop 中使用 Kerberos 认证。

有了这些设置，便可以轻松通过`-conf` 命令行开关来使用各种配置。例如，下面的命令显示了一个在伪分布式模式下运行于本地主机上的 HDFS 服务器上的目录列表：

```
% hadoop fs -conf conf/hadoop-localhost.xml -ls .
Found 2 items
drwxr-xr-x - tom supergroup 0 2014-09-08 10:19 input
drwxr-xr-x - tom supergroup 0 2014-09-08 10:19 output
```

如果省略`-conf` 选项，可以从`$HADOOP_HOME` 的 *etc/hadoop* 子目录中找到 Hadoop 的配置信息。或者如果已经设置了 `HADOOP_CONF_DIR`，Hadoop 的配置信息将从那个位置读取。

这里介绍另一种管理配置设置的方法。将 *etc/hadoop* 目录从 Hadoop 的安装位置拷贝至另一个位置，将*-*site.xml* 配置文件也放于该位置(文件中的各项设置应正确)，再将 `HADOOP_CONF_DIR` 环境变量设置为指向该位置。该方法的主要优点是，不需要每个命令中都指定`-conf`。并且，由于 `HADOOP_CONF_DIR` 路径下有所有配置文件的拷贝，因此，对文件的修改可以与 Hadoop XML 配置文件隔离开来(例如，*log4j.properties*)。详细介绍可以参见 10.3 节。

Hadoop 自带的工具支持-conf 选项，也可以直接用程序(例如运行 MapReduce 作业的程序)通过使用 Tool 接口来支持-conf 选项。

6.2.2 辅助类 GenericOptionsParser，Tool 和 ToolRunner

为了简化命令行方式运行作业，Hadoop 自带了一些辅助类。GenericOptionsParser 是一个类，用来解释常用的 Hadoop 命令行选项，并根据需要，为 Configuration 对象设置相应的取值。通常不直接使用 GenericOptionsParser。更方便的方式是实现 Tool 接口，通过 ToolRunner 来运行应用程序。ToolRunner 内部调用 GenericOptionsParser：

```
public interface Tool extends Configurable {
  int run(String [] args) throws Exception;
}
```

范例 6-4 给出了一个非常简单的 Tool 的实现，用来打印 Tool 的 Configuration 对象中所有属性的键-值对。

范例 6-4. Tool 实现用于打印一个 Configuration 对象的属性的范例

```
public class ConfigurationPrinter extends Configured implements Tool {

  static {
    Configuration.addDefaultResource("hdfs-default.xml");
    Configuration.addDefaultResource("hdfs-site.xml");
    Configuration.addDefaultResource("yarn-default.xml");
    Configuration.addDefaultResource("yarn-site.xml");
    Configuration.addDefaultResource("mapred-default.xml");
    Configuration.addDefaultResource("mapred-site.xml");
  }

  @Override
  public int run(String[] args) throws Exception {
    Configuration conf = getConf();
    for (Entry<String, String> entry: conf) {
      System.out.printf("%s=%s\n", entry.getKey(), entry.getValue());
    }
      return 0;
    }

    public static void main(String[] args) throws Exception {
      int exitCode = ToolRunner.run(new ConfigurationPrinter(), args);
      System.exit(exitCode);
    }
  }
```

我们把 ConfigurationPrinter 作为 Configured 的一个子类，Configured 是 Configurable 接口的一个实现。Tool 的所有实现都需要实现 Configurable(因为 Tool 继承于 Configurable)，Configured 子类通常是一种最简单的实现方式。run()方法通过 Configurable 的 getConf()方法获取 Configuration，然后重复

执行，将每个属性打印到标准输出。

静态代码部分确保核心配置外的 HDFS、YARN 和 MapReduce 配置能够被获取(因为 Configuration 已经获取了核心配置)。

可以设置哪些属性？

作为一个有用的工具，ConfigurationPrinter 可用于了解在环境中某个属性是如何进行设置的。对于运行中的守护进程，例如 namenode，可以通过查看其网络服务器上的*/conf* 页面来了解配置情况。

你也可以在 Hadoop 安装路径的 share/doc 目录中，查看所有公共属性的默认设置，相关文件包括 core-default.xml，hdfs-default.xml，yarn-default.xml 和 mapred-default.xml 这几个文件。每个属性都有用来解释属性作用和取值范围的描述。

默认的配置文档可通过以下网址所链接的页面找到，http://hadoop.apache.org/docs/current/(在导航树中寻找“Configuration”标题)。通过用<version>替换前述 URL 中的 current，可以找到一个特定 Hadoop 发行版本的默认配置，例如，*http://hadoop.apache.org/docs/ r2.5.0/*。

注意，在客户端配置中设置某些属性，将不会产生影响。例如，如果在作业提交时想通过设置 yarn.nodemanager.resource.memory-mb 来改变运行作业的节点管理器能够得到的内存数量，结果会让你失望的，因为这个属性只能在节点管理器的 *yarn-site.xml* 文件中进行设置。一般情况下，我们可以通过属性名的组成部分来获知该属性应该在哪里进行设置。由于 yarn.nodemanager.resource.memory-mb 以 yarn.nodemanager 开头，因此，我们知道它只能为节点管理器守护进程进行设置。但是，这不是硬性的，在有些情况下，我们需要进行尝试，甚至去阅读源代码。

在 Hadoop 2 中，为了让命名结构看着更常规一些，对配置属性名称进行了修改。例如，namenode 相关的 HDFS 属性名称都改为带一个 dfs.namenode 前缀，这样原先的 dfs.name.dir 现在称为 dfs.namenode. name.dir。类似的，MapReduce 属性用 mapreduce 前缀代替了较早的 mapred 前缀，因此，原先的 mapred.job.name 现在称为 mapreduce.job.name。

本书使用了新属性名，以避免弃用警告。然而，旧属性名仍然有效，通常会在旧版本的文档被引用。可以在 Hadoop 网站上找到一张关于弃用属性名及替代名的列表*(http://bit.ly/deprecated_props)*。

本书讨论了 Hadoop 很多重要的配置属性。

ConfigurationPrinter 的 main()方法没有直接调用自身的 run()方法，而是调用 ToolRunner 的静态 run()方法，该方法负责在调用自身的 run()方法之前，为 Tool 建立一个 Configuration 对象。ToolRunner 还使用了 GenericOptionsParser 来获取在命令行方式中指定所有标准的选项，然后，在 Configuration 实例上进行设置。运行下列代码，可以看到在 *conf/hadoop-localhost.xml* 中设置的属性。

```
% mvn compile
% export HADOOP_CLASSPATH: target/classes
% hadoop ConfigurationPrinter -conf conf/hadoop-localhost.xml \
  | grep yarn.resourcemanager.address=
yarn.resourcemanager.address=localhost:8032
```

GenericOptionsParser 也允许设置个别属性。例如：

```
% hadoop ConfigurationPrinter -D color=yellow | grep color
color=yellow
```

-D 选项用于将键 color 的配置属性值设置为 yellow。设置为-D 的选项优先级要高于配置文件里的其他属性。这一点很有用：可以把默认属性放入配置文件中，然后再在需要时用 -D 选项来覆盖。一个常见的例子是，通过 -D mapreduce.job.reduces=*n* 来设置 MapReduce 作业中 reducer 的数量。这样会覆盖集群上或客户端配置属性文件中设置的 reducer 数量。

GenericOptionsParser 和 ToolRunner 支持的其他选项可以参见表 6-1。Hadoop 用于配置的更多 API 可以在 6.1 节中找到。

用-D *property=value* 选项将 Hadoop 属性设为 GenericOptionsParser (和 ToolRunner)，不同于用 Java 命令-D*property=value* 选项对 JVM 系统属性进行设置。JVM 系统属性的语法不允许 D 和属性名之间有任何空格，而 GenericOptionsParser 则允许用空格。

JVM 系统属性来自于 java.lang.System 类，而 Hadoop 属性只能从 Configuration 对象中获取。所以，即使已经设置了系统属性 color(通过 HADOOP_OPTS)，下面的命令行也不会有任何输出，因为 ConfigurationPrinter 没有使用 System 类：

```
% HADOOP_OPTS='-Dcolor=yellow' \
hadoop ConfigurationPrinter | grep color
```

如果希望通过系统属性进行配置，则需要在配置文件中反映相关的系统属性。具体讨论请参见 6.1.2 节。

表 6-1. GenericOptionsParser 选项和 ToolRunner 选项

选项名称	描述
-D property=value	将指定值赋值给某个 Hadoop 配置属性。覆盖配置文件里的默认属性或站点属性，或通过-conf 选项设置的任何属性
-conf filename ...	将指定文件添加到配置的资源列表中。这是设置站点属性或同时设置一组属性的简便方法
-fs uri	用指定的URI设置默认文件系统。这是-D fs.default.FS=uri的快捷方式
-jt host:port	用指定主机和端口设置 YARN 资源管理器。(在 Hadoop 1 中，该选项用于设置 jobtracker 地址，因此沿用了该名称)这是 -D yarn.resourcemanager.address= host:port的快捷方式
-files file1, file2,…	从本地文件系统(或任何指定模式的文件系统)中复制指定文件到 MapReduce 所用的共享文件系统(通常是 HDFS)，确保在任务工作目录的 MapReduce 程序可以访问这些文件，要想进一步了解复制文件到集群机器的分布式缓存机制，请参见 9.4.2 节
-archives archive1, archive2,…	从本地文件系统(或任何指定模式的文件系统)复制指定存档到 MapReduce 所用的共享文件系统(通常是 HDFS)，打开存档文件，确保任务工作目录的 MapReduce 程序可以访问这些存档
-libjars jar1, jar2,…	从本地文件系统(或任何指定模式的文件系统)复制指定 JAR 文件到被 MapReduce 使用的共享文件系统(通常是 HDFS)，把它们加入 MapReduce 任务的类路径中。这个选项适用于传输作业需要的 JAR 文件

6.3 用 MRUnit 来写单元测试

在 MapReduce 中，map 函数和 reduce 函数的独立测试非常方便，这是由函数风格决定的。MRUnit(*http://mrunit.apache.org/*)是一个测试库，它便于将已知的输入传递给 mapper 或者检查 reducer 的输出是否符合预期。MRUnit 与标准的测试执行框架(如 JUnit)一起使用，因此可以在正常的开发环境中运行 MapReduce 作业的测试。例如，这里描述的所有测试都可根据 6.2 节介绍的指令在 IDE 中运行。

6.3.1 关于 Mapper

范例 6-5 是一个 mapper 的测试。

范例 6-5. MaxTemperatureMapper 的单元测试

```
import java.io.IOException;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mrunit.mapreduce.MapDriver;
import org.junit.*;
```

```
public class MaxTemperatureMapperTest {

@Test
public void processesValidRecord() throws IOException,InterruptedException {
    Text value = new Text("0043011990999991950051518004+68750+023550FM-12+0382" +
                                // Year ^^^^
    "99999V0203201N00261220001CN9999999N9-00111+99999999999");
                                // Temperature ^^^^^
  new MapDriver<LongWritable, Text, Text, IntWritable>()
    .withMapper(new MaxTemperatureMapper())
    .withInput(new LongWritable(0), value)
    .withOutput(new Text("1950"), new IntWritable(-11))
    .runTest();
  }
}
```

测试很简单：传递一个天气记录作为 mapper 的输入，然后检查输出是否是读入的年份和气温。

由于测试的是 mapper，所以可以使用 MRUnit 的 MapDriver.在调用 runTest()方法执行这个测试之前，在 test(MaxTemperatureMapper)下配置 mapper、输入 key 和值、期望的输出 key(1950，表示年份的 Text 对象)和期望的输出值(-1.1℃，表示温度的 IntWritable)。如果 mapper 没有输出期望的值，MRUnit 测试失败。注意，由于 mapper 忽略输入 key，因此，输入 key 可以设置为任何值。

在测试驱动的方式下，范例 6-6 创建了一个能够通过测试的 Mapper 实现。由于本章要进行类的扩展，所以每个类被放在包含版本信息的不同包中。例如，v1.MaxTemperatureMapper 是 MaxTemperatureMapper 的第一个版本。当然，不重新打包实际上也可以对类进行扩展。

范例 6-6. 通过了 MaxTemperatureMapper 测试的第一个版本 Mapper 函数

```
public class MaxTemperatureMapper extends Mapper<LongWritable, Text, Text,
IntWritable> {

@Override
public void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {

  String line = value.toString();
  String year = line.substring(15, 19);
  int airTemperature = Integer.parseInt(line.substring(87, 92));
  context.write(new Text(year), new IntWritable(airTemperature));
  }
}
```

这是一个非常简单的实现，从行中抽出年份和气温，并将它们写到 Context 中。现在，让我们增加一个缺失值的测试，该值在原始数据中表示气温+9999：

```
@Test
public void ignoresMissingTemperatureRecord() throws IOException {
InterruptedException {
Text value = new Text("0043011990999991950051518004+68750+023550FM-12+0382" +
             // Year ^^^^
    "99999V0203201N00261220001CN9999999N9+99991+99999999999");
            // Temperature ^^^^^

 new MapDriver<LongWritable, Text, Text, IntWritable>()
    .withMapper(new MaxTemperatureMapper())
    .withInput(new LongWritable(0), value)
    .runTest();
}
```

根据 withOutput()被调用的次数，MapDriver 能用来检查 0、1 或多个输出记录。在这个测试中由于缺失温度的记录已经被过滤掉，该测试保证对于这种特定的输入值不产生任何输出。

由于没有考虑到+9999 这样一种特殊情况，新测试以失败告终。与其在 mapper 中加入更多的逻辑考虑，还不如给出一个解析类来封装解析逻辑更有意义。详情可以参见范例 6-7。

范例 6-7. 该类解析 NCDC 格式的气温记录

```
public class NcdcRecordParser {
  private static final int MISSING_TEMPERATURE = 9999;

  private String year;
  private int airTemperature;
  private String quality;

  public void parse(String record) {
    year = record.substring(15, 19);
    String airTemperatureString;
    // Remove leading plus sign as parseInt doesn't like them (pre-Java 7)
    if (record.charAt(87) == '+') {
        airTemperatureString = record.substring(88, 92);
    } else {
        airTemperatureString = record.substring(87, 92);
    }
    airTemperature = Integer.parseInt(airTemperatureString);
    quality = record.substring(92, 93);
  }
  public void parse(Text record) {
    parse(record.toString());
}
```

```
public boolean isValidTemperature() {
    return airTemperature != MISSING_TEMPERATURE &&
    quality.matches("[01459]");
}

public String getYear() {
    return year;
  }

public int getAirTemperature() {
    return airTemperature;
  }
}
```

最终的 mapper(第二个版本)相当简单(见范例 6-8)。它只需要调用解析类的 parse()方法，就能对输入行中感兴趣的字段进行解析，用 isValidTemperature()查询方法可以检查是否是合法气温，如果是，则使用解析类的获取方法得到年份和气温值。需要注意是，我们在 isValidTemperature()中检查是否有遗漏气温值的同时，也对质量状态字段进行检查，以便过滤掉不准确的气温读取值。

创建解析类的另一个好处是：相似作业的 mapper 不需要重写代码。并且，对于更多的有针对性的测试而言，该方法也提供了一个机会，即可以直接针对解析类编写单元测试。

范例 6-8. 这个 mapper 使用 utility 类来解析记录

```
public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

  private NcdcRecordParser parser = new NcdcRecordParser();

  @Override
  public void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {

    parser.parse(value);
    if (parser.isValidTemperature()) {
    context.write(new Text(parser.getYear()),
      new IntWritable(parser.getAirTemperature()));
    }
  }
}
```

这个对 mapper 的测试通过后，我们接下来写 reducer。

6.3.2 关于 Reducer

reducer 必须找出指定键的最大值。这是针对此特性的一个简单的测试,其中使用了一个 ReduceDriver。

```
@Test
public void returnsMaximumIntegerInValues() throws IOException,
    InterruptedException {

new ReduceDriver<Text, IntWritable, Text, IntWritable>()
    .withReducer(new MaxTemperatureReducer())
    .withInputKey(new Text("1950"))
       Arrays.asList(new IntWritable(10), new IntWritable(5)))
    .withOutput(new Text("1950"), new IntWritable(10))
    .runTest();
}
```

我们对一些 IntWritable 值构建一个迭代器来验证 MaxTemperatureReducer 能找到最大值。范例 6-9 里的代码是一个通过测试的 MaxTemperatureReducer 的实现。

范例 6-9. 用来计算最高气温的 reducer

```
public class MaxTemperatureReducer
  extends Reducer<Text, IntWritable, Text, IntWritable> {

  @Override
  public void reduce(Text key, Iterable<IntWritable> values,
      Context context)
      throws IOException, InterruptedException {

    int maxValue = Integer.MIN_VALUE;
    for (IntWritable value : values) {
      maxValue = Math.max(maxValue, value.get());
    }
    context.write(key, new IntWritable(maxValue));
  }
}
```

6.4 本地运行测试数据

现在 mapper 和 reducer 已经能够在受控的输入上进行工作了，下一步是写一个作业驱动程序(job driver)，然后在开发机器上使用测试数据运行它。

6.4.1 在本地作业运行器上运行作业

通过使用前面介绍的 Tool 接口，可以轻松写一个 MapReducer 作业的驱动程序，

用它来计算按照年度查找最高气温，参见范例 6-10 的 MaxTemperatureDriver。

范例 6-10. 查找最高气温

```
public class MaxTemperatureDriver extends Configured implements Tool {

  @Override
  public int run(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.printf("Usage: %s [generic options] <input> <output>\n",
        getClass().getSimpleName());
      ToolRunner.printGenericCommandUsage(System.err);
      return -1;
    }

    Job job = new Job(getConf(), "Max temperature");
    job.setJarByClass(getClass());

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
    job.setReducerClass(MaxTemperatureReducer.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    return job.waitForCompletion(true) ? 0 : 1;
    }
    public static void main(String[] args) throws Exception {
      int exitCode = ToolRunner.run(new MaxTemperatureDriver(), args);
      System.exit(exitCode);
  }
}
```

MaxTemperatureDriver 实现了 Tool 接口，所以，我们能够设置 GenericOptionsParser 支持的选项。run()方法根据工具的配置创建一个 Job 对象来启动一个作业。在所有可能的作业配置参数中，可以设置输入和输出文件路径，mapper、reducer 和 combiner 以及输出类型(输入类型由输入格式决定，默认为 TextInputFormat，包括 Long Writable 键和 Text 值)。为作业设置一个名称(Max temperature)也是很好的做法，这样可以在执行过程中或作业完成后方便地从作业列表中查找作业。默认情况下，作业名称是 JAR 文件名，通常情况下没有特殊的描述。

现在我们可以在一些本地文件上运行这个应用。Hadoop 有一个本地作业运行器(job runner)，它是在 MapReduce 执行引擎运行单个 JVM 上的 MapReduce 作业的简化版本。它是为测试而设计的，在 IDE 中使用起来非常方便，因为我们可以在

调试器中单步运行 mapper 和 reducer 代码。

如果 mapreduce.framework.name 被设置为 local， 则使用本地作业运行器，这也是默认情况[①]。

可以在命令行方式下输入如下命令来运行驱动程序：

```
% mvn compile
% export HADOOP_CLASSPATH=target/classes/
% hadoop v2.MaxTemperatureDriver -conf conf/hadoop-local.xml \
  input/ncdc/micro output
```

类似地，可以使用 GenericOptionsParser 提供的-fs 和-jt 选项：

```
% hadoop v2.MaxTemperatureDriver -fs file:/// -jt local \
  input/ncdc/micro output
```

这条指令使用本地 *input/ncdc/micro* 目录作为输入来执行 MaxTemperatureDriver，产生的输出存放在本地 *output* 目录中。注意：虽然我们设置了-fs，可以使用本地文件系统(file:///)，但本地作业运行器实际上可以在包括 HDFS 在内的任何文件系统上正常工作(如果 HDFS 里有一些文件，可以马上进行尝试)。

可以如下检查本地文件系统上的输出：

```
% cat output/part-r-00000
1949 111
1950 22
```

6.4.2 测试驱动程序

除了灵活的配置选项可以使应用程序实现 Tool，还可以插入任意 Configuration 来增加可测试性。可以利用这点来编写测试程序，它将利用本地作业运行器在已知的输入数据上运行作业，借此来检查输出是否满足预期。

要实现这个目标，有两种方法。第一种方法是使用本地作业运行器，在本地文件系统的测试文件上运行作业。范例 6-11 的代码给出了一种思路。

范例 6-11. 这个 MaxTemperatureDriver 测试使用了一个正在运行的本地作业运行器

```
@Test
    public void test() throws Exception {
```

① 在 Hadoop 1 中，mapred.job.tracker 的值决定了执行的方式：值为 local 时，表示本地作业运行器；值为冒号分隔开的主机端口对(*host:port*)时，表示一个 jobtracker 地址。

```
    Configuration conf = new Configuration();
    conf.set("fs.defaultFS", "file:///");
    conf.set("mapreduce.framework.name", "local");
    conf.setInt("mapreduce.task.io.sort.mb", 1);

    Path input = new Path("input/ncdc/micro");
    Path output = new Path("output");

    FileSystem fs = FileSystem.getLocal(conf);
    fs.delete(output, true); // delete old output

    MaxTemperatureDriver driver = new MaxTemperatureDriver();
    driver.setConf(conf);

    int exitCode = driver.run(new String[] {
        input.toString(), output.toString() });
    assertThat(exitCode, is(0));

    checkOutput(conf, output);
}
```

测试代码明确设置了 fs.defaultFS 和 mapreduce.framework.name 所以，它使用的是本地文件系统和本地作业运行器。随后，通过其 Tool 接口在少数已知数据上运行 MaxTemperatureDriver。最后，checkOutput()方法调用以逐行对比实际输出与预期输出。

测试驱动程序的第二种方法是使用一个 mini 集群来运行它。Hadoop 有一组测试类，名为 MiniDFSCluster、MiniMRCluster 和 MiniYARNCluster，它以程序方式创建正在运行的集群。不同于本地作业运行器，它们允许在整个 HDFS、MapReduce 和 YARN 机器上运行测试。注意，mini 集群上的节点管理器启动不同的 JVM 来运行任务，这会使调试更困难。

也可以从命令行运行一个 mini 集群，如下所示：

```
% hadoop jar \
$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-*\
-tests.jar minicluster
```

mini 集群广泛应用于 Hadoop 自带的自动测试包中，但也可以用于测试用户代码。Hadoop 的 ClusterMapReduceTestCase 抽象类提供了一个编写此类测试的基础，它的 setUp()和 tearDown()方法可处理启动和停止运行中的 HDFS 和 YARN 集群的细节，同时产生一个可以配置为一起工作的 Configuration 对象。子类只需要得到 HDFS 中的数据(可能从本地文件中复制得到)，运行 MapReduce 作业，然后确认输出是否满足要求。参见本书示例代码中的

MaxTemperatureDriverMiniTest 类。

这样的测试是回归测试，是一个非常有用的输入边界用例和相应的期望结果的资源库。随着测试用例的增加，简单将其加入输入文件，然后更新相应的输出文件即可。

6.5 在集群上运行

目前，程序已经可以在少量测试数据上正确运行，下面可以准备在 Hadoop 集群的完整数据集上运行了。第 10 章将介绍如何建立完全分布的集群，同时，该章中的方法也可以用在伪分布集群上。

6.5.1 打包作业

本地作业运行器使用单 JVM 运行一个作业，只要作业需要的所有类都在类路径(classpath)上，那么作业就可以正常执行。

在分布式的环境中，情况稍微复杂一些。开始的时候作业的类必须打包成一个作业 JAR 文件并发送给集群。Hadoop 通过搜索驱动程序的类路径自动找到该作业 JAR 文件，该类路径包含 JobConf 或 Job 上的 setJarByClass()方法中设置的类。另一种方法，如果你想通过文件路径设置一个指定的 JAR 文件，可以使用 setJar()方法。JAR 文件路径可以是本地的，也可以是一个 HDFS 文件路径。

通过使用像 Ant 或 Maven 的构建工具可以方便地创建作业的 JAR 文件。当给定范例 6-3 中所示的 POM 时，下面的 Maven 命令将在包含所有已编译的类的工程目录中创建一个名为 *hadoop-examples.jar* 的 JAR 文件：

```
% mvn package -DskipTests
```

如果每个 JAR 文件都有一个作业，可以在 JAR 文件的 manifest 中指定要运行的主类。如果主类不在 manifest 中，则必须在命令行指定。

任何有依赖关系的 JAR 文件应该打包到作业的 JAR 文件的 *lib* 子目录中。当然也有其他的方法将依赖包含进来，这我们稍后会讨论。类似地，资源文件也可以打包进一个 *classes* 子目录。这与 Java *Web application archive* 或 WAR 文件类似，只不过 JAR 文件是放在 WAR 文件的 *WEB-INF/lib* 子目录下，而类则是放在 WAR 文件的 *WEB-INF/classes* 子目录中。

1. 客户端的类路径

由 hadoop jar <*jar*>设置的用户客户端类路径包括以下几个组成部分：

- 作业的 JAR 文件
- 作业 JAR 文件的 *lib* 目录中的所有 JAR 文件以及 *classes* 目录(如果定义)
- HADOOP_CLASSPH 定义的类路径(如果已经设置)

顺便说一下，这解释了如果你在没有作业 JAR(hadoop *CLASSNAME*)情况下使用本地作业运行器时，为什么必须设置 HADOOP_CLASSPATH 来指明依赖类和库。

2. 任务的类路径

在集群上(包括伪分布式模式)，map 和 reduce 任务在各自的 JVM 上运行，它们的类路径不受 HADOOP_CLASSPATH 控制。HADOOP_CLASSPATH 是一项客户端设置，并只针对驱动程序的 JVM 的类路径进行设置。

反之，用户任务的类路径有以下几个部分组成：

- 作业的 JAR 文件
- 作业 JAR 文件的 lib 目录中包含的所有 JAR 文件以及 *classes* 目录(如果存在的话)
- 使用 -libjars 选项(参见表 6-1)或 DistributedCache 的 addFileToClassPath()方法(老版本的 API)或 Job(新版本的 API)添加到分布式缓存的所有文件

3. 打包依赖

给定这些不同的方法来控制客户端和类路径上的内容，也有相应的操作处理作业的库依赖：

- 将库解包和重新打包进作业 JAR
- 将作业 JAR 的 *lib* 目录中的库打包
- 保持库与作业 JAR 分开，并且通过 HADOOP_CLASSPATH 将它们添加到客户端的类路径，通过-libjars 将它们添加到任务的类路径

从创建的角度来看，最后使用分布式缓存的选项是最简单的，因为依赖不需要在作业的 JAR 中重新创建。同时，使用分布式缓存意味着在集群上更少的 JAR 文件转移，因为文件可能缓存在任务间的一个节点上了。详情可参见 9.4.2 节。

4. 任务类路径的优先权

用户的 JAR 文件被添加到客户端类路径和任务类路径的最后，如果 Hadoop 使用的库版本和你的代码使用的不同或不相容，在某些情况下可能会引发和 Hadoop 内置库的依赖冲突。有时需要控制任务类路径的次序，这样你的类能够被先提取出来。在客户端，可以通过设置环境变量 `HADOOP_USER_CLASSPATH_FIRST` 为 `true` 强制使 Hadoop 将用户的类路径优先放到搜索顺序中。对于任务的类路径，你可以将 `mapreduce.job.user.classpath.first` 设为 `true`。注意，设置这些选项就改变了针对 Hadoop 框架依赖的类(但仅仅对你的作业而言)，这可能会引起作业的提交失败或者任务失败，因此请谨慎使用这些选项。

6.5.2 启动作业

为了启动作业，我们需要运行驱动程序，使用`-conf` 选项来指定想要运行作业的集群(同样，也可以使用`-fs` 和`-jt` 选项)：

```
% unset HADOOP_CLASSPATH
% hadoop jar hadoop-examples.jar V2.MaxTemperatureDriver \
  -conf conf/hadoop-cluster.xml input/ncdc/all max-temp
```

我们不设置 `HADOOP_CLASSPATH` 环境变量是因为对于该作业没有任何第三方依赖。如果它被设置为`/target/classes/`(本章前面的内容),那么 Hadoop 将找不到作业 JAR，Hadoop 会从 *target/classes* 而不是从 JAR 装载 `MaxTempratureDriver` 类，从而导致作业失败。

`Job` 上的 `waitForCompletion()`方法启动作业并检查进展情况。如果有任何变化，就输出一行 map 和 reduce 进度总结。输入如下(为了清楚起见，有些行特意删除了)：

```
14/09/12 06:38:11 INFO input.FileInputFormat: Total input paths to process : 101
14/09/12 06:38:11 INFO impl.YarnClientImpl: Submitted application
application_1410450250506_0003
14/09/12 06:38:12 INFO mapreduce.Job: Running job: job_1410450250506_0003
14/09/12 06:38:26 INFO mapreduce.Job: map 0% reduce 0%
...
14/09/12 06:45:24 INFO mapreduce.Job: map 100% reduce 100%
14/09/12 06:45:24 INFO mapreduce.Job: Job job_1410450250506_0003 completed
successfully
14/09/12 06:45:24 INFO mapreduce.Job: Counters: 49
   File System Counters
       FILE: Number of bytes read=93995
```

```
        FILE: Number of bytes written=10273563
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=33485855415
        HDFS: Number of bytes written=904
        HDFS: Number of read operations=327
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=16
    Job Counters
        Launched map tasks=101
        Launched reduce tasks=8
        Data-local map tasks=101
        Total time spent by all maps in occupied slots (ms)=5954495
        Total time spent by all reduces in occupied slots (ms)=74934
        Total time spent by all map tasks (ms)=5954495
        Total time spent by all reduce tasks (ms)=74934
        Total vcore-seconds taken by all map tasks=5954495
        Total vcore-seconds taken by all reduce tasks=74934
        Total megabyte-seconds taken by all map tasks=6097402880
        Total megabyte-seconds taken by all reduce tasks=76732416
    Map-Reduce Framework
        Map input records=1209901509
        Map output records=1143764653
        Map output bytes=10293881877
        Map output materialized bytes=14193
        Input split bytes=14140
        Combine input records=1143764772
        Combine output records=234
        Reduce input groups=100
        Reduce shuffle bytes=14193
        Reduce input records=115
        Reduce output records=100
        Spilled Records=379
        Shuffled Maps =808
        Failed Shuffles=0
        Merged Map outputs=808
        GC time elapsed (ms)=101080
        CPU time spent (ms)=5113180
        Physical memory (bytes) snapshot=60509106176
        Virtual memory (bytes) snapshot=167657209856
        Total committed heap usage (bytes)=68220878848
    Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
    File Input Format Counters
        Bytes Read=33485841275
```

```
File Output Format Counters
    Bytes Written=90
```

输出包含很多有用的信息。在作业开始之前，打印作业 ID；如果需要在日志文件中或通过 mapred job 命令查询某个作业，必须要有 ID 信息。作业完成后，统计信息(例如计数器)被打印出来。这对于确认作业是否完成是很有用的。例如，对于这个作业，大约分析 12 亿条记录(“Map input records”)，从 HDFS 读取了大约 34 GB 压缩文件(“HDFS:Number of bytes”)。输入数据被分成 101 个大小合适的 gzipped 文件，因此即使不能划分数据也没有问题。

关于计数器的更多介绍，可以参见 9.1.1 节。

作业、任务和任务尝试 ID

Hadoop 2 中，MapReduce 作业 ID 由 YARN 资源管理器创建的 YARN 应用 ID 生成。一个应用 ID 的格式包含两部分：资源管理器(不是应用)开始时间和唯一标识此应用的由资源管理器维护的增量计数器。例如：ID 为 application_1410450250506_0003 的应用是资源管理器运行的第三个应用(0003，应用 ID 从 1 开始计数)，时间戳 1410450250506 表示资源管理器开始时间。计数器的数字前面由 0 开始，以便于 ID 在目录列表中进行排序。然而，计数器达到 10000 时，不能重新设置，会导致应用 ID 更长(这些 ID 就不能很好地排序了)。

将应用 ID 的 application 前缀替换为 job 前缀即可得到相应的作业 ID，如 job_1410450250506_0003。

任务属于作业，任务 ID 是这样形成的，将作业 ID 的 job 前缀替换为 task 前缀，然后加上一个后缀表示是作业里的哪个任务。例如：task_1410450250506_0003_m_000003 表示 ID 为 job_1410450250506_0003 的作业的第 4 个 map 任务(000003，任务 ID 从 0 开始计数)。作业的任务 ID 在作业初始化时产生，因此，任务 ID 的顺序不必是任务执行的顺序。

由于失败(参见 7.2 节)或推测执行(参见 7.4.2 节)，任务可以执行多次，所以，为了标识任务执行的不同实例，任务尝试(task attempt)都会被指定一个唯一的 ID。例如：attempt_1410450250506_0003_m_000003_0 表示正在运行的 task_1410450250506_0003_m_000003 任务的第一个尝试 (0，任务尝试 ID 从 0 开始计数)。任务尝试在作业运行时根据需要分配，所以，它们的顺序代表被创建运行的先后顺序。

6.5.3 MapReduce 的 Web 界面

Hadoop 的 Web 界面用来浏览作业信息，对于跟踪作业运行进度、查找作业完成后的统计信息和日志非常有用。可以在 *http://resource-manager-host:8088/*找到用户界面信息。

1. 资源管理器页面

图 6-1 展示了主页的截屏。“Cluster Metrics”部分给出了集群的概要信息，包括当前集群上处于运行及其他不同状态的应用的数量，集群上可用的资源数量(“Memory Total”)及节点管理器的相关信息。

图 6-1. 资源管理器页面的屏幕截图

接下来的主表中列出了集群上所有曾经运行或正在运行的应用。有个搜索窗口可以用于过滤寻找所感兴趣的应用。主视图中每页可以显示 100 个条目，资源管理器在同一时刻能够在内存中保存近 10 000 个已完成的应用(通过设置 `yarn.resourcemanager.max-completed-applications`)，随后只能通过作业历史页面获取这些应用信息。注意，作业历史是永久存储的，因此也可以通过作业历史找到资源管理器以前运行过的作业。

作业历史

作业历史指已完成的 MapReduce 作业的事件和配置信息。不管作业是否成功执行，作业历史都将保存下来，为运行作业的用户提供有用信息。

作业历史文件由 MapReduce 的 application master 存放在 HDFS 中，通过 `mapreduce.jobhistory.done-dir` 属性来设置存放目录。作业的历史文件会保存一周，随后被系统删除。

历史日志包括作业、任务和尝试事件，所有这些信息以 JSON 格式存放在文件中。特定作业的历史可以通过作业历史服务器的 Web 界面(通过资源管理器页面链接)查看，或在命令行方法下用 `mapred job -history`(指向作业历史文件中)查看。

2. MapReduce 作业页面

单击“Tracking UI”链接进入 application master 的 web 界面(如果应用已经完成则进入历史页面)。在 MapReduce 中，将进入作业页面，如图 6-2 所示。

图 6-2. 作业页面的屏幕截图

作业运行期间，可以在作业页面监视作业进度。底部的表展示 map 和 reduce 进度。“Total”显示该作业 map 和 reduce 的总数。其他列显示的是这些任务的状态：`Pending`(等待运行)、`Running`(运行中)或 `Complete`(成功完成)。

表下面的部分显示的是 map 或 reduce 任务中失败和被终止的任务尝试的总数。任

务尝试(task attempt)可标记为被终止，如果它们是推测执行的副本，或它们运行的节点已结束，或它们已被用户终止。7.2.1 节对任务失败进行了详细的讨论。

导航栏中还有许多有用的链接。例如，“Configuration”链接指向作业的统一配置文件，该文件包含了作业运行过程中生效的所有属性及属性值。如果不确定某个属性的设置值，可以通过该链接查看文件。

6.5.4 获取结果

一旦作业完成，有许多方法可以获取结果。每个 reducer 产生一个输出文件，因此，在 *max-temp* 目录中会有 30 个部分文件(part file)，命名为 *part-00000* 到 *part-00029*。

正如文件名所示，这些“part”文件可以认为是 *max-temp* 文件的一部分。

如果输出文件很大(本例不是这种情况)，那么把文件分为多个 part 文件很重要，这样才能使多个 reducer 并行工作。通常情况下，如果文件采用这种分割形式，使用起来仍然很方便：例如作为另一个 MapReduce 作业的输入。在某些情况下，可以探索多个分割文件的结构来进行 map 端连接操作(参见 8.3.1 节)。

这个作业产生的输出很少，所以很容易从 HDFS 中将其复制到开发机器上。`hadoop fs` 命令的`-getmerge` 选项在这时很有用，因为它得到了源模式指定目录下所有的文件，并将其合并为本地文件系统的一个文件：

```
% hadoop fs -getmerge max-temp max-temp-local
% sort max-temp-local | tail
1991 607
1992 605
1993 567
1994 568
1995 567
1996 561
1997 565
1998 568
1999 568
2000 558
```

因为 reduce 的输出分区是无序的(使用哈希分区函数的缘故)，我们对输出进行排序。对 MapReduce 的数据做些后期处理是很常见的，把这些数据送入分析工具(例如 R、电子数据表甚至关系型数据库)进行处理。

如果输出文件比较小，另外一种获取输出的方式是使用`-cat` 选项将输出文件打印到控制台：

```
% hadoop fs -cat max-temp/*
```

深入分析后，我们发现某些结果看起来似乎没有道理。比如，1951 年(此处没有显示)的最高气温是 590℃！这个结果是怎么产生的呢？是不正确的输入数据还是程序中的 bug?

6.5.5 作业调试

最经典的方法通过打印语句来调试程序，这在 Hadoop 中同样适用。然而，需要考虑复杂的情况：当程序运行在几十台、几百台甚至几千台节点上时，如何找到并检测调试语句分散在这些节点中的输出呢？为了处理我们这种要查找一个不寻常情况的需求，可以用一个调试语句记录到一个标准错误中，同时配合更新任务状态信息以提示我们查看错误日志。我们将看到，Web UI 简化了这个操作。

我们还要创建一个自定义的计数器来统计整个数据集中不合理的气温记录总数。这就提供了很有价值的信息来处理如下情况，如果这种情况经常发生，我们需要从中进一步了解事件发生的条件以及如何提取气温值，而不是简单地丢掉这些记录。事实上，调试一个作业的时候，应当总想是否能够使用计数器来获得需要找出事件发生来源的相关信息。即使需要使用日志或状态信息，但使用计数器来衡量问题的严重程度仍然也是有帮助的(详情参见 9.1 节)。

如果调试期间产生的日志数据规模比较大，可以有多种选择。一种是将这些信息写到 map 的输出流供 reduce 任务分析和汇总，而不是写到标准错误流。这种方法通常必须改变程序结构，所以先选用其他技术。另一种是可以写一个程序(当然是 MapReduce 程序)来分析作业产生的日志。

我们把调试加入 mapper(版本 3)，而不是 reducer，因为我们希望找到导致这些异常输出的数据源：

```
public class MaxTemperatureMapper
  extends Mapper<LongWritable, Text, Text, IntWritable> {

enum Temperature {
  OVER_100
}

private NcdcRecordParser parser = new NcdcRecordParser();
@ Override
public void map(LongWritable key, Text value, Context context)
       throws IOException, InterruptedException {
```

```
parser.parse(value);
if (parser.isValidTemperature()) {
  int airTemperature = parser.getAirTemperature();
  if (airTemperature > 1000) {
    System.err.println("Temperature over 100 degrees for input: " + value);
    context.setStatus("Detected possibly corrupt record: see logs.");
    context.getCounter(Temperature.OVER_100.increment(1));
    }
    context.write(new Text(parser.getYear()), new IntWritable(airTemperature));
  }
}
}
```

如果气温超过 100℃(表示为 1000，因为气温只保留小数点后一位)，我们输出一行到标准错误流以代表有问题的行，同时使用 Context 的 setStatus()方法来更新 map 的状态信息，引导我们查看日志。我们还增加了计数器，在 Java 中用 enum 类型的字段表示。在这个程序中，定义一个 OVER_100 字段来统计气温超过 100℃的记录数。

完成这些修改，我们重新编译代码，重新创建 JAR 文件，然后重新运行作业并在运行时进入任务页面。

1. 任务和任务尝试页面

作业页面包含了一些查看作业中任务细节的链接。例如，点击“Map”链接，将进入一个列举了所有 map 任务的信息的页面。图 6-3 的屏幕截图显示了一个作业的任务信息页面，该作业带有调试语句，运行时在任务的“Status”列中显示调试信息。

图 6-3. 任务页面的屏幕截图

点击任务链接将进入任务尝试页面，页面显示了该任务的每个任务尝试。每个任务尝试页面都有链接指向日志文件和计数器。如果进入成功任务尝试的日志文件链接，将发现所记录的可疑输入记录。这里考虑到篇幅，已经进行了转行和截断处理：

```
Temperature over 100 degrees for input:
0335999999433181957042302005 + 37950 + 139117SAO  +0004RJSN  V02011359003150070356999
99943320195701010000 5 + 35317 + 139650SAO
+000899999V0200235900265007624 9N0040005...
```

此记录的格式看上去与其他记录不同。可能是因为行中有空格，规范中没有这方面的描述。

作业完成后，查看我们定义的计数器的值，检查在整个数据集中有多少记录超过100℃。通过 Web 界面或命令行，可以查看计数器：

```
% mapred job -counter job_1410450250506_0006 \
'v3.MaxTemperatureMapper$Temperature' OVER_1003
```

-counter 选项的输入参数包括作业 ID，计数器的组名(这里一般是类名)和计数器名称(enum 名)。这里，在超过十亿条记录的整个数据集中，只有三个异常记录。直接扔掉不正确的记录，是许多大数据问题中的标准做法。然而，这里我们需要谨慎处理这种情况，因为我们寻找的是一个极限值-最高气温值，而不是一个累计测量值。当然，在本例中，扔掉三个记录可能并不会影响结果。

2. 处理不合理的数据

捕获引发问题的输入数据是很有价值的，因为我们可以在测试中用它来检查 mapper 的工作是否正常。在这个 MRUnit 测试中，我们将检查对于不合理的输入计数器是否进行了更新：

```
@Test
public void parsesMalformedTemperature() throws IOException,
InterruptedException {
  Text value = new Text("0335999999433181957042302005+37950+139117SAO +0004" +
    // Year ^^^^
  "RJSN V02011359003150070356999999433201957010100005+353");
    // Temperature ^^^^^
Counters counters = new Counters();
new MapDriver<longWritable, Text, Text, IntWritable()>
    withMapper [new MaxTemperature Mapper()]
    withInput(new LongWritable(0), value)
    withCounters(counters)
    runTest();
Counter c = ounters.findCounter(MaxTemperatureMapper.Temperature.MALFORMED);
```

```
    assertThat(c.getValue(), is(1L));
    }
```

引发问题的记录与其他行的格式是不同的。范例 6-12 显示了修改过的程序(版本 4)，它使用的解析器忽略了那些没有首符号(+或-)气温字段的行。我们还引入一个计数器来统计因为这个原因而被忽略的记录数。

范例 6-12. 该 mapper 用于查找最高气温

```
public class MaxTemperatureMapper
  extends Mapper<LongWritable, Text, Text, IntWritable> {

  enum Temperature {
    MALFORMED
  }
private NcdcRecordParser parser = new NcdcRecordParser();

@Override
public void map(LongWritable key, Text value, context context
    throws IOException, InterruptedException {

  parser.parse(value);
  if (parser.isValidTemperature()) {
    int airTemperature = parser.getAirTemperature();
    context.write(new Text(parser.getYear()),new IntWritable(airTemperature));
  } else if (parser.isMalformedTemperature()) {
    System.err.println("Ignoring possibly corrupt input: " + value);
    context.getCounter(Temperature.MALFORMED).increment(1);
   }
  }
}
```

6.5.6 Hadoop 日志

针对不同用户，Hadoop 在不同的地方生成日志。表 6-2 对此进行了总结。

YARN 有一个日志聚合(*log aggregation*)服务,可以取到已完成的应用的任务日志，并把其搬移到 HDFS 中，在那里任务日志被存储在一个容器文件中用于存档。如果服务已被启用(通过在集群上将 `yarn.log-aggregation-enable` 设置为 `true`)，可以通过点击任务尝试 web 界面中 *logs* 链接，或使用 `mapred job -logs` 命令查看任务日志。

默认情况下，日志聚合服务处于关闭状态。此时，可以通过访问节点管理器的界面(http://node-manager-host:8042/logs/userlogs)查看任务日志。

表 6-2. Hadoop 日志的类型

日志	主要对象	描述	更多信息
系统守护进程日志	管理员	每个 Hadoop 守护进程产生一个日志文件(使用 log4j)和另一个(文件合并标准输出和错误)。这些文件分别写入 HADOOP_ LOG_DIR 环境变量定义的目录	参见 10.3.2 节和 11.2.1 节
HDFS 审计日志	管理员	这个日志记录所有 HDFS 请求，默认是关闭状态。虽然该日志存放位置可以配置，但一般写入 namenode 的日志	参见 11.1.3 节
MapReduce 作业历史日志	用户	记录作业运行期间发生的事件(如任务完成)。集中保存在 HDFS 中	参见 6.5.3 节的补充内容“作业历史”
MapReduce 任务日志	用户	每个任务子进程都用 log4j 产生一个日志文件(称作 *syslog*)，一个保存发到标准输出(*stdout*)数据的文件，一个保存标准错误(*stderr*)的文件。这些文件写入到 YARN_LOG_DIR 环境变量定义的目录的 *userlogs* 的子目录中	参见本小节

对这些日志文件的写操作是很直观的。任何到标准输出或标准错误流的写操作都直接写到相关日志文件。当然，在 Streaming 方式下，标准输出用于 map 或 reduce 的输出，所以不会出现在标准输出日志文件中。

在 Java 中，如果愿意的话，用 Apache Commons Logging API(实际上可以使用任何能写入 log4i 的日志 API)就可以写入任务的系统日志文件中(*syslog* 文件)，如范例 6-13 所示。

范例 6-13. 这个等价的 Mapper 写到标准输出(使用 Apache Commons Logging API)

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.hadoop.mapreduce.Mapper;

public class LoggingIdentityMapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
    extends Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {

  private static final Log LOG = LogFactory.getLog(LoggingIdentityMapper.class);

  @Override
@SuppressWarnings("unchecked")
    public void map(KEYIN key, VALUEIN value, Context context)
      throws IOException, InterruptedException {
    // Log to stdout file
  System.out.println("Map key: " + key);
```

```
    // Log to syslog file
    LOG.info("Map key: " + key);
    if (LOG.isDebugEnabled()) {
      LOG.debug("Map value: " + value);
    }
    context.write((KEYOUT) key, (VALUEOUT) value);
  }
}
```

默认的日志级别是 INFO，因此 DEBUG 级别的消息不在 *syslog* 任务日志文件中出现。然而，有时候又希望看到这些消息。这时可以适当设置 mapreduce.map.log.level 或者 mapreduce.reduce.log.level。例如，对于上面的情况你可以为 mapper 进行如下设置，以便能够看到日志中的 map 值。

```
% hadoop jar hadoop-examples.jar LoggingDriver -conf conf/hadoop-cluster.xml \
-D mapreduce.map.log.level=DEBUG input/ncdc/sample.txt logging-out
```

有一些控制用于管理任务日志的大小和记录保留时间。在默认情况下，日志最短在 3 小时后删除(时间可以通过 yarn.nodemanager.log.retain-seconds 属性来设置，当然，如果日志聚合被激活，这个时间可以被忽略)。也可以用 mapreduce.task.userlog.limit.kb 属性为每个日志文件的最大规模设置一个阈值，默认值是 0，表示没有上限。

有时你可能需要调试一个问题，这个问题你怀疑在运行一个 Hadoop 命令的 JVM 上发生，而不是在集群上。可以通过如下调用将 DEBUG 级别的日志发送给控制台：

```
% HADOOP_ROOT_LOGGER=DEBUG,console hadoop fs -text /foo/bar
```

6.5.7 远程调试

当一个任务失败并且没有足够多的记录信息来诊断错误时，可以选择用调试器运行该任务。在集群上运行作业时，很难使用调试器，因为不知道哪个节点处理哪部分输入，所以不能在错误发生之前安装调试器。然而，有其他一些方法可以用。

- **在本地重新产生错误** 对于特定的输入，失败的任务通常总会失败。你可以尝试通过下载致使任务失败的文件到本地运行重现问题，这可以使用到调试器(如 Java 的 VisualVM)。
- **使用 JVM 调试选项** 失败的常见原因是任务 JVM 中 Java 内存溢出。可以将 mapred.child.java.opts 设为包含-XX:HeapDumpOnOutOfMemory

Error-XX:HeapDumpPath=*/path/to/dumps*。该设置将产生一个堆转储(heap dump)，这可以通过 *jhat* 或 Eclipse Memory Analyzer 这样的工具来检查。注意，该 JVM 选项应当添加到由 mapred.child.java.opts 指定的已有内存设置中。10.3.3 节在讨论 YARN 和 MapReduce 的内存时有进一步的讨论。

- **使用任务分析** Java 的 profiler 提供了很多 JVM 的内部细节，Hadoop 提供了分析作业中部分任务的机制。参见 6.6 节。

在一些情况下保存失败的任务尝试的中间结果文件对于以后的检查是有用的，特别是在任务工作路径中建立转储或配置文件。可以将 mapreduce.task.files.preserve.failedtasks 设为 true 来保存失败的任务文件。

也可以保存成功任务的中间结果文件，以便解释任务没有失败。这时，将属性 mapreduce.task.files.preserve.filepattern 设置为一个正则表达式(与保留的任务 ID 匹配)。

对调试有用的另一个属性是 yarn.nodemanager.delete.debug-delay-sec，以秒为单位，表示等待删除本地尝试文件(如用于启动任务容器 JVM 的脚本)的时间。如果在集群上该属性值被设置为一个比较大的合理值(例如，600，表示 10 分钟)，那么在文件删除前有足够的时间查看。

为了检查任务尝试文件，登录到任务失败的节点并找到该任务尝试的目录。它在一个本地 MapReduce 目录下，由 mapreduce.cluster.local.dir 的设置决定(细节请参见 10.3.3 节)。如果这个属性是以逗号分隔的目录列表(在一台机器的物理磁盘上分布负载)，在找到那个特定的任务尝试(task attempt)之前，需要搜索整个目录。task attemp 的目录在以下位置：

mapreduce.cluster.local.dir/usercache/*user*/appcache/*application-ID*/output/*task-attempt-ID*

6.6 作业调优

作业运行后，许多开发人员可能会问："能够让它运行得更快一些吗?"

有一些 Hadoop 相关的"疑点"值得检查一下，看它们是不是引发性能问题的"元凶"。在开始任务级别的分析或优化之前，必须仔细研究表 6-3 所示的检查内容。

表 6-3. 作业调优检查表

范围	最佳实践	更多参考信息
mapper 的数量	mapper 需要运行多长时间?如果平均只运行几秒钟，则可以看是否能用更少 mapper 运行更长的时间，通常是一分钟左右。时间长度取决于使用的输入格式	8.2.1 节
reducer 的数量	检查使用的 reducer 数目是不是超过 1 个。根据经验，Reduce 任务应运行 5 分钟左右，且能生产出至少一个数据块的数据	8.1.1 节
combiner	作业能否充分利用 combiner 来减少通过 shuffle 传输的数据量	2.4.2 节
中间值的压缩	对 map 输出进行压缩几乎总能使作业执行得更快	5.2.3 节
自定义序列	如果使用自定义的 `Writable` 对象或自定义的 comparator，则必须确保已实现 `RawComparator`	5.3.3 节
调整 shuffle	MapReduce 的 shuffle 过程可以对一些内存管理的参数进行调整，以弥补性能的不足	7.3.3 节

分析任务

正如调试一样，对 MapReduce 这类分布式系统上运行的作业进行分析也有诸多挑战。Hadoop 允许分析作业中的一部分任务，并且在每个任务完成时，把分析信息放到用户的机器上，以便日后使用标准分析工具进行分析。

当然，对本地作业运行器中运行的作业进行分析可能稍微简单些。如果你有足够的数据运行 map 和 reduce 任务，那么对于提高 mapper 和 reducer 的性能有很大的帮助。但必须注意一些问题。本地作业运行器是一个与集群完全不同的环境，并且数据流模式也截然不同。如果 MapReduce 作业是 I/O 密集型的(很多作业都属于此类)，那么优化代码的 CPU 性能是没有意义的。为了保证所有调整都是有效的，应该在实际集群上对比新老执行时间。这说起来容易做起来难，因为作业执行时间会随着与其他作业的资源争夺和调度器决定的任务顺序不同而发生改变。为了在这类情况下得到较短的作业执行时间，必须不断运行(改变代码或不改变代码)，并检查是否有明显的改进。

有些问题(如内存溢出)只能在集群上重现，在这些情况下，必须能够在发生问题的地方进行分析。

1. HPROF 分析工具

许多配置属性可以控制分析过程，这些属性也可以通过 JobConf 的简便方法获取。启用分析很简单，将属性 mapreduce.task.profile 设置为 true 即可：

```
% hadoop jar hadoop-examples.jar v4.MaxTemperatureDriver \
  -conf conf/hadoop-cluster.xml \
  -D mapreduce.task.profile=true \
  input/ncdc/all max-temp
```

上述命令正常运行作业，但是给用于启动节点管理器上的任务容器的 Java 命令增加了一个 -agentlib 参数。可以通过设置属性 mapreduce.task.profile.params 来精确地控制该新增参数。默认情况下使用 HPROF，一个 JDK 自带的分析工具，虽然只有基本功能，但是能提供程序的 CPU 和堆使用情况等有价值的信息。

分析作业中的所有任务通常没有意义，因此，默认情况下，只有那些 ID 为 0，1，2 的 map 任务和 reduce 任务将被分析。可以通过设置 mapreduce.task.profile.maps 和 mapreduce.task.profile.reduces 两个属性来指定想要分析的任务 ID 的范围。

每个任务的分析输出和任务日志一起存放在节点管理器的本地日志目录的 *userlogs* 子目录下(和 *syslog*，*stdout*，*stderr* 文件一起)，可以根据日志聚合是否启用，使用 6.5.6 节介绍的方法获取到。

6.7 MapReduce 的工作流

至此，你已经知道 MapReduce 应用开发的机制了。我们目前还未考虑如何将数据处理问题转化成 MapReduce 模型。

本书前面的数据处理都用来解决十分简单的问题(如在指定年份找到最高气温值的记录)。如果处理过程更复杂，这种复杂度一般是因为有更多的 MapReduce 作业，而不是更复杂的 map 和 reduce 函数。换而言之，通常是增加更多的作业，而不是增加作业的复杂度。

对于更复杂的问题，可考虑使用比 MapReduce 更高级的语言，如 Pig、hive、Cascading、Crunch 或 Spark。一个直接的好处是：有了它之后，就用不着处理到 MapReduce 作业的转换，而是集中精力分析正在执行的任务。

最后，Jimmy Lin 和 Chris Dyer 合著的《MapReduce 数据密集型文本处理》(*Data-Intensive Text Processing with MapReduce*)一书是学习 MapReduce 算法设计的优秀资源，强烈推荐。该书由 Morgan & Claypool 出版社于 2010 出版。

6.7.1 将问题分解成 MapReduce 作业

让我们看一个更复杂的问题，我们想把它转换成 MapReduce 工作流。

假设我们想找到每个气象台每年每天的最高气温记录的均值。例如，要计算 029070～99999 气象台的 1 月 1 日的每日最高气温的均值，我们将从这个气象台的 1901 年 1 月 1 日，1902 年 1 月 1 日，直到 2000 年的 1 月 1 日的气温中找出每日最高气温的均值。

我们如何使用 MapReduce 来计算它呢?计算自然分解为下面两个阶段。

(1) 计算每对 station-date 的每日最高气温。
本例中的 MapReduce 程序是最高气温程序的一个变种，不同之处在于本例中的键是一个综合的 station-date 对，而不只是年份。

(2) 计算每个 station-day-month 键的每日最高气温的均值。
mapper 从上一个作业得到输出记录(station-date，最高气温值)，丢掉年份部分，将其值投影到记录(station-day-month，最高气温值)。然后 reducer 为每个 station-day-month 键计算最高气温值的均值。

第一阶段的输出看上去就是我们想要的气象台的信息。范例中的 *mean_max_daily_temp.sh* 脚本提供了 Hadoop Streaming 的一个实现：

```
029070-99999   19010101   0
029070-99999   19020101   -94
...
```

前两个字段形成键，最后一列是指定气象台和日期所有记录中的最高气温。第二阶段计算这些年份中每日最高气温的平均值：

```
029070-99999    0101  -68
```

以上是气象台 029070-99999 在整个世纪中 1 月 1 日的日均最高气温为-6.8℃。

只用一个 MapReduce 过程就能完成这个计算，但是它可能会让部分程序员花更多

精力。[1]

一个作业可以包含多个(简单的)MapReduce 步骤，这样整个作业由多个可分解的、可维护的 mapper 和 reducer 组成。本书第 V 部分提到的一些实例学习包括使用 MapReduce 来解决的大量实际问题，在每个例子中，数据处理任务都是使用两个或更多 MapReduce 作业来实现的。对于理解如何将问题分解成 MapReduce 工作流，第 V 部分所提供的详细介绍非常有价值。

相对于我们已经做的，mapper 和 reducer 完全可以进一步分解。mapper 一般执行输入格式解析、投影(选择相关的字段)和过滤(去掉无关记录)。在前面的 mapper 中，我们在一个 mapper 中实现了所有这些函数。然而，还可以将这些函数分割到不同的 mapper，然后使用 Hadoop 自带的 `ChainMapper` 类库将它们连接成一个 mapper。结合使用 `ChainReducer`，你可以在一个 MapReduce 作业中运行一系列的 mapper，再运行一个 reducer 和另一个 mapper 链。

6.7.2 关于 JobControl

当 MapReduce 工作流中的作业不止一个时，问题随之而来：如何管理这些作业按顺序执行?有几种方法，其中主要考虑是否有一个线性的作业链或一个更复杂的作业有向无环图(DAG，directed acyclic graph)。

对于线性链表，最简单的方法是一个接一个地运行作业，等前一个作业运行结束后再运行下一个：

```
JobClient.runJob(conf1);
JobClient.runJob(conf2);
```

如果一个作业失败，`runJob()`方法就抛出一个 `IOException`，这样一来，管道中后面的作业就无法执行。根据具体的应用程序，你可能想捕获异常，并清除前一个作业输出的中间数据。

这种方法类似新的 MapReduce API，除了需要 Job 上的 `waitForCompletion()`方法的布尔返回值：`true` 表示作业成功，而 `false` 表示失败。

对于比线性链表更复杂的结构，有相关的类库可以帮助你合理安排工作流。它们也适用于线性链表或一次性作业。最简单的是 `org.apache.hadoop.mapreduce.`

① 这个一个很有趣的练习。提示：使用 9.2.4 节介绍的内容。

jobcontrol 包中的 JobControl 类。在 org.apache.hadoop.mapred.jobcontrol 包中也有一个等价的类。JobControl 的实例表示一个作业的运行图，你可以加入作业配置，然后告知 JobControl 实例作业之间的依赖关系。在一个线程中运行 JobControl 时，它将按照依赖顺序来执行这些作业。也可以查看进程，在作业结束后，可以查询作业的所有状态和每个失败相关的错误信息。如果一个作业失败，JobControl 将不执行与之有依赖关系的后续作业。

6.7.3 关于 Apache Oozie

Apache Oozie 是一个运行工作流的系统，该工作流由相互依赖的作业组成。Oozie 由两部分组成：一个工作流引擎，负责存储和运行由不同类型的 Hadoop 作业(MapReduce，Pig，Hive 等)组成的工作流；一个 *coordinator* 引擎，负责基于预定义的调度策略及数据可用性运行工作流作业。Oozie 的设计考虑到了可扩展性，能够管理 Hadoop 集群中数千工作流的及时运行，每个工作流的组成作业都可能有好几十个。

由于在运行工作流中成功的那一部分时不会浪费任何时间，因此在 Oozie 中更易处理失败工作流的重运行。任何一个管理过复杂批处理系统的人都知道，由于宕机或故障而丢失作业后再跟上处理节奏是多么困难，因此他们都会欣赏 Oozie 的这个特性。(更进一步，coordinator 应用代表了单个的数据管线可以被打包在一起，然后作为一个单元一起运行。)

不同于在客户端运行并提交作业的 JobControl，Ooize 作为服务器运行，客户端提交一个立即或稍后执行的工作流定义到服务器。在 Ooize 中，工作流是一个由动作(action)节点和控制流节点组成的 DAG(有向无环图)。

动作节点执行工作流任务，例如在 HDFS 中移动文件，运行 MapReduce、Streaming、Pig 或 Hive 作业，执行 Sqoop 导入，又或者是运行 shell 脚本或 Java 程序。控制流节点通过构建条件逻辑(不同执行分支的执行依赖于前一个动作节点的输出结果)或并行执行来管理活动之间的工作流执行情况。当工作流结束时，Oozie 通过发送一个 HTTP 的回调向客户端通知工作流的状态。还可以在每次进入工作流或退出一个动作节点时接收到回调。

1. 定义 Oozie 工作流

工作流定义是使用 Hadoop Process Difinition Language 以 XML 格式来书写，这个

规范可在 Oozie 网站(*http://oozie.apache.org/*)找到。范例 6-14 展示了一个运行单个 MapReduce 作业的简单 OoZie 工作流定义。

范例 6-14. 用来运行求最高温度的 MapReduce 作业的 Oozie 工作流定义

```
<workflow-app xmlns="uri:oozie:workflow:0.1" name="max-temp-workflow">
  <start to="max-temp-mr"/>
  <action name="max-temp-mr">
    <map-reduce>
      <job-tracker>${resourceManager}</job-tracker>
      <name-node>${nameNode}</name-node>
      <prepare>
       <delete path="${nameNode}/user/${wf:user()}/output"/>
      </prepare>
      <configuration>
       <property>
           <name>mapred.mapper.new-api</name>
           <value>true</value>
       </property>
       <property>
         <name>mapred.reducer.new-api</name>
         <value>true</value>
       </property>
       <property>
         <name>mapreduce.job.map.class</name>
         <value>MaxTemperatureMapper</value>
       </property>
       <property>
         <name>mapreduce.job.combine.class</name>
         <value>MaxTemperatureReducer</value>
       </property>
       <property>
         <name>mapreduce.job.reduce.class</name>
         <value>MaxTemperatureReducer</value>
       </property>
       <property>
<name>mapreduce.job.output.key.class</name>
<value>org.apache.hadoop.io.Text</value>
       </property>
       <property>
         <name>mapreduce.job.output.value.class</name>
         <value>org.apache.hadoop.io.IntWritable</value>
       </property>
       <property>
<name>mapreduce.input.fileinputformat.inputdir</name>
<value>/user/${wf:user()}/input/ncdc/micro</value>
       </property>
       <property>
        <name>mapreduce.output.fileoutputformat.outputdir</name>
         <value>/user/${wf:user()}/output</value>
       </property>
      </configuration>
    </map-reduce>
```

```
    <ok to="end"/>
    <error to="fail"/>
  </action>
  <kill name="fail">
     <message>MapReduce failed,error message[${wf:errorMessage(wf:lastErrorNode())}]
     </message>
  </kill>
  <end name="end"/>
</workflow-app>
```

这个工作流有三个控制流节点和一个动作节点：一个 start 控制节点、一个 map-reduce 动作节点、一个 kill 控制节点和一个 end 控制节点。节点及其之间的转换如图 6-4 所示。

每个工作流都必须有一个 start 节点和一个 end 节点。当工作流作业开始时，它转移到有 start 节点指定的节点上(本例中 max-temp-mr 动作)。当一个工作流作业转移到 end 节点时就意味着它成功完成了。然而，如果一个工作流作业转移到了 kill 节点，那么就被认为失败了并且报告在工作流定义中的 message 元素指定的错误消息。

这个工作流定义文件的大部分都是指定 map-reduce 动作。前两个元素(job-tracker 和 name-node)用于指定提交作业的 YARN 资源管理器(或 Hadoop 1 中的 jobtracker)和输入输出数据的 namenode(实际上是一个 Hadoop 文件系统的 URI)。两者都被参数化，使得工作流定义不受限于特定的集群，更有利于测试。这些参数在提交时指定为工作流属性，我们稍后会看到。

图 6-4. 一个 Oozie 工作流的转移图

与其名称无关，job-tracker 元素是用来指定 YARN 资源管理器地址和端口的。

可选项 prepare 元素在 MapReduce 作业之前运行并用于目录删除(在需要的时候也可以创建目录，但这里没有指明)。通过确保输出目录在运行作业之前处于一致的状态，如果作业失败的话，Oozie 也可以安全地重新执行。

运行 MapReduce 作业是在 configuration 元素中设定的，通过为 Hadoop 配置的名值对来设置嵌套的元素。可以把 MapReduce 配置部分看作本书中其他地方运行 MapReduce 程序(如范例 2-5 所示)使用的驱动类的一个替代。

我们在工作流的定义中的几个地方利用了 JSP Expression Language (EL)语法。Oozie 提供了一组与工作流交互的函数。例如，${wf:user()}返回开始当前工作流作业的用户名，我们用它来指定正确的文件系统路径。Oozie 规范中列出所有 Oozie 支持的 EL 函数。

2. 打包和配置 Oozie 工作流应用

工作流应用由工作流定义和所有运行它所需的资源(例如 MapReduce JAR 文件、Pig 脚本等)构成。应用必须遵循一个简单的目录结构，并在 HDFS 上配置，这样它们才能被 Oozie 访问。对于这个工作流应用，我们将所有的文件放到基础目录 *max-temp-workflow* 中，如下所示：

```
max-temp-workflow/
├── lib/
│   └── hadoop-examples.jar
└── workflow.xml
```

工作流定义文件 *workflow.xml* 必须在该目录的顶层出现。包含应用的 MapReduce 类的 JAR 文件放在 *lib* 目录中。

遵循这种布局的工作流应用能通过很多合适的工具创建，例如 Ant 或 Maven；可以在本书附带的代码中找到样例。一旦创建应用，就使用正规的 Hadoop 工具将它复制到 HDFS。命令如下：

```
% hadoop fs -put hadoop-examples/target/max-temp-workflow max-temp-workflow
```

3. 运行 Oozie 工作流作业

接下来，我们看看如何为刚刚上载的应用运行一个工作流作业。为此，使用 *oozie*

命令行工具，它是用于和 Oozie 服务器通信客户端程序。方便起见，我们输出 OOZIE_URL 环境变量来告诉 oozie 命令使用哪个 Oozie 服务器(这里我们使用本地运行的服务器)：

```
% export OOZIE_URL="http://localhost:11000/oozie"
```

oozie 工具有很多子命令(输入 oozie help 可得到这些子命令的列表)，但我们将调用带有-run 选项的 job 子命令来运行工作流作业：

```
% oozie job -config ch06-mr-dev/src/main/resources/max-temp-workflow.properties
\-run
job: 0000001-140911033236814-oozie-oozi-W
```

-config 选项设定本地 Java 属性文件，它包含工作流 XML 文件里参数的定义(这里是nameNode 和resourceManager)与oozie.wf.application.path，后者告知 Oozi HDFS 中工作流应用的位置。属性文件的内容如下：

```
nameNode=hdfs://localhost:8020
resourceManager=localhost:8032
oozie.wf.application.path=${nameNode}/user/${user.name}/max-temp-workflow
```

为了得到工作流作业的状态信息，要使用-info 选项，指定由前面运行的命令打印的作业 ID(输入 oozie job 将得到所有作业的列表)：

```
% oozie job -info 0000001-140911033236814-oozie-oozi-W
```

输出显示如下状态：RUNNING、KILLED 或者 SUCCEDED。通过 Oozie 的网页 UI(*http://localhost:11000/oozie*)，你也可以找到这些信息。

作业成功后，可以通过以下常用方法检查结果：

```
% hadoop fs -cat output/part-*
1949 111
1950 22
```

这个例子只是简单展示了如何写 Oozie 工作流。Oozie 网站上的文档介绍了如何创建更复杂的工作流以及如何写和运行 coordinator 作业。

第 7 章

MapReduce 的工作机制

在本章中，我们将深入学习 Hadoop 中的 MapReduce 工作机制。这些知识将为我们随后两章学习写 MapReduce 高级编程奠定基础。

7.1 剖析 MapReduce 作业运行机制

可以通过一个简单的方法调用来运行 MapReduce 作业：`Job` 对象的 `submit()`方法。注意，也可以调用 `waitForCompletion()`，它用于提交以前没有提交过的作业，并等待它的完成。[①] `submit()`方法调用封装了大量的处理细节。本小节将揭示 Hadoop 运行作业时所采取的措施。

整个过程描述如图 7-1 所示。在最高层，有以下 5 个独立的实体[②]。

- 客户端，提交 MapReduce 作业。
- YARN 资源管理器，负责协调集群上计算机资源的分配。
- YARN 节点管理器，负责启动和监视集群中机器上的计算容器(container)。
- MapReduce 的 `application master`，负责协调运行 MapReduce 作业的任务。它和 MapReduce 任务在容器中运行，这些容器由资源管理器分配并由节点管理器进行管理。

① 老版本 MapReduce API 中，调用 `JobClient.submitJob(conf)`或 `JobClient.runJob(conf)`。

② 本节中没有涉及作业历史服务端守护进程(负责维护作业历史数据)和 shuffle 处理器辅助服务(负责将 map 输出传送给 reduce 任务)的讨论。

- 分布式文件系统(一般为 HDFS，参见第 3 章)，用来与其他实体间共享作业文件。

7.1.1 作业的提交

Job 的 submit()方法创建一个内部的 JobSummiter 实例，并且调用其 submitJobInternal()方法(参见图 7-1 的步骤 1)。提交作业后，waitForCompletion()每秒轮询作业的进度，如果发现自上次报告后有改变，便把进度报告到控制台。作业完成后，如果成功，就显示作业计数器；如果失败，则导致作业失败的错误被记录到控制台。

图 7-1. Hadoop 运行 MapReduce 作业的工作原理

JobSummiter 所实现的作业提交过程如下所述。

- 向资源管理器请求一个新应用 ID，用于 MapReduce 作业 ID。请参见步骤 2。
- 检查作业的输出说明。例如，如果没有指定输出目录或输出目录已经存在，作业就不提交，错误抛回给 MapReduce 程序。
- 计算作业的输入分片。如果分片无法计算，比如因为输入路径不存在，作业就不提交，错误返回给 MapReduce 程序。
- 将运行作业所需要的资源(包括作业 JAR 文件、配置文件和计算所得的输入分片)复制到一个以作业 ID 命名的目录下的共享文件系统中(请参见步骤 3)。作业 JAR 的复本较多(由 mapreduce.client.submit.file.replication 属性控制，默认值为 10)，因此在运行作业的任务时，集群中有很多个复本可供节点管理器访问。
- 通过调用资源管理器的 submitApplication()方法提交作业。请参见步骤 4。

7.1.2 作业的初始化

资源管理器收到调用它的 submitApplication()消息后，便将请求传递给 YARN 调度器(scheduler)。调度器分配一个容器，然后资源管理器在节点管理器的管理下在容器中启动 application master 的进程(步骤 5a 和 5b)。

MapReduce 作业的 application master 是一个 Java 应用程序，它的主类是 MRAppMaster。由于将接受来自任务的进度和完成报告(步骤 6)，因此 application master 对作业的初始化是通过创建多个簿记对象以保持对作业进度的跟踪来完成的。接下来，它接受来自共享文件系统的、在客户端计算的输入分片(步骤 7)。然后对每一个分片创建一个 map 任务对象以及由 mapreduce.job.reduces 属性(通过作业的 setNumReduceTasks()方法设置)确定的多个 reduce 任务对象。任务 ID 在此时分配。

application master 必须决定如何运行构成 MapReduce 作业的各个任务。如果作业很小，就选择和自己在同一个 JVM 上运行任务。与在一个节点上顺序运行这些任务相比，当 application master 判断在新的容器中分配和运行任务的开销大于并行

运行它们的开销时，就会发生这一情况。这样的作业称为 *uberized*，或者作为 *uber* 任务运行。

哪些作业是小作业？默认情况下，小作业就是少于 10 个 mapper 且只有 1 个 reducer 且输入大小小于一个 HDFS 块的作业(通过设置 `mapreduce.job.ubertask.maxmaps`、`mapreduce.job.ubertask.maxreduces` 和 `mapreduce.job.ubertask.maxbytes` 可以改变这几个值)。必须明确启用 Uber 任务(对于单个作业，或者是对整个集群)，具体方法是将 `mapreduce.job.ubertask.enable` 设置为 `true`。

最后，在任何任务运行之前，application master 调用 `setupJob()` 方法设置 `OutputCommitter`。`FileOutputCommitter` 为默认值，表示将建立作业的最终输出目录及任务输出的临时工作空间。提交协议(commit protocol)将在 7.4.3 节介绍。

7.1.3 任务的分配

如果作业不适合作为 uber 任务运行，那么 application master 就会为该作业中的所有 map 任务和 reduce 任务向资源管理器请求容器(步骤 8)。首先为 Map 任务发出请求，该请求优先级要高于 reduce 任务的请求，这是因为所有的 map 任务必须在 reduce 的排序阶段能够启动前完成(详见 7.3 节)。直到有 5%的 map 任务已经完成时，为 reduce 任务的请求才会发出(详见 10.3.5 节)。

reduce 任务能够在集群中任意位置运行，但是 map 任务的请求有着数据本地化局限，这也是调度器所关注的(详见 4.1.1 节)。在理想的情况下，任务是数据本地化(*data local*)的，意味着任务在分片驻留的同一节点上运行。可选的情况是，任务可能是机架本地化(*rack local*)的，即和分片在同一机架而非同一节点上运行。有一些任务既不是数据本地化，也不是机架本地化，它们会从别的机架，而不是运行所在的机架上获取自己的数据。对于一个特定的作业运行，可以通过查看作业的计数器来确定在每个本地化层次上运行的任务的数量(参见表 9-6)。

请求也为任务指定了内存需求和 CPU 数。在默认情况下，每个 map 任务和 reduce 任务都分配到 1024 MB 的内存和一个虚拟的内核，这些值可以在每个作业的基础上进行配置(遵从于 10.3.3 节描述的最大值和最小值)，分别通过 4 个属性来设置 `mapreduce.map.memory.mb`、`mapreduce.reduce.memory.mb`、`mapreduce.map.cpu.vcores` 和 `mapreduce.reduce.cpu.vcoresp.memory.mb`。

7.1.4 任务的执行

一旦资源管理器的调度器为任务分配了一个特定节点上的容器，application master 就通过与节点管理器通信来启动容器(步骤 9a 和 9b)。该任务由主类为 `YarnChild` 的一个 Java 应用程序执行。在它运行任务之前，首先将任务需要的资源本地化，包括作业的配置、JAR 文件和所有来自分布式缓存的文件(步骤 10，参见 9.4.2 节)。最后，运行 map 任务或 reduce 任务(步骤 11)。

`YarnChild` 在指定的 JVM 中运行，因此用户定义的 map 或 reduce 函数(甚至是 `YarnChild`)中的任何缺陷不会影响到节点管理器，例如导致其崩溃或挂起。

每个任务都能够执行搭建(setup)和提交(commit)动作，它们和任务本身在同一个 JVM 中运行，并由作业的 `OutputCommitter` 确定(参见 7.1.4 节)。对于基于文件的作业，提交动作将任务输出由临时位置搬移到最终位置。提交协议确保当推测执行(speculative execution)被启用时(参见 7.4.2 节)，只有一个任务副本被提交，其他的都被取消。

Streaming

Streaming 运行特殊的 map 任务和 reduce 任务，目的是运行用户提供的可执行程序，并与之通信(参见图 7-2)。

图 7-2. 可执行的 Streaming 与节点管理器及任务容器的关系

Streaming 任务使用标准输入和输出流与进程(可以用任何语言写)进行通信。在任务执行过程中，Java 进程都会把输入键-值对传给外部的进程，后者通过用户定义的 map 函数和 reduce 函数来执行它并把输出键-值对传回 Java 进程。从节点管理器的角度看，就像其子进程自己在运行 map 或 reduce 代码一样。

7.1.5 进度和状态的更新

MapReduce 作业是长时间运行的批量作业，运行时间范围从数秒到数小时。这可能是一个很长的时间段，所以对于用户而言，能够得知关于作业进展的一些反馈是很重要的。一个作业和它的每个任务都有一个状态(status)，包括：作业或任务的状态(比如，运行中，成功完成，失败)、map 和 reduce 的进度、作业计数器的值、状态消息或描述(可以由用户代码来设置)。这些状态信息在作业期间不断改变，它们是如何与客户端通信的呢?

任务在运行时，对其进度(progress，即任务完成百分比)保持追踪。对 map 任务，任务进度是已处理输入所占的比例。对 reduce 任务，情况稍微有点复杂，但系统仍然会估计已处理 reduce 输入的比例。整个过程分成三部分，与 shuffle 的三个阶段相对应(详情参见 7.3 节)。比如，如果任务已经执行 reducer 一半的输入，那么任务的进度便是 5/6，这是因为已经完成复制和排序阶段(每个占 1/3)，并且已经完成 reduce 阶段的一半(1/6)。

MapReduce 中进度的组成

进度并不总是可测量的，但是虽然如此，它能告诉 Hadoop 有个任务正在做一些事情。比如，正在写输出记录的任务是有进度的，即使此时这个进度不能用需要写的总量的百分比来表示(因为即便是产生这些输出的任务，也可能不知道需要写的总量)。

进度报告很重要。构成进度的所有操作如下:

- 读入一条输入记录(在 mapper 或 reducer 中)
- 写入一条输出记录(在 mapper 或 reducer 中)
- 设置状态描述(通过 `Reporter` 或 `TaskAttemptContext` 的 `setStatus()`方法)
- 增加计数器的值(使用 `Reporter` 的 `incrCounter()`方法或 `Counter` 的 `increment()`方法)
- 调用 `Reporter` 或 `TaskAttemptContext` 的 `progress()`方法

任务也有一组计数器，负责对任务运行过程中各个事件进行计数(详情参见 2.3.2 节)，这些计数器要么内置于框架中，例如已写入的 map 输出记录数，要么由用户自己定义。

当 map 任务或 reduce 任务运行时，子进程和自己的父 application master 通过 *umbilical* 接口通信。每隔 3 秒钟，任务通过这个 umbilical 接口向自己的 application master 报告进度和状态(包括计数器)，application master 会形成一个作业的汇聚视图(aggregate view)。

资源管理器的界面显示了所有运行中的应用程序，并且分别有链接指向这些应用各自的 application master 的界面，这些界面展示了 MapReduce 作业的更多细节，包括其进度。

在作业期间，客户端每秒钟轮询一次 application master 以接收最新状态(轮询间隔通过 `mapreduce.client.progressmonitor.pollinterval` 设置)。客户端也可以使用 `Job` 的 `getStatus()`方法得到一个 `JobStatus` 的实例，后者包含作业的所有状态信息。

图 7-3 对上述过程进行了图解。

图 7-3. 状态更新在 MapReduce 系统中的传递流程

7.1.6 作业的完成

当 application master 收到作业最后一个任务已完成的通知后，便把作业的状态设置为“成功”。然后，在 Job 轮询状态时，便知道任务已成功完成，于是 Job 打印一条消息告知用户，然后从 waitForCompletion()方法返回。Job 的统计信息和计数值也在这个时候输出到控制台。

如果 application master 有相应的设置，也会发送一个 HTTP 作业通知。希望收到回调指令的客户端可以通过 mapreduce.job.end-notification.url 属性来进行这项设置。

最后，作业完成时，application master 和任务容器清理其工作状态(这样中间输出将被删除)，OutputCommitter 的 commitJob() 方法会被调用。作业信息由作业历史服务器存档，以便日后用户需要时可以查询。

7.2 失败

在现实情况中，用户代码错误不断，进程崩溃，机器故障，如此种种。使用 Hadoop 最主要的好处之一是它能处理此类故障并让你能够成功完成作业。我们需要考虑以下实体的失败：任务、application master、节点管理器和资源管理器。

7.2.1 任务运行失败

首先考虑任务失败的情况。最常见的情况是 map 任务或 reduce 任务中的用户代码抛出运行异常。如果发生这种情况，任务 JVM 会在退出之前向其父 application master 发送错误报告。错误报告最后被记入用户日志。application master 将此次任务任务尝试标记为 *failed*(失败)，并释放容器以便资源可以为其他任务使用。

对于 Streaming 任务，如果 Streaming 进程以非零退出代码退出，则被标记为失败。这种行为由 stream.non.zero.exit.is.failure 属性(默认值为 true)来控制。

另一种失败模式是任务 JVM 突然退出，可能由于 JVM 软件缺陷而导致 MapReduce 用户代码由于某些特殊原因造成 JVM 退出。在这种情况下，节点管理器会注意到进程已经退出，并通知 application master 将此次任务尝试标记为失败。

任务挂起的处理方式则有不同。一旦 application master 注意到已经有一段时间没

有收到进度的更新，便会将任务标记为失败。在此之后，任务 JVM 进程将被自动杀死。[①]任务被认为失败的超时间隔通常为 10 分钟，可以以作业为基础(或以集群为基础)进行设置，对应的属性为 `mapreduce.task.timeout`，单位为毫秒。

超时(timeout)设置为 0 将关闭超时判定，所以长时间运行的任务永远不会被标记为失败。在这种情况下，被挂起的任务永远不会释放它的容器并随着时间的推移最终降低整个集群的效率。因此，尽量避免这种设置，同时充分确保每个任务能够定期汇报其进度。参见 7.1.5 节的补充材料“MapReduce 中进度的组成”。

application master 被告知一个任务尝试失败后，将重新调度该任务的执行。application master 会试图避免在以前失败过的节点管理器上重新调度该任务。此外，如果一个任务失败过 4 次，将不会再重试。这个值是可以设置的：对于 map 任务，运行任务的最多尝试次数由 `mapreduce.map.maxattempts` 属性控制；而对于 reduce 任务，则由 `mapreduce.reduce.maxattempts` 属性控制。在默认情况下，如果任何任务失败次数大于 4(或最多尝试次数被配置为 4)，整个作业都会失败。

对于一些应用程序，我们不希望一旦有少数几个任务失败就中止运行整个作业，因为即使有任务失败，作业的一些结果可能还是可用的。在这种情况下，可以为作业设置在不触发作业失败的情况下允许任务失败的最大百分比。针对 map 任务和 reduce 任务的设置可以通过 `mapreduce. map.failures.maxpercent` 和 `mapreduce.reduce.failures.maxpercent` 这两个属性来完成。

任务尝试(task attempt)也是可以中止的(killed)，这与失败不同。任务尝试可以被中止是因为它是一个推测副本(相关详情可以参见 7.4.2 节)或因为它所处的节点管理器失败，导致 application master 将它上面运行的所有任务尝试标记为 killed。被中止的任务尝试不会被计入任务运行尝试次数(由 `mapreduce.map.maxattempts` 和 `mapreduce.reduce.maxattempts` 设置)，因为尝试被中止并不是任务的过错。

用户也可以使用 Web UI 或命令行方式(输入 `mapred job` 查看相应的选项)来中止

① 如果一个 Streaming 进程挂起，节点管理器在下面这种情况中将会终止它(与启动它的 JVM 一起)：`yarn.nodemanager.container-executor.class` 被设置为 `org.apache.hadoop.yarn.server.nodemanager.LinuxContainerExecutor` 或者默认的容器执行器正被使用并且系统上可以使用 `setid` 命令(这样，任务 JVM 和它启动的所有进程都在同一个进程群中)。对于其他情况，孤立的 Streaming 进程将堆积在系统上，随着时间的推移，这会影响到利用率。

或取消任务尝试。也可以采用相同的机制来中止作业。

7.2.2 application master 运行失败

YARN 中的应用程序在运行失败的时候有几次尝试机会，就像 MapReduce 任务在遇到硬件或网络故障时要进行几次尝试一样。运行 MapReduce application master 的最多尝试次数由 `mapreduce.am.max-attempts` 属性控制。默认值是 2，即如果 MapReduce application master 失败两次，便不会再进行尝试，作业将失败。

YARN 对集群上运行的 YARN application master 的最大尝试次数加以了限制，单个的应用程序不可以超过这个限制。该限制由 `yarn.resourcemanager. am.max-attempts` 属性设置，默认值是 2，这样如果你想增加 MapReduce application master 的尝试次数，你也必须增加集群上 YARN 的设置。

恢复的过程如下。application master 向资源管理器发送周期性的心跳，当 application master 失败时，资源管理器将检测到该失败并在一个新的容器(由节点管理器管理)中开始一个新的 master 实例。对于 Mapreduce application master，它将使用作业历史来恢复失败的应用程序所运行任务的状态，使其不必重新运行。默认情况下恢复功能是开启的，但可以通过设置 `yarn.app.mapreduce.am.job.recovery.enable` 为 `false` 来关闭这个功能。

Mapreduce 客户端向 application master 轮询进度报告，但是如果它的 application master 运行失败，客户端就需要定位新的实例。在作业初始化期间，客户端向资源管理器询问并缓存 application master 的地址，使其每次需要向 application master 查询时不必重载资源管理器。但是，如果 application master 运行失败，客户端就会在发出状态更新请求时经历超时，这时客户端会折回向资源管理器请求新的 application master 的地址。这个过程对用户是透明的。

7.2.3 节点管理器运行失败

如果节点管理器由于崩溃或运行非常缓慢而失败，就会停止向资源管理器发送心跳信息(或发送频率很低)。如果 10 分钟内(可以通过属性 `yarn.resourcemanager.nm.liveness-monitor.expiry-interval-ms` 设置，以毫秒为单位)没有收到一条心跳信息，资源管理器将会通知停止发送心跳信息的节点管理器，并且将其从自己的节点池中移除以调度启用容器。

在失败的节点管理器上运行的所有任务或 application master 都用前两节描述的机

制进行恢复。另外，对于那些曾经在失败的节点管理器上运行且成功完成的 map 任务，如果属于未完成的作业，那么 application master 会安排它们重新运行。这是由于这些任务的中间输出驻留在失败的节点管理器的本地文件系统中，可能无法被 reduce 任务访问的缘故。

如果应用程序的运行失败次数过高，那么节点管理器可能会被拉黑，即使节点管理自己并没有失败过。由 application master 管理黑名单，对于 MapReduce，如果一个节点管理器上有超过三个任务失败，application master 就会尽量将任务调度到不同的节点上。用户可以通过作业属性 `mapreduce.job.maxtaskfailures.per.tracker` 设置该阈值。

注意，在本书写作时，资源管理器不会执行对应用程序的拉黑操作，因此新作业中的任务可能被调度到故障节点上，即使这些故障节点已经被运行早期作业的 application master 拉黑。

7.2.4 资源管理器运行失败

资源管理器失败是非常严重的问题，没有资源管理器，作业和任务容器将无法启动。在默认的配置中，资源管理器是个单点故障，这是由于在机器失败的情况下(尽管不太可能发生)，所有运行的作业都失败且不能被恢复。

为获得高可用性(HA)，在双机热备配置下，运行一对资源管理器是必要的。如果主资源管理器失败了，那么备份资源管理器能够接替，且客户端不会感到明显的中断。

关于所有运行中的应用程序的信息存储在一个高可用的状态存储区中(由 ZooKeeper 或 HDFS 备份)，这样备机可以恢复出失败的主资源管理器的关键状态。节点管理器信息没有存储在状态存储区中，因为当节点管理器发送它们的第一个心跳信息时，节点管理器的信息能以相当快的速度被新的资源管理器重构。(同样要注意，由于任务是由 application master 管理的，因此任务不是资源管理器的状态的一部分。这样，要存储的状态量比 MapReduce 1 中 jobtracker 要存储的状态量更好管理。)

当新资源管理器启动后，它从状态存储区中读取应用程序的信息，然后为集群中运行的所有应用程序重启 application master 。这个行为不被计为失败的应用程序尝试(所以不会计入 `yarn.resourcemanager.am.max-attempts`)，这是因为应用

程序并不是因为程序代码错误而失败，而是被系统强行中止的。实际情况中，application master 重启不是 MapReduce 应用程序的问题，因为它们是恢复已完成的任务的工作(详见 7.2.2 节)。

资源管理器从备机到主机的切换是由故障转移控制器(failover controller)处理的。默认的故障转移控制器是自动工作的，使用 ZooKeeper 的 leader 选举机制(leader election)以确保同一时刻只有一个主资源管理器。不同于 HDFS 高可用性(详见 3.2.5 节)的实现，故障转移控制器不必是一个独立的进程，为配置方便，默认情况下嵌入在资源管理器中。故障转移也可以配置为手动处理，但不建议这样。

为应对资源管理器的故障转移，必须对客户和节点管理器进行配置，因为他们可能是在和两个资源管理器打交道。客户和节点管理器以轮询(round-robin)方式试图连接每一个资源管理器，直到找到主资源管理器。如果主资源管理器故障，他们将再次尝试直到备份资源管理器变成主机。

7.3 shuffle 和排序

MapReduce 确保每个 reducer 的输入都是按键排序的。系统执行排序、将 map 输出作为输入传给 reducer 的过程称为 *shuffle*。[①]在此，我们将学习 shuffle 是如何工作的，因为它有助于我们理解工作机制(如果需要优化 MapReduce 程序)。shuffle 属于不断被优化和改进的代码库的一部分，因此下面的描述有必要隐藏一些细节(也可能随时间而改变，目前是 0.20 版本)。从许多方面来看，shuffle 是 MapReduce 的“心脏”，是奇迹发生的地方。

7.3.1 map 端

map 函数开始产生输出时，并不是简单地将它写到磁盘。这个过程更复杂，它利用缓冲的方式写到内存并出于效率的考虑进行预排序。图 7-4 展示了这个过程。

每个 map 任务都有一个环形内存缓冲区用于存储任务输出。在默认情况下，缓冲区的大小为 100 MB，这个值可以通过改变 `mapreduce.task.io.sort.mb` 属性来调整。一旦缓冲内容达到阈值(`mapreduce.map.sort.spill.percent`，默认为 0.80，或 80%)，一

① 事实上，shuffle 这个说法并不准确。因为在某些语境中，它只代表 reduce 任务获取 map 输出的这部分过程。在这一小节，将其理解为从 map 产生输出到 reduce 消化输入的整个过程。

个后台线程便开始把内容溢出(spill)到磁盘。在溢出写到磁盘过程中，map 输出继续写到缓冲区，但如果在此期间缓冲区被填满，map 会被阻塞直到写磁盘过程完成。溢出写过程按轮询方式将缓冲区中的内容写到 `mapreduce.cluster.local.dir` 属性在作业特定子目录下指定的目录中。

图 7-4. MapReduce 的 shuffle 和排序

在写磁盘之前，线程首先根据数据最终要传的 reducer 把数据划分成相应的分区(partition)。在每个分区中，后台线程按键进行内存中排序，如果有一个 combiner 函数，它就在排序后的输出上运行。运行 combiner 函数使得 map 输出结果更紧凑，因此减少写到磁盘的数据和传递给 reducer 的数据。

每次内存缓冲区达到溢出阈值，就会新建一个溢出文件(spill file)，因此在 map 任务写完其最后一个输出记录之后，会有几个溢出文件。在任务完成之前，溢出文件被合并成一个已分区且已排序的输出文件。配置属性 `mapreduce.task.io.sort.factor` 控制着一次最多能合并多少流，默认值是 10。

如果至少存在 3 个溢出文件(通过 `mapreduce.map.combine.minspills` 属性设置)时，则 combiner 就会在输出文件写到磁盘之前再次运行。前面曾讲过，combiner 可以在输入上反复运行，但并不影响最终结果。如果只有 1 或 2 个溢出文件，那么由于 map 输出规模减少，因而不值得调用 combiner 带来的开销，因此不会为该 map 输出再次运行 combiner。

在将压缩 map 输出写到磁盘的过程中对它进行压缩往往是个很好的主意，因为这样会写磁盘的速度更快，节约磁盘空间，并且减少传给 reducer 的数据量。在默认情况下，输出是不压缩的，但只要将 `mapreduce.map.output.compress` 设置为

true，就可以轻松启用此功能。使用的压缩库由 mapreduce.map.output.compress.codec 指定，要想进一步了解压缩格式，请参见 5.2 节。

reducer 通过 HTTP 得到输出文件的分区。用于文件分区的工作线程的数量由任务的 mapreduce.shuffle.max.threads 属性控制，此设置针对的是每一个节点管理器，而不是针对每个 map 任务。默认值 0 将最大线程数设置为机器中处理器数量的两倍。

7.3.2 reduce 端

现在转到处理过程的 reduce 部分。map 输出文件位于运行 map 任务的 tasktracker 的本地磁盘(注意，尽管 map 输出经常写到 map tasktracker 的本地磁盘，但 reduce 输出并不这样)，现在，tasktracker 需要为分区文件运行 reduce 任务。并且，reduce 任务需要集群上若干个 map 任务的 map 输出作为其特殊的分区文件。每个 map 任务的完成时间可能不同，因此在每个任务完成时，reduce 任务就开始复制其输出。这就是 reduce 任务的复制阶段。reduce 任务有少量复制线程，因此能够并行取得 map 输出。默认值是 5 个线程，但这个默认值可以修改设置 mapreduce.reduce.shuffle. parallelcopies 属性即可。

reducer 如何知道要从哪台机器取得 map 输出呢？

map 任务成功完成后，它们会使用心跳机制通知它们的 application master。因此，对于指定作业，application master 知道 map 输出和主机位置之间的映射关系。reducer 中的一个线程定期询问 master 以便获取 map 输出主机的位置，直到获得所有输出位置。

由于第一个 reducer 可能失败，因此主机并没有在第一个 reducer 检索到 map 输出时就立即从磁盘上删除它们。相反，主机会等待，直到 application master 告知它删除 map 输出，这是作业完成后执行的。

如果 map 输出相当小，会被复制到 reduce 任务 JVM 的内存(缓冲区大小由 mapreduce.reduce.shuffle.input.buffer.percent 属性控制，指定用于此用途的堆空间的百分比)，否则，map 输出被复制到磁盘。一旦内存缓冲区达到阈值大小(由 mapreduce.reduce.shuffle.merge.percent 决定)或达到 map 输出阈值(由 mapreduce.reduce.merge.inmem.threshold 控制)，则合并后溢出写到磁盘中。如果指定 combiner，则在合并期间运行它以降低写入硬盘的数据量。

随着磁盘上副本增多，后台线程会将它们合并为更大的、排好序的文件。这会为后面的合并节省一些时间。注意，为了合并，压缩的 map 输出(通过 map 任务)都

必须在内存中被解压缩。

复制完所有 map 输出后，reduce 任务进入排序阶段(更恰当的说法是合并阶段，因为排序是在 map 端进行的)，这个阶段将合并 map 输出，维持其顺序排序。这是循环进行的。比如，如果有 50 个 map 输出，而合并因子是 10(10 为默认设置，由 `mapreduce.task.io.sort.factor` 属性设置，与 map 的合并类似)，合并将进行 5 趟。每趟将 10 个文件合并成一个文件，因此最后有 5 个中间文件。

在最后阶段，即 reduce 阶段，直接把数据输入 reduce 函数，从而省略了一次磁盘往返行程，并没有将这 5 个文件合并成一个已排序的文件作为最后一趟。最后的合并可以来自内存和磁盘片段。

每趟合并的文件数实际上比示例中展示有所不同。目标是合并最小数量的文件以便满足最后一趟的合并系数。因此如果有 40 个文件，我们不会在四趟中每趟合并 10 个文件从而得到 4 个文件。相反，第一趟只合并 4 个文件，随后的三趟合并完整的 10 个文件。在最后一趟中，4 个已合并的文件和余下的 6 个(未合并的)文件合计 10 个文件。该过程如图 7-5 所述。

图 7-5. 通过合并因子 10 有效地合并 40 个文件片段

注意，这并没有改变合并次数，它只是一个优化措施，目的是尽量减少写到磁盘的数据量，因为最后一趟总是直接合并到 reduce。

在 reduce 阶段，对已排序输出中的每个键调用 reduce 函数。此阶段的输出直接写到输出文件系统，一般为 HDFS。如果采用 HDFS，由于节点管理器也运行数据节点，所以第一个块复本将被写到本地磁盘。

7.3.3 配置调优

现在我们已经有比较好的基础来理解如何调优 shuffle 过程来提高 MapReduce 性能。表 7-1 和表 7-2 总结了相关设置和默认值，这些设置以作业为单位(除非有特别说明)，默认值适用于常规作业。

表 7-1. map 端的调优属性

属性名称	类型	默认值	说明
mapreduce.task.io.sort.mb	int	100	排序 map 输出时所使用的内存缓冲区的大小，以兆字节为单位
mapreduce.map.sort.spill.percent	float	0.80	map 输出内存缓冲和用来开始磁盘溢出写过程的记录边界索引，这两者使用比例的阈值
mapreduce.task.io.sort.factor	int	10	排序文件时，一次最多合并的流数。这个属性也在 reduce 中使用。将此值增加到 100 是很常见的
mapreduce.map.combine.minspills	int	3	运行 combiner 所需的最少溢出文件数(如果已指定 combiner)
mapreduce.map.output.compress	Boolean	false	是否压缩 map 输出
mapreduce.map.output.compress.codec	Class name	org.apache.hadoop.io.compress.DefaultCodec	用于 map 输出的压缩编解码器
mapreduce.shuffle.max.threads	int	0	每个节点管理器的工作线程数，用于将 map 输出到 reducer。这是集群范围的设置，不能由单个作业设置。0 表示使用 Netty 默认值，即两倍于可用的处理器数

总的原则是给 shuffle 过程尽量多提供内存空间。然而，有一个平衡问题，也就是要确保 map 函数和 reduce 函数能得到足够的内存来运行。这就是为什么写 map 函数和 reduce 函数时尽量少用内存的原因，它们不应该无限使用内存(例如，应避免

在 map 中堆积数据)。

运行 map 任务和 reduce 任务的 JVM，其内存大小由 mapred.child. java.opts 属性设置。任务节点上的内存应该尽可能设置的大些，10.3.3 节讨论 YARN 和 MapReduce 中的内存设置时要讲到需要考虑哪些约束条件。

在 map 端，可以通过避免多次溢出写磁盘来获得最佳性能；一次是最佳的情况。如果能估算 map 输出大小，就可以合理地设置 mapreduce.task. io.sort.*属性来尽可能减少溢出写的次数。具体而言，如果可以，就要增加 mapreduce.task.io.sort.mb 的值。MapReduce 计数器(“SPILLED_RECORDS”，参见 9.1 节“计数器”)计算在作业运行整个阶段中溢出写磁盘的记录数，这对于调优很有帮助。注意，这个计数器包括 map 和 reduce 两端的溢出写。

在 reduce 端，中间数据全部驻留在内存时，就能获得最佳性能。在默认情况下，这是不可能发生的，因为所有内存一般都预留给 reduce 函数。但如果 reduce 函数的内存需求不大，把 mapreduce.reduce.merge. inmem.threshold 设置为 0，把 mapreduce.reduce.input.buffer.percent 设置为 1.0(或一个更低的值，详见表 7-2)就可以提升性能。

表 7-2. reduce 端的调优属性

属性名称	类型	默认值	描述
mapreduce.reduce. shuffle.parallelcopies	int	5	用于把 map 输出复制到 reducer 的线程数
mapreduce.reduce. shuffle.maxfetchfailures	int	10	在声明失败之前，reducer 获取一个 map 输出所花的最大时间
mapreduce.task.io. sort.factor	int	10	排序文件时一次最多合并的流的数量。这个属性也在 map 端使用
mapreduce.reduce. shuffle.input.buffer. percent	float	0.70	在 shuffle 的复制阶段，分配给 map 输出的缓冲区占堆空间的百分比
mapreduce.reduce. shuffle.merge. percent	float	0.66	map 输出缓冲区(由 mapred.job.shuffle. input.buffer.percent 定义)的阈值使用比例，用于启动合并输出和磁盘溢出写的过程
mapreduce.reduce. merge.in mem.threshold	int	1000	启动合并输出和磁盘溢出写过程的 map 输出的阈值数。0 或更小的数意味着没有阈值限制，溢出写行为由 mapreduce.reduce. shuffle.merge.percent 单独控制

续表

属性名称	类型	默认值	描述
mapreduce.reduce.input.buffer.percent	float	0.0	在 reduce 过程中，在内存中保存 map 输出的空间占整个堆空间的比例。reduce 阶段开始时，内存中的 map 输出大小不能大于这个值。默认情况下，在 reduce 任务开始之前，所有 map 输出都合并到磁盘上，以便为 reducer 提供尽可能多的内存。然而，如果 reducer 需要的内存较少，可以增加此值来最小化访问磁盘的次数

2008 年 4 月，Hadoop 在通用 TB 字节排序基准测试中获胜(详见 1.6 节的介绍)，它使用了一个优化方法，即将中间数据保存在 reduce 端的内存中。

更常见的情况是，Hadoop 使用默认为 4 KB 的缓冲区，这是很低的，因此应该在集群中增加这个值(通过设置 io.file.buffer.size，详见 10.3.5 节)。

7.4 任务的执行

在 7.1 节介绍剖析 MapRedue 作业运行机制时，我们结合整个作业的背景知道了 MapReduce 系统是如何执行任务的。在本小节，我们将了解 MapReduce 用户对任务执行的更多的控制。

7.4.1 任务执行环境

Hadoop 为 map 任务或 reduce 任务提供运行环境相关信息。例如，map 任务可以知道它处理的文件的名称(参见 8.2.1 节)，map 任务或 reduce 任务可以得知任务的尝试次数。表 7-3 中的属性可以从作业的配置信息中获得，在老版本的 MapReduce API 中通过为 Mapper 或 Reducer 提供一个 configure()方法实现(其中，配置信息作为参数进行传递)，便可获得这一信息。在新版本的 API 中，这些属性可以从传递给 Mapper 或 Reducer 的所有方法的相关对象中获取。

表 7-3. 任务执行环境的属性

属性名称	类型	说明	范例
mapreduce.job.id	string	作业 ID[1]	job_200811201130_0004
mapreduce.task.id	string	任务 ID	task_200811201130_ 0004_m_000003
mapreduce.task.attemp.id	string	任务尝试 ID	attempt_2008112011300004_m _000003_0
mapreduce.task.partition	int	作业中任务的索引	3
mapreduce.task.ismap	boolean	此任务是否是 map 任务	true

Streaming 环境变量

Hadoop 设置作业配置参数作为 Streaming 程序的环境变量。但它用下划线来代替非字母数字的符号，以确保名称的合法性。下面这个 Python 表达式解释了如何用 Python Streaming 脚本来检索 `mapreduce.job.id` 属性的值。

```
os.environ["mapreduce_job_id"]
```

也可以应用 Streaming 启动程序的`-cmdenv` 选项，来设置 MapReduce 所启动的 Streaming 进程的环境变量（一次设置一个变量）。比如，下面的语句设置了 MAGIC_PARAMETER 环境变量：

```
-cmdenv MAGIC_PARAMETER=abracadabra
```

7.4.2 推测执行

MapReduce 模型将作业分解成任务，然后并行地运行任务以使作业的整体执行时间少于各个任务顺序执行的时间。这使作业执行时间对运行缓慢的任务很敏感，因为只运行一个缓慢的任务会使整个作业所用的时间远远长于执行其他任务的时间。当一个作业由几百或几千个任务组成时，可能出现少数“拖后腿”的任务，这是很常见的。

任务执行缓慢可能有多种原因，包括硬件老化或软件配置错误，但是，检测具体原因很困难，因为任务总能够成功完成，尽管比预计执行时间长。Hadoop 不会尝试诊断或修复执行慢的任务，相反，在一个任务运行比预期慢的时候，它会尽量

① 格式描述参见 6.5.2 节的补充内容“作业、任务和任务任务尝试 ID”。

检测，并启动另一个相同的任务作为备份。这就是所谓的任务的“推测执行”(speculative execution)。

必须认识到一点：如果同时启动两个重复的任务，它们会互相竞争，导致推测执行无法工作。这对集群资源是一种浪费。相反，调度器跟踪作业中所有相同类型(map 和 reduce)任务的进度，并且仅仅启动运行速度明显低于平均水平的那一小部分任务的推测副本。一个任务成功完成后，任何正在运行的重复任务都将被中止，因为已经不再需要它们了。因此，如果原任务在推测任务前完成，推测任务就会被终止；同样，如果推测任务先完成，那么原任务就会被中止。

推测执行是一种优化措施，它并不能使作业的运行更可靠。如果有一些软件缺陷会造成任务挂起或运行速度减慢，依靠推测执行来避免这些问题显然是不明智的，并且不能可靠地运行，因为相同的软件缺陷可能会影响推测式任务。应该修复软件缺陷，使任务不会挂起或运行速度减慢。

在默认情况下，推测执行是启用的。可以基于集群或基于每个作业，单独为 map 任务和 reduce 任务启用或禁用该功能。相关的属性如表 7-4 所示。

表 7-4. 推测执行的属性

属性名称	类型	默认值	描述
mapreduce.map.speculative	boolean	true	如果任务运行变慢，该属性决定着是否要启动 map 任务的另外一个实例
mapreduce.reduce.speculative	boolean	true	如果任务运行变慢，该属性决定着是否要启动 reduce 任务的另一个实例
Yarn.app.mapreduce.am.job.speculator.class	Class	Org.apache.hadoop.mapreduce.v2.app.speculate.DefaultSpeculator	Speculator 类实现推测执行策略(只针对 MapReduce2)
Yarn.app.mapreduce.am.job.estimator.class	Class	Org.apache.hadoop.mapreduce.v2.app.speculate.LegacyTaskRuntimeEstimator	Speculator 实例使用的 TaskRuntimeEstimator 的实现，提供任务运行时间的估计值(只针对 MapReduce 2)

为什么会想到关闭推测执行？推测执行的目的是减少作业执行时间，但这是以集群效率为代价的。在一个繁忙的集群中，推测执行会减少整体的吞吐量，因为冗

余任务的执行时会减少作业的执行时间。因此，一些集群管理员倾向于在集群上关闭此选项，而让用户根据个别作业需要而开启该功能。Hadoop 老版本尤其如此，因为在调度推测任务时，会过度使用推测执行方式。

对于 reduce 任务，关闭推测执行是有益的，因为任意重复的 reduce 任务都必须将取得 map 输出作为最先的任务，这可能会大幅度地增加集群上的网络传输。

关闭推测执行的另一种情况是为了非幂等(nonidempotent)任务。然而在很多情况下，将任务写成幂等的并使用 OutputCommitter 来提升任务成功时输出到最后位置的速度，这是可行的。详情将在下一节介绍。

7.4.3 关于 OutputCommitters

Hadoop MapReduce 使用一个提交协议来确保作业和任务都完全成功或失败。这个行为通过对作业使用 OutputCommitter 来实现，在老版本 MapReduce API 中通过调用 JobConf 的 setOutputCommitter()或配置中的 mapred.output.committer.class 来设置。在新版本的 MapReduce API 中，OutputCommitter 由 OutputFormat 通过它的 getOutputCommitter()方法确定。默认值为 FileOutputCommitter，这对基于文件的 MapReduce 是适合的。可以定制已有的 OutputCommitter，或者在需要时还可以写一个新的实现以完成对作业或任务的特别设置或清理。

OutputCommitter的 API 如下所示(在新旧版本中的 MapReduce API 中)：

```
public abstract class OutputCommitter {
public abstract void setupJob(JobContext jobContext) throws IOException;
public void commitJob(JobContext jobContext) throws IOException { }
public void abortJob(JobContext jobContext, JobStatus.State state)
  throws IOException { }

public abstract void setupTask(TaskAttemptContext taskContext)
  throws IOException;
public abstract boolean needsTaskCommit(TaskAttemptContext taskContext)
  throws IOException;
public abstract void commitTask(TaskAttemptContext taskContext)
  throws IOException;
public abstract void abortTask(TaskAttemptContext taskContext)
  throws IOException;
}
}
```

setupJob()方法在作业运行前调用，通常用来执行初始化操作。当 OutputCommitter 设为 FileOutputCommitter 时，该方法创建最终的输出目录${mapreduce.output.

fileoutputformat.outputdir}，并且为任务输出创建一个临时的工作空间，*_temporary*，作为最终输出目录的子目录。

如果作业成功，就调用 commitJob()方法，在默认的基于文件的实现中，它用于删除临时的工作空间并在输出目录中创建一个名为*_SUCCESS* 的隐藏的标志文件，以此告知文件系统的客户端该作业成功完成了。如果作业不成功，就通过状态对象调用 abortJob()，意味着该作业是否失败或终止(例如由用户终止)。在默认的实现中，将删除作业的临时工作空间。

在任务级别上的操作与此类似。在任务执行之前先调用 setupTask()方法，默认的实现不做任何事情，因为针对任务输出命名的临时目录在写任务输出的时候被创建。

任务的提交阶段是可选的，并通过从 needsTaskCommit()返回的 false 值关闭它。这使得执行框架不必为任务运行分布提交协议，也不需要 commitTask()或者 abortTask()。当一个任务没有写任何输出时，FileOutputCommitter 将跳过提交阶段。

如果任务成功，就调用 commitTask()，在默认的实现中它将临时的任务输出目录(它的名字中有任务尝试的 ID，以此避免任务尝试间的冲突)移动到最后的输出路径 ${mapreduce.output.fil eoutputformat.outputdir}。否则，执行框架调用 abortTask()，它负责删除临时的任务输出目录。

执行框架保证特定任务在有多次任务尝试的情况下，只有一个任务会被提交，其他的则被取消。这种情况是可能出现的，因为第一次尝试出于某个原因而失败(这种情况下将被取消)，提交的是稍后成功的尝试。另一种情况是如果两个任务尝试作为推测副本同时运行，则提交先完成的，而另一个被取消。

任务附属文件

对于 map 任务和 reduce 任务的输出，常用的写方式是通过 OutputCollector 来收集键-值对。有一些应用需要比单个键-值对模式更灵活的方式，因此直接将 map 或 reduce 任务的输出文件写到分布式文件系统中，如 HDFS。还有其他方法用于产生多个输出，详情参见 8.3.3 节。

注意，要确保同一个任务的多个实例不向同一个文件进行写操作。如前所述，OutputCommitter 协议解决了该问题。如果应用程序将附属文件导入其任务的工作

目录中，那么成功完成的这些任务就会将其附属文件自动推送到输出目录，而失败的任务，其附属文件则被删除。

任务通过从作业配置文件中查询 mapreduce.task.out put.dir 属性值找到其工作目录。另一种方法，使用 Java API 的 MapReduce 程序可以调用 FileOutputFormat 上的 getWorkOutputPath()静态方法获取描述工作目录的 Path 对象。执行框架在执行任务之前首先创建工作目录，因此不需要我们创建。

举一个简单的例子，假设有一个程序用来转换图像文件的格式。一种实现方法是用一个只有 map 任务的作业，其中每个 map 指定一组要转换的图像(可以使用 NlineInputFormat，详情参见 8.2.2 节)。如果 map 任务把转换后的图像写到其工作目录，那么在任务成功完成之后，这些图像会被传到输出目录。

第 8 章

MapReduce 的类型与格式

MapReduce 数据处理模型非常简单：map 和 reduce 函数的输入和输出是键-值对。本章深入讨论 MapReduce 模型，重点介绍各种类型的数据(从简单文本到结构化的二进制对象)如何在 MapReduce 中使用。

8.1 MapReduce 的类型

Hadoop 的 MapReduce 中，map 函数和 reduce 函数遵循如下常规格式：

```
map: (K1, V1) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)
```

一般来说，map 函数输入的键/值类型(K1 和 V1)不同于输出类型(K2 和 V2)。然而，reduce 函数的输入类型必须与 map 函数的输出类型相同，但 reduce 函数的输出类型(K3 和 V3)可以不同于输入类型。例如以下 Java 接口代码：

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
  public class Context extends MapContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
    // ...
  }
  protected void map(KEYIN key, VALUEIN value, Context context)
                    throws IOException, InterruptedException {
    // ...
  }
}
public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {

  public class Context extends ReducerContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
    // ...
  }
```

```
  protected void reduce(KEYIN key, Iterable<VALUEIN> values,
                        Context context) throws IOException,
InterruptedException {
  // ...
  }
}
```

Context 类对象用于输出键-值对，因此它们通过输出类型参数化，这样 write()方法的说明如下：

```
Public void write(KEYOUT key, VALUEOUT value)
    throws IOException, InterruptedException
```

由于 Mapper 和 Reducer 是单独的类，因此类型参数可能会不同，所以 Mapper 中 KEYIN(say)的实际类型可能与 Reducer 中同名的类型参数(KEYIN)的类型不一致。例如，在前面章节的求最高温度例子中，Mapper 中 KEYIN 为 LongWritable 类型，而 Reducer 中为 Text 类型。

类似的，即使 map 输出类型与 reduce 的输入类型必须匹配，但这在 Java 编译器中并不是强制要求的。

类型参数(type parameter)的命名不同于抽象类型的命名(KEYIN 对应于 K1 等)，但它们的形式是相同的。

如果使用 combiner 函数，它与 reduce 函数(是 Reducer 的一个实现)的形式相同，不同之处是它的输出类型是中间的键-值对类型(K2 和 V2)，这些中间值可以输入 reduce 函数：

```
map:  (K1, V1) →list(K2, V2)
combiner:  (K2,  list(V2))  →list(K2, V2)
reduce:  (K2,  list(V2)) → list(K3, V3)
```

combiner 函数与 reduce 函数通常是一样的，在这种情况下，K3 与 K2 类型相同，V3 与 V2 类型相同。

partition 函数对中间结果的键-值对(K2 和 V2)进行处理，并且返回一个分区索引(partition index)。实际上，分区由键单独决定 (值被忽略)。

```
partition: (K2, V2) → integer
```

或用 Java：

```
public abstrack class Partitioner<KEY, VALUE> {
  public abstract int gerPartition(KRY key, VALUE value, int numPartitions);
}
```

旧版本 API 中 MapReduce 的用法说明

在旧版本的 API(见附录D)中，MapReduce 的用法非常类似，类型参数的实际命名也为 K1、V1 等。在新旧版本 API 中类型上的约束也是完全一样的：

```
public interface Mapper<K1, V1, K2, V2> extends JobConfigurable, Closeable {
  void map(K1 key, V1 value, OutputCollector<K2, V2> output, Reporter reporter) throws
IOException;
}
public interface Reducer<K2, V2, K3, V3> extends JobConfigurable, Closeable {
  void reduce(K2 key, Iterator<V2> values,
    OutputCollector<K3, V3> output, Reporter reporter) throws IOException;
}
public interface Partitioner<K2, V2> extends JobConfigurable {
  int getPartition(K2 key, V2 value, int numPartitions);
}
```

这些理论对配置 MapReduce 作业有帮助吗？表 8 -1 总结了新版本 API 的配置选项(表 8-2 为旧版本 API 的)，把属性分为可以设置类型的属性和必须与类型相容的属性。

输入数据的类型由输入格式进行设置。例如，对应于 TextInputFormat 的键类型是 LongWritable，值类型是 Text。其他的类型通过调用 Job 类的方法来进行显式设置(旧版本 API 中使用 JobConf 类的方法)。如果没有显式设置，则中间的类型默认为(最终的)输出类型，也就是默认值 LongWritable 和 Text。因此，如果 K2 与 K3 是相同类型，就不需要调用 setMapOutputKeyClass()，因为它将调用 setOutputKeyClass()来设置；同样，如果 V2 与 V3 相同，只需要使用 setOutputValueClass()。

这些为中间和最终输出类型进行设置的方法似乎有些奇怪。为什么不能结合 mapper 和 reducer 导出类型呢？原来，Java 的泛型机制有很多限制：类型擦除(type erasure)导致运行过程中类型信息并非一直可见，所以 Hadoop 不得不进行明确设定。这也意味着可能会在 MapReduce 配置的作用中遇到不兼容的类型，因为这些配置在编译时无法检查。与 MapReduce 类型兼容的设置列在表 8-1 中。类型冲突是在作业执行过程中被检测出来的，所以一个比较明智的做法是先用少量数据跑一次测试任务，发现并修正任何一个类型不兼容的问题。

表 8-1. 新的 MapReduce API 中的设置类型

属性	属性设置方法	输入类型		中间类型		输出类型	
		K1	V1	K2	V2	K3	V3
可以设置类型的属性		*	*				
mapreduce.job.inputformat.class	setInputFormatClass()		*				
mapreduce.map.output.key.class	setMapOutputKeyClass()			*			
mapreduce.map.output.value.class	setMapOutputValueClass()				*		
mapreduce.job.output.key.class	setOutputKeyClass()					*	
mapreduce.job.output.value.class	setOutputValueClass()						*
类型必须一致的属性							
mapreduce.job.map.class	setMapperClass()	*	*	*	*		
mapreduce.job.combine.class	setCombinerClass()			*	*		
mapreduce.job.partitioner.class	setPartitionerClass()			*	*		
mapreduce.job.output.key.comparator.class	setSortComparatorClass()						
mapreduce.job.output.group.comparator.class	setGroupingComparatorClass()			*			
mapreduce.job.reduce.class	setReducerClass()			*	*	*	*
mapreduce.job.outputformat.class	setOutputFormatClass()					*	*

表 8-2　旧版本 MapReduce API 的设置类型

属性	属性设置方法	输入类型		中间类型		输出类型	
		K1	V1	K2	V2	K3	V3
可以设置类型的属性							
mapred.input.format.class	setInputFormat()	*	*				
mapred.mapoutput.key.class	setMapOutputKeyClass()			*			
mapred.mapoutput.value.class	setMapOutputValueClass()				*		
mapred.output.key.class	setOutputKeyClass()					*	
mapred.output.value.class	setOutputValueClass()						*
类型必须一致的属性							
mapred.mapper.class	setMapperClass()	*	*	*	*		
mapred.map.runner.class	setMapRunnerClass()						
mapred.combiner.class	setCombinerClass()			*	*		
mapred.partitioner.class	setPartitionerClass()			*	*		
mapred.output.key.comparator.class	setOutputkeyComparatorClass()			*			
mapred.output.value.groupfn.class	setOutputValueGroupingComparator()			*			
mapred.reducer.class	setReducerClass()			*	*	*	*
mapred.output.format.class	setOutputFormat()					*	*

8.1.1 默认的 MapReduce 作业

如果不指定 mapper 或 reducer 就运行 MapReduce，会发生什么情况?我们运行一个最简单的 MapReduce 程序来看看：

```
public class MinimalMapReduce extends Configured implements Tool {

@Override
public int run(String[] args) throws Exception {
  if (args.length != 2) {
    System.err.printf("Usage: %s [generic options] <input> <output>\n",
        getClass().getSimpleName());
    ToolRunner.printGenericCommandUsage(System.err);
  return -1;
}

Job job = new Job(getConf());
job.setJarByClass(getClass());
FileInputFormat.addInputPath(conf, new Path(args[0]));
FileOutputFormat.setOutputPath(conf, new Path(args[1]));
return job.waitForCompletion(true) ? 0 : 1;
}

public static void main(String[] args) throws Exception {
  int exitCode = ToolRunner.run(new MinimalMapReduce(), args);
  System.exit(exitCode);
 }
}
```

我们唯一设置的是输入路径和输出路径。在气象数据的子集上运行以下命令：

```
% hadoop MinimalMapReduce "input/ncdc/all/190{1,2}.gz" output
```

输出目录中得到命名为 *part-r-00000* 的输出文件。这个文件的前几行如下(为适应页面而进行了截断处理)：

```
0→0029029070999991901010106004+64333+023450FM-12+000599999V0202701N01591...
0→0035029070999991902010106004+64333+023450FM-12+000599999V0201401N01181...
135→0029029070999991901010113004+64333+023450FM-12+000599999V0202901N00821...
141→0035029070999991902010113004+64333+023450FM-12+000599999V0201401N01181...
270→0029029070999991901010120004+64333+023450FM-12+000599999V0209991C00001...
282→0035029070999991902010120004+64333+023450FM-12+000599999V0201401N01391...
```

每一行以整数开始，接着是制表符(Tab)，然后是一段原始气象数据记录。虽然这并不是一个有用的程序，但理解它如何产生输出确实能够洞悉 Hadoop 是如何使用默认设置运行 MapReduce 作业的。范例 8-1 的示例与前面 MinimalMapReduce 完成的事情一模一样，但是它显式地把作业环境设置为默认值。

范例 8-1. 最小的 MapReduce 驱动程序，默认值显式设置

```
public class MinimalMapReduceWithDefaults extends Configured implements Tool {

  @Override
  public int run(String[] args) throws IOException {
      Job job = JobBuilder.parseInputAnOutput(this, getConf(), args);
    if (job == null) {
      return -1;
    }

    job.setInputFormat(TextInputFormat.class);

    job.setMapperClass(Mapper.class);

    job.setMapOutputKeyClass(LongWritable.class);
    job.setMapOutputValueClass(Text.class);

    job.setPartitionerClass(HashPartitioner.class);

    job.setNumReduceTasks(1);
    job.setReducerClass(Reducer.class);

    job.setOutputKeyClass(LongWritable.class);
    job.setOutputValueClass(Text.class);

    job.setOutputFormat(TextOutputFormat.class);

    return job.waitForCompletion(true) ? 0 : 1;
    }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new MinimalMapReduceWithDefaults(), args);
    System.exit(exitCode);
    }
}
```

通过把打印使用说明的逻辑抽取出来并把输入/输出路径设定放到一个帮助方法中，实现对 run()方法的前几行进行了简化。几乎所有 MapReduce 驱动程序都有两个参数(输入与输出)，所以此处进行这样的代码约简是可行的。以下是 JobBuilder 类中的相关方法，供大家参考：

```
public static Job parseInputAndOutput(Tool tool, Configuration conf,
    String[] args) throws IOException {

  if (args.length != 2) {
    printUsage(tool, "<input> <output>");
    return null;
  }
  Job job = new Job(conf);
  job.setJarByClass(tool.getClass());
  FileInputFormat.addInputPath(job, new Path(args[0]));
```

```
FileOutputFormat.setOutputPath(job, new Path(args[1]));
return job;
}

public static void printUsage(Tool tool, String extraArgsUsage) {
  System.err.printf("Usage: %s [genericOptions] %s\n\n",
      tool.getClass().getSimpleName(), extraArgsUsage);
  GenericOptionsParser.printGenericCommandUsage(System.err);
}
```

回到范例 8-1 中的 MinimalMapReducewithDefaults 类，虽然有很多其他的默认作业设置，但加粗显示的部分是执行一个作业最关键的代码。接下来我们逐一讨论。

在默认的输入格式是 TextInputFormat，它产生的键类型是 LongWritable(文件中每行中开始的偏移量值)，值类型是 Text(文本行)。这也解释了最后输出的整数的含义：行偏移量。

默认的 mapper 是 Mapper 类，它将输入的键和值原封不动地写到输出中：

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
  protected void map(KEYIN key, VALUEIN value,
                     Context context) throws IOException, InterruptedException {
    context.write((KEYOUT) key, (VALUEOUT) value);
  }
}
```

Mapper 是一个泛型类型(generic type)，它可以接受任何键或值的类型。在这个例子中，map 的输入输出键是 LongWritable 类型，map 的输入输出值是 Text 类型。

默认的 partitioner 是 HashPartitioner，它对每条记录的键进行哈希操作以决定该记录应该属于哪个分区。每个分区由一个 reduce 任务处理，所以分区数等于作业的 reduce 任务个数：

```
public class HashPartitioner<K, V> extends Partitioner<K, V> {

  public int getPartition(K key, V value,
                          int numPartitions) {
    return (key.hashCode() & Integer.MAX_VALUE) % numPartitions;
  }
}
```

键的哈希码被转换为一个非负整数，它由哈希值与最大的整型值做一次按位与操作而获得，然后用分区数进行取模操作，来决定该记录属于哪个分区索引。

默认情况下，只有一个 reducer，因此，也就只有一个分区，在这种情况下，由于所有数据都放入同一个分区，partitioner 操作将变得无关紧要了。然而，如果有多个 reduce 任务，了解 HashPartitioner 的作用就非常重要。假设基于键的散列函数足够好，那么记录将被均匀分到若干个 reduce 任务中，这样，具有相同键的记录将由同一个 reduce 任务进行处理。

你可能已经注意到我们并没有设置 map 任务的数量。原因是该数量等于输入文件被划分成的分块数，这取决于输入文件的大小以及文件块的大小(如果此文件在 HDFS 中)。关于控制块大小的操作，可以参见 8.2.1 节。

选择 reducer 的个数

对 Hadoop 新手而言，单个 reducer 的默认配置很容易上手。但在真实的应用中，几乎所有作业都把它设置成一个较大的数字，否则由于所有的中间数据都会放到一个 reduce 任务中，作业处理极其低效。

为一个作业选择多少个 reducer 与其说是一门技术，不如说更多是一门艺术。由于并行化程度提高，增加 reducer 的数量能缩短 reduce 过程。然而，如果做过了，小文件将会更多，这又不够优化。一条经验法则是，目标 reducer 保持在每个运行 5 分钟左右、且产生至少一个 HDFS 块的输出比较合适。

默认的 reducer 是 Reducer 类型，它也是一个泛型类型，只是把所有的输入写到输出中：

```
public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
  protected void reduce(KEYIN key, Iterable<VALUEIN> values, Context context
                        Context context) throws IOException, InterruptedException {
    for (VALUEIN value: values) {
      context.write((KEYOUT) key, (VALUEOUT) value);
    }
  }
}
```

对于这个任务来说，输出的键是 LongWritable 类型，而值是 Text 类型。事实上，对于这个 MapReduce 程序来说，所有键都是 LongWritable 类型，所有值都是 Text 类型，因为它们是输入键/值，并且 map 函数和 reduce 函数是恒等函数。然而，大多数 MapReduce 程序不会一直用相同的键或值类型，所以就像上一节所描述的那样，必须配置作业来声明使用的类型。

记录在发送给 reducer 之前，会被 MapReduce 系统进行排序。在这个例子中，键是按照数值的大小进行排序的，因此来自输入文件中的行会被交叉放入一个合并

后的输出文件。

默认的输出格式是 TextOutputFormat，它将键和值转换成字符串并用制表符分隔开，然后一条记录一行地进行输出。这就是为什么输出文件是用制表符(Tab)分隔的，这是 TextOutputFormat 的特点。

8.1.2 默认的 Streaming 作业

在 Streaming 方式下，默认的作业与 Java 方式是相似的，但也有差别。基本形式如下：

```
% hadoop jar  $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
    -input input/ncdc/sample.txt \
    -output output \
    -mapper /bin/cat
```

如果我们开发一个非 Java 的 mapper，并且当前是默认的文本模式(-io text)，那么 Streaming 会做一些特殊的处理。它并不会把键传递给 mapper，而是只传递值。对于其他输入类型，将 stream.map.input.ignoreKey 设置为 true 也可以达到相同的效果。这样做事实上是非常有用的，因为键只是文件中的行偏移量，而值是行中的数据，这才是几乎所有应用都关心的内容。这个作业的效果就是对输入的值进行排序。

将更多的默认设置写出来，那么命令行看起来如下所示(注意，Streaming 使用的是旧版本 MapReduce API 类)：

```
% hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
    -input input/ncdc/sample.txt \
    -output output \
    -inputformat org.apache.hadoop.mapred.TextInputFormat \
    -mapper /bin/cat \
    -partitioner org.apache.hadoop.mapred.lib.HashPartitioner \
    -numReduceTasks 1 \
    -reducer org.apache.hadoop.mapred.lib.IdentityReducer \
    -outputformat org.apache.hadoop.mapred.TextOutputFormat
    -io text
```

参数-mapper 和参数-reducer 可以是一条命令或一个 Java 类。我们可以用-combiner 参数指定一个 combiner。

Streaming 中的键和值

Streaming 应用可以决定分隔符的使用，该分隔符用于通过标准输入把键-值对转换

为一串比特值发送到 map 函数或 reduce 函数。默认情况下是 Tab(制表符)，但是如果键或值中本身含有 Tab 分隔符，能将分隔符修改成其他符号是很有用的。

类似地，当 map 和 reduce 输出结果键-值对时，也需要一个可配置的分隔符来进行分隔。更进一步，来自输出的键可以由多个字段进行组合：它可以由一条记录的前 *n* 个字段组成(由 `stream.num.map.output.key.fields` 或 `stream.num.reduce.output.key.fields` 进行定义)，剩下的字段就是值。例如，一个 Streaming 处理的输出是“a，b，c”(分隔符是逗号)，*n* 设为 2，则键解释为“a、b”，而值是“c”。

mapper 和 reducer 的分隔符是单独配置的。这些属性可以参见表 8-3，数据流图可以参见图 8-1。

表 8-3. Streaming 的分隔符属性

属性名称	类型	默认值	描述
stream.map.input.field.separator	String	\t	此分隔符用于将输入键/值字符串作为字节流传递到流 map
stream.map.output.field.separator	String	\t	此分隔符用于把流 map 处理的输出分割成 map 输出需要的键/值字符串
tream.num.map.output.key.fields	int	1	由 stream.map.output.field.separator 分隔的字段数，这些字段作为 map 输出键
stream.reduce.input.field.separator	String	\t	此分隔符用于将输入键/值字符串作为字节流传递到流 reduce
stream.reduce.output.field.separator	String	\t	此分隔符用于将来自流 reduce 处理的输出分成 reduce 最终输出需要的键/值字符串
stream.num.reduce.output.key.fields	int	1	由 stream.reduce.output.field.separator 分隔的字段数，这些字段作为 reduce 输出键

图 8-1. 在 Streaming MapReduce 作业中使用分隔符的位置

这些属性与输入和输出的格式无关。例如，如果 stream.reduce.output.field.separator 被设置成冒号，reduce 的 stream 过程就把 a：b 行写入标准输出，那么 Streaming 的 reducer 就会知道 a 作为键，b 作为值。如果使用标准的 TextOutputFormat，那么这条记录就用 Tab 将 a 和 b 分隔开并写到输出文件。可以设置 mapreduce.output.textoutput format.separator 来修改 TextOutputFormat 的分隔符。

8.2 输入格式

从一般的文本文件到数据库，Hadoop 可以处理很多不同类型的数据格式。本节将探讨数据格式问题。

8.2.1 输入分片与记录

第 2 章中讲过，一个**输入分片**(split)就是一个由单个 map 操作来处理的输入块。每一个 map 操作只处理一个输入分片。每个分片被划分为若干个记录，每条记录就是一个键-值对，map 一个接一个地处理记录。输入分片和记录都是逻辑概念，不必将它们对应到文件，尽管其常见形式都是文件。在数据库的场景中，一个输入分片可以对应于一个表上的若干行，而一条记录对应到一行(如同 DBInputFormat，这种输入格式用于从关系型数据库读取数据)。

输入分片在 Java 中表示为 InputSplit 接口 (和本章提到的所有类一样，它也在 org.apache.hadoop.mapreduce 包中)。[①]

```
public abstract class InputSplit {
    public abstract long getLength() throws IOException, InterruptedException;
    public abstract String[] getLocations() throws IOException,
      InterruptedException;
}
```

InputSplit 包含一个以字节为单位的长度和一组存储位置(即一组主机名)。注意，分片并不包含数据本身，而是指向数据的引用(reference)。存储位置供 MapReduce 系统使用以便将 map 任务尽量放在分片数据附近，而分片大小用来排序分片，以便优先处理最大的分片，从而最小化作业运行时间(这也是贪婪近似算

① 如果是老版本的 MapReduce API，这些类包含在 org.apache.hadoop.mapred 中。

法的一个实例)。

MapReduce 应用开发人员不必直接处理 InputSplit，因为它是由 InputFormat 创建的(InputFormat 负责创建输入分片并将它们分割成记录)。在我们探讨 InputFormat 的具体例子之前，先简单看一下它在 MapReduce 中的用法。接口如下：

```
public abstract class InputFormat<K, V> {
  public abstract List<InputSplit> getSplits(JobContext context)
      throws IOException, InterruptedException;

public abstract RecordReader<K, V>
  createRecordReader(InputSplit split, TaskAttemptContext context)
      throws IOException, InterruptedException;
}
```

运行作业的客户端通过调用 getSplits()计算分片，然后将它们发送到 application master，application master 使用其存储位置信息来调度 map 任务从而在集群上处理这些分片数据。map 任务把输入分片传给 InputFormat 的 createRecordReader()方法来获得这个分片的 RecordReader。RecordReader 就像是记录上的迭代器，map 任务用一个 RecordReader 来生成记录的键-值对，然后再传递给 map 函数。查看 Mapper 的 run()方法可以看到这些情况：

```
public void run(Context context) throws IOException, InterruptedException {
  setup(context);
  while (context.nextKeyValue()) {
    map(context.getCurrentKey(), context.getCurrentValue(), context);
  }
  cleanup(context);
}
```

运行 setup()之后，再重复调用 Context 上的 nextKeyValue()(委托给 RecordRader 的同名方法)为 mapper 产生键-值对象。通过 Context，键/值从 RecordReader 中被检索出并传递给 map()方法。当 reader 读到 stream 的结尾时，nextKeyValue()方法返回 false，map 任务运行其 cleanup()方法，然后结束。

尽管这段代码没有显示，由于效率的原因，RecordReader 程序每次调用 gerCurrentKey()和 getCurrentValue()时将返回相同的键-值对象。只是这些对象的内容被 reader 的 nextKeyValue()方法改变。用户对此可能有些惊讶，他们可能希望键/值是不可变的且不会被重用。在 map()方法之外有对键/值的引用时，这可能引起问题，因为它的值会在没有警告的情况下被改变。如果确

实需要这样的引用，那么请保存你想保留的对象的一个副本，例如，对于 Text 对象，可以使用它的复制构造函数：new Text(value)。

这样的情况在 reducer 中也会发生。reducer 迭代器中的值对象被反复使用，所以，在调用迭代器之间，一定要复制任何需要保留的任何对象(参见范例 9-11)。

最后，注意Mapper 的run()方法是公共的，可以由用户定制。MultithreadedMapRunner 是另一个 MapRunnable 接口的实现，它可以使用可配置个数的线程来并发运行多个 mapper(用 mapreduce.mapper. multithreadedmapper.threads 设置)。对于大多数数据处理任务来说，默认的执行机制没有优势。但是，对于因为需要连接外部服务器而造成单个记录处理时间比较长的 mapper 来说，它允许多个 mapper 在同一个 JVM 下以尽量避免竞争的方式执行。

1. FileInputFormat 类

FileInputFormat 是所有使用文件作为其数据源的 InputFormat 实现的基类(参见图 8-2)。它提供两个功能：一个用于指出作业的输入文件位置；一个是为输入文件生成分片的代码实现。把分片分割成记录的作业由其子类来完成。

图 8-2. InputFormat 类的层次结构

2. FileInputFormat 类的输入路径

作业的输入被设定为一组路径，这对限定输入提供了很强的灵活性。FileInputFormat 提供四种静态方法来设定 Job 的输入路径：

```
public static void addInputPath(Job job, Path path)
public static void addInputPaths(Job job, String commaSeparatedPaths)
public static void setInputPaths(Job job, Path... inputPaths)
public static void setInputPaths(Job job, String commaSeparatedPaths)
```

其中，addInputPath()和 addInputPaths()方法可以将一个或多个路径加入路径列表。可以分别调用这两种方法来建立路径列表。setInputPaths()方法一次设定完整的路径列表(替换前面调用中在 Job 上所设置的所有路径)。

一条路径可以表示一个文件、一个目录或是一个 glob，即一个文件和目录的集合。路径是目录的话，表示要包含这个目录下所有的文件，这些文件都作为作业的输入。关于 glob 的使用，3.5.5 节在讲到“文件模式”时有详细讨论。

一个被指定为输入路径的目录，其内容不会被递归处理。事实上，这些目录只包含文件：如果包含子目录，也会被解释为文件(从而产生错误)。处理这个问题的方法是：使用一个文件 glob 或一个过滤器根据命名模式(name pattern)限定选择目录中的文件。另一种方法是将 mapreduce.input.fileinputformat.input.dir.recursive 设置为 true 从而强制对输入目录进行递归地读取。

add 方法和 set 方法允许指定包含的文件。如果需要排除特定文件，可以使用 FileInputFormat 的 setInputPathFilter()方法设置一个过滤器。过滤器的详细讨论参见 3.5.5 节中对 PathFilter 的讨论。

即使不设置过滤器，FileInputFormat 也会使用一个默认的过滤器来排除隐藏文件(名称中以“.”和“_”开头的文件)。如果通过调用 setInputPathFilter()设置了过滤器，它会在默认过滤器的基础上进行过滤。换句话说，自定义的过滤器只能看到非隐藏文件。

路径和过滤器也可以通过配置属性来设置(参见表 8-4)，这对于 Streaming 作业来说很方便。Streaming 接口使用-input 选项来设置路径，所以通常不需要直接进行手动设置。

表 8-4. 输入路径和过滤器属性

属性名称	类型	默认值	描述
mapreduce.input.fileinputformat.inputdir	逗号分隔的路径	无	作业的输入文件。包含逗号的路径中的逗号由“\”符号转义。例如，glob {a,b}变成了{a\, b}
mapreduce.input.pathFilter.class	PathFilter 类名	无	应用于作业输入文件的过滤器

3. FileInputFormat 类的输入分片

假设有一组文件，FileInputFormat 如何把它们转换为输入分片呢？FileInputFormat 只分割大文件。这里的“大”指的是文件超过 HDFS 块的大小。分片通常与 HDFS 块大小一样，这在大多应用中是合理的；然而，这个值也可以通过设置不同的 Hadoop 属性来改变，如表 8-5 所示。

表 8-5. 控制分片大小的属性

属性名称	类型	默认值	描述
mapreduce.input.fileinputformat.split.minsize	int	1	一个文件分片最小的有效字节数
mapreduce.input.fileinputformat.split.maxsize[①]	long	Long.MAX_VALUE，即 9223372036854775807	一个文件分片中最大的有效字节数(以字节算)
dfs.blocksize	long	128 MB，即 134217728	HDFS 中块的大小(按字节)

① 这个属性在老版本的 MapReduce API 中没有出现(除了 CombineFileInputFormat)。然而，这个值是被间接计算的。计算方法是作业总的输入大小除以 map 任务数，该值由 mapreduce.job.maps（或 JobConf 上的 SetNumMapTasks()方法)设置。因为 map 任务的数目默认情况下是 1，所以，分片的最大值就是输入的大小

最小的分片大小通常是 1 个字节，不过某些格式可以使分片大小有一个更低的下界。例如，顺序文件在流中每次插入一个同步入口，所以，最小的分片大小不得不足够大以确保每个分片有一个同步点，以便 reader 根据记录边界进行重新同步。详见 5.4.1 节。

应用程序可以强制设置一个最小的输入分片大小：通过设置一个比 HDFS 块更大一些的值，强制分片比文件块大。如果数据存储在 HDFS 上，那么这样做是没有好处的，因为这样做会增加对 map 任务来说不是本地文件的文件块数。

最大的分片大小默认是由 Java 的 long 类型表示的最大值。只有把它的值被设置成小于块大小才有效果，这将强制分片比块小。

分片的大小由以下公式计算，参见FileInputFormat的computeSplitSize()方法：

```
max(minimumSize, min(maximumSize, blockSize))
```

在默认情况下：

```
minimumSize < blockSize < maximumSize
```

所以分片的大小就是 blocksize。这些参数的不同设置及其如何影响最终分片大小，请参见表 8-6 的详细说明。

表 8-6. 举例说明如何控制分片的大小

最小分片大小	最大分片大小	块的大小	分片大小	说明
1(默认值)	Long.MAX_VALUE (默认值)	128 MB (默认值)	128 MB	默认情况下，分片大小与块大小相同
1(默认值)	Long.MAX_VALUE (默认值)	256 MB	256 MB	增加分片大小最自然的方法是提供更大的 HDFS 块，通过 dfs.blocksize 或在构建文件时以单个文件为基础进行设置
256MB	Long.MAX VALUE (默认值)	128 MB (默认值)	256 MB	通过使最小分片大小的值大于块大小的方法来增大分片大小，但代价是增加了本地操作
1(默认值)	64 MB	128 MB (默认值)	64 MB	通过使最大分片大小的值大于块大小的方法来减少分片大小

4. 小文件与 CombineFileInputFormat

相对于大批量的小文件，Hadoop 更合适处理少量的大文件。一个原因是 FileInputFormat 生成的分块是一个文件或该文件的一部分。如果文件很小（“小”意味着比 HDFS 的块要小很多），并且文件数量很多，那么每次 map 任务只处理很少的输入数据，(一个文件)就会有很多 map 任务，每次 map 操作都会造成额外的开销。请比较一下把 1GB 的文件分割成 8 个 128 MB 块与分成 10000 个左右 100 KB 的文件。10000 个文件每个都需要使用一个 map 任务，作业时间比一个输入文件上用 8 个 map 任务慢几十倍甚至几百倍。

CombineFileInputFormat 可以缓解这个问题，它是针对小文件而设计的。FileInputFormat 为每个文件产生 1 个分片，而 CombineFileInputFormat 把多个文件打包到一个分片中以便每个 mapper 可以处理更多的数据。关键是，决定哪些块放入同一个分片时，CombineFileInputFormat 会考虑到节点和机架的因素，所以在典型 MapReduce 作业中处理输入的速度并不会下降。

当然，如果可能的话应该尽量避免许多小文件的情况，因为 MapReduce 处理数据的最佳速度最好与数据在集群中的传输速度相同，而处理小文件将增加运行作业而必需的寻址次数。还有，在 HDFS 集群中存储大量的小文件会浪费 namenode 的内存。一个可以减少大量小文件的方法是使用顺序文件(sequence file)将这些小文件合并成一个或多个大文件 (参见范例 8-4)：可以将文件名作为键(如果不需要键，可以用 NullWritable 等常量代替)，文件的内容作为值。但如果 HDFS 中已经有大批小文件，CombineFileInputFormat 方法值得一试。

CombineFileInputFormat 不仅可以很好地处理小文件，在处理大文件的时候也有好处。这是因为，它在每个节点生成一个分片，分片可能由多个块组成。本质上，CombineFileInputFormat 使 map 操作中处理的数据量与 HDFS 中文件的块大小之间的耦合度降低了。

5. 避免切分

有些应用程序可能不希望文件被切分，而是用一个 mapper 完整处理每一个输入文件。例如，检查一个文件中所有记录是否有序，一个简单的方法是顺序扫描每一条记录并且比较后一条记录是否比前一条要小。如果将它实现为一个 map 任务，那么只有一个 map 操作整个文件时，这个算法才可行。[①]

有两种方法可以保证输入文件不被切分。第一种(最简单但不怎么漂亮)方法就是增加最小分片大小，将它设置成大于要处理的最大文件大小。把它设置为最大值 long.MAX_VALUE 即可。第二种方法就是使用 FileInputFormat 具体子类，并且重写 isSplitable()方法[②]把返回值设置为 false。例如，以下就是一个不可分割的 TextInputFormat：

```
import org.apache.hadoop.fs.path;
import org.apache.hadoop.mapreduce.JobContenxt;
```

① SortValidator.RecordStatsChecker 中的 mapper 就是这样实现的。

② isSplitable()的方法名中，“splitable”只有一个“t”(通常拼写为“splittable”)，此书中使用的是这种拼写。

```
import org.apache.hadoop.mapreduce.lib.input.TextInpusFormat;

public class NonSplittableTextInputFormat extends TextInputFormat  {
   @override
   protected boolean isSplitable(JobContext context, Path file)  {
      return false;
  }
}
```

6. mapper 中的文件信息

处理文件输入分片的 mapper 可以从作业配置对象的某些特定属性中读取输入分片的有关信息，这可以通过调用在 Mapper 的 Context 对象上的 getInputSplit() 方法来实现。当输入的格式源自于 FileInputFormat 时，该方法返回的 InputSplit 可以被强制转换为一个 FileSplit，以此来访问表 8-7 列出的文件信息。

在旧版本的 MapReduce API 和 Streaming 接口中，同一个文件分片的信息可通过从 mapper 配置的可读属性获取。(在旧版本的 MapReduce API 中，可以通过在 Mappe 类中写 configure()方法访问 JobConf 对象来实现。)

除了表 8-7 中的属性，所有 mapper 和 reducer 都可以访问 7.4.1 节中列出的属性。

表 8-7. 文件输入分片的属性

FileSplit 方法	属性名称	类型	说明
getPath()	mapreduce.map.input.file	Path/String	正在处理的输入文件的路径
getStart()	mapreduce.map.input.start	long	分片开始处的字节偏移量
getLength()	mapreduce.map.input.length	long	分片的长度(按字节)

下一节将讨论在需要访问分块的文件名时如何使用 FileSplit。

7. 把整个文件作为一条记录处理

有时，mapper 需要访问一个文件中的全部内容。即使不分割文件，仍然需要一个 RecordReader 来读取文件内容作为 record 的值。范例 8-2 的 WholeFileInputFormat 展示了实现的方法。

范例 8-2. 把整个文件作为一条记录的 InputFormat

```
public class WholeFileInputFormat
   extends FileInputFormat<NullWritable, BytesWritable> {

@Override
```

```
protected boolean isSplitable(JobContext context, Path file) {
  return false;
}

@Override
public RecordReader<NullWritable, BytesWritable> createRecordReader(
    InputSplit split, TaskAttemptContext context) throws IOException,
    InterruptedException {
  WholeFileRecordReader reader = new WholeFileRecordReader();
  reader.initialize(split, context);
   return reader;
 }
}
```

WholeFileInputFormat 中没有使用键，此处表示为 NullWritable，值是文件内容，表示成 BytesWritable 实例。它定义了两个方法：一个是将 isSplitable() 方法重写返回 false 值，以此来指定输入文件不被分片；另一个是实现了 createRecordReader()方法，以此来返回一个定制的 RecordReader 实现，如范例 8-3 所示。

范例 8-3. WholeFileInputFormat 使用 RecordReader 将整个文件读为一条记录

```
class WholeFileRecordReader extends RecordReader<NullWritable, BytesWritable> {

  private FileSplit fileSplit;
  private Configuration conf;
  private BytesWritable value = new BytesWritable();
  private boolean processed = false;

  @Override
  public void initialize(InputSplit split, TaskAttemptContext context)
      throws IOException, InterruptedException {
    this.fileSplit = (FileSplit) split;
    this.conf = context.getConfiguration();
  }

  @Override
  public boolean nextKeyValue() throws IOException, InterruptedException {
    if (!processed) {
      byte[] contents = new byte[(int) fileSplit.getLength()];
      Path file = fileSplit.getPath();
      FileSystem fs = file.getFileSystem(conf);
      FSDataInputStream in = null;
      try {
        in = fs.open(file);
        IOUtils.readFully(in, contents, 0, contents.length);
        value.set(contents, 0, contents.length);
      } finally {
        IOUtils.closeStream(in);
      }
      processed = true;
```

```
    return true;
  }
  return false;
}

@Override
public NullWritable getCurrentKey() throws IOException, InterruptedException {
  return NullWritable.get();
}

@Override
public BytesWritable getCurrentValue() throws IOException,
    InterruptedException {
  return value;
}

@Override
public float getProgress() throws IOException {
   return processed ? 1.0f : 0.0f;
}

@Override
public void close() throws IOException {
   // do nothing
}
}
```

WholeFileRecordReader 负责将 FileSplit 转换成一条记录，该记录的键是 null，值是这个文件的内容。因为只有一条记录，WholeFileRecordReader 要么处理这条记录，要么不处理，所以它维护一个名称为 processed 的布尔变量来表示记录是否被处理过。如果当 nextKeyValue()方法被调用时，文件没有被处理过，就打开文件，产生一个长度是文件长度的字节数组，并用 Hadoop 的 IOUtils 类把文件的内容放入字节数组。然后再被传递到 next()方法的 BytesWritable 实例上设置数组，返回值为 true 则表示成功读取记录。

其他一些方法都是一些直接的用来访问当前的键和值类型、获取 reader 进度的方法，还有一个 close()方法，该方法由 MapReduce 框架在 reader 完成后调用。

现在演示如何使用 WholeFileInputFormat。假设有一个将若干个小文件打包成顺序文件的 MapReduce 作业，键是原来的文件名，值是文件的内容。如范例 8-4 所示。

范例 8-4. 将若干个小文件打包成顺序文件的 MapReduce 程序

```
public class SmallFilesToSequenceFileConverter extends Configured
  implements Tool {

static class SequenceFileMapper
    extends Mapper<NullWritable, BytesWritable, Text, BytesWritable> {
```

```
  private Text filenameKey;

  @Override
  protected void setup(Context context) throws IOException,
      InterruptedException {
    InputSplit split = context.getInputSplit();
    Path path = ((FileSplit) split).getPath();
    filenameKey = new Text(path.toString());
  }

  @Override
  protected void map(NullWritable key, BytesWritable value, Context context)
      throws IOException, InterruptedException {
  context.write(filenameKey, value);
   }

  }

  @Override
  public int run(String[] args) throws IOException {
    Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
    if (conf == null) {
      return -1;
    }

    job.setInputFormatClass(WholeFileInputFormat.class);
    job.setOutputFormatClass(SequenceFileOutputFormat.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(BytesWritable.class);

    job.setMapperClass(SequenceFileMapper.class);

    return job.waitForCompletion(true) ? 0 : 1;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new SmallFilesToSequenceFileConverter(), args);
    System.exit(exitCode);
  }
}
```

由于输入格式是 wholeFileInputFormat，所以 mapper 只需要找到文件输入分片的文件名。通过将 InputSplit 从 context 强制转换为 FileSplit 来实现这点，后者包含一个方法可以获取文件路径。路径存储在键对应的的一个 Text 对象中。reducer 的类型是相同的(没有明确设置)，输出格式是 SequenceFileOutputFormat。

以下是在一些小文件上运行样例。此处使用了两个 reducer，所以生成两个输出顺序文件：

```
% hadoop jar job. jar SmallFilesToSequenceFileConverter\
- conf conf/hadoop-localhost. xml -D mapreduce.job.reduces=2 \
input/smallfiles output
```

由此产生两部分文件，每一个对应一个顺序文件，可以通过文件系统 shell 的-text 选项来进行检查：

```
% hadoop fs -conf conf/hadoop-localhost.xml -text output/part-r-00000
  hdfs://localhost/user/tom/input/smallfiles/a     61 61 61 61 61 61 61 61 61 61
  hdfs://localhost/user/tom/input/smallfiles/c     63 63 63 63 63 63 63 63 63 63
  hdfs://localhost/user/tom/input/smallfiles/e
% hadoop fs -conf conf/hadoop-localhost.xml -text output/part-r-00001
  hdfs://localhost/user/tom/input/smallfiles/b     62 62 62 62 62 62 62 62 62 62
  hdfs://localhost/user/tom/input/smallfiles/d     64 64 64 64 64 64 64 64 64 64
  hdfs://localhost/user/tom/input/smallfiles/f     66 66 66 66 66 66 66 66 66 66
```

输入文件的文件名分别是 *a*、*b*、*c*、*d*、*e* 和 *f*，每个文件分别包含 10 个相应字母(比如，a 文件中包含 10 个“a”字母)，e 文件例外，它的内容为空。我们可以看到这些顺序文件的文本表示，文件名后跟着文件的十六进制的表示。

至少有一种方法可以改进我们的程序。前面提到，一个 mapper 处理一个文件的方法是低效的，所以较好的方法是继承 CombineFileInputFormat 而不是 FileInputFormat。

8.2.2 文本输入

Hadoop 非常擅长处理非结构化文本数据。本节讨论 Hadoop 提供的用于处理文本的不同 InputFormat 类。

1. TextInputFormat

TextInputFormat 是默认的 InputFormat。每条记录是一行输入。键是 LongWritable 类型，存储该行在整个文件中的字节偏移量。值是这行的内容，不包括任何行终止符(换行符和回车符)，它被打包成一个 Text 对象。所以，包含如下文本的文件被切分为包含 4 条记录的一个分片：

```
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

每条记录表示为以下键-值对：

```
(0, On the top of the Crumpetty Tree)
```

```
(33,  The Quangle Wangle sat,)
(57,  But his face you could not see,)
(89,  On account of his Beaver Hat.)
```

很明显，键并不是行号。一般情况下，很难取得行号，因为文件按字节而不是按行切分为分片。每个分片单独处理。行号实际上是一个顺序的标记，即每次读取一行的时候需要对行号进行计数。因此，在分片内知道行号是可能的，但在文件中是不可能的。

然而，每一行在文件中的偏移量是可以在分片内单独确定的，而不需要知道分片的信息，因为每个分片都知道上一个分片的大小，只需要加到分片内的偏移量上，就可以获得每行在整个文件中的偏移量了。通常，对于每行需要唯一标识的应用来说，有偏移量就足够了。如果再加上文件名，那么它在整个文件系统内就是唯一的。当然，如果每一行都是定长的，那么这个偏移量除以每一行的长度即可算出行号。

输入分片与 HDFS 块之间的关系

`FileInputFormat` 定义的逻辑记录有时并不能很好地匹配 HDFS 的文件块。例如，`TextInputFormat` 的逻辑记录是以行为单位的，那么很有可能某一行会跨文件块存放。虽然这对程序的功能没有什么影响，如行不会丢失或出错，但这种现象应该引起注意，因为这意味着那些“本地的”map(即 map 运行在输入数据所在的主机上)会执行一些远程的读操作。由此而来的额外开销一般不是特别明显。

图 8-3 展示了一个例子。一个文件分成几行，行的边界与 HDFS 块的边界没有对齐。分片的边界与逻辑记录的边界对齐(这里是行边界)，所以第一个分片包含第 5 行，即使第 5 行跨第一块和第二块。第二个分片从第 6 行开始。

图 8-3. TextInputFormat 的逻辑记录和 HDFS 块

2. 控制一行最大的长度

如果你正在使用这里讨论的文本输入格式中的一种，可以为预期的行长设一个最

大值，对付被损坏的文件。文件的损坏可以表现为一个超长行，这会导致内存溢出错误，进而任务失败。通过将 mapreduce.input. linerecordreader.line.maxlength 设置为用字节数表示的、在内存范围内的值(适当超过输入数据中的行长)，可以确保记录 reader 跳过(长的)损坏的行，不会导致任务失败。

3. 关于 KeyValueTextInputFormat

TextInputFormat 的键，即每一行在文件中的字节偏移量，通常并不是特别有用。通常情况下，文件中的每一行是一个键-值对，使用某个分界符进行分隔，比如制表符。例如由 TextOutputFormat (即 Hadoop 默认 OutputFormat)产生的输出就是这种。如果要正确处理这类文件，KeyValueTextInputFormat 比较合适。

可以通过 mapreduce.input.keyvaluelinerecordreader.key.value.separator 属性来指定分隔符。它的默认值是一个制表符。以下是一个范例，其中→表示一个(水平方向的)制表符：

```
line1 →On the top of the Crumpetty Tree
line2 →The Quangle Wangle sat,
line3 →But his face you could not see,
line4 →On account of his Beaver Hat.
```

与 TextInputFormat 类似，输入是一个包含 4 条记录的分片，不过此时的键是每行排在制表符之前的 Text 序列：

```
(line1, On the top of the Crumpetty Tree)
(line2, The Quangle Wangle sat,)
(line3,  But his face you could not see,)
(line4,  On account of his Beaver Hat.)
```

4. 关于 NLineInputFormat

通过 TextInputFormat 和 KeyValueTextInputFormat，每个 mapper 收到的输入行数不同。行数取决于输入分片的大小和行的长度。如果希望 mapper 收到固定行数的输入，需要将 NLineInputFormat 作为 InputFormat 使用。与 TextInputFormat 一样，键是文件中行的字节偏移量，值是行本身。

N 是每个 mapper 收到的输入行数。N 设置为 1(默认值)时，每个 mapper 正好收到一行输入。mapreduce.input.lineinputformat.linespermap 属性控制 N 值的设定。仍然以刚才的 4 行输入为例：

```
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

例如，如果 N 是 2，则每个输入分片包含两行。一个 mapper 收到前两行键-值对：

```
(0,  On the top of the Crumpetty Tree)
(33,  The Quangle Wangle sat,)
```

另一个 mapper 则收到后两行：

```
(57,  But his face you could not see,)
(89,  On account of his Beaver Hat.)
```

键和值与 `TextInputFormat` 生成的一样。不同的是输入分片的构造方法。

通常来说，对少量输入行执行 map 任务是比较低效的(任务初始化的额外开销造成的)，但有些应用程序会对少量数据做一些扩展的(也就是 CPU 密集型的)计算任务，然后产生输出。仿真是一个不错的例子。通过生成一个指定输入参数的输入文件，每行一个参数，便可以执行一个参数扫描分析(parameter sweep)：并发运行一组仿真试验，看模型是如何随参数不同而变化的。

在一些长时间运行的仿真实验中，可能会出现任务超时的情况。一个任务在 10 分钟内没有报告状态，application master 就会认为任务失败，进而中止进程(参见 7.2.1 节的详细讨论)。

这个问题最佳解决方案是定期报告状态，如写一段状态信息，或增加计数器的值。详情可以参见 7.1.5 节的补充材料“MapReduce 中进度的组成”。

另一个例子是用 Hadoop 引导从多个数据源(如数据库)加载数据。创建一个“种子”输入文件，记录所有的数据源，一行一个数据源。然后每个 mapper 分到一个数据源，并从这些数据源中加载数据到 HDFS 中。这个作业不需要 reduce 阶段，所以 reducer 的数量应该被设成 0(通过调用 `Job` 的 `setNumReduceTasks()`来设置)。进而可以运行 MapReduce 作业处理加载到 HDFS 中的数据。范例参见附录 C。

5. 关于 XML

大多数 XML 解析器会处理整个 XML 文档，所以如果一个大型 XML 文档由多个输入分片组成，那么单独解析每个分片就相当有挑战。当然，可以在一个 mapper 上(如果这个文件不是很大)，可以用 8.2.1 节介绍的方法来处理整个 XML 文档。

由很多“记录”(此处是 XML 文档片断)组成的 XML 文档，可以使用简单的字符串匹配或正则表达式匹配的方法来查找记录的开始标签和结束标签，而得到很多记录。这可以解决由 MapReduce 框架进行分割的问题，因为一条记录的下一个开始标签可以通过简单地从分片开始处进行扫描轻松找到，就像 TextInputFormat 确定新行的边界一样。

Hadoop 提供了 StreamXmlRecordReader 类(在 org.apache.hadoop.streaming.mapreduce 包中，还可以在 Streaming 之外使用)。通过把输入格式设为 StreamInputFormat，把 stream.recordreader.class 属性设为 org.apache.hadoop.streaming.mapreduce.StreamXmlRecordReader 来用 StreamXmlRecordReader 类。reader 的配置方法是通过作业配置属性来设 reader 开始标签和结束标签(详情参见这个类的帮助文档)。[①]

例如，维基百科用 XML 格式来提供大量数据内容，非常适合用 MapReduce 来并行处理。数据包含在一个大型的 XML 打包文档中，文档中有一些元素，例如包含每页内容和相关元数据的 page 元素。使用 StreamXmlRecordReader 后，这些 page 元素便可解释为一系列的记录，交由一个 mapper 来处理。

8.2.3 二进制输入

Hadoop 的 MapReduce 不只是可以处理文本信息，它还可以处理二进制格式的数据。

1. 关于 SequenceFileInputFormat 类

Hadoop 的顺序文件格式存储二进制的键-值对的序列。由于它们是可分割的(它们有同步点，所以 reader 可以从文件中的任意一点与记录边界进行同步，例如分片的起点)，所以它们很符合 MapReduce 数据的格式要求，并且它们还支持压缩，可以使用一些序列化技术来存储任意类型。详情参见 5.4.1 节。

如果要用顺序文件数据作为 MapReduce 的输入，可以使用 SequenceFileInputFormat。键和值是由顺序文件决定，所以只需要保证 map 输入的类型匹配。例如，如果顺序文件中键的格式是 IntWritable，值是 Text，就

① 对于完善的 XML 输入格式说明，可以参见 Mahout 的 XmlInputFormat，网址为 *http://mahout.apache.org/*。

像第 5 章中生成的那样，那么 mapper 的格式应该是 Mapper<IntWritable, Text, K, V>，其中 K 和 V 是这个 mapper 输出的键和值的类型。

虽然从名称上看不出来，但 SequenceFileInputFormat 可以读 map 文件和顺序文件。如果在处理顺序文件时遇到目录，SequenceFileInputFormat 会认为自己正在读 map 文件，使用的是其数据文件。因此，如果没有 MapFileInputFormat 类，也是可以理解的。

2. 关于 SequenceFileAsTextInputFormat 类

SequenceFileAsTextInputFormat 是 SequenceFileInputFormat 的变体，它将顺序文件的键和值转换为 Text 对象。这个转换通过在键和值上调用 toString()方法实现。这个格式使顺序文件作为 Streaming 的合适的输入类型。

3. 关于 SequenceFileAsBinaryInputFormat 类

SequenceFileAsBinaryInputFormat 是 SequenceFileInputFormat 的一种变体，它获取顺序文件的键和值作为二进制对象。它们被封装为 BytesWritable 对象，因而应用程序可以任意解释这些字节数组。与使用 SequenceFile.Reader 的 appendRaw()方法或 SequenceFileAsBinary OutputFormat 创建顺序文件的过程相配合，可以提供在 MapReduce 中可以使用任意二进制数据类型的方法(作为顺序文件打包)，不过呢，插入 Hadoop 序列化机制通常更简洁，详情参见 5.3.4 节。

4. 关于 FixedLengthInputFormat 类

FixedLengthInputFormat 用于从文件中读取固定宽度的二进制记录，当然这些记录没有用分隔符分开。必须通过 fixedlengthinputformat. record.length 设置每个记录的大小。

8.2.4 多个输入

虽然一个 MapReduce 作业的输入可能包含多个输入文件(由文件 glob、过滤器和路径组成)，但所有文件都由同一个 InputFormat 和同一个 Mapper 来解释。然而，数据格式往往会随时间演变，所以必须写自己的 mapper 来处理应用中的遗留数据格式问题。或者，有些数据源会提供相同的数据，但是格式不同。对不同的数据集进行连接(join，也称“联接”)操作时，便会产生这样的问题。详情参见 9.3.2 节。例如，有些数据可能是使用制表符分隔的文本文件，另一些可能是二进制的

顺序文件。即使它们格式相同，它们的表示也可能不同，因此需要分别进行解析。

这些问题可以用 MultipleInputs 类来妥善处理，它允许为每条输入路径指定 InputFormat 和 Mapper。例如，我们想把英国 Met Office[①]的气象数据和 NCDC 的气象数据放在一起来分析最高气温，则可以按照下面的方式来设置输入路径：

```
MultipleInputs.addInputPath(job, ncdcInputPath,
    TextInputFormat.class, MaxTemperatureMapper.class);
MultipleInputs.addInputPath(job,metofficeInputPath,
    TextInputFormat.class, MetofficeMaxTemperatureMapper.class);
```

这段代码取代了对 FileInputFormat.addInputPath()和 job.setMapperClass()的常规调用。Met Office 和 NCDC 的数据都是文本文件，所以对两者都使用 TextInputFormat 数据类型。但这两个数据源的行格式不同，所以我们使用了两个不一样的 mapper。MaxTemperatureMapper 读取 NCDC 的输入数据并抽取年份和气温字段的值。MetOfficeMaxTemperatureMapper 读取 Met Office 的输入数据，抽取年份和气温字段的值。重要的是两个 mapper 的输出类型一样，因此，reducer 看到的是聚集后的 map 输出，并不知道这些输入是由不同的 mapper 产生的。

MultipleInputs 类有一个重载版本的 addInputPath()方法，它没有 mapper 参数：

```
public static void addInputPath(Job job,  Path path,
                                class<? extends InputFormat> inputFormatClass)
```

如果有多种输入格式而只有一个 mapper(通过 Job 的 setMapperClass()方法设定)，这种方法很有用。

8.2.5 数据库输入(和输出)

DBInputFormat 这种输入格式用于使用 JDBC 从关系型数据库中读取数据。因为它没有任何共享能力，所以在访问数据库的时候必须非常小心，在数据库中运行太多的 mapper 读数据可能会使数据库受不了。正是由于这个原因，DBInputFormat 最好用于加载小量的数据集，如果需要与来自 HDFS 的大数据集

① Met Office 数据一般只用于科研和学术领域。然而，有少部分每月气象站数据可以从以下网址获取：*http://www.metoffice.gov.uk/climate/uk/stationdata/*。

连接，要使用 MultipleInputs。与之相对应的输出格式是 DBOutputFormat，它适用于将作业输出数据(中等规模的数据)转储到数据库。

在关系型数据库和 HDFS 之间移动数据的另一个方法是：使用 Sqoop，具体描述可以参见第 15 章。

HBase 的 TableInputFormat 的设计初衷是让 MapReduce 程序操作存放在 HBase 表中的数据。而 TableOutputFormat 则是把 MapReduce 的输出写到 HBase 表。

8.3 输出格式

针对前一节介绍的输入格式，Hadoop 都有相应的输出格式。OutputFormat 类的层次结构如图 8-4 所示。

图 8-4. OutputFormat 类的层次结构

8.3.1 文本输出

默认的输出格式是 TextOutputFormat，它把每条记录写为文本行。它的键和值可以是任意类型，因为 TextOutputFormat 调用 toString()方法把它们转换为

字符串。每个键-值对由制表符进行分隔，当然也可以设定 mapreduce.output.textoutputformat.separator 属性改变默认的分隔符。与 TextOutputFormat 对应的输入格式是 KeyValueTextInputFormat，它通过可配置的分隔符将键-值对文本行分隔，详情参见 8.2.2 节。

可以使用 NullWritable 来省略输出的键或值(或两者都省略，相当于 NullOutputFormat 输出格式，后者什么也不输出)。这也会导致无分隔符输出，以使输出适合用 TextInputFormat 读取。

8.3.2 二进制输出

1. 关于 SequenceFileOutputFormat

正如名称所示，SequenceFileOutputFormat 将它的输出写为一个顺序文件。如果输出需要作为后续 MapReduce 任务的输入，这便是一种好的输出格式，因为它的格式紧凑，很容易被压缩。压缩由 SequenceFileOutputFormat 的静态方法来实现，详情参见 5.2.3 节。9.2 节用一个例子展示了如何使用 SequenceFileoutputFormat。

2. 关于 SequenceFileAsBinaryOuputFormat

SequenceFileAsBinaryOutputFormat 与 SequenceFileAsBinaryInputFormat 相对应，它以原始的二进制格式把键-值对写到一个顺序文件容器中。

3. 关于 MapFileOutputFormat

MapFileOutputFormat 把 map 文件作为输出。MapFile 中的键必须顺序添加，所以必须确保 reducer 输出的键已经排好序。

reduce 输入的键一定是有序的，但输出的键由 reduce 函数控制，MapReduce 框架中没有硬性规定 reduce 输出键必须是有序的。所以 reduce 输出的键必须有序是对 MapFileOutputFormat 的一个额外限制。

8.3.3 多个输出

FileOutputFormat 及其子类产生的文件放在输出目录下。每个 reducer 一个文件并且文件由分区号命名：*part-r-00000*，*part-r-00001*，等等。有时可能需要对输出

的文件名进行控制或让每个 reducer 输出多个文件。MapReduce 为此提供了 MultipleOutputFormat 类。[①]

1. 范例：数据分割

考虑这样一个需求：按气象站来区分气象数据。这需要运行一个作业，作业的输出是每个气象站一个文件，此文件包含该气象站的所有数据记录。

一种方法是每个气象站对应一个 reducer。为此，我们必须做两件事。第一，写一个 partitioner，把同一个气象站的数据放到同一个分区。第二，把作业的 reducer 数设为气象站的个数。partitioner 如下：

```
public class StationPartitioner extend Partitioner<LongWritable, Text> {

  private NcdcRecordParser parser = new NcdcRecordParser();

  @Override
  public int getPartition(LongWritable key, Text value, int numPartitions) {
    parser.parse(value);
    return getPartition(parser.getStationId());
  }

  private int getPartition(String stationId) {
    ...
  }
}
```

这里没有给出 getPartition(String)方法的实现，它将气象站 ID 转换成分区索引号。为此，它的输入是一个列出所有气象站 ID 的列表，然后返回列表中气象站 ID 的索引。

这样做有两个缺点。第一，需要在作业运行之前知道分区数和气象站的个数。虽然 NCDC 数据集提供了气象站的元数据，但无法保证数据中的气象站 ID 与元数据匹配。如果元数据中有某个气象站但数据中却没有该气象站的数据，就会浪费一个 reduce 任务。更糟糕的是，数据中有但元数据中却没有的气象站，也没有对应的 reduce 任务，只好将这个气象站扔掉。解决这个问题的方法是写一个作业来

① 在旧版本的 MapReduce API 中，有两个类用于产生多个输出：MultipleOutputFormat 和 MultipleOutputs。简单地说，虽然 MultipleOutputs 更具有特色，但 MultipleOutputs 在输出目录结构和文件命名上有更多的控制。新版本 API 中的 MultipleOutputs 结合了旧版本 API 中两种多个输出类的特点。本书网站上的代码包含了本节例子的旧版本 API 等价样例，该样例使用了 MultipleOutputs 和 MultipleOutputFormat。

抽取唯一的气象站 ID，但很遗憾，这需要额外的作业来实现。

第二个缺点更微妙。一般来说，让应用程序来严格限定分区数并不好，因为可能导致分区数少或分区不均。让很多 reducer 做少量工作不是一个高效的作业组织方法，比较好的办法是使用更少 reducer 做更多的事情，因为运行任务的额外开销减少了。分区不均的情况也是很难避免的。不同气象站的数据量差异很大：有些气象站是一年前刚投入使用的，而另一些气象站可能已经工作近一个世纪了。如果其中一些 reduce 任务运行时间远远超过另一些，那么作业执行时间将由它们来决定，从而导致作业运行时间超出预期。

在以下两种特殊情况下，让应用程序来设定分区数(等价于 reducer 的个数)是有好处的。

- 0 个 reducer　这个情况很罕见：没有分区，因为应用只需执行 map 任务
- 1 个 reducer　可以很方便地运行若干小作业，把以前作业的输出合并成单个文件。前提是数据量足够小，以便一个 reducer 能轻松处理

最好让集群为作业决定分区数：可用的集群资源越多，作业完成就越快。这就是默认的 `HashPartitioner` 表现如此出色的原因，因为它处理的分区数不限，并且确保每个分区都有一个很好的键组合使分区更均匀。

如果使用 `HashPartitioner`，每个分区就会包含多个气象站，因此，要实现每个气象站输出一个文件，必须安排每个 reducer 写多个文件，由此就有了 `MultipleOutput`。

2. 关于 MultipleOutput 类

`MultipleOutput` 类可以将数据写到多个文件，这些文件的名称源于输出的键和值或者任意字符串。这允许每个 reducer(或者只有 map 作业的 mapper)创建多个文件。采用 *name-m-nnnnn* 形式的文件名用于 map 输出，*name-r-nnnnn* 形式的文件名用于 reduce 输出，其中 *name* 是由程序设定的任意名字，*nnnnn* 是一个指明块号的整数(从 *00000* 开始)。块号保证从不同分区(mapper 或 reducer)写的输出在相同名字情况下不会冲突。

范例 8-5 显示了如何使用 `MultipleOutputs` 按照气象站划分数据。

范例 8-5. 用 MultipleOutput 类将整个数据集分区到以气象站 ID 命名的文件

```
public class PartitionByStationUsingMultipleOutputs extends Configured
  implements Tool {

  static class StationMapper
    extends Mapper<LongWritable, Text, Text, Text> {

  private NcdcRecordParser parser = new NcdcRecordParser();

  @Override
  protected void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
    parser.parse(value);
    context.write(new Text(parser.getStationId()), value);
  }
}

static class MultipleOutputsReducer
    extends Reducer<Text, Text, NullWritable, Text> {
  private MultipleOutputs<NullWritable, Text> multipleOutputs;

  @Override
  protected void setup(Context context)
      throws IOException, InterruptedException {
    multipleOutputs = new MultipleOutputs<NullWritable, Text>(context);
  }

  @Override
  public void reduce(Text key, Iterable<Text> values, Context context)
      throws IOException, InterruptedException {
    for (Text value : values) {
      multipleOutputs.write(NullWritable.get(), value, key.toString());
    }
  }

  @Override
  protected void cleanup(Context context)
      throws IOException, InterruptedException {
    multipleOutputs.close();
  }
}

@Override
public int run(String[] args) throws Exception {
  Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
  if (job == null) {
    return -1;
  }

  job.setMapperClass(StationMapper.class);
  job.setMapOutputKeyClass(Text.class);
  job.setReducerClass(MultipleOutputsReducer.class);
  job.setOutputKeyClass(NullWritable.class);
```

```
  return job.waitForCompletion(true) ? 0 : 1;
}
public static void main(String[] args) throws Exception {
  int exitCode = ToolRunner.run(new PartitionByStationUsingMultipleOutputs(),
      args);
  System.exit(exitCode);
  }
}
```

在生成输出的 reducer 中，在 setup()方法中构造一个 MultipleOutputs 的实例并将它赋给一个实例变量。在 reduce()方法中使用 MultipleOutputs 实例来写输出，而不是 context。write()方法作用于键、值和名字。这里使用气象站标识符作为名字，因此最后产生的输出名字的形式为 *station_identifier_r-nnnnn*。

运行一次后，前面几个输出文件的命名如下：

```
/output/010010-99999-r-00027
/output/010050-99999-r-00013
/output/010100-99999-r-00015
/output/010280-99999-r-00014
/output/010550-99999-r-00000
/output/010980-99999-r-00011
/output/011060-99999-r-00025
/output/012030-99999-r-00029
/output/012350-99999-r-00018
/output/012620-99999-r-00004
```

在 MultipleOutputs 的 write()方法中指定的基本路径相对于输出路径进行解释，因为它可以包含文件路径分隔符(/)，创建任意深度的子目录是有可能的。例如，下面的改动将数据根据气象站和年份进行划分，这样每年的数据就被包含到一个名为气象站 ID 的目录中(例如 *029070-99999/1901/part-r-00000*)：

```
@Override
protected void reduce(Text key, Iterable<Text> values, Context context)
    throws IOException, InterruptedException {
  for (Text value : values) {
    parser.parse(value);
    String basePath = String.format("%s/%s/part",
        parser.getStationId(), parser.getYear());
    multipleOutputs.write(NullWritable.get(), value, basePath);
  }
}
```

MultipleOutput 传递给 mapper 的 OutputFormat，该例子中为 TextOutputFormat，但可能有更复杂的情况。例如，可以创建命名的输出，每个都有自己的 OutputForamt、键和值的类型(这可以与 mapper 或 reducer 的输出类型不相同)。此外，mapper 或 reducer 可以为每条处理的记录写多个输出文件。可以查阅 Java 帮助文档，获取更多信息。

8.3.4 延迟输出

FileOutputFormat 的子类会产生输出文件(*part-r-nnnnn*)，即使文件是空的。有些应用倾向于不创建空文件，此时 LazyOutputFormat 就有用武之地了。它是一个封装输出格式，可以保证指定分区第一条记录输出时才真正创建文件。要使用它，用 JobConf 和相关的输出格式作为参数来调用 setOutputFormatClass() 方法即可。

Streaming 支持-LazyOutput 选项来启用 LazyOutputFormat 功能。

8.3.5 数据库输出

写到关系型数据库和 HBase 的输出格式可以参见 8.2.5 节。

第 9 章

MapReduce 的特性

本章探讨 MapReduce 的一些高级特性，其中包括计数器、数据集的排序和连接。

9.1 计数器

在许多情况下，用户需要了解待分析的数据，尽管这并非所要执行的分析任务的核心内容。以统计数据集中无效记录数目的任务为例，如果发现无效记录的比例相当高，那么就需要认真思考为何存在如此多无效记录。是所采用的检测程序存在缺陷，还是数据集质量确实很低，包含了大量无效记录？如果确实是数据集的质量问题，则可能需要扩大数据集的规模以增大有效记录的比例，从而进行有意义的分析。

计数器是收集作业统计信息的有效手段之一，用于质量控制或应用级统计。计数器还可辅助诊断系统故障。如果需要将日志信息传输到 map 或 reduce 任务，更好的方法通常是看能否用一个计数器值来记录某一特定事件的发生。对于大型分布式作业而言，使用计数器更为方便。除了因为获取计数器值比输出日志更方便，还有根据计数器值统计特定事件的发生次数要比分析一堆日志文件容易得多。

9.1.1 内置计数器

Hadoop 为每个作业维护若干内置计数器，以描述多项指标。例如，某些计数器记录已处理的字节数和记录数，使用户可监控已处理的输入数据量和已产生的输出数据量。

这些内置计数器被划分为若干个组，参见表 9-1。

表 9-1. 内置的计数器分组

组别	名称/类别	参考
MapReduce 任务计数器	org.apache.hadoop.mapreduce.TaskCounter	表 9-2
文件系统计数器	org.apache.hadoop.mapreduce.FileSystemCounter	表 9-3
FileInputFormat 计数器	org.apache.hadoop.mapreduce.lib.input.FileInputFormatCounter	表 9-4
FileOutputFormat 计数器	org.apache.hadoop.mapreduce.lib.output.FileOutputFormatCounter	表 9-5
作业计数器	org.apache.hadoop.mapreduce.JobCounter	表 9-6

各组要么包含任务计数器(在任务处理过程中不断更新)，要么包含作业计数器(在作业处理过程中不断更新)。这两种类型将在后续章节中进行介绍。

1. 任务计数器

在任务执行过程中，任务计数器采集任务的相关信息，每个作业的所有任务的结果会被聚集起来。例如，MAP_INPUT_RECORDS 计数器统计每个 map 任务输入记录的总数，并在一个作业的所有 map 任务上进行聚集，使得最终数字是整个作业的所有输入记录的总数。

任务计数器由其关联任务维护，并定期发送给 application master。因此，计数器能够被全局地聚集。(参见 7.1.5 节中对进度和状态更新的介绍。)任务计数器的值每次都是完整传输的，而非传输自上次传输之后的计数值，从而避免由于消息丢失而引发的错误。另外，如果一个任务在作业执行期间失败，则相关计数器的值会减小。

虽然只有当整个作业执行完之后计数器的值才是完整可靠的，但是部分计数器仍然可以在任务处理过程中提供一些有用的诊断信息，以便由 Web 界面监控。例如，PHYSICAL_MEMORY_BYTES、VIRTUAL_MEMORY_BYTES 和 COMMITTED_HEAP_BYTES 计数器显示特定任务执行过程中的内存使用变化情况。

内置的任务计数器包括在 MapReduce 任务计数器分组中的计数器(表 9-2)以及在文件相关的计数器分组(表 9-3、表 9-4 和表 9-5)中的计数器。

表 9-2. 内置的 MapReduce 任务计数器

计数器名称	说明
map 输入的记录数 (MAP_INPUT_RECORDS)	作业中所有 map 已处理的输入记录数。每次 RecordReader 读到一条记录并将其传给 map 的 map()函数时，该计数器的值递增
分片(split)的原始字节数 (SPLIT_RAW_BYTES)	由 map 读取的输入-分片对象的字节数。这些对象描述分片元数据(文件的位移和长度)，而不是分片的数据自身，因此总规模是小的
map 输出的记录数 (MAP_OUTPUT_RECORDS)	作业中所有 map 产生的 map 输出记录数。每次某一个 map 的 OutputCollector 调用 collect()方法时，该计数器的值增加
map 输出的字节数 (MAP_OUTPUT_BYTES)	作业中所有 map 产生的未经压缩的输出数据的字节数。每次某一个 map 的 OutCollector 调用 collect()方法时，该计数器的值增加
map 输出的物化字节数 (MAP_OUTPUT_MATERIALIZED _BYTES)	map 输出后确实写到磁盘上的字节数；若 map 输出压缩功能被启用，则会在计数器值上反映出来
combine 输入的记录数 (COMBINE_INPUT_RECORDS)	作业中所有 combiner(如果有)已处理的输入记录数。combiner 的迭代器每次读一个值，该计数器的值增加。注意：本计数器代表 combiner 已经处理的值的个数，并非不同的键组数(后者并无实质意义，因为对于 combiner 而言，并不要求每个键对应一个组，详情参见 2.4.2 节和 7.3 节
combine 输出的记录数 (COMBINE_OUTPUT_RECORDS)	作业中所有 combiner(如果有)已产生的输出记录数。每当一个 combiner 的 OutputCollector 调用 collect()方法时，该计数器的值增加
reduce 输入的组 (REDUCE_INPUT_GROUPS)	作业中所有 reducer 已经处理的不同的码分组的个数。每当某一个 reducer 的 reduce()被调用时，该计数器的值增加
reduce 输入的记录数 (REDUCE_INPUT_RECORDS)	作业中所有 reducer 已经处理的输入记录的个数。每当某个 reducer 的迭代器读一个值时，该计数器的值增加。如果所有 reducer 已经处理数完所有输入，则该计数器的值与计数器“map 输出的记录”的值相同
reduce 输出的记录数 (REDUCE_OUTPUT_RECORDS)	作业中所有 map 已经产生的 reduce 输出记录数。每当某个 reducer 的 OutputCollector 调用 collect()方法时，该计数器的值增加
reduce 经过 shuffle 的字节数 (REDUCE_SHUFFLE_BYTES)	由 shuffle 复制到 reducer 的 map 输出的字节数
溢出的记录数 (SPILLED_RECORDS)	作业中所有 map 和 reduce 任务溢出到磁盘的记录数
CPU 毫秒 (CPU_MILLISECONDS)	一个任务的总 CPU 时间，以毫秒为单位，可由 /proc/cpuinfo 获取
物理内存字节数 (PHYSICAL_ MEMORY_BYTES)	一个任务所用的物理内存，以字节数为单位，可由 /proc/meminfo 获取

续表

计数器名称	说明
虚拟内存字节数 (VIRTUAL_MEMORY_BYTES)	一个任务所用虚拟内存的字节数，由*/proc/meminfo* 获取
有效的堆字节数 (COMMITTED_HEAP_BYTES)	在 JVM 中的总有效内存量(以字节为单位)，可由 Runtime.getRuntime().totalMemory()获取
GC 运行时间毫秒数 (GC_TIME_MILLIS)	在任务执行过程中，垃圾收集器(garbage collection)花费的时间(以毫秒为单位)，可由 GarbageCollector MXBean.getCollectionTime()获取
由 shuffle 传输的 map 输出数 (SHUFFLED_MAPS)	由 shuffle 传输到 reducer 的 map 输出文件数，详情参见 7.3 节
失败的 shuffle 数 (FAILED_SHUFFLE)	shuffle 过程中，发生 map 输出拷贝错误的次数
被合并的 map 输出数 (MERGED_MAP_OUTPUTS)	shuffle 过程中，在 reduce 端合并的 map 输出文件数

表 9-3. 内置的文件系统任务计数器

计数器名称	说明
文件系统的读字节数 (BYTES_READ)	由 map 任务和 reduce 任务在各个文件系统中读取的字节数，各个文件系统分别对应一个计数器，文件系统可以是 Local、HDFS、S3 等
文件系统的写字节数 (BYTES_WRITTEN)	由 map 任务和 reduce 任务在各个文件系统中写的字节数
文件系统读操作的数量 (READ_OPS)	由 map 任务和 reduce 任务在各个文件系统中进行的读操作的数量(例如，open 操作，file status 操作)
文件系统大规模读操作的数量 (LARGE_READ_OPS)	由 map 和 reduce 任务在各个文件系统中进行的大规模读操作(例如，对于一个大容量目录进行 list 操作)的数量
文件系统写操作的数量 (WRITE_OPS)	由 map 任务和 reduce 任务在各个文件系统中进行的写操作的数量(例如，create 操作，append 操作)

表 9-4. 内置的 FileInputFormat 任务计数器

计数器名称	说明
读取的字节数 (BYTES_READ)	由 map 任务通过 FileInputFormat 读取的字节数

表 9-5. 内置的 FileOutputFormat 任务计数器

计数器名称	说明
写的字节数 (BYTES_WRITTEN)	由 map 任务(针对仅含 map 的作业)或者 reduce 任务通过 FileOutputFormat 写的字节数

2. 作业计数器

作业计数器(表 9-6)由 application master 维护，因此无需在网络间传输数据，这一点与包括“用户定义的计数器”在内的其他计数器不同。这些计数器都是作业级别的统计量，其值不会随着任务运行而改变。例如，TOTAL_LAUNCHED_MAPS 统计在作业执行过程中启动的 map 任务数，包括失败的 map 任务。

表 9-6. 内置的作业计数器

计数器名称	说明
启用的 map 任务数 (TOTAL_LAUNCHED_MAPS)	启动的 map 任务数，包括以“推测执行”方式启动的任务，详情参见 7.4.2 节
启用的 reduce 任务数 (TOTAL_LAUNCHED_REDUCES)	启动的 reduce 任务数，包括以“推测执行”方式启动的任务
启用的 uber 任务数 (TOTAL_LAUNCHED_UBERTASKS)	启用的 uber 任务数，详情参见 7.1 节
uber 任务中的 map 数 (NUM_UBER_SUBMAPS)	在 uber 任务中的 map 数
Uber 任务中的 reduce 数 (NUM_UBER_SUBREDUCES)	在 uber 任务中的 reduce 数
失败的 map 任务数 (NUM_FAILED_MAPS)	失败的 map 任务数，用户可以参见 7.2.1 节对任务失败的讨论，了解失败原因
失败的 reduce 任务数 (NUM_FAILED_REDUCES)	失败的 reduce 任务数
失败的 uber 任务数 (NUM_FAILED_UBERTASKS)	失败的 uber 任务数
被中止的 map 任务数 (NUM_KILLED_MAPS)	被中止的 map 任务数，可以参见 7.2.1 节对任务失败的讨论，了解中止原因
被中止的 reduce 任务数 (NUM_KILLED_REDUCES)	被中止的 reduce 任务数
数据本地化的 map 任务数 (DATA_LOCAL_MAPS)	与输入数据在同一节点上的 map 任务数
机架本地化的 map 任务数 (RACK_LOCAL_MAPS)	与输入数据在同一机架范围内但不在同一节点上的 map 任务数
其他本地化的 map 任务数 (OTHER_LOCAL_MAPS)	与输入数据不在同一机架范围内的 map 任务数。由于机架之间的带宽资源相对较少，Hadoop 会尽量让 map 任务靠近输入数据执行，因此该计数器值一般比较小。详情参见图 2-2

续表

计数器名称	说明
map 任务的总运行时间 (MILLIS_MAPS)	map 任务的总运行时间，单位毫秒。包括以推测执行方式启动的任务。可参见相关的度量内核和内存使用的计数器(VCORES_MILLIS_MAPS 和 MB_MILLIS_MAPS)
reduce 任务的总运行时间 (MILLIS_REDUCES)	reduce 任务的总运行时间，单位毫秒。包括以推测执行方式启动的任务。可参见相关的度量内核和内存使用的计数器(VCORES_MILLIS_REDUCES 和 MB_MILLIS_REDUCES)

9.1.2 用户定义的 Java 计数器

MapReduce 允许用户编写程序来定义计数器，计数器的值可在 mapper 或 reducer 中增加，计数器由一个 Java 枚举(enum)类型来定义，以便对有关的计数器分组。一个作业可以定义的枚举类型数量不限，各个枚举类型所包含的字段数量也不限。枚举类型的名称即为组的名称，枚举类型的字段就是计数器名称。计数器是全局的。换言之，MapReduce 框架将跨所有 map 和 reduce 聚集这些计数器，并在作业结束时产生一个最终结果。

在第 6 章中，我们创建了若干计数器来统计天气数据集中不规范的记录数。范例 9-1 中的程序对此做了进一步扩展，能统计缺失记录和气温质量代码的分布情况。

范例 9-1. 统计最高气温的作业，包括统计气温值缺失的记录、不规范的字段和质量代码

```
public class MaxTemperatureWithCounters extends Configured implements Tool {

  enum Temperature {
    MISSING,
    MALFORMED
  }

  static class MaxTemperatureMapperWithCounters
    extends Mapper<LongWritable, Text, Text, IntWritable> {

    private NcdcRecordParser parser = new NcdcRecordParser();

    @Override
    protected void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {

    parser.parse(value);
    if (parser.isValidTemperature()) {
      int airTemperature = parser.getAirTemperature();
      context.write(new Text(parser.getYear()),
        new IntWritable(airTemperature));
    } else if (parser.isMalformedTemperature()) {
```

```
      System.err.println("Ignoring possibly corrupt input: " + value);
      context.getCounter(Temperature.MALFORMED).increment(1);
    } else if (parser.isMissingTemperature()) {
      context.getCounter(Temperature.MISSING).increment(1);
    }

    // dynamic counter
    context.getCounter("TemperatureQuality", parser.getQuality()).increment(1);
  }
}

@Override
public int run(String[] args) throws Exception {
  Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
  if (job == null) {
    return -1;
  }

  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  job.setMapperClass(MaxTemperatureMapperWithCounters.class);
  job.setCombinerClass(MaxTemperatureReducer.class);
  job.setReducerClass(MaxTemperatureReducer.class);

  return job.waitForCompletion(true) ? 0 : 1;
}

public static void main(String[] args) throws Exception {
  int exitCode = ToolRunner.run(new MaxTemperatureWithCounters(), args);
  System.exit(exitCode);
}
}
```

理解上述程序的最佳方法是在完整的数据集上运行一遍:

```
% hadoop jar hadoop-examples.jar MaxTemperatureWithCounters \
  input/ncdc/all output-counters
```

作业成功执行完毕之后会输出各计数器的值(由作业的客户端完成)。以下是一组我们感兴趣的计数器的值。

```
Air Temperature Records
  Malformed=3
  Missing=66136856
TemperatureQuality
  0=1
  1=973422173
  2=1246032
  4=10764500
  5=158291879
  6=40066
  9=66136858
```

注意，为使得气温计数器的名称可读性更好，使用了 Java 枚举类型的命名方式(使用下划线作为嵌套类的分隔符)，这种方式称为“资源捆绑”(resource bundle)，例如本例中为 *MaxTemperatureWithCounters_Temperature.properties*，该捆绑包含显示名称的映射关系。

1. 动态计数器

上述代码还使用了动态计数器，这是一种不由 Java 枚举类型定义的计数器。由于 Java 枚举类型的字段在编译阶段就必须指定，因而无法使用枚举类型动态新建计数器。范例 8-1 统计气温质量的分布，尽管通过格式规范定义了可以取的值，但相比之下，使用动态计数器来产生实际值更加方便。在该例中，`Context` 对象的 `getCounter()` 方法有两个 `String` 类型的输入参数，分别代表组名称和计数器名称：

```
public Counter getCounter(String groupName, String counterName)
```

鉴于 Hadoop 需先将 Java 枚举类型转变成 `String` 类型，再通过 RPC 发送计数器值，这两种创建和访问计数器的方法(即使用枚举类型和 `String` 类型)事实上是等价的。相比之下，枚举类型易于使用，还提供类型安全，适合大多数作业使用。如果某些特定场合需要动态创建计数器，则可以使用 `String` 接口。

2. 获取计数器

除了通过 Web 界面和命令行(执行 `mapred job -counter` 指令)之外，用户还可以使用 Java API 获取计数器的值。通常情况下，用户一般在作业运行完成、计数器的值已经稳定下来时再获取计数器的值，而 Java API 还支持在作业运行期间就能够获取计数器的值。范例 9-2 展示了如何统计整个数据集中气温信息缺失记录的比例。

范例 9-2. 统计气温信息缺失记录所占的比例

```
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.util.*;

public class MissingTemperatureFields extends Configured implements Tool {

  @Override
  public int run(String[] args) throws Exception {
    if (args.length != 1) {
      JobBuilder.printUsage(this, "<job ID>");
      return -1;
    }
    String jobID = args[0];
```

```
    Cluster cluster = new Cluster(getConf());
    Job job = cluster.getJob(JobID.forName(jobID));
    if (job == null) {
      System.err.printf("No job with ID %s found.\n", jobID);
      return -1;
    }
    if (!job.isComplete()) {
      System.err.printf("Job %s is not complete.\n", jobID);
      return -1;
    }

    Counters counters = job.getCounters();
    long missing = counters.findCounter(
        MaxTemperatureWithCounters.Temperature.MISSING).getValue();
    long total = counters.findCounter(TaskCounter.MAP_INPUT_RECORDS).getValue();

    System.out.printf("Records with missing temperature fields: %.2f%%\n",
        100.0 * missing / total);
    return 0;
  }
  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new MissingTemperatureFields(), args);
    System.exit(exitCode);
  }
}
```

首先，以作业 ID 为输入参数调用 getJob()方法，从 cluster 中获取一个 Job 对象。通过检查返回是否为空来判断是否有一个作业与指定 ID 相匹配。有多种因素可能导致无法找到一个有效的 Job 对象，例如，错误地指定了作业 ID，或是该作业不再保留在作业历史中。

其次，如果确认该作业已经完成，则调用该 Job 对象的 getCounters()方法会返回一个 Counters 对象，封装了该作业的所有计数器。Counters 类提供了多个方法用于获取计数器的名称和值。上例调用 findCounter()方法，它通过一个枚举值来获取气温信息缺失的记录数和被处理的记录数(根据一个内置计数器)。

最后，输出气温信息缺失记录的比例。针对整个天气数据集的运行结果如下所示：

```
% hadoop jar hadoop-examples.jar MissingTemperatureFields job_1410450250506_0007
Records with missing temperature fields: 5.47%
```

9.1.3 用户定义的 Streaming 计数器

使用 Streaming 的 MapReduce 程序可以向标准错误流发送一行特殊格式的信息来

增加计数器的值，这种技术可以作为一种计数器控制手段。信息的格式如下：

```
reporter:counter:group,counter,amount
```

以下 Python 代码片段将 Temperature 组的 Missing 计数器的值增加 1：

```
sys.stderr.write("reporter:counter:Temperature,Missing,1\n")
```

状态信息也可以类似方法发出，格式如下所示：

```
reporter:status:message
```

9.2 排序

排序是 MapReduce 的核心技术。尽管应用本身可能并不需要对数据排序，但仍可能使用 MapReduce 的排序功能来组织数据。本节将讨论几种不同的数据集排序方法，以及如何控制 MapReduce 的排序。12.8 节介绍了如何对 Avro 数据进行排序。

9.2.1 准备

下面将按气温字段对天气数据集排序。由于气温字段是有符号整数，所以不能将该字段视为 Text 对象并以字典顺序排序。[①]反之，我们要用顺序文件存储数据，其 IntWritable 键代表气温(并且正确排序)，其 Text 值就是数据行。

范例 9-3 中的 MapReduce 作业只包含 map 任务，它过滤输入数据并移除包含有无效气温的记录。各个 map 创建并输出一个块压缩的顺序文件。相关指令如下：

```
% hadoop jar hadoop-examples.jar SortDataPreprocessor input/ncdc/all \
  input/ncdc/all-seq
```

范例 9-3. 该 MapReduce 程序将天气数据转成 SequenceFile 格式

```
public class SortDataPreprocessor extends Configured implements Tool {

  static class CleanerMapper
    extends Mapper<LongWritable, Text, IntWritable, Text> {
```

① 有一个常用的方法能解决这个问题(特别是针对基于文本的 Streaming 应用)：首先，增加偏移量以消除所有负数；其次，在数字前面增加 0，使所有数字的长度相等。9.2.4 节要介绍另一种方法。

```
  private NcdcRecordParser parser = new NcdcRecordParser();

  @Override
  protected void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    parser.parse(value);
    if (parser.isValidTemperature()) {
      context.write(new IntWritable(parser.getAirTemperature()), value);
    }
  }
}

@Override
public int run(String[] args) throws Exception {
  Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
  if (job == null) {
    return -1;
  }

  job.setMapperClass(CleanerMapper.class);
  job.setOutputKeyClass(IntWritable.class);
  job.setOutputValueClass(Text.class);
  job.setNumReduceTasks(0);
  job.setOutputFormatClass(SequenceFileOutputFormat.class);
  SequenceFileOutputFormat.setCompressOutput(job, true);
  SequenceFileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
  SequenceFileOutputFormat.setOutputCompressionType(job,
      CompressionType.BLOCK);

  return job.waitForCompletion(true) ? 0 : 1;
}
public static void main(String[] args) throws Exception {
  int exitCode = ToolRunner.run(new SortDataPreprocessor(), args);
  System.exit(exitCode);
}
}
```

9.2.2 部分排序

如 9.1.1 节所述，在默认情况下， MapReduce 根据输入记录的键对数据集排序。范例 9-4 则是一个变种，它利用 `IntWritable` 键对顺序文件排序。

范例 9-4. 程序调用默认 HashPartitioner 按 IntWritable 键排序顺序文件

```
public class SortByTemperatureUsingHashPartitioner extends Configured
  implements Tool {

  @Override
  public int run(String[] args) throws Exception {
    Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
```

```
    if (job == null) {
      return -1;
    }

    job.setInputFormatClass(SequenceFileInputFormat.class);
    job.setOutputKeyClass(IntWritable.class);
    job.setOutputFormatClass(SequenceFileOutputFormat.class);
    SequenceFileOutputFormat.setCompressOutput(job, true);
    SequenceFileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
    SequenceFileOutputFormat.setOutputCompressionType(job,
        CompressionType.BLOCK);

    return job.waitForCompletion(true) ? 0 : 1;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new SortByTemperatureUsingHashPartitioner(),
        args);
    System.exit(exitCode);
  }
}
```

控制排列顺序

键的排列顺序是由 RawComparator 控制的，规则如下。

1. 若属性 mapreduce.job.output.key.comparator.class 已经显式设置，或者通过 Job 类的 setSortComparatorClass()方法进行设置，则使用该类的实例，旧版 API 使用 JobConf 类的 setOutputKeyComparatorClass()方法。

2. 否则，键必须是 WritableComparable 的子类，并使用针对该键类的已登记的 comparator。

3. 如果还没有已登记的 comparator，则使用 RawComparator。RawComparator 将字节流反序列化为一个对象，再由 WritableComparable 的 compareTo()方法进行操作。

上述规则彰显了为自定义 Writable 类登记 RawComparators 优化版本的重要性，详情可参见 5.3.3 节介绍的如何为提高速度实现一个 RawComparator。同时，通过定制 comparator 来重新定义排序顺序也很直观，详情可参见 9.2.4 节对辅助排序的讨论。

假设采用 30 个 reducer 来运行该程序：[1]

```
% hadoop jar hadoop-examples.jar SortByTemperatureUsingHashPartitioner \
  -D mapreduce.job.reduces=30 input/ncdc/all-seq output-hashsort
```

该指令产生 30 个已排序的输出文件。但是如何将这些小文件合并成一个有序的文件却并非易事。例如，直接将纯文本文件连接起来无法保证全局有序。

幸运的是，许多应用并不强求待处理的文件全局有序。例如，对于通过键进行查找来说，部分排序的文件就已经足够了。本书范例代码中的 `SortByTemperatureToMapFile` 类和 `LookupRecordsByTemperature` 类对这个问题进行了探究。通过使用 map 文件代替顺序文件，第一时间发现一个键所属的相关分区(使用 partitioner)是可能的，然后在 map 文件分区中执行记录查找操作效率将会更高。

9.2.3 全排序

如何用 Hadoop 产生一个全局排序的文件？最简单的方法是使用一个分区(a single partition)。[2]但该方法在处理大型文件时效率极低，因为一台机器必须处理所有输出文件，从而完全丧失了 MapReduce 所提供的并行架构的优势。

事实上仍有替代方案：首先，创建一系列排好序的文件；其次，串联这些文件；最后，生成一个全局排序的文件。主要的思路是使用一个 partitioner 来描述输出的全局排序。例如，可以为上述文件创建 4 个分区，在第一分区中，各记录的气温小于-10℃，第二分区的气温介于-10℃和 0℃之间，第三个分区的气温在 0℃和 10℃之间，最后一个分区的气温大于 10℃。

该方法的关键点在于如何划分各个分区。理想情况下，各分区所含记录数应该大致相等，使作业的总体执行时间不会受制于个别 reducer。在前面提到的分区方案中，各分区的相对大小如下所示。

气温范围	<-10℃	[-10℃, 0℃)	[0℃, 10℃)	≥10℃
记录所占的比例	11%	13%	17%	59%

① 参见 5.4.1 节对排序和合并顺序文件的介绍，了解如何运用 Hadoop 的排序程序来实现相同的功能。

② 更好的回答是使用 Pig(参见 16.6.4 节)、Hive(参见 17.7.1 节)、Crunch 或 Spark，两者都可以用一条指令来进行排序。

显然，记录没有均匀划分。只有深入了解整个数据集的气温分布才能建立更均匀的分区。写一个 MapReduce 作业来计算落入各个气温桶的记录数，并不困难。例如，图 9-1 显示了桶大小为 1℃时各桶的分布情况，各点分别对应一个桶。

获得气温分布信息意味着可以建立一系列分布非常均匀的分区。但由于该操作需要遍历整个数据集，因此并不实用。通过对键空间进行采样，就可较为均匀地划分数据集。采样的核心思想是只查看一小部分键，获得键的近似分布，并由此构建分区。幸运的是，Hadoop 已经内置若干采样器，不需要用户自己写。

`InputSampler` 类实现了 `Sampler` 接口，该接口的唯一成员方法(即 `getSample`)有两个输入参数(一个 `InputFormat` 对象和一个 `Job` 对象)，返回一系列样本键：

```
public interface Sampler<K, V> {
    K[] getSample(InputFormat<K, V> inf, Job job)
    throws IOException, InterruptedException;
}
```

图 9-1. 天气数据集合的气温分布

该接口通常不直接由客户端调用，而是由 `InputSampler` 类的静态方法 `writePartitionFile()`调用，目的是创建一个顺序文件来存储定义分区的键：

```
public static <K, V> void writePartitionFile(Job job, Sampler<K, V> sampler)
    throws IOException, ClassNotFoundException, InterruptedException
```

顺序文件由 TotalOrderPartitioner 使用，为排序作业创建分区。范例 9-5 整合了上述内容。

范例 9-5. 调用 TotalOrderPartitioner 按 IntWritable 键对顺序文件进行全局排序

```
public class SortByTemperatureUsingTotalOrderPartitioner extends Configured
  implements Tool {

  @Override
  public int run(String[] args) throws Exception {
    Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
    if (job == null) {
      return -1;
    }

    job.setInputFormatClass(SequenceFileInputFormat.class);
    job.setOutputKeyClass(IntWritable.class);
    job.setOutputFormatClass(SequenceFileOutputFormat.class);
    SequenceFileOutputFormat.setCompressOutput(job, true);
    SequenceFileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
    SequenceFileOutputFormat.setOutputCompressionType(job,
        CompressionType.BLOCK);

    job.setPartitionerClass(TotalOrderPartitioner.class);

    InputSampler.Sampler<IntWritable, Text> sampler =
      new InputSampler.RandomSampler<IntWritable, Text>(0.1, 10000, 10);

    InputSampler.writePartitionFile(job, sampler);

    // Add to DistributedCache
    Configuration conf = job.getConfiguration();
    String partitionFile =TotalOrderPartitioner.getPartitionFile(conf);
    URI partitionUri = new URI(partitionFile);    job.addCacheFile(partitionUri);

    return job.waitForCompletion(true) ? 0 : 1;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(
        new SortByTemperatureUsingTotalOrderPartitioner(), args);
    System.exit(exitCode);
  }
}
```

该程序使用 RandomSampler 以指定的采样率均匀地从一个数据集中选择样本。在

本例中，采样率被设为 0.1。RamdomSampler 的输入参数还包括最大样本数和最大分区(本例中这两个参数分别是 10,000 和 10，这也是 InputSampler 作为应用程序运行时的默认设置)，只要任意一个限制条件满足，即停止采样。采样器在客户端运行，因此，限制分片的下载数量以加速采样器的运行就尤为重要。在实践中，采样器的运行时间仅占作业总运行时间的一小部分。

为了和集群上运行的其他任务共享分区文件，InputSampler 需将其所写的分区文件加到分布式缓存中(参见 9.4.2 节)。

以下方案别以-5.6℃、13.9℃和 22.0℃为边界得到 4 个分区。易知，新方案比旧方案更为均匀。

气温范围	<5.6℃	[-5.6℃, 13.9℃)	[13.9℃, 22.0℃)	≥ 22.0℃
记录所占的比例	29%	24%	23%	24%

输入数据的特性决定如何挑选最合适的采样器。以 SplitSampler 为例，它只采样一个分片中的前 n 条记录。由于并未从所有分片中广泛采样，该采样器并不适合已经排好序的数据。[①]

另一方面，IntervalSample 以一定的间隔定期从分片中选择键，因此对于已排好序的数据来说是一个更好的选择。RandomSampler 是优秀的通用采样器。如果没有采样器可以满足应用需求(记住，采样目的是创建大小近似相等的一系列分区)，则只能写程序来实现 Sampler 接口。

InputSampler 类和 TotalOrderPartitioner 类的一个好特性是用户可以自由定义分区数，即 reducer 的数目。然而，由于 TotalOrderPartitioner 只用于分区边界均不相同的时候，因而当键空间较小时，设置太大的分区数可能会导致数据冲突。

以下是运行方式：

```
% hadoop jar hadoop-examples.jar SortByTemperatureUsingTotalOrderPartitioner \
  -D mapreduce.job.reduces=30 input/ncdc/all-seq output-totalsort
```

该程序输出 30 个已经内部排好序的分区。且分区 i 中的所有键都小于分区 i+1 中的键。

① 在某些应用中，待处理的数据文件通常要么已经排好序，要么至少已部分排序，例如，天气数据集就是按时间排序的。对于这类数据，使用 SplitSampler 会造成采样集合偏倚，因此最好采用 RandomSampler。

9.2.4 辅助排序

MapReduce 框架在记录到达 reducer 之前按键对记录排序，但键所对应的值并没有排序。甚至在不同的执行轮次中，这些值的排序也不固定，因为它们来自不同的 map 任务且这些 map 任务在不同轮次中的完成时间各不相同。一般来说，大多数 MapReduce 程序会避免让 reduce 函数依赖于值的排序。但是，有时也需要通过特定的方法对键进行排序和分组等以实现对值的排序。

例如，考虑如何设计一个 MapReduce 程序以计算每年的最高气温。如果全部记录均按照气温降序排列，则无需遍历整个数据集即可获得查询结果——获取各年份的首条记录并忽略剩余记录。尽管该方法并不是最佳方案，但演示了辅助排序的工作机理。

为此，首先构建一个同时包含年份和气温信息的组合键，然后对键值先按年份升序排序，再按气温降序排列：

```
1900 35℃
1900 34℃
1900 34℃
...
1901 36℃
1901 35℃
```

如果仅仅是使用组合键的话，并没有太大的帮助，因为这会导致同一年的记录可能有不同的键，通常这种情况下记录并不会被送到同一个 reducer 中。例如，(1900, 35℃)和(1900, 34℃)就可能被送到不同的 reducer 中。通过设置一个按照键的年份进行分区的 patitioner，可以确保同一年的记录会被发送到同一个 reducer 中。但是，这样做还不够。因为 partitioner 只保证每一个 reducer 接受一个年份的所有记录，而在一个分区之内，reducer 仍是通过键进行分组的分区：

```
          分区    组
1900 35℃ |      |
1900 34℃ |      |
1900 34℃ |      |
...
1900 36℃ |      |
1900 35℃ |      |
```

该问题的最终解决方案是进行分组设置。如果 reducer 中的值按照键的年份进行分组，则一个 reducer 组将包括同一年份的所有记录。鉴于这些记录已经按气温降序

排列，所以各组的首条记录就是这一年的最高气温：

```
              分区     组
1900 35℃ |       |
1900 34℃ |       |
1900 34℃ |       |
…
1900 36℃ |       |
1900 35℃ |       |
```

下面对记录按值排序的方法做一个总结。

- 定义包括自然键和自然值的组合键。
- 根据组合键对记录进行排序，即同时用自然键和自然值进行排序。
- 针对组合键进行分区和分组时均只考虑自然键。

1. Java 代码

综合起来便得到范例 9-6 中的源代码，该程序再一次使用了纯文本输入。

范例 9-6. 该应用程序通过对键中的气温进行排序来找出最高气温

```
public class MaxTemperatureUsingSecondarySort
  extends Configured implements Tool {

  static class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, IntPair, NullWritable> {

    private NcdcRecordParser parser = new NcdcRecordParser();

    @Override
    protected void map(LongWritable key, Text value,
        Context context) throws IOException, InterruptedException {

      parser.parse(value);
      if (parser.isValidTemperature()) {
        context.write(new IntPair(parser.getYearInt(),
            parser.getAirTemperature()), NullWritable.get());
      }
    }
  }

  static class MaxTemperatureReducer
    extends Reducer<IntPair, NullWritable, IntPair, NullWritable> {

    @Override
    protected void reduce(IntPair key, Iterable<NullWritable> values,
        Context context) throws IOException, InterruptedException {
```

```
    context.write(key, NullWritable.get());
  }
}

public static class FirstPartitioner
  extends Partitioner<IntPair, NullWritable> {

  @Override
  public int getPartition(IntPair key, NullWritable value, int numPartitions) {
    // multiply by 127 to perform some mixing
    return Math.abs(key.getFirst() * 127) % numPartitions;
  }
}

public static class KeyComparator extends WritableComparator {
  protected KeyComparator() {
    super(IntPair.class, true);
  }
  @Override
  public int compare(WritableComparable w1, WritableComparable w2) {
    IntPair ip1 = (IntPair) w1;
    IntPair ip2 = (IntPair) w2;
    int cmp = IntPair.compare(ip1.getFirst(), ip2.getFirst());
    if (cmp != 0) {
      return cmp;
    }
    return -IntPair.compare(ip1.getSecond(), ip2.getSecond()); //reverse
  }
}

public static class GroupComparator extends WritableComparator {
  protected GroupComparator() {
    super(IntPair.class, true);
  }
  @Override
  public int compare(WritableComparable w1, WritableComparable w2) {
    IntPair ip1 = (IntPair) w1;
    IntPair ip2 = (IntPair) w2;
    return IntPair.compare(ip1.getFirst(), ip2.getFirst());
  }
}

@Override
public int run(String[] args) throws Exception {
  Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
  if (job == null) {
    return -1;
  }

  job.setMapperClass(MaxTemperatureMapper.class);
  job.setPartitionerClass(FirstPartitioner.class);
  job.setSortComparatorClass(KeyComparator.class);
  job.setGroupingComparatorClass(GroupComparator.class);
  job.setReducerClass(MaxTemperatureReducer.class);
```

```
    job.setOutputKeyClass(IntPair.class);
    job.setOutputValueClass(NullWritable.class);

    return job.waitForCompletion(true) ? 0 : 1;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new MaxTemperatureUsingSecondarySort(), args);
    System.exit(exitCode);
  }
}
```

在上述 mapper 中，我们利用 IntPair 类定义了一个代表年份和气温的组合键，该类实现了 Writable 接口。IntPair 与 TextPair 类相似，后者可以参见 5.3.3 节的相关讨论。由于可以根据各 reducer 的组合键获得最高气温，因此无需在值上附加其他信息，使用 NullWritable 即可。根据辅助排序，reducer 输出的第一个键就是包含年份和最高气温信息的 IntPair 对象。IntPair 的 toString()方法返回一个以制表符分隔的字符串，因而该程序输出一组由制表符分隔的年份/气温对。

许多应用需要访问所有已排序的值，而非像上例一样只需要第一个值。鉴于在 reducer 中用户只能够获取第一个键，所以必须通过填充值字段来获取所有已排序的值，这样不可避免会在键和值之间产生一些冗余信息。

我们创建一个自定义的 partitioner 以按照组合键的首字段(年份)进行分区，即 FirstPartitioner。为了按照年份(升序)和气温(降序)排列键，我们使用 setSortComparatorClass()设置一个自定义键 comparator(即 KeyComparator)，以抽取字段并执行比较操作。类似的，为了按年份对键进行分组，我们使用 SetGroupingComparatorClass 来自定义一个分组 comparator，只取键的首字段进行比较。[①]

运行该程序，返回各年的最高气温：

```
% hadoop jar hadoop-examples.jar MaxTemperatureUsingSecondarySort input/ncdc/all \
> output-secondarysort
% hadoop fs -cat output-secondarysort/part-* | sort | head
1901    317
1902    244
1903    289
1904    256
```

① 为简单起见，这里自定义的 comparator 并未经过优化。参见 5.3.3 节对 RawComparator 的介绍，了解如何提高运行效率。

```
1905    283
1906    294
1907    283
1908    289
1909    278
1910    294
```

2. Streaming

我们可以借助 Hadoop 所提供的一组库来实现 Streaming 的辅助排序，下面就是用来进行辅助排序的驱动：

```
% hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
    -D stream.num.map.output.key.fields=2 \
    -D mapreduce.partition.keypartitioner.options=-k1,1 \
    -D mapreduce.job.output.key.comparator.class=\
  org.apache.hadoop.mapred.lib.KeyFieldBasedComparator \
    -D mapreduce.partition.keycomparator.options="-k1n -k2nr" \
    -files secondary_sort_map.py,secondary_sort_reduce.py \
    -input input/ncdc/all \
    -output output-secondarysort-streaming \
    -mapper ch09-mr-features/src/main/python/secondary_sort_map.py \
    -partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner \
    -reducer ch09-mr-features/src/main/python/secondary_sort_reduce.py
```

范例 9-7 中的 map 函数输出年份和气温两个字段。为了将这两个字段看成一个组合键，需要将 `stream.num.map.output.key.fields` 的值设为 2。这意味着值是空的，就像 Java 程序(范例 9-6)一样。

范例 9-7. 针对辅助排序的 map 函数(Python 版本)

```
#!/usr/bin/env python

import re
import sys

for line in sys.stdin:
  val = line.strip()
  (year, temp, q) = (val[15:19], int(val[87:92]), val[92:93])
  if temp == 9999:
    sys.stderr.write("reporter:counter:Temperature,Missing,1\n")
  elif re.match("[01459]", q):
    print "%s\t%s" % (year, temp)
```

鉴于我们并不期望根据整个组合键来划分数据集，因此可以利用 `KeyFieldBasedPatitioner` 类以组合键的一部分进行划分。具体实现是使用 `mapreduce.partition.keypartitioner.options` 配置该 partitioner。在上例中，值`-k1,1` 表示该 partitioner 只使用组合键的第一个字段。`mapreduce.map.`

output.key.field.separator 属性所定义的字符串能分隔各个字段(默认是制表符)。

接下来，我们还需要一个 comparator 以对年份字段升序排列、对气温字段降序排列，使 reduce 函数能够方便地返回各组中的第一个记录。Hadoop 提供的 KeyFieldBasedComparator 类能有效解决这个问题。该类通过 mapreduce.partition.keycomparator.options 属性来设置排列次序，其格式规范与 GNU *sort* 类似。本例中的-k1n -k2nr 选项表示“首字段按数值顺序排序，字段按数值顺序反向排序”。与 KeyFieldBasedPartitioner 类似，KeyFieldBasedComparator 使用 map 输出键分隔符将一个键划分成多个字段。

Java 版本的程序需要定义分组 comparator。但是在 Streaming 中，组并未以任何方式划分，因此必须在 reduce 函数中不断地查看年份是否改变来检测组的边界(范例 9-8)。

范例 9-8. 针对辅助排序的 reduce 函数(Python 版本)

```
#!/usr/bin/env python

import sys

last_group = None
for line in sys.stdin:
  val = line.strip()
  (year, temp) = val.split("\t")
  group = year
  if last_group != group:
    print val
    last_group = group
```

运行此程序之后，得到与 Java 版本一样的结果。

最后牢记一点：KeyFieldBasedPartitioner 和 KeyFieldBasedComparator 不仅在 Streaming 程序中使用，也能在 Java 版本的 MapReduce 程序中使用。

9.3 连接

MapReduce 能够执行大型数据集间的“连接”(join)操作，但是，自己从头写相关代码来执行连接的确非常棘手。除了写 MapReduce 程序，还可以考虑采用更高级的框架，如 Pig、Hive、Cascading、Cruc 或 Spark 等，它们都将连接操作视为整个实现的核心部分。

先简要地描述待解决的问题。假设有两个数据集：气象站数据库和天气记录数据集，并考虑如何合二为一。一个典型的查询是：输出各气象站的历史信息，同时

各行记录也包含气象站的元数据信息，如图 9-2 所示。

连接操作的具体实现技术取决于数据集的规模及分区方式。如果一个数据集很大(例如天气记录)而另外一个集合很小，以至于可以分发到集群中的每一个节点之中(例如气象站元数据)，则可以执行一个 MapReduce 作业，将各个气象站的天气记录放到一块(例如，根据气象站 ID 执行部分排序)，从而实现连接。mapper 或 reducer 根据各气象站 ID 从较小的数据集合中找到气象站元数据，使元数据能够被写到各条记录之中。该方法将在 9.4 节中详细介绍，它侧重于将数据分发到集群中节点的机制。

图 9-2. 两个数据集的内连接

连接操作如果由 mapper 执行，则称为“map 端连接”；如果由 reducer 执行，则称为“reduce 端连接”。

如果两个数据集的规模均很大，以至于没有哪个数据集可以被完全复制到集群的每个节点，我们仍然可以使用 MapReduce 来进行连接，至于到底采用 map 端连接还是 reduce 端连接，则取决于数据的组织方式。最常见的一个例子便是用户数据库和用户活动日志(例如访问日志)。对于一个热门服务来说，将用户数据库(或日志)分发到所有 MapReduce 节点中是行不通的。

9.3.1 map 端连接

在两个大规模输入数据集之间的 map 端连接会在数据到达 map 函数之前就执行连接操作。为达到该目的，各 map 的输入数据必须先分区并且以特定方式排序。各个输入数据集被划分成相同数量的分区，并且均按相同的键(连接键)排序。同一键的所有记录均会放在同一分区之中。听起来似乎要求非常严格(的确如此)，但这的确合乎 MapReduce 作业的输出。

map 端连接操作可以连接多个作业的输出，只要这些作业的 reducer 数量相同、键相同并且输出文件是不可切分的(例如，借助于小于一个 HDFS 块、或进行 gzip 压缩来实现)。在天气例子中，如果气象站文件以气象站 ID 部分排序，记录文件也以气象站 ID 部分排序，而且 reducer 的数量相同，则就满足了执行 map 端连接的前提条件。

利用 `org.apache.hadoop.mapreduce.join` 包中的 `CompositeInputFormat` 类来运行一个 map 端连接。`CompositeInputFormat` 类的输入源和连接类型(内连接或外连接)可以通过一个连接表达式进行配置，连接表达式的语法简单。详情与示例可参见包文档。

`org.apache.hadoop.examples.Join` 是一个通用的执行 map 端连接的命令行程序样例。该例运行一个基于多个输入数据集的 mapper 和 reducer 的 MapReduce 作业，以执行给定的连接操作。

9.3.2 reduce 端连接

由于 reduce 端连接并不要求输入数据集符合特定结构，因而 reduce 端连接比 map 端连接更为常用。但是，由于两个数据集均需经过 MapReduce 的 shuffle 过程，所以 reduce 端连接的效率往往要低一些。基本思路是 mapper 为各个记录标记源，并且使用连接键作为 map 输出键，使键相同的记录放在同一个 reducer 中。以下技术能帮助实现 reduce 端连接。

1. 多输入

数据集的输入源往往有多种格式，因此可以使用 MultipleInputs 类(参见 8.2.4 节)来方便地解析和标注各个源。

2. 辅助排序

如前所述，reducer 将从两个源中选出键相同的记录，但这些记录不保证是经过排序的。然而，为了更好地执行连接操作，一个源的数据排列在另一个源的数据前是非常重要的。以天气数据连接为例，对应每个键，气象站记录的值必须是最先看到的，这样 reducer 能够将气象站名称填到天气记录之中再马上输出。虽然也可以不指定数据传输次序，并将待处理的记录缓存在内存之中，但应该尽量避免这种情况，因为其中任何一组的记录数量可能非常庞大，远远超出 reducer 的可用内存容量。

9.2.4 节介绍如何对 reducer 所看到的每个键的值进行排序，所以在此也用到了辅助排序技术。

为标记每个记录，我们使用第 5 章的 TextPair 类，包括键(存储气象站 ID)和“标记”。在这里，“标记”是一个虚拟的字段，其唯一目的是用于记录的排序，使气象站记录比天气记录先到达。一种简单的做法就是：对于气象站记录，“标记”值为 0；对于天气记录，“标记”值为 1。范例 9-9 和范例 9-10 分别描述了执行该任务的两个 mapper 类。

范例 9-9. 在 reduce 端连接中，标记气象站记录的 mapper

```
public class JoinStationMapper
    extends Mapper<LongWritable, Text, TextPair, Text> {
  private NcdcStationMetadataParser parser = new NcdcStationMetadataParser();

  @Override
  protected void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
    if (parser.parse(value)) {
      context.write(new TextPair(parser.getStationId(), "0"),
          new Text(parser.getStationName()));
    }
  }
}
```

范例 9-10. 在 reduce 端连接中标记天气记录的 mapper

```
public class JoinRecordMapper
    extends Mapper<LongWritable, Text, TextPair, Text> {
```

```
  private NcdcRecordParser parser = new NcdcRecordParser();

  @Override
  protected void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
    parser.parse(value);
    context.write(new TextPair(parser.getStationId(), "1"), value);
  }

}
```

reducer 知道自己会先接收气象站记录。因此从中抽取出值，并将其作为后续每条输出记录的一部分写到输出文件。如范例 9-11 所示。

范例 9-11. 用于连接已标记的气象站记录和天气记录的 reducer

```
public class JoinReducer extends Reducer<TextPair, Text, Text, Text> {

  @Override
  protected void reduce(TextPair key, Iterable<Text> values, Context context)
      throws IOException, InterruptedException {
    Iterator<Text> iter = values.iterator();
    Text stationName = new Text(iter.next());
    while (iter.hasNext()) {
      Text record = iter.next();
      Text outValue = new Text(stationName.toString() + "\t" + record.toString());
      context.write(key.getFirst(), outValue);
    }
  }
}
```

上述代码假设天气记录的每个气象站 ID 恰巧与气象站数据集中的一条记录准确匹配。如果该假设不成立，则需要泛化代码，使用另一个 TextPair 将标记放入值的对象中。reduce()方法在处理天气记录之前，要能够区分哪些记录是气象站名称，检测(和处理)缺失或重复的记录。

在 reducer 的迭代部分中，对象被重复使用(为了提高效率)。因此，从第一个 Text 对象获得站点名称(即 stationName)就非常关键。

```
Text stationName = new Text(iter.next());
```

如果不执行该语句，stationName 就会指向上一条记录的值，这显然是错的。

将作业连接在一起通过驱动类来完成，如范例 9-12 所示。这里，关键点在于根据组合键的第一个字段(即气象站 ID)进行分区和分组，即使用一个自定义的 partitioner(即 KeyPartitioner)和一个自定义的分组 comparator (FirstComparator，作为 TextPair 的嵌套类)。

范例 9-12. 对天气记录和气象站名称执行连接操作

```
public class JoinRecordWithStationName extends Configured implements Tool {

  public static class KeyPartitioner extends Partitioner<TextPair, Text> {
    @Override
    public int getPartition(TextPair key, Text value, int numPartitions) {
      return (key.getFirst().hashCode() & Integer.MAX_VALUE) % numPartitions;
    }
  }

  @Override
  public int run(String[] args) throws Exception {
    if (args.length != 3) {
      JobBuilder.printUsage(this, "<ncdc input> <station input> <output>");
      return -1;
    }

    Job job = new Job(getConf(), "Join weather records with station names");
    job.setJarByClass(getClass());

    Path ncdcInputPath = new Path(args[0]);
    Path stationInputPath = new Path(args[1]);
    Path outputPath = new Path(args[2]);

    MultipleInputs.addInputPath(job, ncdcInputPath,
        TextInputFormat.class, JoinRecordMapper.class);
    MultipleInputs.addInputPath(job, stationInputPath,
        TextInputFormat.class, JoinStationMapper.class);
    FileOutputFormat.setOutputPath(job, outputPath);

    job.setPartitionerClass(KeyPartitioner.class);
    job.setGroupingComparatorClass(TextPair.FirstComparator.class);

    job.setMapOutputKeyClass(TextPair.class);

    job.setReducerClass(JoinReducer.class);

    job.setOutputKeyClass(Text.class);

    return job.waitForCompletion(true) ? 0 : 1;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new JoinRecordWithStationName(), args);
    System.exit(exitCode);
  }
}
```

在样本数据上运行这个程序，获得以下输出：

```
011990-99999 SIHCCAJAVRI        0067011990999991950051507004...
011990-99999 SIHCCAJAVRI        0043011990999991950051512004...
011990-99999 SIHCCAJAVRI        0043011990999991950051518004...
```

```
012650-99999 TYNSET-HANSMOEN      0043012650999991949032412004...
012650-99999 TYNSET-HANSMOEN      0043012650999991949032418004...
```

9.4 边数据分布

“边数据”(side data)是作业所需的额外的只读数据，以辅助处理主数据集。所面临的挑战在于如何使所有 map 或 reduce 任务(这些任务散布在集群内部)都能够方便而高效地使用边数据。

9.4.1 利用 JobConf 来配置作业

Configuration 类(或者旧版 MapReduce API 的 JobConf 类)的各种 setter 方法能够方便地配置作业的任一键-值对。如果仅需向任务传递少量元数据则非常有用。

在任务中，用户可以通过 Context 类的 getConfiguration()方法获得配置信息。(在旧版 API 中，做法更加复杂一点：需要重写 Mapper 或者 Reducer 类的 configure()方法，并调用传入 JobConf 对象的 getter 方法。通常情况下，可将数据以实例字段的形式保存，使得其可在 map()或者 reduce()方法中使用。)

一般情况下，基本类型足以应付元数据编码。但对于更复杂的对象，用户要么自己处理序列化工作(这需要实现一个对象与字符串之间的双向转换机制)，要么使用 Hadoop 提供的 Stringifier 类。DefaultStringifier 使用 Hadoop 的序列化框架来序列化对象。详情参见 5.3 节。

但是这种机制会加大 MapReduce 组件的内存开销压力，因此，并不适合传输多达几千字节的数据量。作业配置总是由客户端、application master 和任务 JVM 读取。每次读取配置时，所有项都被读入到内存(即使暂时不用的属性项也不例外)。

9.4.2 分布式缓存

与在作业配置中序列化边数据的技术相比，Hadoop 的分布式缓存机制更受青睐，它能够在任务运行过程中及时地将文件和存档复制到任务节点以供使用。为了节约网络带宽，在每一个作业中，各个文件通常只需要复制到一个节点一次。

1. 用法

对于使用 GenericOptionsParser(本书中多处程序均用到该类，参见 6.2.2 节的相关讨

论)的工具来说，用户可以使用-files 选项指定待分发的文件，文件内包含以逗号隔开的 URI 列表。文件可以存放在本地文件系统、HDFS 或其他 Hadoop 可读文件系统(例如 S3)之中。如果尚未指定文件系统，则这些文件被默认是本地的。即使默认文件系统并非本地文件系统，这也是成立的。

用户可以使用-archives 选项向自己的任务中复制存档文件(JAR 文件、ZIP 文件、tar 文件和 gzipped tar 文件)，这些文件会被解档到任务节点。-libjars 选项会把 JAR 文件添加到 mapper 和 reducer 任务的类路径中。如果作业 JAR 文件并没包含库 JAR 文件，这点会很有用。

以下指令显示如何使用分布式缓存来共享元数据文件，从而得到气象站的名称：

```
% hadoop jar hadoop-examples.jar \
    MaxTemperatureByStationNameUsingDistributedCacheFile \
    -files input/ncdc/metadata/stations-fixed-width.txt input/ncdc/all output
```

该命令将本地文件 *stations-fixed-width.txt*(未指定文件系统，从而被自动解析为本地文件)复制到任务节点，从而可以查找气象站名称。范例 9-13 描述了类 MaxTemperatureByStationNameUsingDistributedCacheFile 的代码。

范例 9-13. 查找各气象站的最高气温并显示气象站名称，气象站文件是一个分布式缓存文件

```
public class MaxTemperatureByStationNameUsingDistributedCacheFile
  extends Configured implements Tool {

  static class StationTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

    private NcdcRecordParser parser = new NcdcRecordParser();

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

      parser.parse(value);
      if (parser.isValidTemperature()) {
        context.write(new Text(parser.getStationId()),
            new IntWritable(parser.getAirTemperature()));
      }
    }
  }
}
```

```
static class MaxTemperatureReducerWithStationLookup
  extends Reducer<Text, IntWritable, Text, IntWritable> {

  private NcdcStationMetadata metadata;

  @Override
  protected void setup(Context context)
      throws IOException, InterruptedException {
    metadata = new NcdcStationMetadata();
    metadata.initialize(new File("stations-fixed-width.txt"));
  }

  @Override
  protected void reduce(Text key, Iterable<IntWritable> values,
      Context context) throws IOException, InterruptedException {

    String stationName = metadata.getStationName(key.toString());

    int maxValue = Integer.MIN_VALUE;
    for (IntWritable value : values) {
      maxValue = Math.max(maxValue, value.get());
    }
    context.write(new Text(stationName), new IntWritable(maxValue));
  }
}

@Override
public int run(String[] args) throws Exception {
  Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
  if (job == null) {
    return -1;
  }

  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);

  job.setMapperClass(StationTemperatureMapper.class);
  job.setCombinerClass(MaxTemperatureReducer.class);
  job.setReducerClass(MaxTemperatureReducerWithStationLookup.class);

  return job.waitForCompletion(true) ? 0 : 1;
}
```

```
  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(
        new MaxTemperatureByStationNameUsingDistributedCacheFile(), args);
    System.exit(exitCode);
  }
}
```

该程序通过气象站查找最高气温，因此 mapper (StationTemperatureMapper)仅仅输出(气象站 ID，气温)对。对于 combiner，该程序重用 MaxTemperatureReducer (参见第 2 章和第 6 章)来为 map 端的 map 输出分组获得最高气温。reducer (MaxTemperatureReducerWithStationLookup)则有所不同，它不仅要查找最高气温，还需要根据缓存文件查找气象站名称。

该程序调用 reducer 的 setup()方法来获取缓存文件；输入参数是文件的原始名称，文件的路径与任务的工作目录相同。

当文件无法整个放到内存中时，可以使用分布式缓存进行复制。由于充当了一种在盘检索格式(参见 5.4.2 节)，Hadoop map 文件在这方面非常有用。由于 map 文件是一组已定义目录结构的文件，用户可以将这些文件整理成存档格式(JAR、ZIP、tar 或 gzipped tar)，再用-archives 选项将其加入缓存。

以下是输出的小片段，显示部分气象站的最高气温值。

```
PEATS RIDGE WARATAH          372
STRATHALBYN RACECOU          410
SHEOAKS AWS                  399
WANGARATTA AERO              409
MOOGARA                      334
MACKAY AERO                  331
```

2. 工作机制

当用户启动一个作业，Hadoop 会把由-files、-archives 和-libjars 等选项所指定的文件复制到分布式文件系统(一般是 HDFS)之中。接着，在任务运行之前，节点管理器将文件从分布式文件系统复制到本地磁盘(缓存)使任务能够访问文件。此时，这些文件就被视为“本地化”了。从任务的角度来看，这些文件就已经在那儿了，以符号链接的方式指向任务的工作目录。此外，由-libjars 指定的文件会在任务启动前添加到任务的类路径(classpath)中。

节点管理器为缓存中的文件各维护一个计数器来统计这些文件的被使用情况。当任务即将运行时，该任务所使用的所有文件的对应计数器值增 1；当任务执行完毕

之后，这些计数器值均减 1。仅当文件不在使用中(此时计数达到 0)，才有资格删除。当节点缓存的容量超过一定范围(默认 10 GB)时，需要根据最近最少使用原则删除文件以腾出空间来装载新文件。缓存的大小可以通过属性 yarn.nodemanager.localizer.cache.target-size-mb 来配置。

尽管该机制并不确保在同一个节点上运行的同一作业的后续任务肯定能在缓存中找到文件，但是成功的概率相当大。原因在于作业的多个任务在调度之后几乎同时开始运行，因此，不会有足够多的其他作业在运行而导致原始任务的文件从缓存中被删除。

3. 分布式缓存 API

由于可以通过 GenericOptionsParser 间接使用分布式缓存(如范例 9-13 所示)，大多数应用不需要使用分布式缓存 API。然而，如果没有使用 GenericOptionsParser，那么可以使用 Job 中的 API 将对象放进分布式缓存中[①]。以下是 Job 中相关的方法：

```
public void addCacheFile(URI uri)
public void addCacheArchive(URI uri)
public void setCacheFiles(URI[] files)
public void setCacheArchives(URI[] archives)
public void addFileToClassPath(Path file)
public void addArchiveToClassPath(Path archive)
public void createSymlink()
```

在缓存中可以存放两类对象：文件(files)和存档(achives)。文件被直接放置在任务节点上，而存档则会被解档之后再将具体文件放置在任务节点上。每种对象类型都包含三种方法：addCache*XXXX*()、setCache*XXXX*s()和add*XXXX*ToClassPath()。其中，addCache*XXXX*()方法将文件或存档添加到分布式缓存，setCache*XXXX*s()方法将一次性向分布式缓存中添加一组文件或存档(之前调用所生成的集合将被替换)，add*XXXX*ToClassPath()方法将文件或存档添加到 MapReduce 任务的类路径。表 9-7 对上述 API 方法与 GenericOptionsParser 选项(参见表 6-1)做了一个比较。

① 如果用的是老版本的 MapReduce API，可以在 org.apache.hadoop.filecache.DistributedCache 中找到同样的方法。

表 9-7. 分布式缓存 API

Job 的 API 名称	GenericOptionsParser 的等价选项	说明
addCacheFile(URI uri) setCacheFiles(URI[] files)	-files *file1,file2,...*	将文件添加到分布式缓存，以备将来被复制到任务节点
addCacheArchive(URI uri) setCacheArchives(URI[] files)	-archives archive1, archive2, ...	将存档添加到分布式缓存，以备将来被复制到任务节点，并在节点解档
addFileToClassPath (Path file)	-libjars jar1, jar2, ...	将文件添加到分布式缓存，以备将来被复制到 MapReduce 任务的类路径中。文件并不会被解档，因此适合向类路径添加 JAR 文件
addArchiveToClassPath (Path archive)	无	将存档添加到分布式缓存，以备将来解档、添加到 MapReduce 的类路径中。当想向类路径添加目录和文件时，这种方式比较有用，因为用户可以创建一个包含指定文件的存档。此外，用户也可以创建一个 JAR 文件，并使用addFileTo ClassPath()，效果相同

add 和 set 方法中的输入参数 URI 是指在作业运行时位于共享文件系统中的(一组)文件。而 GenericOptionsParser 选项(例如，-files)所指定的文件可以是本地文件；如果是本地文件的话，则会被复制到默认的共享文件系统(一般是 HDFS)。

这也是使用 Java API 和使用 GenericOptionsParser 的关键区别：Java API 的 add 和 set 方法不会将指定文件复制到共享文件系统中，但 GenericOptionsParser 会这样做。

从任务中获取分布式缓存文件在工作机理上和以前是一样的：通过名称访问本地化的文件，如范例 9-13 中所示。之所以起作用，是因为 MapReduce 总会在任务的工作目录和添加到分布式缓存中的每个文件或存档之间建立符号化链接。[①] 存档被解档后就能使用嵌套的路径访问其中的文件。

① 在 Hadoop 1 中，本地化文件并不总是符号化链接的，有时必须得用 JobContext 提供的方法来获取本地化文件的路径。Hadoop 2 中这个限制没有了。

9.5 MapReduce 库类

Hadoop 还为 mapper 和 reducer 提供了一个包含了常用函数的库。表 9-8 简要描述了这些类。如需了解详细用法，可参考相关 Java 文档。

表 9-8. MapReduce 库的类

类的名称	描述
ChainMapper，ChainReducer	在一个 mapper 中运行多个 mapper，再运行一个 reducer，随后在一个 reducer 中运行多个 mapper。(符号表示：M+RM*，其中 M 是 mapper，R 是 reducer。)与运行多个 MapReduce 作业相比，该方案能够显著降低磁盘 I/O 开销
FieldSelectionMapReduce (旧版 API)：FieldSelectionMapper 和 FieldSelectionReducer (新版 API)	能从输入键和值中选择字段(类似 Unix 的 cut 命令)，并输出键和值的 mapper 和 reducer
IntSumReducer，LongSumReducer	对各键的所有整数值执行求和操作的 reducer
InverseMapper	一个能交换键和值的 mapper
MultithreadedMapRunner (旧版 API) MultithreadedMapper (新版 API)	一个能在多个独立线程中分别并发运行 mapper 的 mapper(或者旧版 API 中的 map runner)。该技术对于非 CPU 受限的 mapper 比较有用
TokenCounterMapper	将输入值分解成独立的单词(使用 Java 的 StringTokenizer)并输出每个单词和计数值 1 的 mapper
RegexMapper	检查输入值是否匹配某正则表达式，输出匹配字符串和计数为 1 的 mapper

第III部分
Hadoop 的操作

第 10 章　构建 Hadoop 集群
第 11 章　管理 Hadoop

第 10 章

构建 Hadoop 集群

本章介绍如何在一个计算机集群上构建 Hadoop 系统。尽管在单机上运行 HDFS、MapReduce 和 YARN 有助于学习这些系统，但是要想执行一些有价值的工作，必须在多节点系统上运行。

有多个选择来获得一个 Hadoop 集群，从建立一个专属集群，到在租借的硬件设备上运行 Hadoop 系统，乃至于使用云端作为托管服务提供的 Hadoop。被托管的选项数很多，这里就不逐一列举，但是即使你选择自己建立一个 Hadoop 集群，仍然会有很多安装选项要考虑。

Apache tarball

Apache Hadoop 项目及相关的项目为每次发布提供了二进制(和源)压缩包(tarball)。用二进制压缩包安装最灵活，但工作量也最大，这是由于需要确定安装文件、配置文件和日志文件在文件系统中的位置、正确设置文件访问权限等等。

Packages

从 Apache Bigtop 项目(*http://bigtop.apache.org*)及所有 Hadoop 供应商那里都可以获取 RPM 和 Debian 包。这些包比压缩包有更多的优点，它们提供了一个一致性的文件系统布局，可以作为一个整体进行测试(这样可以知道 Hadoop 和 Hive 的多个版本能够在一起运行)，并且它们可以和配置管理工具如 Puppet 一起运行。

Hadoop 集群管理工具

有一些工具用于 Hadoop 集群全生命期的安装和管理，Cloudera Manager 和 Apache Ambari 就是这样的专用工具。它们提供了简单的 WebUI，并且被推荐给大多数用户和操作者用以构建 Hadoop 集群。这些工具集成了 Hadoop 运行有关的操

作知识。例如，它们基于硬件特点使用启发式方法来选择好的默认值用于 Hadoop 配置设置。对于更复杂的构建，例如 HA，或安全 Hadoop，这些管理工具提供了经过测试的向导，能够帮助在短时间内建立一个能够工作的集群。最后，它们增加了额外的、其他安装选项没有提供的特性，例如统一监控和日志搜索，滚动升级(升级集群时不用经历停机)。

本章和下一章提供了足够的信息来构建和操作基础的 Hadoop 集群。然而，即使有些读者可能正在使用 Hadoop 集群管理工具或 Hadoop 服务，这些工具和服务帮助完成了大量常规的构建和维护工作，对于这些读者，阅读这两章内容仍然有助于从操作的角度深入理解 Hadoop 的工作机制。如果想深入了解，建议阅读 Eric Sammer 所著的 *Hadoop Operation* 一书(O'Reilly，2012)。

10.1 集群规范

Hadoop 运行在商业硬件上。用户可以选择普通硬件供应商生产的标准化的、广泛有效的硬件来构建集群，无需使用特定供应商生产的昂贵、专有的硬件设备。

首先澄清两点。第一，商业硬件并不等同于低端硬件。低端机器常常使用便宜的零部件，其故障率远高于更贵一些(但仍是商业级别)的机器。当用户管理几十台、上百台，甚至几千台机器时，选择便宜的零部件并不划算，因为更高的故障率推高了维护成本。第二，也不推荐使用大型的数据库级别的机器，因为这类机器的性价比太低了。用户可能会考虑使用少数几台数据库级别的机器来构建一个集群，使其性能达到一个中等规模的商业机器集群。然而，某一台机器所发生的故障会对整个集群产生更大的负面影响，因为大多数集群硬件将无法使用。

硬件规格很快就会过时。但为了举例说明，下面列举一下硬件规格。在 2014 年，运行 Hadoop 的 datanode 和 YARN 节点管理器的典型机器有以下规格：

- 处理器，两个六核/八核 3 GHz CPU
- 内存，64～512 GB ECC RAM[①]
- 存储器，12～24×1～4 TB SATA 硬盘
- 网络，带链路聚合的千兆以太网

① 据有部分用户报告称，在 Hadoop 集群中使用非 ECC 内存时会产生校验和错误，因此我强烈建议大家采用 ECC 内存。

尽管各个集群采用的硬件规格肯定有所不同，但是 Hadoop 一般使用多核 CPU 和多磁盘，以充分利用硬件的强大功能。

为何不使用 RAID?

尽管建议采用 RAID(Redundant Array of Independent Disk，即磁盘阵列)作为 namenode 的存储器以保护元数据，但是若将 RAID 作为 datanode 的存储设备则不会给 HDFS 带来益处。HDFS 所提供的节点间数据复制技术已可满足数据备份需求，无需使用 RAID 的冗余机制。

此外，尽管 RAID 条带化技术(RAID 0)被广泛用于提升性能，但是其速度仍然比用在 HDFS 里的 JBOD(Just a Bunch Of Disks)配置慢。JBOD 在所有磁盘之间循环调度 HDFS 块。RAID 0 的读/写操作受限于磁盘阵列中响应最慢的盘片的速度，而 JBOD 的磁盘操作均独立，因而平均读/写速度高于最慢盘片的读/写速度。需要强调的是，各个磁盘的性能在实际使用中总存在相当大的差异，即使对于相同型号的磁盘。在一些针对某一雅虎集群的基准评测中，在一项测试(Gridmix)中，JBOD 比 RAID 0 快 10%；在另一测试(HDFS 写吞吐量)中，JBOD 比 RAID 0 快 30%。

最后，若 JBOD 配置的某一磁盘出现故障，HDFS 可以忽略该磁盘，继续工作。而 RAID 的某一盘片故障会导致整个磁盘阵列不可用，进而使相应节点失效。

10.1.1 集群规模

一个 Hadoop 集群到底应该多大？这个问题并无确切的答案。但是，Hadoop 的魅力在于用户可以在初始阶段构建一个小集群(大约 10 个节点)，并随存储与计算需求增长持续扩充。从某种意义上讲，更恰当的问题是“你的集群需要增长得多快？” 用户可以通过下面这个关于存储的例子得到更深的体会。

假如数据每天增长 1 TB。如果采用三路 HDFS 复制技术，则每天需要增加 3 TB 存储能力。再加上一些中间文件和日志文件(约占 30%)所需空间，基本上相当于每周添设一台机器(2014 年的典型机器)。实际上，用户一般不会每周买一台新机器并将其加入集群。类似粗略计算的意义在于让用户了解集群的规模。本例中，一个集群保存两年的数据大致需要 100 台机器。

Master 节点场景

集群的规模不同，运行 master 守护进程的配置也不同，包括：namenode、辅助 namenode、资源管理器及历史服务器。对于一个小集群(几十个节点)而言，在一台 master 机器上同时运行 namenode 和资源管理器通常是可接受的(只要至少一份 namenode 的元数据被另存在远程文件系统中)。然而，随着集群规模增大，完全有理由分离它们。

由于 namenode 在内存中保存整个命名空间中的所有文件元数据和块元数据，其内存需求很大。辅助 namenode 在大多数时间里空闲，但是它在创建检查点时的内存需求与主 namenode 差不多。详情参见 11.1.1 节对文件系统映像和编辑日志的讨论。一旦文件系统包含大量文件，单台机器的物理内存便无法同时运行主 namenode 和辅助 namenode。

除了简单的资源需求，在分开的机器上运行 master 的主要理由是为了高可用性。HDFS 和 YARN 都支持以主备方式运行 master 的配置。如果主 master 故障，在不同硬件上运行的备机将接替主机，且几乎不会对服务造成干扰。在 HDFS 中，辅助 namenode 的检查点功能由备机执行，所以不需要同时运行备机和辅助 namenode。

配置和运行 Hadoop HA 不是本书的内容。可以参考 Hadoop 网站或供应商文档以获取更详细的信息。

10.1.2 网络拓扑

Hadoop 集群架构通常包含两级网络拓扑，如图 10-1 所示。一般来说，各机架装配 30～40 个服务器，共享一个 10 GB 的交换机(该图中各机架只画了 3 个服务器)，各机架的交换机又通过上行链路与一个核心交换机或路由器(至少为 10 GB 或更高)互联。该架构的突出特点是同一机架内部的节点之间的总带宽要远高于不同机架上的节点间的带宽。

机架的注意事项

为了达到 Hadoop 的最佳性能，配置 Hadoop 系统以让其了解网络拓扑状况就极为关键。如果集群只包含一个机架，就无需做什么，因为这是默认配置。但是对于多机架的集群来说，描述清楚节点-机架的映射关系就很有必要。这使得 Hadoop

将 MapReduce 任务分配到各个节点时，会倾向于执行机架内的数据传输(拥有更多带宽)，而非跨机架数据传输。HDFS 还能够更加智能地放置复本(replica)，以取得性能和弹性的平衡。

图 10-1. Hadoop 集群的典型二级网络架构

诸如节点和机架等的网络位置以树的形式来表示，从而能够体现出各个位置之间的网络“距离”。namenode 使用网络位置来确定在哪里放置块的复本(参见 3.6.1 节)；MapReduce 的调度器根据网络位置来查找最近的复本，将它作为 map 任务的输入。

在图 10-1 所示的网络中，机架拓扑由两个网络位置来描述，即*/switch1/rack1* 和*/switch1/rack2*。由于该集群只有一个顶层路由器，这两个位置可以简写为*/rack1*和*/rack2*。

Hadoop 配置需要通过一个 Java 接口 `DNSToSwitchMapping` 来指定节点地址和网络位置之间的映射关系。该接口定义如下：

```
public interface DNSToSwitchMapping {
  public List<String> resolve(List<String> names);
}
```

`resolve()`函数的输入参数 `names` 描述 IP 地址列表，返回相应的网络位置字符串列表。`net.topology.node.switch.mapping.impl` 配置属性实现了 `DNSToSwitchMapping` 接口，namenode 和资源管理器均采用它来解析工作节点的网络位置。

在上例的网络拓扑中，可将 node1、node2 和 node3 映射到/rack1，将 node4、node5 和 node6 映射到/rack2 中。

但是，大多数安装并不需要自己实现接口，只需使用默认的 `ScriptBasedMapping` 实现即可，它运行用户定义的脚本来描述映射关系。脚本的存放路径由属性 `net.topology.script.file.name` 控制。脚本接受一系列输入参数，描述带映射的主机名称或 IP 地址，再将相应的网络位置以空格分开，输出到标准输出。用户可以参考 Hadoop 英文维基页面的一个例子，网址为 *http://wiki.apache.org/hadoop/topology_rack_awareness_scripts*。

如果没有指定脚本位置，默认情况下会将所有节点映射到单个网络位置，即 */default-rack*。

10.2 集群的构建和安装

本节介绍如何在 Unix 操作系统下使用 Apache Hadoop 分发包安装和配置一个基础的 Hadoop 集群。同时也介绍一些在安装 Hadoop 过程中需要仔细思考的背景知识。对于产品安装，大部分用户和操作者应该考虑使用本章开始部分列举的 Hadoop 集群管理工具。

10.2.1 安装 Java

Hadoop 在 Unix 和 Windows 操作系统上都可以运行，但都需要安装 Java。对于产品安装，应该选择一个经过 Hadoop 产品供应商认证的，操作系统、Java 和 Hadoop 的组合。Hadoop 英文维基页面上(*http://wiki.apache.org/hadoop/HadoopJavaVersions*)列出了能够成功运行的组合。

10.2.2 创建 Unix 用户账号

最好创建特定的 Unix 用户帐号以区分各 Hadoop 进程，及区分同一机器上的其他服务。HDFS， MapReduce 和 YARN 服务通常作为独立的用户运行，分别命名为 `hdfs`，`mapred` 和 `yarn`。它们都属于同一 hadoop 组。

10.2.3 安装 Hadoop

从 Apache Hadoop 的发布页面(*http://hadoop.apache.org/core/releases.html*)下载

Hadoop 发布包，并在某一本地目录解压缩，例如 */usr/local* (*/opt* 是另一标准选项)。注意，鉴于 hadoop 用户的 home 目录可能挂载在 NFS 上，Hadoop 系统最好不要安装在该目录上：

```
% cd /usr/local
% sudo tar xzf hadoop-x.y.z.tar.gz
```

此外，还需将 Hadoop 文件的拥有者改为 hadoop 用户和组：

```
% sudo chown -R hadoop:hadoop hadoop-x.y.z
```

在 shell 路径里添加 Hadoop 执行程序所在目录很方便：

```
% export HADOOP_HOME=/usr/local/hadoop-x.y.z
% export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
```

10.2.4 SSH 配置

Hadoop 控制脚本(并非守护进程)依赖 SSH 来执行针对整个集群的操作。例如，某个脚本能够终止并重启集群中的所有守护进程。值得注意的是，控制脚本并非唯一途径，用户也可以利用其他方法执行集群范围的操作，例如，分布式 shell 或专门的 Hadoop 管理应用)。

为了支持无缝式工作，SSH 安装好之后，需要允许来自集群内机器的 `hdfs` 用户和 `yarn` 用户能够无需密码即可登陆。[1] 最简单的方法是创建一个公钥/私钥对，存放在 NFS 之中，让整个集群共享该密钥对。

首先，键入以下指令来产生一个 RSA 密钥对。你需要做两次，一次以 `hdfs` 用户身份，一次以 `yarn` 用户身份：

```
% ssh-keygen -t rsa -f ~/.ssh/id_rsa
```

尽管期望无密码登录，但无口令的密钥并不是一个好的选择(运行在本地伪分布集群上时，倒也不妨使用一个空口令，参见附录 A)。因此，当系统提示输入口令时，用户最好指定一个口令。可以使用 *ssh-agent* 以免为每个连接逐一输入密码。

私钥放在由 `-f` 选项指定的文件之中，例如*~/.ssh/id_rsa*。存放公钥的文件名称与私

① mapred 用户不使用 SSH，如同 Hadoop 2 和以后版本中的一样，唯一的 MapReduce 守护进程是作业历史服务器。

钥类似，但是以“*.pub*”作为后缀，例如*~/.ssh/id_rsa.pub*。

接下来，需确保公钥存放在用户打算连接的所有机器的*~/.ssh/authorized_keys* 文件中。如果用户的 home 目录是存储在 NFS 文件系统中，则可以键入以下指令在整个集群内共享密钥(第一次作为 hdfs 用户，第二次作为 yarn 用户)：

```
% cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

如果 home 目录并没有通过 NFS 共享，则需要利用其他方法共享公钥(比如 *ssh-copy-id*)。

测试是否可以从主机器 SSH 到工作机器。若可以，则表明 *ssh-agent* 正在运行[①]，再运行 *ssh-add* 来存储口令。这样的话，用户即可不用再输入口令就能 SSH 到一台工作机器。

10.2.5 配置 Hadoop

如果希望 Hadoop 以分布式模式在集群上运行，必须正确对其进行配置。10.3 节中将详细讨论为达到此目的所需的重要配置。

10.2.6 格式化 HDFS 文件系统

在能够使用之前，全新的 HDFS 安装需要进行格式化。通过创建存储目录和初始版本的 namenode 持久数据结构，格式化进程将创建一个空的文件系统。由于 namenode 管理所有的文件系统元数据，datanode 可以动态加入或离开集群，所以初始的格式化进程不涉及到 datanode。同样原因，创建文件系统时也无需指定大小，这是由集群中的 datanode 数目决定的，在文件系统格式化之后的很长时间内都可以根据需要增加。

格式化 HDFS 是一个快速操作。以 hdfs 用户身份运行以下命令：

```
% hdfs namenode -format
```

10.2.7 启动和停止守护进程

Hadoop 自带脚本，可以运行命令并在整个集群范围内启动和停止守护进程。为使用这些脚本(在 *sbin* 目录下)，需要告诉 Hadoop 集群中有哪些机器。文件 *slaves* 正

① 可以参阅 *ssh-agent* 指令的操作手册，了解如何启动 *ssh-agent*。

是用于此目的，该文件包含了机器主机名或 IP 地址的列表，每行代表一个机器信息。文件 *slaves* 列举了可以运行 datanode 和节点管理器的机器。文件驻留在 Hadoop 配置目录下，尽管通过修改 *hadoop-env.sh* 中的 HADOOP_SLAVES 设置可能会将文件放在别的地方(并赋予一个别的名称)。并且，不需要将该文件分发给工作节点，因为仅有运行在 namenode 和资源管理器上的控制脚本使用它。

以 hdfs 用户身份运行以下命令可以启动 HDFS 守护进程：

```
% start-dfs.sh
```

namenode 和辅助 namenode 运行所在的机器通过向 Hadoop 配置询问机器主机名来决定。例如，通过执行以下命令，脚本能够找到 namenode 的主机名。

```
% hdfs getconf -namenodes
```

默认情况下，该命令从 fs.defaultFS 中找到 namenode 的主机名。更具体一些，*start-dfs.sh* 脚本所做的事情如下。

1. 在每台机器上启动一个 namenode，这些机器由执行 hdfs getconf -namenodes 得到的返回值所确定。[①]
2. 在 *slaves* 文件列举的每台机器上启动一个 datanode。
3. 在每台机器上启动一个辅助 namenode，这些机器由执行 hdfs get conf -secondarynamenodes 得到的返回值所确定。

YARN 守护进程以相同的方式启动，通过以 yarn 用户身份在托管资源管理器的机器上运行以下命令：

```
% start-yarn.sh
```

在这种情况下，资源管理器总是和 *start- yarn.sh* 脚本运行在同一机器上。脚本明确完成以下事情。

1. 在本地机器上启动一个资源管理器。
2. 在 *slaves* 文件列举的每台机器上启动一个节点管理器。

同样，还提供了 *stop-dfs.sh* 和 *stop-yarn.sh* 脚本用于停止由相应的启动脚本启动的守护进程。

① 运行 HDFS HA 时，可能会有多个 namenode。

这些脚本实质是使用了 *hadoop-daemon.sh* 脚本(YARN 中是 *yarn-daemon.sh* 脚本)启动和停止 Hadoop 守护进程。如果你使用了前面提到的脚本，那么你不能直接调用 *hadoop-daemon.sh*。但是如果你需要从另一个系统或从你自己的脚本来控制 Hadoop 守护进程， *hadoop-daemon.sh* 脚本是一个很好的切入点。类似的，当需要一个主机集上启动相同的守护进程时，使用 *hadoop-daemons.sh*(带有“s”)会很方便。

最后，仅有一个 MapReduce 守护进程，即作业历史服务器，是以 `mapred` 用户身份用以下命令启动的：

```
% mr-jobhistory-daemon.sh start historyserver
```

10.2.8 创建用户目录

一旦建立并运行了 Hadoop 集群，就需要给用户提供访问手段。涉及到为每个用户创建 home 目录，给目录设置用户访问许可：

```
% hadoop fs -mkdir /user/username
% hadoop fs -chown username:username /user/username
```

此时是给目录设置空间限制的好时机。以下命令为给定的用户目录设置了 1TB 的容量限制：

```
% hdfs dfsadmin -setSpaceQuota 1t /user/username
```

10.3 Hadoop 配置

有多个配置文件适用于 Hadoop 安装，表 10-1 列举了最重要的几个文件。

这几个重要文件都放在 Hadoop 分发包的 *etc/hadoop* 目录中。配置目录可以被重新安置在文件系统的其他地方(Hadoop 安装路径的外面，以便于升级)，只要启动守护进程时使用`--config` 选项(或等价的，使用 `HADOOP_CONF_DIR` 环境变量集)说明这个目录在本地文件系统的位置就可以了。

表 10-1. Hadoop 配置文件

文件名称	格式	描述
hadoop-env.sh	Bash 脚本	脚本中要用到的环境变量，以运行 Hadoop
mapred-env.sh	Bash 脚本	脚本中要用到的环境变量，以运行 MapReduce(覆盖 *hadoop-env.sh* 中设置的变量)

续表

文件名称	格式	描述
yarn-env.sh	Bash 脚本	脚本中要用到的环境变量，以运行 YARN(覆盖 *hadoop-env.sh* 中设置的变量)
core-site.xml	Hadoop 配置 XML	Hadoop Core 的配置项，例如 HDFS、MapReduce 和 YARN 常用的 I/O 设置等
hdfs-site.xml	Hadoop 配置 XML	Hadoop 守护进程的配置项，包括 namenode、辅助 namenode 和 datanode 等
mapred-site.xml	Hadoop 配置 XML	MapReduce 守护进程的配置项，包括作业历史服务器
yarn-site.xml	Hadoop 配置 XML	YARN 守护进程的配置项，包括资源管理器、web 应用代理服务器和节点管理器
slaves	纯文本	运行 datanode 和节点管理器的机器列表(每行一个)
hadoop-metrics2.properties	Java 属性	控制如何在 Hadoop 上发布度量的属性(参 11.2.2 节)
log4j.properties	Java 属性	系统日志文件、namenode 审计日志、任务 JVM 进程的任务日志的属性(参见 6.5.6 节)
hadoop-policy.xml	Hadoop 配置 XML	安全模式下运行 Hadoop 时的访问控制列表的配置项

10.3.1 配置管理

Hadoop 并没有将所有配置信息放在一个单独的全局位置中。反之，集群的每个 Hadoop 节点都各自保存一系列配置文件，并由管理员完成这些配置文件的同步工作。有并行 shell 工具帮助完成同步工作，诸如 *dsh* 或 *pdsh*。在这方面，Hadoop 集群管理工具例如 Cloudera Manager 和 Apache Ambari 表现突出，因为在集群间传递修改信息是它们的关注点。

Hadoop 也支持为所有 master 机器和 worker 机器采用同一套配置文件。这个做法的最大优势在于简单，不仅体现在理论上(仅需处理一套配置文件)，也体现在可操作性上(使用 Hadoop 脚本就能进行管理)。

但是，这种一体适用的配置模型并不合适某些集群。以扩展集群为例，当试图为集群添加新机器，且新机器的硬件规格与现有机器不同时，则需要新建一套配置文件，以充分利用新硬件的额外资源。

在这种情况下，需要引入“机器类”的概念，为每一机器类维护单独的配置文件。Hadoop 没有提供执行这个操作的工具，需要借助外部工具来执行该配置操作，例如 Chef、Puppet、CFEngine 和 Bcfg2 等。

对于任何规模的集群来说，同步所有机器上的配置文件都极具挑战性。例如，假

设某台机器正好处于异常状态，而此时用户正好发出一条更新配置的指令，如何保证这台机器在恢复正常状态之后也能够更新配置？这个问题很严重，可能会导致集群中各机器的配置不一致。因此，尽管用户能够使用控制脚本来管理 Hadoop，仍然推荐使用控制管理工具管理集群。使用这些工具也可以顺利完成日常维护，例如为安全漏洞打补丁、升级系统包等。

10.3.2 环境设置

本节探讨如何设置 *hadoop-env.sh* 文件中的变量。MapReduce 和 YARN(HDFS 除外)都有类似的配置文件，分别为 *mapred-env.sh* 和 *yarn-env.sh*，文件中的变量和组件相关，并且可以进行设置。注意，*hadoop-env.sh* 文件里设置的值会被 MapReduce 和 YARN 文件覆盖。

1. Java

需要设置 Hadoop 系统的 Java 安装的位置。方法一是在 *hadoop-env.sh* 文件中设置 `JAVA_HOME` 项；方法二是在 shell 中设置 `JAVA_HOME` 环境变量。相比之下，方法一更好，因为只需操作一次就能够保证整个集群使用同一版本的 Java。

2. 内存堆大小

在默认情况下，Hadoop 为各个守护进程分配 1000 MB(1GB)内存。该内存值由 *hadoop-env.sh* 文件的 `HADOOP_HEAPSIZE` 参数控制。也可以通过设置环境变量为单个守护进程修改堆大小。例如，在 *yarn-env.sh* 文件中设置 `YARN_RESOURCEMANAGER_HEAPSIZE`，即可覆盖资源管理器的堆大小。

令人惊讶的是，尽管为 namenode 分配更多的堆空间是很常见的事，但对于 HDFS 守护进程而言并没有相应的环境变量。当然有别的途径可以设置 namenode 堆空间大小，见接下来的讨论。

除了守护进程对内存的需求，节点管理器还需为应用程序分配容器(container)，因此需要综合考虑上述因素来计算一个工作机器的总体内存需求，详见 10.3.3 节中 YARN 和 MapReduce 内存设置的有关内容。

一个守护进程究竟需要多少内存？

由于 namenode 会在内存中维护所有文件的每个数据块的引用，因此 namenode 很可能会“吃光”分配给它的所有内存。很难套用一个公式来精确计算内存需求量，因为内存需求量取决于多个因素，包括每个文件包含的数据块数、文件名称的长度、文件系统中的目录数等。此外，在不同 Hadoop 版本下，namenode 的内存需求也不相同。

1000 MB 内存(默认配置)通常足够管理数百万个文件。但是根据经验来看，保守估计需要为每 1 百万个数据块分配 1000 MB 内存空间。

以一个含 200 节点的集群为例，假设每个节点有 24 TB 磁盘空间，数据块大小是 128 MB，复本数是 3 的话，则约有 2 百万个数据块(甚至更多)：200×24000000 MB/(128 MB×3)。因此，在本例中，namenode 的内存空间最好一开始设为 12000 MB。

也可以只增加 namenode 的内存分配量而不改变其他 Hadoop 守护进程的内存分配，即设置 *hadoop-env.sh* 文件的 `HADOOP_NAMENODE_OPTS` 属性包含一个 JVM 选项以设定内存大小。`HADOOP_NAMENODE_OPTS` 允许向 namenode 的 JVM 传递额外的选项。以 Sun JVM 为例，`-Xmx2000m` 选项表示为 namenode 分配 2000 MB 内存空间。

由于辅助 namenode 的内存需求量和主 namenode 差不多，所以一旦更改 namenode 的内存分配的话还需对辅助 namenode 做相同更改(使用 `HADOOP_SECONDARYNAMENODE_OPTS` 变量)。

3. 系统日志文件

默认情况下，Hadoop 生成的系统日志文件存放在*$HADOOP_HOME/logs* 目录之中，也可以通过 *hadoop-env.sh* 文件中的 HADOOP_LOG_DIR 来进行修改。建议修改默认设置，使之独立于 Hadoop 的安装目录。这样的话，即使 Hadoop 升级之后安装路径发生变化，也不会影响日志文件的位置。通常可以将日志文件存放在 */var/log/hadoop* 目录中。实现方法很简单，就是在 *hadoop-env.sh* 中加入一行：

```
export HADOOP_LOG_DIR=/var/log/hadoop
```

如果日志目录并不存在，则会首先创建该目录(如果操作失败，请确认相关的 Unix

Hadoop 用户是否有权创建该目录)。运行在各台机器上的各个 Hadoop 守护进程会产生两类日志文件。第一类日志文件(以.log 作为后缀名)是通过 log4j 记录的。鉴于大部分应用程序的日志消息都写到该日志文件中，故障诊断的首要步骤即为检查该文件。标准的 Hadoop log4j 配置采用日常滚动文件追加方式(daily rolling file appender)来循环管理日志文件。系统不自动删除过期的日志文件，而是留待用户定期删除或存档，以节约本地磁盘空间。

第二类日志文件后缀名为*.out*，记录标准输出和标准错误日志。由于 Hadoop 使用 log4j 记录日志，所以该文件通常只包含少量记录，甚至为空。重启守护进程时，系统会创建一个新文件来记录此类日志。系统仅保留最新的 5 个日志文件。旧的日志文件会附加一个介于 1 和 5 之间的数字后缀，5 表示最旧的文件。

日志文件的名称(两种类型)包含运行守护进程的用户名称、守护进程名称和本地主机名等信息。例如，*hadoop-hdfsdatanode-ip-10-45-174-112.log.2014-09-20* 就是一个日志文件的名称。这种命名方法保证集群内所有机器的日志文件名称各不相同，从而可以将所有日志文件存到一个目录中。

日志文件名称中的“用户名称”部分实际对应 *hadoop-env.sh* 文件中的 `HADOOP_IDENT_STRING` 项。如果想采用其他名称，可以修改 `HADOOP_IDENT_STRING` 项。

4. SSH 设置

借助 SSH 协议，用户在主节点上使用控制脚本就能在(远程)工作节点上运行一系列指令。自定义 SSH 设置会带来诸多益处。例如，减小连接超时设定(使用 `ConnectTimeout` 选项)可以避免控制脚本长时间等待宕机节点的响应。当然，也不可设得过低，否则会导致繁忙节点被跳过。

`StrictHostKeyChecking` 也是一个很有用的 SSH 设置。设置为 `no` 会自动将新主机键加到已知主机文件之中。该项默认值是 `ask`，提示用户确认是否已验证了“键指纹”(key fingerprint)，因此不适合大型集群环境[①]。

在 *hadoop-env.sh* 文件中定义 `HADOOP_SSH_OPTS` 环境变量还能够向 SSH 传递更多选项。参考 `ssh` 和 `ssh_config` 使用手册，了解更多 SSH 设置。

① 如果想进一步了解 SSH 主机密钥的安全性内容，请参阅 Brian Hatch 的文章，标题为“SSH Host Key Protection”，网址为 *http://www.securityfocus.com/infocus/1806*。

10.3.3 Hadoop 守护进程的关键属性

Hadoop 的配置属性之多简直让人眼花缭乱。本节讨论对于真实的工作集群来说非常关键的一些属性(或至少能够理解默认属性的含义)，这些属性分散在 Hadoop 的站点文件之中，包括 *core-site.xml*、*hdfs-site.xml* 和 *yarn-site.xml*。范例 10-1、10-2 和 10-3 分别列举了这些文件的典型实例[①]。要想进一步了解 Hadoop 配置文件的格式，可参见 6.1 节。

对于一个正在运行的守护进程，要想知道其实际配置，可以访问该进程 web 服务器上的*/conf* 页面。例如，*http://resource-manager-host:8088/conf* 表示资源管理器当前的运行配置。该页面展示了守护进程正在运行的组合站点和默认配置文件，并给出了每个属性的来源文件。

范例 10-1. 典型的 core-site.xml 配置文件

```
<?xml version="1.0"?>
<!-- core-site.xml -->
<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://namenode/</value>
  </property>
</configuration>
```

范例 10-2. 典型的 hdfs-site.xml 配置文件

```
<?xml version="1.0"?>
<!-- hdfs-site.xml -->
<configuration>
  <property>
    <name>dfs.namenode.name.dir</name>
    <value>/disk1/hdfs/name,/remote/hdfs/name</value>
  </property>

  <property>
    <name>dfs.datanode.data.dir</name>
    <value>/disk1/hdfs/data,/disk2/hdfs/data</value>
  </property>

  </property>
    <name>dfs.namenode.checkpoint.dir</name>
    <value>/disk1/hdfs/namesecondary,/disk2/hdfs/namesecondary</value>
  </property>
```

① 注意，这里没有列出 MapReduce 的站点文件。这是因为仅有的 MapReduce 守护进程就是作业历史服务器，对于它而言，默认属性值就够用了。

```
</configuration>
```

范例 10-3. 典型的 yarn-site.xml 配置文件

```
<?xml version="1.0"?>
<!-- yarn-site.xml -->
<configuration>
  <property>
    <name>yarn.resourcemanager.hostname</name>
    <value>resourcemanager</value>
  </property>

  <property>
    <name>yarn.nodemanager.local-dirs</name>
    <value>/disk1/nm-local-dir,/disk2/nm-local-dir</value>
  </property>

  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce.shuffle</value>
  </property>

  <property>
    <name>yarn.nodemanager.resource.memory-mb</name>
    <value>16384</value>
  </property>

  <property>
    <name>yarn.nodemanager.resource.cpu-vcores</name>
    <value>16</value>
    </property>
  </configuration>
```

1. HDFS

运行 HDFS 需要将一台机器指定为 namenode。在本例中，属性 `fs.defaultFS` 描述 HDFS 文件系统的 URI，其主机是 namenode 的主机名称或 IP 地址，端口是 namenode 监听 RPC 的端口。如果没有指定，那么默认端口是 8020。

属性 `fs.defaultFS` 也指定了默认文件系统，可以解析相对路径。相对路径的长度更短，使用更便捷(不需要了解特定 namenode 的地址)。例如，假设默认文件系统如范例 10-1 所示的那样，则相对 URI */a/b* 解析为 *hdfs://namenode/a/b*。

当用户在运行 HDFS 时，鉴于 `fs.defaultFS` 指定了 HDFS 的 namenode 和默认文件系统，则 HDFS 必须是服务器配置的默认文件系统。值得注意的是，为了操作方便，也允许在客户端配置中将其他文件系统指定为默认文件系统。

例如，假设系统使用 HDFS 和 S3 两种文件系统，则可以在客户端配置中将任一文件系统指定为默认文件系统。这样的话，就能用相对 URI 指向默认文件系统，用绝对 URI 指向其他文件系统。

此外，还有其他一些关于 HDFS 的配置选项，包括 namenode 和 datanode 存储目录的属性。属性项 `dfs.namenode.name.dir` 指定一系列目录来供 namenode 存储永久性的文件系统元数据(编辑日志和文件系统映像)。这些元数据文件会同时备份在所有指定目录中。通常情况下，通过配置 `dfs.namenode.name.dir` 属性可以将 namenode 元数据写到一两个本地磁盘和一个远程磁盘(例如 NFS 挂载的目录)之中。这样的话，即使本地磁盘发生故障，甚至整个 namenode 发生故障，都可以恢复元数据文件并且重构新的 namenode。(辅助 namenode 只是定期保存 namenode 的检查点，不维护 namenode 的最新备份。)

属性 `dfs.datanode.data.dir` 可以设定 datanode 存储数据块的目录列表。前面提到，`dfs.namenode.name.dir` 描述一系列目录，其目的是支持 namenode 进行冗余备份。虽然 `dfs.datanode.data.dir` 也描述了一系列目录，但是其目的是使 datanode 循环地在各个目录中写数据。因此，为了提高性能，最好分别为各个本地磁盘指定一个存储目录。这样一来，数据块跨磁盘分布，针对不同数据块的读操作可以并发执行，从而提升读取性能。

为了充分发挥性能，需要使用 noatime 选项挂载磁盘。该选项意味着执行读操作时，所读文件的最近访问时间信息并不刷新，从而显著提升性能。

最后，还需要指定辅助 namenode 存储文件系统的检查点的目录。属性 `dfs.namenode.checkpoint.dir` 指定一系列目录来保存检查点。与 namenode 类似，检查点映像文件会分别存储在各个目录之中，以支持冗余备份。

表 10-2 总结了 HDFS 的关键配置属性。

表 10-2. HDFS 守护进程的关键属性

属性名称	类型	默认值	说明
`fs.defaultFS`	URI	`file:///`	默认文件系统。URI 定义主机名称和 namenode 的 RPC 服务器工作的端口号，默认值是 8020。本属性保存在 *core-site.xml* 中
`dfs.namenode.name.dir`	以逗号分隔的目录名称	`file://${hadoop.tmp.dir}/dfs/name`	namenode 存储永久性的元数据的目录列表。namenode 在列表上的各个目录中均存放相同的元数据文件

续表

属性名称	类型	默认值	说明
dfs.datanode.data.dir	以逗号分隔的目录名称	file://${hadoop.tmp.dir}/dfs/data	datanode 存放数据块的目录列表。各个数据块分别存放于某一个目录中
dfs.namenode.checkpoint.dir	以逗号分隔的目录名称	file://${hadoop.tmp.dir}/dfs/namesecondary	辅助 namenode 存放检查点的目录列表。在所列每个目录中均存放一份检查点文件的副本

在默认情况下，HDFS 的存储目录放在 Hadoop 的临时目录下(通过 hadoop.tmp.dir 属性配置，默认值是/tmp/hadoop-${user.name})。因此，正确设置这些属性的重要性在于，即使清除了系统的临时目录。数据也不会丢失。

2. YARN

为了运行 YARN，需要指定一台机器作为资源管理器。最简单的做法是将属性 yarn.resourcemanager.hostname 设置为用于运行资源管理器的机器的主机名或 IP 地址。资源管理器服务器的地址基本都可以从该属性获得。例如，yarn.resourcemanager.address 的格式为主机-端口对，yarn.resourcemanager.hostname 表示默认主机。在 MapReduce 客户端配置中，需要通过 RPC 连接到资源管理器时，会用到这个属性。

在执行 MapReduce 作业的过程中所产生的中间数据和工作文件被写到临时本地文件之中。由于这些数据包括 map 任务的输出数据，数据量可能非常大，因此必须保证 YARN 容器本地临时存储空间(由 yarn.nodemanager.local-dirs 属性设置)的容量足够大。yarn.nodemanager.local-dirs 属性使用一个逗号分隔的目录名称列表，最好将这些目录分散到所有本地磁盘，以提升磁盘 I/O 操作的效率。通常情况下，YARN 本地存储会使用与 datanode 数据块存储相同的磁盘和分区(但是不同的目录)。如前所述，datanode 数据块存储目录由 dfs.datanode.data.dir 属性项指定。

与 MapReduce 1 不同，YARN 没有 tasktracker，它依赖于 shuffle 句柄将 map 任务的输出送给 reduce 任务。Shuffle 句柄是长期运行于节点管理器的附加服务。由于 YARN 是一个通用目的的服务，因此要通过将 *yarn-site.xml* 文件中的 yarn.nodemanager.aux-services 属性设置为 mapreduce_shuffle 来显式启用 MapReduce 的 shuffle 句柄。

表 10-3 总结了 YARN 的关键配置属性，资源相关设置在下一章中有更详细的介绍。

表 10-3. YARN 守护进程的关键属性

属性名称	类型	默认值	说明
yarn.resourcemanager.hostname	主机名	0.0.0.0	运行资源管理器的机器主机名。以下缩写为${y.rm.hostname}
yarn.resourcemanager.address	主机名和端口号	${y.rm.hostname}:8032	运行资源管理器的 RPC 服务器的主机名和端口
yarn.nodemanager.local-dirs	逗号分隔的目录名称	${hadoop.tmp.dir}/nm-local-dir	目录列表，节点管理器允许容器将中间数据存于其中。当应用结束时，数据被清除
yarn.nodemanager.aux-services	逗号分隔的服务名称		节点管理器运行的附加服务列表。每项服务由属性 yarn.nodemanager.auxservices.servicename.class 所定义的类实现。默认情况下，不指定附加服务
yarn.nodemanager.resource.memorymb	int	8192	节点管理器运行的容器可以分配到的物理内存容量(单位是 MB)
yarn.nodemanager.vmem-pmem-ratio	float	2.1	容器所占的虚拟内存和物理内存之比。该值指示了虚拟内存的使用可以超过所分配内存的量
yarn.nodemanager.resource.cpuvcores	int	8	节点管理器运行的容器可以分配到的 CPU 核数目

3. YARN 和 MapReduce 中的内存设置

与 MapReduce 1 的基于 slot 的模型相比，YARN 以更精细化的方式来管理内存。YARN 不会立刻指定一个节点上可以运行的 map 和 reduce slot 的最大数目，相反，它允许应用程序为一个任务请求任意规模的内存(在限制范围内)。在 YARN 模型中，节点管理器从一个内存池中分配内存，因此，在一个特定节点上运行的任务数量取决于这些任务对内存的总需求量，而不简单取决于固定的 slot 数量。

计算为一个运行容器的节点管理器分配多少内存要取决于机器上的物理内存。每个 Hadoop 守护进程使用 1000 MB 内存，因此需要 2000 MB 内存来运行 1 个 datanode 和 1 个节点管理器。为机器上运行的其他进程留出足够的内存后，通过

将配置属性 yarn.nodemanager.resource.memory-mb 设置为总分配量(单位是MB)，剩余的内存就可以被指定给节点管理器的容器使用了。默认是 8192 MB，对于大多数设置来说太低了。

接下来是确定如何为单个作业设置内存选项。有两种主要控制方法：一个是控制YARN 分配的容器大小，另一个是控制容器中运行的 Java 进程堆大小。

MapReduce 的内存控制都由客户端在作业配置中设置。YARN 设置是集群层面的设置，客户端不能修改。

容器大小由属性 mapreduce.map.memory.mb 和 mapreduce.reduce.memory.mb 决定，默认值都为 1024 MB。application master 会使用这些设置以从集群中请求资源；此外，节点管理器也会使用这些设置来运行、监控任务容器。Java 进程的堆大小由 mapred.child.java.opts 设置，默认是 200 MB。也可以单独为 map 和 reduce 任务设置 Java 选项(参见表 10-4)。

表 10-4. MapReduce 作业内存属性(由客户端设置)

属性名称	类型	默认值	说明
mapreduce.map.memory.mb	int	1024	map 容器所用的内存容量
mapreduce.reduce.memory.mb	int	1024	reduce 容器所用的内存容量
mapred.child.java.opts	String	-Xmx200m	JVM 选项，用于启动运行 map 和 reduce 任务的容器进程。除了用于设置内存，该属性还包括 JVM 属性设置，以支持调试
mapreduce.map.java.opts	String	-Xmx200m	JVM 选项，针对运行 map 任务的子进程
mapreduce.reduce.java.opts	string	-Xmx200m	JVM 选项，针对运行 reduce 任务的子进程

例如，假设 mapred.child.java.opts 被设为-Xmx800m，mapreduce.map.memory.mb 被设为默认值 1024 MB，当 map 任务启动时，节点管理器会为该任务分配 1 个 1024 MB 的容器(在该任务运行期间，节点管理器的内存池也会相应降低1024MB)，并启动配置为最大堆为 800MB 的任务 JVM。注意，JVM 进程的内存开销将比该堆的规模要大，开销依赖于所使用的本地库(native libraries)、永久生成空间(permanent generation space)等因素。需要注意的是，JVM 进程(包括它创建的任何进程，如 Streaming)所使用的物理内存必须不超出分配给它的内存大小(1024

MB)。如果一个容器使用的内存超过所分配的量，就会被节点管理器终止，并标记为失败。

YARN 调度器会指定一个最小和最大内存分配量。默认情况下，最小内存分配量是 1024 MB(由 yarn.scheduler.minimum-allocation-mb 设置)，默认情况下，最大内存分配量是 8192 MB(由 yarn.scheduler. maximum-allocation-mb 设置)。

容器还需要满足对虚拟内存的限制。如果容器所使用的虚拟内存量超出预定系数和所分配的物理内存的乘积，则节点管理器也会终止进程。该系数由 yarn.nodemanager.vmem-pmem-ratio 属性指定，默认值是 2.1。在前面的例子中，虚拟内存规模的上限值为2150 MB，即 2.1 × 1024 MB。

除了使用参数来配置内存使用之外，还可以使用 MapReduce 任务计数器来监控任务执行过程中的真实内存消费量。这些计数器包括：PHYSICAL_EMORY_BYTES、VIRTUAL_MEMORY_BYTES 和 COMMITTED_HEAP_BYTES(参见表 9-2)，分别描述了在某一时刻各种内存的使用情况，因此也适用于在任务尝试期间的观察。

Hadoop 也提供了一些设置方法，用于控制 MapReduce 操作的内存使用。这些设置可以针对每个作业进行，详情参见 7.3 节。

4. YARN 和 MapReduce 中的 CPU 设置

除了内存外，YARN 将 CPU 的使用作为一种资源进行管理，应用程序可以申请所需要的核数量。通过属性 yarn.nodemanager.resource.cpuvcores 可以设置节点管理器分配给容器的核数量。应该设置为机器的总核数减去机器上运行的每个守护进程(datanode、节点管理器和其他长期运行的进程)占用的核数(每个进程占用 1 个核)。

通过设置属性 mapreduce.map.cpu.vcores 和 mapreduce.reduce.cpu.vcores，MapReduce 作业能够控制分配给 map 和 reduce 容器的核数量。两者的默认值均为 1，适合通常的单线程 MapReduce 任务，因为这些任务使用单核就足够了。

当调度过程中对核数量进行掌控后(这样，当机器没有空余核时，一个容器将不会分到核)，节点管理器默认情况下将不会限制运行中的容器对 CPU 的实际使用。这意味着一个容器可能会出现滥用配额的情况，例如使用超额的 CPU，而这可能会饿死在同一主机上运行的其他容器。YARN 提供了基于 Linux 的 cgroup 技术的、强制实施 CPU 限制的手段。为此，节点管理器的容

器执行类(yarn.nodemanager.containerexecutor.class)必须被设置为使用 LinuxContainerExecutor 类，并且必须将 LinuxContainerExecutor 类配置为使用 cgroup，详情查阅 under yarn.nodemanager.linux-container-executor 的属性介绍。

10.3.4 Hadoop 守护进程的地址和端口

Hadoop 守护进程一般同时运行 RPC 和 HTTP 两个服务器，RPC 服务器(表 10-5)支持守护进程间的通信，HTTP 服务器则提供与用户交互的 Web 页面(表 10-6)。需要分别为各个服务器配置网络地址和监听端口号。端口号 0 表示服务器会选择一个空闲的端口号：但由于这种做法与集群范围的防火墙策略不兼容，所以我通常不推荐。

表 10-5. RPC 服务器的属性

属性名称	默认值	说明
fs.defaultFS	file:///	设为一个 HDFS 的 URI 时，该属性描述 namenode 的 RPC 服务器地址和端口。如果不指定，则默认的端口号是 8020
dfs.namenode.rpc-bind-host		namenode 的 RPC 服务器将绑定的地址。如果没有设置(默认情况)，绑定地址由 fs.defaultFS 决定。可以设为 0.0.0.0，使得 namenode 可以监听所有接口
dfs.datanode.ipc.address	0.0.0.0:50020	datanode 的 RPC 服务器地址和端口
apreduce.jobhistory.address	0.0.0.0:10020	作业历史服务器的 RPC 服务器地址和端口，客户端(一般在集群外部)用于查询作业历史
mapreduce.jobhistory.bind-host		作业历史服务器的 RPC 和 HTTP 服务器将绑定的地址
yarn.resourcemanager.hostname	0.0.0.0	资源管理器运行所在的机器主机名。以下缩写为${y.rm.hostname}
yarn.resourcemanager.bind-host		资源管理器的 RPC 和 HTTP 服务器将绑定的地址
yarn.resourcemanager.address	${y.rm.hostname}:8032	资源管理器的 RPC 服务器地址和端口。客户端(一般在集群外部)通过它与资源管理器通信

续表

属性名称	默认值	说明
yarn.resourcemanager.admin.address	${y.rm.hostname}:8033	资源管理器的 admin RPC 服务器地址和端口。admin 客户端(由 yarn rmadmin 调用，一般在集群外部)借此与资源管理器通信
yarn.resourcemanager.scheduler.address	${y.rm.hostname}:8030	资源管理器的调度器 RPC 服务器地址和端口。application master(在集群内部)借此与资源管理器通信
yarn.resourcemanager.resourcetracker.address	${y.rm.hostname}:8031	资源管理器的 resource tracker 的 RPC 服务器地址和端口。节点管理器(在集群内)借此与资源管理器通信
yarn.nodemanager.hostname	0.0.0.0	节点管理器运行所在的机器的主机名。以下缩写为${y.nm.hostname}
yarn.nodemanager.bind-host		节点管理器的 RPC 和 HTTP 服务器将绑定的地址
yarn.nodemanager.address	${y.nm.hostname}:0	节点管理器的 RPC 服务器地址和端口。application master(在集群内部)借此与节点管理器通信
yarn.nodemanager.localizer.address	${y.nm.hostname}:8040	节点管理器的 localizer 的 RPC 服务器地址和端口

表 10-6. HTTP 服务器的属性

属性名称	默认值	说明
dfs.namenode.http-address	0.0.0.0:50070	namenode 的 HTTP 服务器地址和端口
dfs.namenode.http-bind-host		namenode 的 HTTP 服务器将绑定的地址
dfs.namenode.secondary.http-address	0.0.0.0:50090	辅助 namenode 的 HTTP 服务器地址和端口
dfs.datanode.http.address	0.0.0.0:50075	datanode 的 HTTP 服务器地址和端口。注意，属性名和 namenode 的属性名不一样
mapreduce.jobhistory.webapp.address	0.0.0.0:19888	MapReduce 作业历史服务器地址和端口。该属性在 *mapred-site.xml* 文件中设置

续表

属性名称	默认值	说明
mapreduce.shuffle.port	13562	Shuffle 句柄的 HTTP 端口号。为 map 输出结果服务，但不是用户可访问的 Web UI。该属性在 *mapred-site.xml* 文件中设置
yarn.resourcemanager.webapp.address	${y.rm.hostname}:8088	资源管理器的 HTTP 服务器地址和端口
yarn.nodemanager.webapp.address	${y.nm.hostname}:8042	节点管理器的 HTTP 服务器地址和端口
yarn.web-proxy.address		Web 应用代理服务器的 HTTP 服务器地址和端口。如果该属性没有设置(默认情况)，Web 应用代理服务器将在资源管理器进程中运行

通常，用于设置服务器 RPC 和 HTTP 地址的属性担负着双重责任：一方面它们决定了服务器将绑定的网络接口，另一方面，客户端或集群中的其他机器使用它们连接服务器。例如，节点管理器使用 yarn.resourcemanager. resource-tracker.address 属性来确定它们的资源管理器的地址。

用户经常希望服务器同时可以绑定多个网络接口，将网络地址设为 0.0.0.0 可以达到这个目的，但是却破坏了上述第二种情况，因为这个地址无法被客户端或集群中的其他机器解析。一种解决方案是将客户端和服务器的配置分开，但是更好的一种方案是为服务器绑定主机。通过将 yarn.resourcemanager.hostname 设定为主机名或 IP 地址，yarn.resourcemanager.bind-host 设定为 0.0.0.0，可以确保资源管理器能够与机器上的所有地址绑定，且同时能为节点管理器和客户端提供可解析的地址。

除了 RPC 服务器之外，各个 datanode 还运行 TCP/IP 服务器以支持块传输。服务器地址和端口号由属性 dfs.datanode.address 设定，默认值是 0.0.0.0:50010。

有多个网络接口时，还可以为各个 datanode 选择某一个网络接口作为 IP 地址(针对 HTTP 和 RPC 服务器)。相关属性是 dfs.datanode.dns.interface，默认值是 default，表示使用默认的网络接口。可以修改该属性项来变更网络接口的地址(例如，eth0)。

10.3.5 Hadoop 的其他属性

本节讨论其他一些可能会用到的 Hadoop 属性。

1. 集群成员

为了便于在将来添加或移除节点，可以通过文件来指定一些允许作为 datanode 或节点管理器加入集群的经过认证的机器。属性 `dfs.hosts` 记录允许作为 datanode 加入集群机器列表；属性 `yarn.resourcemanager. nodes.include-path` 记录允许作为节点管理器加入集群的机器列表。与之相对应的，属性 `dfs.hosts.exclude` 和 `yarn.resourcemanager. nodes.exclude-path` 所指定的文件分别包含待解除的机器列表。更深入的讨论可以参见 11.3.2 节。

2. 缓冲区大小

Hadoop 使用一个 4 KB(4096 字节)的缓冲区辅助 I/O 操作。对于现代硬件和操作系统来说，这个容量实在过于保守了。增大缓冲区容量会显著提高性能，例如 128 KB(131072 字节)更常用。可以通过 *core-site.xml* 文件中的 `io.file.buffer.size` 属性来设置缓冲区大小(以字节为单位)。

3. HDFS 块大小

在默认情况下，HDFS 块大小是 128 MB，但是许多集群把块大小设得更大(如 256 MB，268435456 字节)以降低 namenode 的内存压力，并向 mapper 传输更多数据。可以通过 *hdfs-site.xml* 文件中的 `dfs.blocksize` 属性设置块的大小(以字节为单位)。

4. 保留的存储空间

默认情况下，datanode 能够使用存储目录上的所有闲置空间。如果计划将部分空间留给其他应用程序(非 HDFS)，则需要设置 `dfs.datanode.du. reserved` 属性来指定待保留的空间大小(以字节为单位)。

5. 回收站

Hadoop 文件系统也有回收站设施，被删除的文件并未被真正删除，仅只转移到回收站(一个特定文件夹)中。回收站中的文件在被永久删除之前仍会至少保留一段时

间。该信息由 *core-site.xml* 文件中的 `fs.trash.interval` 属性(以分钟为单位)设置。默认情况下，该属性的值是 0，表示回收站特性无效。

与许多操作系统类似，Hadoop 的回收站设施是用户级特性，换句话说，只有由文件系统 shell 直接删除的文件才会被放到回收站中，用程序删除的文件会被直接删除。当然，也有例外的情况，如使用 Trash 类。构造一个 `Trash` 实例，调用 `moveToTrash()`方法会把指定路径的文件移到回收站中。如果操作成功，该方法返回一个值；否则，如果回收站特性未被启动，或该文件已经在回收站中，该方法返回 false。

当回收站特性被启用时，每个用户都有独立的回收站目录，即：*home* 目录下的.*Trash* 目录。恢复文件也很简易：在.*Trash* 的子目录中找到文件，并将其移出.*Trash* 目录。

HDFS 会自动删除回收站中的文件，但是其他文件系统并不具备这项功能。对于这些文件系统，必须定期手动删除。执行以下命令可以删除已在回收站中超过最小时限的所有文件：

```
% hadoop fs -expunge
```

`Trash` 类的 `expunge()`方法也具有相同的效果。

6. 作业调度

在针对多用户的设置中，可以考虑升级作业调度器队列配置，以反映在组织方面的需求。例如，可以为使用集群的每个组设置一个队列。详见 4.3 节中对作业调度的讨论。

7. 慢启动 reduce

在默认情况下，调度器将会一直等待，直到该作业的 5%的 map 任务已经结束才会调度 reduce 任务。对于大型作业来说，这可能会降低集群的利用率，因为在等待 map 任务执行完毕的过程之中，占用了 reduce 容器。可以将 `mapreduce.job.reduce.slowstart.completedmaps` 的值设得更大，例如0.80(80%)，能够提升吞吐率。

8. 短回路本地读

当从 HDFS 读取文件时，客户端联系 datanode，然后数据通过 TCP 连接发送给客

户端。如果正在读取的数据块和客户端在同一节点上，那么客户端绕过网络从磁盘上直接读取数据效率会更高。这又称作“短回路本地读”(*short-circuit local read*)，这种方式能够让应用程序如 HBase 执行效率更高。

将属性 `dfs.client.read.shortcircuit` 设置为 true，即可启用短回路本地读。该读操作基于 Unix 域套接字实现，在客户端和 datanode 之间的通信中使用了一个本地路径。该路径使用属性 `dfs.domain.socket.path` 进行设置，且必须是一条仅有 datanode 用户(典型的为 `hdfs`)或 root 用户能够创建的路径，例如 */var/run/hadoop-hdfs/dn_socket*。

10.4 安全性

早期版本的 Hadoop 假定 HDFS 和 MapReduce 集群运行在安全环境中，由一组相互合作的用户所操作，因而访问控制措施的目标是防止偶然的数据丢失，而非阻止非授权的数据访问。例如，HDFS 中的文件许可模块会阻止用户由于程序漏洞而毁坏整个文件系统，也会阻止运行不小心输入的 `hadoop fs -rmr /`指令，但却无法阻止某个恶意用户假冒 root 身份来访问或删除集群中的某些数据。

从安全角度分析，Hadoop 缺乏一个安全的认证机制，以确保正在操作集群的用户恰是所声称的安全用户。Hadoop 的文件许可模块只提供一种简单的认证机制来决定各个用户对特定文件的访问权限。例如，某个文件的读权限仅开放给某一组用户，从而阻止其他用户组的成员读取该文件。然而，这种认证机制仍然远远不够，恶意用户只要能够通过网络访问集群，就有可能伪造合法身份来攻击系统。

包含个人身份信息的数据(例如终端用户的全名或 IP 地址)非常敏感。一般情况下，需要严格限制组织内部的能够访问这类信息的员工数。相比之下，敏感性不强(或匿名化)的数据则可以开放给更多用户。如果把同一集群上的数据划分不同的安全级别，在管理上会方便很多，且低安全级别的数据也能够被广泛共享。然而，为了迎合数据保护的常规需求，共享集群的安全认证是不可或缺的。

雅虎公司在 2009 年就遇到了该难题，因此组织了一个工程师团队来实现 Hadoop 的安全认证。这个团队提出了一个方案：用 Kerberos(一个成熟的开源网络认证协议)实现用户认证，Hadoop 不直接管理用户隐私，而 Kerberos 也不关心用户的授权细节。换句话说，Kerberos 的职责在于鉴定登录帐号是否就是所声称的用户，Hadoop 则决定该用户到底拥有多少权限。

Hadoop 中的安全技术涉及内容较多，因此这里只介绍一些突出的内容。若想了解更多背景，可以参阅 Ben Spivey 和 Jason Garman 的《Hadoop 安全》(Hadoop Security)(O’Reilly,2014)。

10.4.1 Kerberos 和 Hadoop

从宏观角度来看，使用 Kerberos 时，一个客户端要经过三个步骤才可以获得服务。在各个步骤，客户端需要和一个服务器交换报文。

1. **认证**。客户端向认证服务器发送一条报文，并获取一个含时间戳的票据授予票据(Ticket-Granting Ticket，TGT)。
2. **授权**。客户端使用 TGT 向票据授予服务器(Ticket-Granting Server，TGS)请求一个服务票据。
3. **服务请求**。客户端向服务器出示服务票据，以证实自己的合法性。该服务器提供客户端所需服务，在 Hadoop 应用中，服务器可以是 namenode 或资源管理器。

同时，认证服务器和票据授予服务器构成了密钥分配中心(*Key Distribution Center*，KDC)。整个过程如图 10-2 所示。

图 10-2. Kerberos 票据交换协议的三个步骤

授权和服务请求步骤并非用户级别的行为：客户端系统会代替用户来执行这些步骤。但是认证步骤通常需要由用户调用 kinit 命令来执行，该过程会提示用户输入密码。需要指出的是，这并不意味着每次运行一个作业或访问 HDFS 的时候都会强迫用户键入密码，因为用户所申请到的 TGT 具备一定的有效期。TGT 有效期的默认值是 10 个小时(可以更新至一周)。更通用的做法是采用自动认证：即在登录操作系统的时候自动执行认证操作，从而只需单次登录(single sign-on)到 Hadoop。

如果用户不期望被提示输入密码(例如，运行一个无人值守的 MapReduce 作业)，则可以使用 `ktutil` 命令创建一个 Kerberos 的 *keytab* 文件，该文件保存了用户密码并且可以通过`-t`选项应用于`kinit`命令。

示例

下面我们来看一个例子。首先，将 *coresite.xml* 文件中的 `hadoop.security.authentication` 属性项设置为 `kerberos`，启用 Kerberos 认证。[①]该属性项的默认值是 `simple`，表示将采用传统的向后兼容(但是不安全)方式，即利用操作系统用户名称来决定登录者的身份。

其次，还需要将同一文件中的 `hadoop.security.authorization` 属性项设置为 `true`，以启用服务级别的授权。可以配置 *hadoop-policy.xml* 文件中的访问控制列表(ACL)以决定哪些用户和组能够访问哪些 Hadoop 服务。这些服务在协议级别定义，包括针对 MapReduce 作业提交的服务、针对 namenode 通信的服务等。默认情况下，各个服务的 ACL 都被设置为*，表示所有用户能够访问所有服务。但在现实情况下，还是有必要充分考虑 ACL 策略，控制访问服务的用户和组的范围。

ACL 的格式很简单，前一段是以逗号隔开的用户名称列表，后一段是以逗号隔开的组名称列表，两段间以空格隔开。例如，ACL 片段 `preston,howard directors,inventors` 会将某服务的访问权限授予用户 `preston` 或用户 `howard`，或组 `directors`、组 `inventors`。

当 Kerberos 认证启用后，以下输出内容显示了从本地复制一个文件到 HDFS 中时系统反馈的结果。

① 为了在 Hadoop 中使用 Kerberos 认证，用户需要安装、配置和运行一个 KDC(Hadoop 并不自带一个 KDC)。用户所在的组织机构可能已经拥有一个 KDC 了(例如，已经安装 Active Directory)；如果还没有任何 KDC，可以新建一个 MIT Kerberos 5 KDC。

```
% hadoop fs -put quangle.txt.
10/07/03 15:44:58 WARN ipc.Client: Exception encountered while connecting to the
server: javax.security.sasl.SaslException: GSS initiate failed [Caused by GSSEx
ception: No valid credentials provided (Mechanism level: Failed to find any Ker  beros
tgt)]
Bad connection to FS. command aborted. exception: Call to localhost/127.0.0.1:80
20 failed on local exception: java.io.IOException: javax.security.sasl.SaslExcep
tion: GSS initiate failed [Caused by GSSException: No valid credentials provided
(Mechanism level: Failed to find any Kerberos tgt)]
```

由于用户没有 Kerberos 票据，所以上述操作失败。用户可以使用 kinit 指令向 KDC 认证，并获得一张票据。

```
% kinit
Password for hadoop-user@LOCALDOMAIN: password
% hadoop fs -put quangle.txt .
% hadoop fs -stat %n quangle.txt
quangle.txt
```

现在，可以看到文件已成功写入 HDFS。注意，由于 Kerberos 票据的有效期是 10 小时，所以尽管执行的是两条文件系统指令，但实际上只需调用一次 kinit 命令。另外，klist 命令能查看票据的过期时间，kdestroy 指令可销毁票据。在获取票据之后，各项工作与平常无异。

10.4.2 委托令牌

在诸如 HDFS 或 MapReduce 的分布式系统中，客户端和服务器之间频繁交互，且每次交互均需认证。例如，一个 HDFS 读操作不仅会与 namenode 多次交互、还会与一个或多个 datanode 交互。如果在一个高负载集群上采用三步骤 Kerberos 票据交换协议来认证每次交互，则会对 KDC 造成很大压力。因此，Hadoop 使用委托令牌来支持后续认证访问，避免了多次访问 KDC。委托令牌的创建和使用过程均由 Hadoop 代表用户透明地进行，因而用户执行 kinit 命令登录之后，无需再做额外的操作。当然，了解委托令牌的基本用法仍然是有必要的。

委托令牌由服务器创建(在这里是指 namenode)，可以视为客户端和服务器之间共享的一个密文。当客户端首次通过 RPC 访问 namenode 时，客户端并没有委托令牌，因而需要利用 Kerberos 进行认证。之后，客户端从 namenode 取得一个委托令牌。在后续 RPC 调用中，客户端只需出示委托令牌，namenode 就能验证委托令牌的真伪(因为该令牌是由 namenode 使用密钥创建的)，并因此向服务器认证客户

端的身份。

客户端需要使用一种特殊类型的委托令牌来执行 HDFS 块操作，称为“块访问令牌”(block access token)。当客户端向 namenode 发出元数据请求时，namenode 创建相应的块访问令牌并发送回客户端。客户端使用块访问令牌向 datanode 认证自己的访问权限。由于 namenode 会和 datanode 分享它创建块访问令牌时用的密钥(通过心跳消息传送)，datanode 也能够验证这些块访问令牌。这样的话，仅当客户端已经从 namenode 获取了针对某一个 HDFS 块的块访问令牌时，才可以访问该块。相比之下，在不安全的 Hadoop 系统中，客户端只需知道块 ID 就能够访问一个块了。可以通过将 `dfs.block.access.token.enable` 的值设置为 `true` 来启用块访问令牌特性。

在 MapReduce 中，application master 共享 HDFS 中的作业资源和元数据(例如 JAR 文件、输入分片和配置文件)。用户代码运行在节点管理器上，并可以访问 HDFS 上的文件(该过程在 7.1 节中介绍过)。在作业运行过程中，这些组件使用委托令牌访问 HDFS。作业结束时，委托令牌失效。

默认的 HDFS 实例会自动获得委托令牌。但是若一个作业试图访问其他 HDFS 集群，则用户必须将 `mapreduce.job.hdfs-servers` 作业属性设置为一个由逗号隔开的 HDFS URI 列表，才能够获取相应的委托令牌。

10.4.3 其他安全性改进

Hadoop 已经全面强化了安全措施，以阻止用户在未授权的情况下访问资源。一些显著的变化如下。

- 任务可以由提交作业的用户以操作系统帐号启动运行，而不一定要由运行节点管理器的用户启动。这意味着，在这种情况下，可以借助操作系统来隔离正在运行的任务，使它们之间无法相互传送指令(例如，终止其他用户的任务)，这样的话，诸如任务数据等本地信息的隐私即可通过本地文件系统的安全性而得到保护。

  要启用这项特性，需要将 `yarn.nodemanager.containerexecutor.class` 设为 `org.apache.hadoop.yarn.server.nodemanager.LinuxContainer`

Executor。[1]此外，管理员还需确保各用户在集群的每个节点上都已经分配帐号(一般用 LDAP)。

- 当任务由提交作业的用户启动运行时，分布式缓存(参见 9.4.2 节)是安全的：把所有用户均可读的文件放到共享缓存中(默认的非安全方式)，把其他文件放在私有缓存中，仅限拥有者读取。
- 用户只能查看和修改自己的作业，无法操控他人的作业。为了启动该特性，需要将 mapreduce.cluster.acls.enabled 属性项设为 true。另外，mapreduce.job.acl-view-job 和 mapreduce.job.acl-modify-job 属性分别对应一个逗号分隔的用户列表，描述能够查看或修改指定作业的所有用户。
- shuffle 是安全的，可以阻止恶意用户请求获取其他用户的 map 输出。
- 正确配置之后，可以阻止恶意的辅助 namenode、datanode 或节点管理器加入集群，从而破坏集群中的数据。这可以通过要求 master 节点对试图与之连接的守护进程进行认证来实现。

  为了启用该特性，需要使用先前由 ktutil 命令创建的 *keytab* 文件来配置 Hadoop。以 datanode 为例，首先，把 dfs.datanode.keytab.file 属性设置为 keytab 文件名称；其次，把 dfs.datanode.kerberos.principal 属性设置为要用的 datanode 用户名称；最后，把 *hadoop-policy.xml* 文件中 security.datanode.protocol.acl 属性设置为 datanode 的用户名称，以设置 DataNodeProtocol 的 ACL。DatanodeProtocol 是 datanode 用于和 namenode 通信的协议类。
- datanode 最好运行在特权端口(端口号小于 1024)，使客户端确信它是安全启动的。
- 任务只与其父 application master 通信，从而阻止攻击者经由其他用户的作业获取 MapReduce 数据。
- Hadoop 的多个不同部件都提供了配置属性以支持网络数据加密，包括

① LinuxTaskController 使用一个已设置 setuid 位的可执行文件，即 *bin* 目录中的 *container-executor* 文件。用户需要确保这个二进制文件的拥有者是 root，并且它的 setuid 位已设置(用 chmod +s)。

RPC(hadoop.rpc.protection)，HDFS 块传输(dfs.encrypt.data.transfer)、MapReduce shuffle(mapreduce. shuffle.ssl.enabled) 和 UI (hadokop.ssl.enabled)。并且，在数据休息期间对数据的加密也是持续进行的，例如，通过这种方式，HDFS 块能够以加密方式进行存储。

10.5 利用基准评测程序测试 Hadoop 集群

集群是否已被正确建立？这个问题最好通过实验来回答：运行若干作业，并确信获得了预期结果。基准评测程序能获得满意的测试结果，用户可以拿结果数据和其他集群做比较，以检测新集群是否达到预期效果。此外，还可以据此调整集群设置以优化整体性能。这点一般通过监控系统实现(参见 11.2 节)，用户可以监测集群中的资源使用情况。

为了获得最佳评测结果，最好不要在运行基准评测程序时还同时运行其他任务。实际上，在集群入役之前进行评测最为合适，此时用户尚未对集群有依赖性。一旦用户已经依赖集群执行常规性作业，想要找到集群完全空闲的时间就非常困难了(除非和所有其他用户协商一个停止服务的时间段)。总而言之，基准评测程序最好在此之前就执行。

实践表明，硬盘驱动器故障是新系统最常见的硬件故障。通过运行含有高强度 I/O 操作的基准评测程序，例如即将提到的基准评测程序，就能在系统正式上线前对集群做“烤机”测试。

10.5.1 Hadoop 基准评测程序

Hadoop 自带若干基准评测程序，安装开销小、运行方便。基准评测程序打包为一个名为 tests.jar 的文件，经无参数解压缩之后，就可以获取文件列表和说明文档：

```
% hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-*-tests.jar
```

如果不指定参数，大多数基准测试程序都会显示具体用法。示例如下：

```
% hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-*-tests.jar \
  TestDFSIO
TestDFSIO.1.7
Missing arguments.
Usage: TestDFSIO [genericOptions] -read [-random | -backward |
-skip [-skipSize Size]] | -write | -append | -clean [-compression codecClassName]
```

```
[-nrFiles N] [-size Size[B|KB|MB|GB|TB]] [-resFile resultFileName]
[-bufferSize Bytes] [-rootDir]
```

1. 使用 TeraSort 来评测 HDFS

Hadoop 自带一个名为 *TeraSort* 的 Mapreduce 程序，该程序对输入进行全排序。[1] 由于全部输入数据集通过 shuffle 传输，所以 TeraSort 对于同时评测 HDFS 和 MapReduce 非常有用。测评分为三步：生成随机数据、执行排序和验证结果。

首先，使用 `teragen` 生成随机数据(可以在示例 JAR 文件中找到，而不是测试用 JAR 文件)。`teragen` 运行一个仅有 map 任务的作业，可以生成指定行数的二进制数据。每一行是 100 字节长，这样使用 1000 个 map 任务可生成 1TB 的数据，执行命令如下(10t 是 10 trillion 的缩写)：

```
% hadoop jar \
$HADOOP_HOME/share/hadoop /mapreduce/hadoop-mapreduce-examples-*.jar \  teragen
-Dmapreduce.job.maps=1000 10t random-data
```

接下来，运行 `TeraSort`：

```
% hadoop jar \
$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-*.jar \  terasort
random-data sorted-data
```

排序的总执行时间会是用户感兴趣的度量值。此外，通过 Web 界面(*http://resource-manager-host:8088/*)来观察作业的进度更有意义，这样可以了解作业在各个阶段的开销。在此基础上，可以练习如何调整系统参数(参见 6.6 节对作业调优的讨论)。

最后，需要验证在 *sorted-data* 文件中的数据是否已经排好序：

```
% hadoop jar \
$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-*.jar \
teravalidate sorted-data report
```

该命令运行了一个小的 MapReduce 作业，对排序后的数据执行一系列检查，以验证排序结果是否正确。任何错误都可以在输出文件 *report/part-r-00000* 中找到。

① 在 2008 年，TeraSort 用于 1 TB 数据的排序比赛并打破了世界纪录。参见 1.6 节。

2. 其他基准评测程序

Hadoop 的基准评测程序有很多，以下几种最常用。

- *TestDFSIO* 主要用于测试 HDFS 的 I/O 性能。该程序使用一个 MapReduce 作业作为并行读/写文件的一种便捷途径。
- *MRBench*(使用 `mrbench`)会多次运行一个小型作业。与 TeraSort 相互映衬，该基准的主要目的是检验小型作业能否快速响应。
- *NNBench*(使用 `nnbench`)测试 namenode 硬件的加载过程。
- *Gridmix* 是一个基准评测程序套装。通过模拟一些真实常见的数据访问模式，Gridmix 能逼真地为一个集群的负载建模。用户可参阅分发包中的文档来了解如何运行 Gridmix。
- *SWIM*(Statistical Workload Injector for MapReduce)，是一个真实的 MapReduce 工作负载库，可以用来为被测系统生成代表性的测试负载。
- *TPCx-HS*，基于 TeraSort 的标准基准评测程序，来自事务处理性能委员会(Transaction Processing Performance Council)。

10.5.2 用户作业

出于集群性能调优的目的，最好包含若干代表性强、使用频繁的作业。这样的话，调优操作可以更有针对性，而不只是对通用场景进行调优。但如果待测集群是用户的第一个 Hadoop 集群，且还没有任何作业，则 Gridmix 或 SWIM 都不失为一个好的评测方案。

如果想把自己的作业作为基准评测时，用户还需要为作业选择数据集合。这样的话，不管运行多少次，作业始终都基于相同的数据集合，便于分析、比较性能变迁。当新建或升级集群时，使用同一数据集合还可以比较新旧集群的性能。

第 11 章

管理 Hadoop

第 10 章介绍如何搭建 Hadoop 集群。本章将关注如何保障集群的平稳运行。

11.1 HDFS

11.1.1 永久性数据结构

作为管理员，深入了解 namenode、辅助 namenode 和 datanode 等 HDFS 组件如何在磁盘上组织永久性数据非常重要。洞悉各文件的用法有助于进行故障诊断和故障检出。

1. namenode 的目录结构

运行中的 namenode 有如下所示的目录结构：

```
${dfs.namenode.name.dir}/
├── current
│   ├── VERSION
│   ├── edits_0000000000000000001-0000000000000000019
│   ├── edits_inprogress_0000000000000000020
│   ├── fsimage_0000000000000000000
│   ├── fsimage_0000000000000000000.md5
│   ├── fsimage_0000000000000000019
│   ├── fsimage_0000000000000000019.md5
│   └── seen_txid
└── in_use.lock
```

如第 10 章所示，dfs.namenode.name.dir 属性描述了一组目录，各个目录存储着镜像内容。该机制使系统具备了一定的复原能力，特别是当其中一个目录是 NFS 的一个挂载时（推荐配置）。

VERSION 文件是一个 Java 属性文件，其中包含正在运行的 HDFS 的版本信息。该文件一般包含以下内容：

```
#Mon Sep 29 09:54:36 BST 2014
namespaceID=1342387246
clusterID=CID-01b5c398-959c-4ea8-aae6-1e0d9bd8b142
cTime=0
storageType=NAME_NODE
blockpoolID=BP-526805057-127.0.0.1-1411980876842
layoutVersion=-57
```

属性 layoutVersion 是一个负整数，描述 HDFS 持久性数据结构(也称布局)的版本，但是该版本号与 Hadoop 发布包的版本号无关。只要布局变更，版本号便会递减(例如，版本号-57 之后是-58)，此时，HDFS 也需要升级。否则，磁盘仍然使用旧版本的布局，新版本的 namenode(或 datanode)就无法正常工作。要想知道如何升级 HDFS，请参见 11.3.3 节。

属性 namespaceID 是文件系统命名空间的唯一标识符，是在 namenode 首次格式化时创建的。clusterID 是将 HDFS 集群作为一个整体赋予的唯一标识符，对于联邦 HDFS 非常重要(见 3.2.4 节)，这里一个集群由多个命名空间组成，且每个命名空间由一个 namenode 管理。blockpoolID 是数据块池的唯一标识符，数据块池中包含了由一个 namenode 管理的命名空间中的所有文件。

cTime 属性标记了 namenode 存储系统的创建时间。对于刚刚格式化的存储系统，这个属性值为 0；但是在文件系统升级之后，该值会更新到新的时间戳。

storageType 属性说明该存储目录包含的是 namenode 的数据结构。

in_use.lock 文件是一个锁文件，namenode 使用该文件为存储目录加锁。可以避免其他 namenode 实例同时使用(可能会破坏)同一个存储目录的情况。

namenode 的存储目录中还包含 *edits*、*fsimage* 和 *seen_txid* 等二进制文件。只有深入学习 namenode 的工作机理，才能够理解这些文件的用途。

2. 文件系统映像和编辑日志

文件系统客户端执行写操作时(例如创建或移动文件)，这些事务首先被记录到编辑

日志中。namenode 在内存中维护文件系统的元数据；当编辑日志被修改时，相关元数据信息也同步更新。内存中的元数据可支持客户端的读请求。

编辑日志在概念上是单个实体，但是它体现为磁盘上的多个文件。每个文件称为一个“段”(*segment*)，名称由前缀 *edits* 及后缀组成，后缀指示出该文件所包含的事务 ID。任一时刻只有一个文件处于打开可写状态(前述例子中为 *edits_inprogress_0000000000000000020*)，在每个事务完成之后，且在向客户端发送成功代码之前，文件都需要更新和同步。当 namenode 向多个目录写数据时，只有在所有写操作更新并同步到每个复本之后方可返回成功代码，以确保任何事务都不会因为机器故障而丢失。

每个 *fsimage* 文件都是文件系统元数据的一个完整的永久性检查点。(前缀表示映像文件中的最后一个事务。)并非每一个写操作都会更新该文件，因为 *fsimage* 是一个大型文件(甚至可高达几个 GB)，如果频繁地执行写操作，会使系统运行极为缓慢。但这个特性根本不会降低系统的恢复能力，因为如果 namenode 发生故障，最近的 *fsimage* 文件将被载入到内存以重构元数据的最近状态，再从相关点开始向前执行编辑日志中记录的每个事务。事实上，namenode 在启动阶段正是这样做的(参见 11.1.2 节对安全模式的讨论)。

每个 *fsimage* 文件包含文件系统中的所有目录和文件 inode 的序列化信息。每个 inode 是一个文件或目录的元数据的内部描述方式。对于文件来说，包含的信息有“复本级别”(replication level)、修改时间和访问时间、访问许可、块大小、组成一个文件的块等；对于目录来说，包含的信息有修改时间、访问许可和配额元数据等信息。

数据块存储在 datanode 中，但 *fsimage* 文件并不描述 datanode。取而代之的是，namenode 将这种块映射关系放在内存中。当 datanode 加入集群时，namenode 向 datanode 索取块列表以建立块映射关系；namenode 还将定期征询 datanode 以确保它拥有最新的块映射。

如前所述，编辑日志会无限增长(即使物理上它是分布在多个 *edits* 文件中)。尽管这种情况对于 namenode 的运行没有影响，但由于需要恢复(非常长的)编辑日志中的各项事务，namenode 的重启操作会比较慢。在这段时间内，文件系统将处于离线状态，这会有违用户的期望。

解决方案是运行辅助 namenode，为主 namenode 内存中的文件系统元数据创建检

查点。[1]创建检查点的步骤如下所示(图 11-1 中也概略展现了前述的编辑日志和映像文件)。

(1) 辅助 namenode 请求主 namenode 停止使用正在进行中的 *edits* 文件，这样新的编辑操作记录到一个新文件中。主 namenode 还会更新所有存储目录中的 *seen_txid* 文件。

(2) 辅助 namenode 从主 namenode 获取最近的 *fsimage* 和 *edits* 文件(采用 HTTP GET)。

(3) 辅助 namenode 将 *fsimage* 文件载入内存，逐一执行 *edits* 文件中的事务，创建新的合并后的 *fsimage* 文件。

(4) 辅助 namenode 将新的 *fsimage* 文件发送回主 namenode(使用 HTTP PUT)，主 namenode 将其保存为临时的 *.ckpt* 文件。

(5) 主 namenode 重新命名临时的 *fsimage* 文件，便于日后使用。

最终，主 namenode 拥有最新的 *fsimage* 文件和一个更小的正在进行中的 *edits* 文件(*edits* 文件可能非空，因为在创建检查点过程中主 namenode 还可能收到一些编辑请求)。当 namenode 处在安全模式时，管理员也可调用 `hdfs dfsadmin -saveNamespace` 命令来创建检查点。

这个过程清晰解释了辅助 namenode 和主 namenode 拥有相近内存需求的原因(因为辅助 namenode 也把 *fsimage* 文件载入内存)。因此，在大型集群中，辅助 namenode 需要运行在一台专用机器上。

创建检查点的触发条件受两个配置参数控制。通常情况下，辅助 namenode 每隔一小时(由 `dfs.namenode.checkpoint.period` 属性设置，以秒为单位)创建检查点；此外，如果从上一个检查点开始编辑日志的大小已经达到 100 万个事务(由 `dfs.namenode.checkpoint.txns` 属性设置)时，那么即使不到一小时，也会创建检查点，检查频率为每分钟一次(由 `dfs.namenode.checkpoint.check.period` 属性设置，以秒为单位)。

① 实际上用户可以使用`-checkpoint` 选项来启动 namenode，它将运行一个检查点过程以应对另一个(主)namenode。在功能上，这等价于运行一个辅助 namenode；但直到本书写就之际，这个技术并未体现出强于辅助 namenode 的能力(实际上，辅助 namenode 仍然是目前最常见的用法)。当在一个高有效的环境之中运行时(参见 3.2.5 节对 HTTP 高可用性的讨论)，备用节点执行检查点功能。

图 11-1. 创建检查点的过程

3. 辅助 namenode 的目录结构

辅助 namenode 的检查点目录(`dfs.namenode.checkpoint.dir`)的布局和主 namenode 的检查点目录的布局相同。这种设计方案的好处是，在主 namenode 发生故障时(假设没有及时备份，甚至在 NFS 上也没有)，可以从辅助 namenode 恢复数据。有两种实现方法。方法一是将相关存储目录复制到新的 namenode 中；方法二是使用`-importCheckpoint` 选项启动 namenode 守护进程，从而将辅助 namenode 用作新的主 namenode。借助该选项，仅当 `dfs.namenode.name.dir` 属性定义的目录中没有元数据时，辅助 namenode 会从 `dfs.namenode.`

checkpoint.dir 属性定义的目录载入最新的检查点 namenode 元数据，因此，不必担心这个操作会覆盖现有的元数据。

4. datanode 的目录结构

和 namenode 不同的是，datanode 的存储目录是初始阶段自动创建的，不需要额外格式化。datanode 的关键文件和目录如下所示：

```
${dfs.datanode.data.dir}/
├ current
| ├── BP-526805057-127.0.0.1-1411980876842
| | └── current
| |   ├── VERSION
| |   ├── finalized
| |   |   ├─── blk_1073741825
| |   |   ├─── blk_1073741825_1001.meta
| |   |   ├─── blk_1073741826
| |   |   └─── blk_1073741826_1002.meta
| |   └── rbw
| └── VERSION
└ in_use.lock
```

HDFS 数据块存储在以 *blk_*为前缀名的文件中，文件名包含了该文件存储的块的原始字节数。每个块有一个相关联的带有*.meta* 后缀的元数据文件。元数据文件包括头部(含版本和类型信息)和该块各区段的一系列的校验和。

每个块属于一个数据块池，每个数据块池都有自己的存储目录，目录根据数据块池 ID 形成(和 namenode 的 VERSION 文件中的数据块池 ID 相同)。

当目录中数据块的数量增加到一定规模时，datanode 会创建一个子目录来存放新的数据块及其元数据信息。如果当前目录已经存储了 64 个(通过 dfs.datanode.numblocks 属性设置)数据块时，就创建一个子目录。终极目标是设计一棵高扇出的目录树，即使文件系统中的块数量非常多，目录树的层数也不多。通过这种方式，datanode 可以有效管理各个目录中的文件，避免大多数操作系统遇到的管理难题，即很多(成千上万个)文件放在同一个目录之中。

如果 dfs.datanode.data.dir 属性指定了不同磁盘上的多个目录，那么数据块会以轮转 (round-robin)的方式写到各个目录中。注意，同一个 datanode 上的每个磁盘上的块不会重复，只有不同 datanode 之间的块才有可能重复。

11.1.2 安全模式

namenode 启动时，首先将映像文件(*fsimage*)载入内存，并执行编辑日志(*edits*)中的各项编辑操作。一旦在内存中成功建立文件系统元数据的映像，则创建一个新的 *fsimage* 文件(该操作不需要借助辅助 namenode)和一个空的编辑日志。在这个过程中，namenode 运行在安全模式，意味着 namenode 的文件系统对于客户端来说是只读的。

严格来说，在安全模式下，只有那些访问文件系统元数据的文件系统操作是肯定成功执行的，例如显示目录列表等。对于读文件操作来说，只有集群中当前 datanode 上的块可用时，才能够工作。但文件修改操作(包括写、删除或重命名)均会失败。

需要强调的是，系统中数据块的位置并不是由 namenode 维护的，而是以块列表的形式存储在 datanode 中(每个 datanode 存储的块组成的列表)。在系统的正常操作期间，namenode 会在内存中保留所有块位置的映射信息。在安全模式下，各个 datanode 会向 namenode 发送最新的块列表信息，namenode 了解到足够多的块位置信息之后，即可高效运行文件系统。如果 namenode 认为向其发送更新信息的 datanode 节点过少，则它会启动块复制进程，以将数据块复制到新的 datanode 节点。然而，在大多数情况下上述操作都是不必要的(因为实际上 namenode 只需继续等待更多 datanode 发送更新信息即可)，并浪费了集群的资源。实际上，在安全模式下 namenode 并不向 datanode 发出任何块复制或块删除的指令。

如果满足“最小复本条件”(minimal replication condition)，namenode 会在 30 秒钟之后就退出安全模式。所谓的最小复本条件指的是在整个文件系统中有 99.9%的块满足最小复本级别(默认值是 1，由 `dfs.namenode.replication.min` 属性设置，参见表 11-1)。

在启动一个刚刚格式化的 HDFS 集群时，因为系统中还没有任何块，所以 namenode 不会进入安全模式。

进入和离开安全模式

要想查看 namenode 是否处于安全模式，可以像下面这样用 `dfsadmin` 命令：

```
% hdfs dfsadmin -safemode get
Safe mode is ON
```

HDFS 的网页界面也能够显示 namenode 是否处于安全模式。

表 11-1. 安全模式的属性

属性名称	类型	默认值	说明
dfs.namenode.replication.min	int	1	成功执行写操作所需要创建的最少复本数目(也称为最小复本级别)
dfs.namenode.safemode.threshold-pct	float	0.999	在 namenode 退出安全模式之前，系统中满足最小复本级别(由 dfs.namenode.replication.min 定义)的块的比例。将这项值设为 0 或更小会令 namenode 无法启动安全模式；设为高于 1 则永远不会退出安全模式
dfs.namenode.safemode.extension	int	30000	在满足最小复本条件(由 dfs.namenode.safemode.threshold-pct 定义)之后，namenode 还需要处于安全模式的时间(以毫秒为单位)。对于小型集群(几十个节点)来说，这项值可以设为 0

有时，用户期望在执行某条命令之前 namenode 先退出安全模式，特别是在脚本中。使用 wait 选项能够达到这个目的：

```
%hdfs dfsadmin -safemode wait
# command to read or write a file
```

管理员随时可以让 namenode 进入或离开安全模式。这项功能在维护和升级集群时非常关键，因为需要确保数据在指定时段内是只读的。使用以下指令进入安全模式：

```
% hdfs dfsadmin -safemode enter
Safe mode is ON
```

前面提到过，namenode 在启动阶段会处于安全模式。在此期间也可使用这条命令，从而确保 namenode 在启动完毕之后不离开安全模式。另一种使 namenode 永远处于安全模式的方法是将属性 dfs.namenode.safemode. threshold-pct 的值设为大于 1。

运行以下指令即可使得 namenode 离开安全模式：

```
% hdfs dfsadmin -safemode leave
Safe mode is OFF
```

11.1.3 日志审计

HDFS 的日志能够记录所有文件系统访问请求，有些组织需要这项特性来进行审计。对日志进行审计是 log4j 在 INFO 级别实现的。在默认配置下，此项特性并未启用，但是通过在文件 *hadoop-env.sh* 中增加以下这行命令，很容易启动该日志审计特性：

```
export HDFS_AUDIT_LOGGER="INFO,RFAAUDIT"
```

每个 HDFS 事件均在审计日志(*hdfs-audit.log*)中生成一行日志记录。下例说明如何对*/user/tom* 目录执行 `list status` 命令(列出指定目录下的文件/目录的状态)：

```
2014-09-30 21:35:30,484 INFO FSNamesystem.audit: allowed=true ugi=tom
(auth:SIMPLE) ip=/127.0.0.1 cmd=listStatus src=/user/tom dst=null
perm=null proto=rpc
```

11.1.4 工具

1. dfsadmin 工具

dfsadmin 工具用途较广，既可以查找 HDFS 状态信息，又可在 HDFS 上执行管理操作。以 `hdfs dfsadmin` 形式调用，且需要超级用户权限。

表 11-2 列举了部分 *dfsadmin* 命令。要想进一步了解详情，可以用`-help` 命令。

表 11-2. dfsadmin 命令

命令	说明
`-help`	显示指定命令的帮助，如果未指明命令，则显示所有命令的帮助
`-report`	显示文件系统的统计信息(类似于在网页界面上显示的内容)，以及所连接的各个 datanode 的信息
`-metasave`	将某些信息存储到 Hadoop 日志目录中的一个文件中，包括正在被复制或删除的块的信息以及已连接的 datanode 列表
`-safemode`	改变或查询安全模式，参见 11.1.2 节对安全模式的讨论
`-saveNamespace`	将内存中的文件系统映像保存为一个新的 *fsimage* 文件，重置 *edits* 文件。该操作仅在安全模式下执行
`-fetchImage`	从 namenode 获取最新的 *fsimage* 文件，并保存为本地文件
`-refreshNodes`	更新允许连接到 namenode 的 datanode 列表，参见 11.3.2 节对委任和解除节点的讨论

续表

命令	说明
-upgradeProgress	获取有关 HDFS 升级的进度信息或强制升级。参见 11.3.3 对升级的讨论
-finalizeUpgrade	移除 datanode 和 namenode 的存储目录上的旧版本数据。这个操作一般在升级完成而且集群在新版本下运行正常的情况下执行。参见 11.3.3 节对升级的讨论
-setQuota	设置目录的配额，即设置以该目录为根的整个目录树最多包含多少个文件和目录。这项配置能有效阻止用户创建大量小文件，从而保护 namenode 的内存(文件系统中的所有文件、目录和块的各项信息均存储在内存中)
-clrQuota	清理指定目录的配额
-setSpaceQuota	设置目录的空间配额，以限制存储在目录树中的所有文件的总规模。分别为各用户指定有限的存储空间很有必要
-clrSpaceQuota	清理指定的空间配额
-refreshServiceAcl	刷新 namenode 的服务级授权策略文件
-allowSnapshot	允许为指定的目录创建快照
-disallowSnapshot	禁止为指定的目录创建快照

2. 文件系统检查 fsck 工具

Hadoop 提供 *fsck* 工具来检查 HDFS 中文件的健康状况。该工具会查找那些在所有 datanode 中均缺失的块以及过少或过多复本的块。下例演示如何检查某个小型集群的整个文件系统：

```
% hdfs fsck /
.....................Status: HEALTHY
 Total size:    511799225 B
 Total dirs:    10
 Total files:    22
 Total blocks (validated):    22 (avg. block size 23263601 B)
 Minimally replicated blocks:    22 (100.0 %)
 Over-replicated blocks:    0 (0.0 %)
 Under-replicated blocks:    0 (0.0 %)
 Mis-replicated blocks:    0 (0.0 %)
 Default replication factor:    3
 Average block replication:    3.0
 Corrupt blocks:    0
 Missing replicas:    0 (0.0 %)
 Number of data-nodes:    4
 Number of racks:    1

The filesystem under path '/' is HEALTHY
```

fsck 工具从给定路径(本例是文件系统的根目录)开始循环遍历文件系统的命名空

间，并检查它所找到的所有文件。对于检查过的每个文件，都会打印一个点“.”。在此过程中，该工具获取文件数据块的元数据并找出问题或检查它们是否一致。注意，*fsck* 工具只是从 namenode 获取信息，并不与任何 datanode 进行交互，因此并不真正获取块数据。

fsck 输出文件的大部分内容都容易理解，以下仅说明部分信息。

- **过多复制的块** 指复本数超出最小复本级别的块。严格意义上讲，这并非一个大问题，HDFS 会自动删除多余复本。
- **仍需复制的块** 指复本数目低于最小复本级别的块。HDFS 会自动为这些块创建新的复本，直到达到最小复本级别。可以调用 `hdfs dfsadmin -metasave` 指令了解正在复制的(或等待复制的)块的信息。
- **错误复制的块** 是指违反块复本放置策略的块(参见 3.6.2 节“复本的放置”相关内容)。例如，在最小复本级别为 3 的多机架集群中，如果一个块的三个复本都存储在一个机架中，则可认定该块的复本放置错误，因为一个块的复本要分散在至少两个机架中，以提高可靠性。
- **损坏的块** 指所有复本均已损坏的块。如果虽然部分复本损坏，但至少还有一个复本完好，则该块就未损坏；namenode 将创建新的复本，直到达到最小复本级别。
- **缺失的复本** 指在集群中没有任何复本的块。

损坏的块和缺失的块是最需要考虑的，因为这意味着数据已经丢失了。默认情况下，*fsck* 不会对这类块进行任何操作，但也可以让 *fsck* 执行如下某一项操作。

- **移动** 使用`-move` 选项将受影响的文件移到 HDFS 的*/lost+found* 目录。这些受影响的文件会分裂成连续的块链表，可以帮助用户挽回损失。
- **删除** 使用`-delete` 选项删除受影响的文件。记住，在删除之后，这些文件无法恢复。

查找一个文件的数据块 *fsck* 工具能够帮助用户轻松找到属于特定文件的数据块。例如：

```
% hdfs fsck /user/tom/part-00007 -files -blocks -racks
/user/tom/part-00007 25582428 bytes, 1 block(s): OK
0. blk_-3724870485760122836_1035 len=25582428 repl=3 [/default-rack/10.251.43.2:50010,
```

```
/default-rack/10.251.27.178:50010, /default-rack/10.251.123.163:50010]
```

输出内容表示文件*/user/tom/part-00007* 包含一个块，该块的三个复本存储在不同 datanode。*fsck* 所使用的三个选项的含义如下。

- `-files` 选项显示第一行信息，包括文件名称、大小、块数量和健康状况(是否有缺失的块)
- `-blocks` 选项描述文件中各个块的信息，每个块一行
- `-racks` 选项显示各个块的机架位置和 datanode 的地址

如果不指定任何参数，运行不带参数的 `hdfs fsck` 命令会显示完整的使用说明。

3. datanode 块扫描器

各个 datanode 运行一个块扫描器，定期检测本节点上的所有块，从而在客户端读到坏块之前及时地检测和修复坏块。可以依靠扫描器所维护的块列表依次扫描块，查看是否有校验和错误。扫描器还使用节流机制，来维持 datanode 的磁盘带宽(换句话说，块扫描器工作时仅占用一小部分磁盘带宽)。

在默认情况下，块扫描器每隔三周就会检测块，以应对可能的磁盘故障，该周期由 `dfs.datanode.scan.period.hours` 属性设置，默认值是 504 小时。损坏的块被报给 namenode，并被及时修复。

用户可以访问 datanode 的网页(*http://datanode:50075/blockScannerReport*)来获取该 datanode 的块检测报告。以下是一份报告的范例，很容易理解：

```
Total Blocks                 : 21131
Verified in last hour        :    70
Verified in last day         :  1767
Verified in last week        :  7360
Verified in last four weeks  : 20057
Verified in SCAN_PERIOD      : 20057
Not yet verified             :  1074
Verified since restart       : 35912
Scans since restart          :  6541
Scan errors since restart    :     0
Transient scan errors        :     0
Current scan rate limit KBps :  1024
Progress this period         :   109%
Time left in cur period      : 53.08%
```

通过指定 listblocks 参数，*http://datanode:50075/blockScannerReport? Listblocks* 会在报告中列出该 datanode 上所有的块及其最新验证状态。下面节选部分内容(由于页面宽度限制，报告中的每行内容被显示成两行)：

```
blk_6035596358209321442   : status : ok type : none scan time   :
 0                    not yet verified
blk_3065580480714947643 : status : ok type : remote scan time :
1215755306400       2008-07-11 05:48:26,400
blk_8729669677359108508 : status : ok type : local scan time :
1215755727345       2008-07-11 05:55:27,345
```

第一列是块 ID，接下来是一些键-值对。块的状态(status)要么 failed(损坏的)，要么 ok(良好的)，由最近一次块扫描是否检测到校验和来决定。扫描类型(type)可以是 local(本地的)、remote(远程的)或 none(没有)。如果扫描操作由后台线程执行，则是 local；如果扫描操作由客户端或其他 datanode 执行，则是 remote；如果针对该块的扫描尚未执行，则是 none。最后一项信息是扫描时间，从 1970 年 1 月 1 号午夜开始到扫描时间为止的毫秒数，另外也提供更易读的形式。

4. 均衡器

随着时间推移，各个 datanode 上的块分布会越来越不均衡。不均衡的集群会降低 MapReduce 的本地性，导致部分 datanode 相对更加繁忙。应避免出现这种情况。

均衡器(balancer)程序是一个 Hadoop 守护进程，它将块从忙碌的 datanode 移到相对空闲的 datanode，从而重新分配块。同时坚持块复本放置策略，将复本分散到不同机架，以降低数据损坏率(参见 3.6.2 节)。它不断移动块，直到集群达到均衡，即每个 datanode 的使用率(该节点上已使用的空间与空间容量之间的比率)和集群的使用率(集群中已使用的空间与集群的空间容量之间的比率)非常接近，差距不超过给定的阀值。可调用下面指令启动均衡器：

```
% start-balancer.sh
```

-threshold 参数指定阈值(百分比格式)，以判定集群是否均衡。该标记是可选的；若省略，默认阈值是 10%。任何时刻，集群中都只运行一个均衡器。

均衡器会一直运行，直到集群变得均衡为止，此时，均衡器不能移动任何块，或失去与 namenode 的联络。均衡器在标准日志目录中创建一个日志文件，记录它所执行的每轮重新分配过程(每轮次输出一行)。以下是针对一个小集群的日志输出

(为适应页面显示要求稍微调整了格式)：

```
Time Stamp              Iteration# Bytes Already Moved..Left To Move.. Being Moved
Mar 18, 2009 5:23:42 PM 0          0 KB              219.21 MB   150.29 MB
Mar 18, 2009 5:27:14 PM 1          195.24 MB          22.45 MB    150.29 MB
The cluster is balanced. Exiting...
Balancing took 6.072933333333333 minutes
```

为了降低集群负荷、避免干扰其他用户，均衡器被设计为在后台运行。在不同节点之间复制数据的带宽也是受限的。默认值是很小的 1 MB/s，可以通过 *hdfs-site.xml* 文件中的 `dfs.datanode.balance.bandwidthPerSec` 属性重新设定(单位是字节)。

11.2 监控

监控是系统管理的重要内容。在本小节中，我们概述 Hadoop 的监控工具，看看它们如何与外部监控系统相结合。

监控的目标在于检测集群在何时未提供所期望的服务。主守护进程是最需要监控的，包括主 namenode、辅助 namenode 和资源管理器。我们可以预期少数 datanode 和节点管理器会出现故障，特别是在大型集群中。因此，需要为集群预留额外的容量，即使有一小部分节点宕机，也不会影响整个系统的运作。

除了以下即将介绍的工具之外，管理员还可以定期运行一些测试作业来检查集群的健康状况。

11.2.1 日志

所有 Hadoop 守护进程都会产生日志文件，这些文件非常有助于查明系统中已发生的事件。10.3.2 节在讨论系统日志文件时解释了如何配置这些文件。

1. 设置日志级别

在故障排查过程中，若能够临时变更特定组件的日志的级别的话，将非常有益。

可以通过 Hadoop 守护进程的网页(在守护进程的网页的*/logLevel* 目录下)来改变任何 log4j 日志名称的日志级别。一般来说，Hadoop 中的日志名称对应着执行相关日志操作的类名称。此外，也有例外情况，因此最好从源代码中查找日志名称。

也可以为所有以给定前缀开始的类包启用日志。例如，为了启用资源管理器相关的所有类的日志调试特性，可以访问它的网页 *http://resource-manager-host:8088/logLevel*，并将日志名 `org.apache.hadoop.yarn.server.resourcemanager` 设置为 DEBUG 级别。

也可以通过以下命令实现上述目标：

```
% hadoop daemonlog -setlevel resource-manager-host:8088 \
  org.apache.hadoop.yarn.server.resourcemanager DEBUG
```

按照上述方式修改的日志级别会在守护进程重启时被复位，通常这也符合用户预期。如果想永久性地变更日志级别，只需在配置目录下的 *log4j.properties* 文件中添加如下这行代码：

```
log4j.logger.org.apache.hadoop.yarn.server.resourcemanager=DEBUG
```

2. 获取堆栈跟踪

Hadoop 守护进程提供一个网页(网页界面的*/stacks* 目录)对正在守护进程的 JVM 中运行着的线程执行线程转储(thread dump)。例如，可通过 *http://resource-manager-host:8088/stacks* 获得资源管理器的线程转储。

11.2.2 度量和 JMX(Java 管理扩展)

Hadoop 守护进程收集事件和度量相关的信息，这些信息统称为“度量”(metric)。例如，各个 datanode 会收集以下度量(还有更多)：写入的字节数、块的复本数和客户端发起的读操作请求数(包括本地的和远程的)。

有时用 *metrics2* 指代 Hadoop 2 及后续版本的度量系统，以区别早期版本 Hadoop 的旧度量系统(现在已经不支持)。

度量从属于特定的上下文(context)。目前，Hadoop 使用“dfs”“mapred”“yarn”和“rpc”四个上下文。Hadoop 守护进程通常在多个上下文中收集度量。例如，datanode 会分别为“dfs”和“rpc”上下文收集度量。

度量和计数器的差别在哪里？

主要区别是应用范围不同：度量由 Hadoop 守护进程收集，而计数器(参见 9.1 节对计数器的讨论)先针对 MapReduce 任务进行采集，再针对整个作业进行汇

总。此外，用户群也不同，从广义上讲，度量为管理员服务，而计数器主要为 MapReduce 用户服务。

二者的数据采集和聚集过程也不相同。计数器是 MapReduce 的特性，MapReduce 系统确保计数器值由任务 JVM 产生，再传回 application master，最终传回运行 MapReduce 作业的客户端。(计数器是通过 RPC 的心跳[heartbeat]传播的，详情可以参见 7.1.5 节。)在整个过程中，任务进程和 application master 都会执行汇总操作。

度量的收集机制独立于接收更新的组件。有多种输出度量的方式，包括本地文件、Ganglia 和 JMX。守护进程收集度量，并在输出之前执行汇总操作。

所有的 Hadoop 度量都发布给 JMX(Java management Extensions)，可以使用标准的 JMX 工具，如 JConsole(JDK 自带)，来查看这些度量。对于远程监控，必须将 JMX 系统属性 `com.sun.management.jmxremote.port`(及其他一些用于安全的属性)设置为允许访问。为了在 namenode 实现这些，需要在 *hadoop-env.sh* 文件中设置以下语句：

```
HADOOP_NAMENODE_OPTS="-Dcom.sun.management.jmxremote.port=8004"
```

也可以通过特定 Hadoop 守护进程的*/jmx* 网页查看该守护进程收集的 JMX 度量(JSON 格式)，这为调试带来了便利。例如，可以在网页 *`http://namenode-host:50070/jmx`* 查看 namenode 的度量。

Hadoop 自带大量的度量通道用于向外部系统发布度量，例如本地文件或 Ganglia 监控系统。通道在 *hadoop-metrics2.properties* 文件中配置，可以参考该文件，了解如何进行配置设置。

11.3 维护

11.3.1 日常管理过程

1. 元数据备份

如果 namenode 的永久性元数据丢失或损坏，则整个文件系统无法使用。因此，元数据备份非常关键。可以在系统中分别保存若干份不同时间的备份(例如，1 小时前、1 天前、1 周前或 1 个月前)，以保护元数据。方法一是直接保存这些元数据

文件的复本；方法二是整合到 namenode 上正在使用的文件中。

最直接的元数据备份方法是使用 `dfsadmin` 命令下载 `namenode` 最新的 *fsimage* 文件的复本：

```
% hdfs dfsadmin -fetchImage fsimage.backup
```

可以写一个脚本从准备存储 *fsimage* 存档文件的异地站点运行该命令。该脚本还需测试复本的完整性。测试方法很简单，只要启动一个本地 namenode 守护进程，查看它是否能够将 *fsimage* 和 *edits* 文件载入内存(例如，通过扫描 namenode 日志以获得操作成功信息)。[①]

2. 数据备份

尽管 HDFS 已经充分考虑了如何可靠地存储数据，但是正如任何存储系统一样，仍旧无法避免数据丢失。因此，备份机制就很关键。Hadoop 中存储着海量数据，判断哪些数据需要备份以及在哪里备份就极具挑战性。关键在于为数据划分不同优先级。那些无法重新生成的数据的优先级最高，这些数据对业务非常关键。同理，可再生数据和一次性数据商业价值有限，所以优先级最低，无需备份。

不要误以为 HDFS 的复本技术足以胜任数据备份任务。HDFS 的程序纰漏、硬件故障都可能导致复本丢失。尽管 Hadoop 的设计方案可确保硬件故障极不可能导致数据丢失，但是这种可能性无法完全排除，特别是软件 bug 和人工误操作情况在所难免。

再比较 HDFS 的备份技术和 RAID。RAID 可以确保在某一个 RAID 盘片发生故障时数据不受损坏。但是，如果发生 RAID 控制器故障、软件纰漏(可能重写部分数据)或整个磁盘阵列故障，数据肯定会丢失。

通常情况下，HDFS 的用户目录还会附加若干策略，例如目录容量限制和夜间备份等。用户需要熟悉相关策略，才可以预料执行结果。

distcp 是一个理想的备份工具，其并行的文件复制功能可以将备份文件存储到其他 HDFS 集群(最好软件版本不同，以防 Hadoop 软件纰漏而丢失数据)或其他 Hadoop 文件系统(例如 S3)。此外，还可以用 3.4 节提到的方法将数据从 HDFS 导出到完全不同的存储系统中。

① Hadoop 自带的 Offline Image Viewer 和 Offline Edits Viewer 工具能检测 *fsimage* 和 *edits* 文件的一致性。这两种工具均支持旧版文件。因此，用户可以用这些工具来诊断 Hadoop 的早期发布版本中存在的问题。可以键入 `hdfs oiv` 和 `hdfs oev` 来调用这些工具。

HDFS 允许管理者和用户对文件系统进行快照。快照是对文件系统子树在给定时刻的一个只读复本。由于并不真正复制数据，因此快照非常高效，它们简单地记录每个文件的元数据和块列表，这对于重构快照时刻的文件系统内容已经足够了。

快照不是数据备份的替代品，但是对于恢复用户误删文件在特定时间点的数据而言，它们是一个有用的工具。可以制定一个周期性快照的策略，根据年份将快照保存一段特定的时间。例如，可以对前一天的数据进行每小时一次的快照，对前一个月的数据进行每天一次的快照。

3. 文件系统检查(fsck)

建议定期地在整个文件系统上运行 HDFS 的 *fsck*(文件系统检查)工具(例如，每天执行)，主动查找丢失的或损坏的块。参见 11.1.4 节对文件系统检查的详细介绍。

4. 文件系统均衡器

定期运行均衡器工具(参见 11.1.4 节对均衡器的详细介绍)，保持文件系统的各个 datanode 比较均衡。

11.3.2 委任和解除节点

Hadoop 集群的管理员经常需要向集群中添加节点，或从集群中移除节点。例如，为了扩大存储容量，需要委任节点。相反的，如果想要缩小集群规模，则需解除节点。如果某些节点表现反常，例如故障率过高或性能过于低下，则需要解除该节点。

通常情况下，节点同时运行 datanode 和节点管理器，因而两者一般同时被委任或解除。

1. 委任新节点

委任一个新节点非常简单。首先，配置 *hdfs-site.xml* 文件，指向 namenode；其次，配置 *yarn-site.xml* 文件，指向资源管理器；最后，启动 datanode 和资源管理器守护进程。然而，预先指定一些经过审核的节点以从中挑选新节点仍不失为一种好的方法。

随便允许一台机器以 datanode 身份连接到 namenode 是不安全的，因为该机器很可

能会访问未授权的数据。此外，这种机器并非真正的 datanode，不在集群的控制之下，随时可能停止，导致潜在的数据丢失。(想象一下，如果有多台这类机器连接到集群，而且某一个块的全部复本恰巧只存储在这类机器上，安全性如何？)由于错误配置的可能性，即使这些机器都在本机构的防火墙之内，这种做法的风险也很高。因此所有工作集群上的 datanode(以及节点管理器)都应该被明确管理。

允许连接到 namenode 的所有 datanode 放在一个文件中，文件名称由 `dfs.hosts` 属性指定。该文件放在 namenode 的本地文件系统中，每行对应一个 datanode 的网络地址(由 datanode 报告——可以通过 namenode 的网页查看)。如果需要为一个 datanode 指定多个网络地址，可将多个网络地址放在一行，由空格隔开。

类似的，可能连接到资源管理器的各个节点管理器也在同一个文件中指定(该文件的名称由 `yarn.resourcemanager.nodes.include-path` 属性指定。在通常情况下，由于集群中的节点同时运行 datanode 和节点管理器守护进程，`dfs.hosts` 和 `yarn.resourcemanager.nodes.include-path` 会同时指向一个文件，即 *include* 文件。

`dfs.hosts` 属性和 `yarn.resourcemanager.nodes.include-path` 属性指定的(一个或多个)文件不同于 *slaves* 文件。前者供 namenode 和资源管理器使用，用于决定可以连接哪些工作节点。Hadoop 控制脚本使用 *slaves* 文件执行面向整个集群范围的操作，例如重启集群等。Hadoop 守护进程从不使用 *slaves* 文件。

向集群添加新节点的步骤如下。

(1) 将新节点的网络地址添加到 *include* 文件中。

(2) 运行以下指令，将审核过的一系列 datanode 集合更新至 namenode 信息：

```
% hdfs dfsadmin -refreshNodes
```

(3) 运行以下指令，将审核过的一系列节点管理器信息更新至资源管理器：

```
% yarn rmadmin -refreshNodes
```

(4) 以新节点更新 *slaves* 文件。这样的话，Hadoop 控制脚本会将新节点包括在未来操作之中。

(5) 启动新的 datanode 和节点管理器。

(6) 检查新的 datanode 和节点管理器是否都出现在网页界面中。

HDFS 不会自动将块从旧的 datanode 移到新的 datanode 以平衡集群。用户需要自行运行均衡器，详情参考 11.1.4 节对均衡器的讨论。

2. 解除旧节点

HDFS 能够容忍 datanode 故障，但这并不意味着允许随意终止 datanode。以三复本策略为例，如果同时关闭不同机架上的三个 datanode，则数据丢失的概率会非常高。正确的方法是，用户将拟退出的若干 datanode 告知 namenode，Hadoop 系统就可在这些 datanode 停机之前将块复制到其他 datanode。

有了节点管理器的支持，Hadoop 对故障的容忍度更高。如果关闭一个正在运行 MapReduce 任务的节点管理器，application master 会检测到故障，并在其他节点上重新调度任务。

解除节点的过程由 *exclude* 文件控制。对于 HDFS 来说，文件由 `dfs.hosts.exclude` 属性设置；对于 YARN 来说，文件由 `yarn.resourcemanager.nodes.exclude-path` 属性设置。这些文件列出若干未被允许连接到集群的节点。通常，这两个属性指向同一个文件。

判断一个节点管理器能否连接到资源管理器非常简单。仅当节点管理器出现在 *include* 文件且不出现在 *exclude* 文件中时，才能够连接到资源管理器。注意，如果未指定 *include* 文件，或 *include* 文件为空，则意味着所有节点都包含在 *include* 文件中。

HDFS 的规则稍有不同。如果一个 datanode 同时出现在 *include* 和 *exclude* 文件中，则该节点可以连接，但是很快会被解除委任。表 11-3 总结了 datanode 的不同组合方式。与节点管理器类似，如果未指定 *include* 文件或 *include* 文件为空，都意味着包含所有节点。

表 11-3. HDFS 的 include 文件和 exclude 文件

节点是否出现在 *include* 文件中	节点是否出现在 *exclude* 文件中	解释
否	否	节点无法连接
否	是	节点无法连接
是	否	节点可连接
是	是	节点可连接，将被解除

从集群中移除节点的步骤如下。

(1) 将待解除节点的网络地址添加到 *exclude* 文件中，不更新 *include* 文件。

(2) 执行以下指令，使用一组新的审核过的 datanode 来更新 namenode 设置：

```
% hdfs dfsadmin -refreshNodes
```

(3) 使用一组新的审核过的节点管理器来更新资源管理器设置：

```
% yarn rmadmin -refreshNodes
```

(4) 转到网页界面，查看待解除 datanode 的管理状态是否已经变为“正在解除”(Decommission In Progress)，因为此时相关的 datanode 正在被解除过程之中。这些 datanode 会把它们的块复制到其他 datanode 中。

(5) 当所有 datanode 的状态变为“解除完毕”(Decommissioned)时，表明所有块都已经复制完毕。关闭已经解除的节点。

(6) 从 *include* 文件中移除这些节点，并运行以下命令：

```
% hdfs dfsadmin -refreshNodes
% yarn rmadmin -refreshNodes
```

(7) 从 *slaves* 文件中移除节点。

11.3.3 升级

升级 Hadoop 集群需要细致的规划，特别是 HDFS 的升级。如果文件系统的布局的版本发生变化，升级操作会自动将文件系统数据和元数据迁移到兼容新版本的格式。与其他涉及数据迁移的过程相似，升级操作暗藏数据丢失的风险，因此需要确保数据和元数据都已经备份完毕。参见 11.3.1 节对日常管理过程的讨论。

规划过程最好包括在一个小型测试集群上的测试过程，以评估是否能够承担(可能的)数据丢失的损失。测试过程使用户更加熟悉升级过程、了解如何配置本集群和工具集，从而为在产品集群上进行升级工作消除技术障碍。此外，一个测试集群也有助于测试客户端的升级过程。用户可以阅读以下补充内容中对客户端兼容性的讨论。

兼容性

将 Hadoop 版本升级成另外一个版本时，需要仔细考虑需要升级步骤。同时还要考虑几个方面：API 兼容性、数据兼容性和连接兼容性。

API 兼容性重点考虑用户代码和发行的 Hadoop API 之间的对比，例如 Java MapReduce API。主发行版本(例如从 1.x.y 到 2.0.0)是允许破坏 API 兼容性的，因此，用户的程序要修改并重新编译。次重点发行版本(例如从到 1.0.x 到 1.1.0)和单点发行版本(例如从 1.0.1 到 1.0.2)不应该破坏兼容性。

Hadoop 针对 API 函数使用分类模式来表征其稳定性。按照先前的命名规则，API 兼容性包括标记为 `InterfaceStability.Stable`。公开发行的 Hadoop API 中包含有部分函数，标记为 `InterfaceStability.Evolving` 或者 `InterfaceStability.Unstable`(上述标注包含在 `org.apache.hadoop.classification` 软件包中)，这意味允许它们分别在次重点发行版本和单点发行版本中破坏兼容性。

数据兼容性主要考虑持久数据和元数据的格式，例如在 HDFS namenode 中用于存储持久数据的格式。这些格式允许在主版本和次重点版本之间修改，但是这类修改对用户透明，因为系统升级时数据会自动迁移。系统升级路径有一些限制，这些限制包含在发行须知中。例如，在系统升级过程中可能需要通过某个中间发行版本依次升级，而非一步直接升级到最新版本。

连接兼容性主要考虑通过利用 RPC 和 HTTP 这样的连接协议来实现客户端和服务器之间的互操作性。连接兼容性的规则是，客户端与服务器必须有相同的主版本号，但次版本号或单点发行版本号可以不同(例如，客户端 2.0.2 版可以和服务器 2.0.1 版或 2.1.0 版一起工作，但是与服务器 3.0.0 版不能一起工作)。

这条连接兼容性规则和 Hadoop 早期版本中要求的不同。早期版本中，内部客户端(例如 datanode)必须和服务端一起加锁升级。目前这种客户端和服务端的版本可以不同的事实使得 Hadoop 2 能够支持滚动升级。

在 *http://bit.ly/hadoop_compatibility* 可以查阅到 Hadoop 遵从的全部兼容性规则。

如果文件系统的布局并未改变，升级集群就非常容易：在集群上安装新版本的 Hadoop(客户端也同步安装)，关闭旧的守护进程，升级配置文件，启动新的守护

进程，令客户端使用新的库。整个过程是可逆的，换言之，也可以方便地还原到旧版本。

成功升级版本之后，还需要执行两个清理步骤。

(1) 从集群中移除旧的安装和配置文件。

(2) 在代码和配置文件中针对“被弃用”(deprecation)警告信息进行修复。

升级功能是 Hadoop 集群管理工具如 Cloudera Manager 和 Apache Ambari 的一个亮点。它们简化了升级过程，且使得滚动升级变得容易。节点以批量方式升级(或对于主节点，一次升级一个)，这样客户端不会感受到服务中断。

HDFS 的数据和元数据升级

如果采用前述方法来升级 HDFS，且新旧 HDFS 的文件系统布局恰巧不同，则 namenode 无法正常工作，在其日志文件中产生如下信息：

```
File system image contains an old layout version -16.
An upgrade to version -18 is required.
Please restart NameNode with -upgrade option.
```

最可靠的判定文件系统升级是否必要的方法是在一个测试集群做实验。

升级 HDFS 会保留前一版本的元数据和数据的复本，但这并不意味着需要两倍的存储开销，因为 datanode 使用硬链接保存指向同一块的两个应用(分别为当前版本和前一版本)，从而能够在需要时方便地回滚到前一版本。需要强调的是，系统回滚到旧版本之后，原先的升级改动都将被取消。

用户可以保留前一个版本的文件系统，但无法回滚多个版本。为了执行 HDFS 数据和元数据上的另一次升级任务，需要删除前一版本，该过程被称为“定妥升级”(finalizing the upgrade)。一旦执行该操作，就无法再回滚到前一个版本。

一般来说，升级过程可以忽略中间版本。但在某些情况下还是需要先升级到中间版本，这种情况会在发布说明文件中明确指出。

仅当文件系统健康时，才可升级，因此有必要在升级之前调用 *fsck* 工具全面检查文件系统的状态(参见 11.1.4 节对 *fsck* 工具的讨论)。此外，最好保留 *fsck* 的输出报告，该报告列举了所有文件和块信息；在升级之后，再次运行 *fsck* 新建一份输出报告并比较两份报告的内容。

在升级之前最好清空临时文件，包括 HDFS 的 MapReduce 系统目录和本地的临时文件等。

综上所述，如果升级集群会导致文件系统的布局变化，则需要采用下述步骤进行升级。

(1) 在执行升级任务之前，确保前一升级已经定妥。

(2) 关闭 YARN 和 MapReduce 守护进程。

(3) 关闭 HDFS，并备份 namenode 目录。

(4) 在集群和客户端安装新版本的 Hadoop。

(5) 使用 `-upgrade` 选项启动 HDFS。

(6) 等待，直到升级完成。

(7) 检验 HDFS 是否运行正常。

(8) 启动 YARN 和 MapReduce 守护进程。

(9) 回滚或定妥升级任务(可选的)。

运行升级任务时，最好移除 PATH 环境变量下的 Hadoop 脚本，这样的话，用户就不会混淆针对不同版本的脚本。通常可以为新的安装目录定义两个环境变量。在后续指令中，我们定义了 OLD_HADOOP_HOME 和 NEW_HADOOP_ HOME 两个环境变量。

启动升级 为了执行升级，可运行以下命令(即前述的步骤 5)：

```
% $NEW_HADOOP_HOME/bin/start-dfs.sh -upgrade
```

该命令的结果是让 namenode 升级元数据，将前一版本放在 `dfs.namenode.name.dir` 下的名为 *previous* 的新目录中。类似地，datanode 升级存储目录，保留原先的复本，将其存放在 *previous* 目录中。

等待，直到升级完成 升级过程并非一蹴即就，可以用 *dfsadmin* 查看升级进度，升级事件同时也出现在守护进程的日志文件中，步骤(6)：

```
% $NEW_HADOOP_HOME/bin/hdfs dfsadmin -upgradeProgress status
Upgrade for version -18 has been completed.
Upgrade is not finalized.
```

查验升级情况 显示升级完毕。在本阶段中，用户可以检查文件系统的状态，例

如使用 *fsck*(一个基本的文件操作)检验文件和块，参见步骤(7)。检验系统状态时，最好让 HDFS 进入安全模式(所有数据只读)，以防止其他用户修改数据。详见 11.1.2 节安全模式的有关内容。

回滚升级(可选的) 如果新版本无法正确工作，可以回滚到前一版本，参见步骤(9)，前提是尚未定妥更新。

回滚操作会将文件系统的状态转回到升级之前的状态，同期所做的任何改变都会丢失。换句话说，将回滚到文件系统的前一状态，而非将当前的文件系统降级到前一版本。

首先，关闭新的守护进程：

```
% $NEW_HADOOP_HOME/bin/stop-dfs.sh
```

其次，使用`-rollback`选项启动旧版本的 HDFS：

```
% $OLD_HADOOP_HOME/bin/start-dfs.sh -rollback
```

该命令会让 namenode 和 datanode 使用升级前的复本替换当前的存储目录。文件系统返回之前的状态。

定妥升级(可选) 如果用户满意于新版本的 HDFS，可以定妥升级，参见步骤(9)，以移除升级前的存储目录。

一旦升级定妥，就再也无法回滚到前一版本。

在执行新的升级任务之前，必须执行这一步：

```
% $NEW_HADOOP_HOME/bin/hdfs dfsadmin -finalizeUpgrade
% $NEW_HADOOP_HOME/bin/hdfs dfsadmin -upgradeProgress status
There are no upgrades in progress.
```

现在，HDFS 已经完全升级到新版本了。

第Ⅳ部分

Hadoop 相关开源项目

第 12 章　关于 Avro

第 13 章　关于 Parquet

第 14 章　关于 Flume

第 15 章　关于 Sqoop

第 16 章　关于 Pig

第 17 章　关于 Hive

第 18 章　关于 Crunch

第 19 章　关于 Spark

第 20 章　关于 HBase

第 21 章　关于 ZooKeeper

第 12 章

关于 Avro

Apache Avro(*http://avro.apache.org/*)[1]是一个独立于编程语言的数据序列化系统。该项目由 Doug Cutting(Hadoop 之父)创建，旨在解决 Hadoop 中 Writable 类型的不足：缺乏语言的可移植性。拥有一个可被多种语言(当前是 C、C++、C#、Java、PHP、Python 和 Ruby)处理的数据格式与绑定到单一语言的数据格式相比，前者更易于与公众共享数据集。Avro 同时也更具生命力，该语言将使得数据具有更长的生命周期，即使原先用于读/写该数据的语言已经不再使用。

但为什么要有一个新的数据序列化系统？与 Apache Thrift 和 Google 的 Protocol Buffers 相比，Avro 有其独有的特性。[2]与前述系统及其他系统相似，Avro 数据是用语言无关的模式定义的。但与其他系统不同的是，在 Avro 中，代码生成是可选的，这意味着你可以对遵循指定模式的数据进行读/写操作，即使在此之前代码从来没有见过这个特殊的数据模式。为此，Avro 假设数据模式总是存在的(在读/写数据时)，它形成的是非常精简的编码，因为编码后的数值不需要用字段标识符来打标签。

Avro 模式通常用 JSON 来写，数据通常采用二进制格式编码，但也有其他选择。还有一种高级语言称为 Avro IDL，可以使用开发人员更为熟悉的类 C 语言来写模

① 得名于 20 世纪英国一家飞机制造商。

② 基准测试(*http://code.google.com/p/thrift-protobuf-compare/*)表明，和其他序列化类库相比，Avro 的性能更好。

式。还有一个基于 JSON 的数据编码方式(对构建原型和调试 Avro 数据很有用，因为它是我们人类可读的)。

Avro 规范(*http://avro.apache.org/docs/current/spec.html*)对所有实现都必须支持的二进制格式进行了精确定义，同时还指定这些实现需要支持的其他 Avro 特性。但是，该规范并没有为 API 制定规范：实现可以根据自己的需求操作 Avro 数据并给出相应的 API，因为每个 API 都与语言相关。重要的二进制格式只有一种这一事实意味着绑定新的编程语言的门槛比较低，可以避免语言和格式组合爆炸问题，否则将对互操作性造成一定的问题。

Avro 有丰富的**模式解析**(schema resolution)能力。在精心定义的约束条件下，读数据所用的模式不必与写数据所用的模式相同。由此，Avro 是支持模式演化的。例如，如果有一个新的、可选择的字段要加入记录中，那么只需在用来读取老数据的模式中声明它即可。新客户端和以前的客户端非常相似，均能读取按旧模式存储的数据，同时新的客户端可以使用新字段写入新的内容。相反，如果老客户端读取新客户端写入的数据，会忽略新加入的字段并按照先前的数据模式处理。

Avro 为一系列的对象指定了一个对象容器格式，类似于 Hadoop 的顺序文件。Avro 数据文件包含元数据项(模式数据存储在其中)，使此文件可以自我声明。Avro 数据文件支持压缩，并且是可切分的，这对 MapReduce 的输入格式至关重要。对 Avro 的支持远不止于 MapReduce，事实上本书中所有的数据处理框架(Pig、Hive、Crunch、Spark)都能读/写 Avro 数据文件。

Avro 还可用于 RPC，但在这里不做详细说明。详情参见规范文档。

12.1 Avro 数据类型和模式

Avro 定义了少量的基本数据类型，通过编写模式的方式，它们可被用于构建应用特定的数据结构。考虑到互操作性，实现必须支持所有的 Avro 类型。

表 12-1 列举了 Avro 的基本类型。每个基本类型也可以使用 `type` 属性来指定，其结果是形式更加冗长，示例如下：

```
{ "type": "null" }
```

每个 Avro 语言的 API 都包含该语言特定的 Avro 类型表示。例如，Avro 的 double 类型可以用 C、C++和 Java 语言的 double 类型，Python 的 float 类型以及 Ruby 的 Float 类型来表示。

表 12-1. Avro 基本类型

类型	描述	模式示例
null	空值	"null"
boolean	二进制值	"boolean"
int	32 位带符号整数	"int"
long	64 位带符号整数	"long"
float	单精度(32 位)IEEE 754 浮点数	"float"
double	双精度(64 位)IEEE 754 浮点数	"double"
bytes	8 位无符号字节序列	"bytes"
string	Unicode 字符序列	"string"

表 12-2 列举了 Avro 的复杂类型，并为每种类型给出相应的模式示例。

表 12-2. Avro 复杂类型

类型	描述	模式示例
array	一个排过序的对象集合。特定数组中的所有对象必须模式相同	{ "type": "array", "items": "long" }
map	未排过序的键-值对。键必须是字符串，值可以是任何一种类型，但一个特定 map 中所有值必须模式相同	{ "type": "map", "values": "string" }
record	一个任意类型的命名字段集合	{ "type": "record", "name": "WeatherRecord", "doc": "A weather reading.", "fields": [{"name": "year", "type": "int"}, {"name": "temperature", "type": int"}, {"name": "stationId", "type": string"}] }

续表

类型	描述	模式示例

enum	一个命名的值集合	{
		"type": "enum",
		"name": "Cutlery",
		"doc": "An eating utensil.",
		"symbols": ["KNIFE", "FORK", "SPOON"]
		}

fixed	一组固定数量的 8 位无符号字节	{
		"type": "fixed",
		"name": "Md5Hash",
		"size": 16
		}

union	模式的并集。并集可用 JSON 数组表示，其中每个元素为一个模式。并集表示的数据必须与其内的某个模式相匹配	["null", "string", {"type": "map", "values": "string"}]

而且，一种语言可能有多种表示或映射。所有的语言都支持动态映射，即使运行前并不知道具体模式，也可以使用动态映射。对此，Java 称为**通用映射**(generic mapping)。

另外，Java 和 C++实现可以自动生成代码来表示符合某种 Avro 模式的数据。如果在读/写数据之前就有模式备份的话，代码生成(code generation)能优化数据处理，这在 Java 中被称为**特殊映射**(specific mapping)。同时，为用户代码生成的类与为通用代码生成的类相比，前者提供的 API 更贴近问题域。

Java 还有第三类映射，即**自反映射**(reflect mapping，将 Avro 类型映射到已有的 Java 类型)。它的速度比通用映射和特殊映射都慢，但不失为定义一个类型的方便途径，原因在于 Avro 能够自动推断模式。

表 12-3 列举了 Java 的类型映射。如表中所示，除非有特别说明，否则特殊映射和通用映射相同。类似地，除非有特别说明，否则自反映射与特殊映射相同。特殊映射与通用映射仅在 record、enum 和 fixed 三个类型上有区别，它们的特殊映射都有自动生成的类，类名由 name 属性和可选的 namesapce 属性决定。

表 12-3. Avro 的 Java 类型映射

Avro 类型	Java 通用映射	Java 特殊映射	Java 自反映射
null	null 类型		
boolean	boolean		
int	int		short 或 int
long	long		
float	float		
double	double		
bytes	java.nio.bytebuffer		字节数组
string	org.apache.avro.util.utf8		java.lang.String
array	org.apache.avro.generic.GenericArray		数组或 java.util.Collection
map	java.util.map		
record	org.apache.avro.generic.genericrecord	生成实现 org.apache.avro.specific.SpecificRecord 类的实现	具有零参数构造函数的任意用户类。继承了所有不传递的实例字段
enum	java.lang.string	生成 Java enum 类型	任意 Java enum 类型
fixed	org.apache.avro.generic.genericfixed	生成实现 org.apache.avro.specific.SpecificFixed 的类	org.apache.avro.generic.genericFixed
union	java.lang.Object		

Avro string 类型既可以通过 Java String 类型来表示，也可以通过 Avro Utf8 Java 类型来表示。选择使用 Utf8 的原因在于其高效率：因为 Avro Utf8 类型是易变的，单个 Utf8 实例可以重用，并可对一系列值进行读/写操作。另外，Java String 在新建对象时就要进行 UTF-8 解码，而 Avro 执行 Utf8 解码较晚一些，在某些情况下，这样做可以提高系统性能。

Utf8 实现了 Java 的 java.lang.CharSequence 接口，该接口可以与 Java 类库实现互操作。在其他情况下可能需要通过调用 toString()方法将 Utf8 实例转化成 String 对象。

在通用映射和特殊映射中，Utf8 都是默认的，但是对于特定的映射也可以使用 String。有两个方法可以达到该目的。第一个方法是将模式中的 avro.java.string 属性设置成 String：

```
{ "type": "string", "avro.java.string": "String" }
```

另外一个方法是，对于一些特定的映射操作，你可以构建基于 String 的 get() 和 set() 方法的类。如果使用 Avro Maven 插件，该功能可以通过将 stringType 属性设置成 String 来实现(在 12.2.1 节中有一个相关的例子)。

最后，请注意，Java 自反映射始终使用 Java 的 String 对象，其主要原因是 Java 的兼容性，而非性能。

12.2 内存中的序列化和反序列化

Avro 为序列化和反序列化提供了一些 API，如果想把 Avro 集成到现有系统(比如已定义帧格式的消息系统)，这些 API 函数就很有用，而对于其他情况，请考虑使用 Avro 的数据文件格式。

让我们写一个 Java 程序来从数据流读/写 Avro 数据。首先以一个简单的 Avro 模式为例，它用于表示以记录形式出现的一对字符串：

```
{
  "type": "record",
  "name": "StingPair",
  "doc": "A pair of strings.",
  "fields": [
    {"name": "left", "type": "string"},
    {"name": "right", "type": "string"}
  ]
}
```

如果此模式存储在类路径下一个名为 *StringPair.avsc* 的文件中(.avsc 是 Avro 模式文件的常用扩展名)，我们可以通过下面的两行代码进行加载：

```
Schema.Parser parser = new Schema.Parser();
Schema schema = parser.parse(getClass().getResourceAsStream("StringPair.avsc"));
```

可以使用以下的通用 API 新建一个 Avro 记录实例：

```
GenericRecord datum = new GenericData.Record(schema);
datum.put("left", "L");
datum.put("right", "R");
```

接下来，我们将记录序列化到输出流中：

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
DatumWriter<GenericRecord> writer =
    new GenericDatumWriter<GenericRecord>(schema);
Encoder encoder = EncoderFactory.get().binaryEncoder(out, null);
writer.write(datum, encoder);
encoder.flush();
out.close();
```

其中有两个重要的对象：DatumWriter 和 Encoder。DatumWriter 对象将数据对象翻译成 Encoder 对象可以理解的类型，然后由后者写入输出流。这里，我们使用了 GenericDatumWriter 对象，它将 GenericRecord 字段的值传递给 Encoder 对象。由于没有重用先前构建的 encoder，此处我们将 null 传递 encoder 工厂。

在本例中，只有一个对象需要写到输出流，但如果需要写若干个对象，可以调用 write()方法，然后再关闭输入流。

我们需要将该模式传递给 GenericDatumWriter 对象，因为它会根据模式来确定将数据对象中的哪些数值写到输出流。在调用 writer 的 write()方法后刷新 encoder，然后关闭输出流。

我们也可以使用反向的处理过程来从字节缓冲区中读回对象：

```
DatumReader<GenericRecord> reader =
    new GenericDatumReader<GenericRecord>(schema);
Decoder decoder = DecoderFactory.get().binaryDecoder(out.toByteArray(),
    null);
GenericRecord result = reader.read(null, decoder);
assertThat(result.get("left").toString(), is("L"));
assertThat(result.get("right").toString(), is("R"));
```

我们需要给 binaryDecoder()和 read()的调用传递空值(null)，因为这里没有重用对象(分别是 decoder 或记录)。

由于 result.get("left")和 result.get("right")返回的对象是 Utf8 类型的，因此我们需要通过调用 toString()方法将它们转型为 Java String 类型。

特定 API

现在，让我们来看看使用特定 API 的等价代码。通过使用 Avro 的 Maven 插件编译模式，我们可以根据模式文件生成 StringPair 类。以下是与 Maven Project Object Model(POM)相关的部分：

```
<project>
 ...
 <build>
  <plugins>
   <plugin>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro-maven-plugin</artifactId>
    <version>${avro.version}</version>
```

```
<executions>
 <execution>
  <id>schemas</id>
  <phase>generate-sources</phase>
  <goals>
   <goal>schema</goal>
  </goals>
  <configuration>
   <includes>
    <include>StringPair.avsc</include>
   </includes>
   <stringType>String</stringType>
   <sourceDirectory>src/main/resources</sourceDirectory>
   <outputDirectory>${project.build.directory}/generated-sources/java
   </outputDirectory>
  </configuration>
 </execution>
</executions>
</plugin>
</plugins>
</build>
...
</project>
```

也可以不使用 Maven，而是使用 Avro 的 Ant 任务(org.apache.avro.specific.SchemaTask)或者 Avro 的命令行工具[1]来为一个模式生成 Java 代码。

在序列化和反序列化的代码中，我们通过构建一个 StringPair 实例来替代 GenericRecord 对象(使用 SpecificDatumWriter 类将该对象写入数据流，并使用 SpecificDatumReader 类读回数据)：

```
StringPair datum = new StringPair();
datum.setLeft("L");
datum.setRight("R");

ByteArrayOutputStream out = new ByteArrayOutputStream();
DatumWriter<StringPair> writer =
    new SpecificDatumWriter<StringPair>(StringPair.class);
Encoder encoder = EncoderFactory.get().binaryEncoder(out, null);
writer.write(datum, encoder);
encoder.flush();
out.close();

DatumReader<StringPair> reader =
    new SpecificDatumReader<StringPair>(StringPair.class);
Decoder decoder = DecoderFactory.get().binaryDecoder(out.toByteArray(),
    null);
```

① 可以下载获得 Avro 的源文件和二进制文件，网址为 *http://avro.apache.org/releases.html*。键入命令 java -jar avro-tools-*.jar，即可获得使用指南。

```
StringPair result = reader.read(null, decoder);
assertThat(result.getLeft(), is("L"));
assertThat(result.getRight(), is("R"));
```

12.3 Avro 数据文件

Avro 的对象容器文件格式主要用于存储 Avro 对象序列。这与 Hadoop 顺序文件的设计非常相似，详见 5.4.1 节，它们之间的最大区别在于 Avro 数据文件主要是面向跨语言使用而设计的，因此，我们可以用 Python 语言写入文件，并用 C 语言来读取文件，下一节将详细探讨。

在数据文件的头部中含有元数据，它包括一个 Avro 模式和一个 *sync marker*(同步标识)，紧接着是一系列包含序列化 Avro 对象的数据块(压缩可选)。数据块通过 sync marker 分隔，而 sync marker 对该文件来说是唯一的(特定文件的标识信息存储在文件头部)，并允许在文件中搜索到任意位置之后通过块边界快速地重新进行同步。因此，Avro 数据文件是可切分的，适合 MapReduce 快速处理。

将 Avro 的对象写到数据文件与写到数据流类似。就像前面一样，我们需要使用 `DatumWriter`，但并没有用到 Encoder，而是通过 `DatumWriter` 来创建一个 `DataFileWriter` 实例。然后便可以新建一个数据文件(该文件一般有`.avro` 扩展名)，并向它附加新写入的对象：

```
File file = new File("data.avro");
DatumWriter<GenericRecord> writer =
    new GenericDatumWriter<GenericRecord>(schema);
DataFileWriter<GenericRecord> dataFileWriter =
    new DataFileWriter<GenericRecord>(writer);
dataFileWriter.create(schema, file);
dataFileWriter.append(datum);
dataFileWriter.close();
```

写入数据文件的对象必须遵循相应的文件模式，否则在调用 `append()`方法时会抛出异常。

这个例子演示了如何将对象写到本地文件(前面代码段中的 `java.io.File`)，但如果使用重载的 `DataFileWriter` 的 `create()`方法，则可以将数据对象写到任何一个 `java.io.OutputStream` 对象中。例如，通过对 `FileSystem` 对象调用 `create()` 方法，可以返回 `OutputStream` 对象，进而将文件写入 HDFS，更多详情可以参见 3.5.3 节。

从数据文件读取对象与前面例子中从内存数据流读取数据类似，只有一个重要的区别：我们不需要指定模式，因为可以从文件的元数据中读取它。事实上，还可以对 DataFileReader 实例调用 getSchema()方法来获取模式，并验证该模式是否和原始写入对象的模式相同：

```
DatumReader<GenericRecord> reader = new GenericDatumReader<GenericRecord>();
DataFileReader<GenericRecord> dataFileReader =
  new DataFileReader<GenericRecord>(file, reader);
assertThat("Schema is the same", schema, is(dataFileReader.getSchema()));
```

DataFileReader 对象是一个常规的 Java 迭代器，由此我们可以调用 hashNext()和 next()方法来迭代数据对象。下面的代码检查是否只有一条记录以及该记录是否有期望的字段值：

```
assertThat(dataFileReader.hasNext(), is(true));
GenericRecord result = dataFileReader.next();
assertThat(result.get("left").toString(), is("L"));
assertThat(result.get("right").toString(), is("R"));
assertThat(dataFileReader.hasNext(), is(false));
```

但是，更适合的做法是使用重载并将返回对象实例作为输入参数(该例中，为 GenericRecord 对象)，而非直接使用 next()方法，因为这样可以重用对象，减少对象分配和垃圾回收所产生的开销，特别是当文件中包含有很多对象时。代码如下所示：

```
GenericRecord record = null;
while (dataFileReader.hasNext()) {
  record = dataFileReader.next(record);
  // process record
}
```

如果对象重用不是那么重要，则可以使用如下更简短的形式：

```
for (GenericRecord record : dataFileReader) {
  // process record
}
```

如果只是普通的从 Hadoop 文件系统中读取文件，可以使用 Avro 的 FsInput 对象来指定使用 Hadoop Path 对象作为输入对象。事实上，DataFileReader 对象提供对 Avro 数据文件的随机访问(通过 seek()和 sync()方法)，不过在大多数情况下，顺序访问数据流就足够了，而此时应当使用 DataFileStream 对象。DataFileStream 对象可以从任意 Java InputStream 对象中读取数据。

12.4 互操作性

为了说明 Avro 的语言互操作性，让我们试着用一种语言(Python)来写入数据文件，并用另一种语言(Java)来读取这个文件。

12.4.1 Python API

范例 12-1 中的程序从标准输入中读取由逗号分隔的字符串并将其以 `StringPair` 记录的方式写入 Avro 数据文件。与写数据文件的 Java 代码类似，我们需要新建一个 `DatumWriter` 对象和一个 `DataFileWriter` 对象，注意，我们在代码中嵌入了 Avro 模式，尽管没有这个模式，我们仍然可以从文件中正确读取数据。

Python 以目录形式表示 Avro 记录，从标准输入中读取的每一行都被转换为 `dict` 对象并附加到 `DataFileWriter` 对象的末尾。

范例 12-1. 这个 Python 程序将 Avro 的成对形式的记录写入一个数据文件

```
import os
import string
import sys

from avro import schema
from avro import io
from avro import datafile

if __name__ == '__main__':
  if len(sys.argv) != 2:
    sys.exit('Usage: %s <data_file>' % sys.argv[0])
  avro_file = sys.argv[1]
  writer = open(avro_file, 'wb')
  datum_writer = io.DatumWriter()
  schema_object = schema.parse("\
{ "type": "record",
  "name": "StringPair",
  "doc": "A pair of strings.",
  "fields": [
    {"name": "left", "type": "string"},
    {"name": "right", "type": "string"}
  ]
}")
  dfw = datafile.DataFileWriter(writer, datum_writer, schema_object)
  for line in sys.stdin.readlines():
    (left, right) = string.split(line.strip(), ',')
    dfw.append({'left':left, 'right':right});
  dfw.close()
```

在运行该程序之前，我们需要为 Python 安装 Avro：

```
% easy_install avro
```

为了运行该程序，我们需要指定文件名(*pairs.avro*，输出结果会写到这个文件)，并通过标准输入来发送输入的成对记录，结束文件输入时键入快捷键 Ctrl-D：

```
% python ch12-avro/src/main/py/write_pairs.py pairs.avro
a,1
c,2
b,3
b,2
^D
```

12.4.2 Avro 工具集

下面我们将使用 Avro 工具(Java)显示 *pairs.avro* 的内容。JAR 工具可以从 Avro 网站上获得，此处假设它已经被存放在本地目录*$AVRO_HOME* 中。使用 `tojson` 命令可以将 Avro 数据文件中的内容转储为 JSON 并打印输出到控制台。

```
% java -jar $AVRO_HOME/avro-tools-*.jar tojson pairs.avro
{"left":"a","right":"1"}
{"left":"c","right":"2"}
{"left":"b","right":"3"}
{"left":"b","right":"2"}
```

这样，我们便成功交换了两个 Avro 实现(Python 和 Java)的复杂数据。

12.5 模式解析

我们可以选择使用不同于写入数据的模式(writer 的模式)来读回数据(reader 的模式)。这非常有用，因为它意味着模式的演化。例如，为了便于说明，我们考虑新增一个 `description` 字段，从而形成一个新的模式：

```
{
  "type": "record",
  "name": "StringPair",
  "doc": "A pair of strings with an added field.",
  "fields": [
    {"name": "left", "type": "string"},
    {"name": "right", "type": "string"},
    {"name": "description", "type": "string", "default": ""}
  ]
}
```

由于已经为 description 字段指定了一个默认值(空字符串[1])以供 Avro 在读取没有定义字段的记录时使用，因此我们可以使用该模式来读取之前已经序列化的数据。如果忽略 default 属性，那么在读取旧数据时会报错。

要想将默认值设为 null 而非空字符串，我们需要使用具有 Avro 的 null 类型的并集来定义 description 字段：

```
{"name": "description", "type": ["null", "string"], "default": null}
```

当读模式不同于写模式时，则需要调用 GenericDatumReader 的构造函数，它以两个模式对象作为输入参数，即 reader 的模式对象和 writer 的模式对象，并按照以下顺序传递：

```
DatumReader<GenericRecord> reader =
    new GenericDatumReader<GenericRecord>(schema, newSchema);
Decoder decoder = DecoderFactory.get().binaryDecoder(out.toByteArray(),
    null);
GenericRecord result = reader.read(null, decoder);
assertThat(result.get("left").toString(), is("L"));
assertThat(result.get("right").toString(), is("R"));
assertThat(result.get("description").toString(), is(""));
```

对于 writer 的模式已经被存储在元数据中的数据文件，我们只需要显式指定其 reader 的模式，具体做法是将 null 作为 writer 的模式传递：

```
DatumReader<GenericRecord> reader =
  new GenericDatumReader<GenericRecord>(null, newSchema);
```

模式不同的另外一种常见用法是去掉记录中的某些字段，这种操作也可以称为投影(projection)。当记录中包含大量的字段，但需要读取的只是其中的一部分时，这种做法非常有用。例如，使用下面这个模式可以仅读取 StringPair 对象中的 right 字段：

```
{
  "type": "record",
  "name": "StringPair",
  "doc": "The right field of a pair of strings.",
  "fields": [
    {"name": "right", "type": "string"}
  ]
}
```

① 使用 JSON 对字段默认值进行编码。参见 Avro 规范中对每个数据类型进行编码描述。

模式解析规则可以直接解决模式从一个版本演化为另一个版本时可能产生的问题。在 Avro 规范中对所有 Avro 类型均有详细说明。表 12-4 从类型读/写(客户端和服务器端)的角度总结了记录演化规则。

表 12-4. 记录的模式解析

新模式	写入	读取	操作
增加的字段	旧	新	通过默认值读取新字段，因为写入时没有该字段
	新	旧	读取时不知道新写入的新字段，所以忽略该字段(投影)
删除的字段	旧	新	读取时忽略已删除的字段(投影)
	新	旧	写入时不写入已删除的字段。如果旧模式对该字段有默认值，那么读取时可以使用该默认值，否则报错。在这种情况下，最好同时更新读取模式或在更新写入模式之前更新读取模式

对于 Avro 模式演化来说，另一种有用的技术是使用**别名**(*aliases*)。别名允许你在读 Avro 数据的模式与写 Avro 数据的模式中使用不同的字段名称。例如，下面这个 reader 的模式能够以新的字段名称(即 `first` 和 `second`)来读取 `StringPair` 数据，而非写入数据时所使用的字段名称(即 `left` 和 `right`)。

```
{
  "type": "record",
  "name": "StringPair",
  "doc": "A pair of strings with aliased field names.",
  "fields": [
    {"name": "first", "type": "string", "aliases": ["left"]},
    {"name": "second", "type": "string", "aliases": ["right"]}
  ]
}
```

注意，别名的主要作用是(在读取的时候)将 writer 的模式转换为 reader 的模式，但是别名对读取程序是不可见的。在上述例子中，读取程序无法再使用字段名称 `left` 和 `right`，因为它们已经被转换为 `first` 和 `second`。

12.6 排列顺序

Avro 定义了对象的排列顺序。大多数 Avro 类型的排列顺序与用户期望符合，例如，数值型按照数值的升序进行排序。而其他有一些类型则没那么巧妙了，例如，枚举通过符号的定义而非符号字符串的值来排序。

除了 record 之外，所有类型均按照 Avro 规范中预先定义的规则来排序，这些规则不能被用户改写。但对于记录，可以通过指定 order 属性来控制排列顺序，它有三个值：ascending(默认值)、descending(降序)或 ignore(如此一来，在排序比较时可以忽略此字段。)

例如，通过将 right 字段的排列顺序设置为 descending，可以使下述模式(*SortedStringPair.avsc*)定义的 StringPair 记录按降序排序。在排序时忽略了 left 字段，但它依旧保留在投影中：

```
{
  "type": "record",
  "name": "StringPair",
  "doc": "A pair of strings, sorted by right field descending.",
  "fields": [
    {"name": "left", "type": "string", "order": "ignore"},
    {"name": "right", "type": "string", "order": "descending"}
  ]
}
```

按照文档中 reader 的模式所指定的顺序，记录中的字段两两进行比较，因此，通过指定一个恰当的 reader 的模式，可以对数据记录使用任意顺序。下面这个模式(*SwitchedStringPair.avsc*)定义的是先按 right 字段，再按 left 字段排序：

```
{
  "type": "record",
  "name": "StringPair",
  "doc": "A pair of strings, sorted by right then left.",
  "fields": [
    {"name": "right", "type": "string"},
    {"name": "left", "type": "string"}
  ]
}
```

Avro 实现了高效的二进制比较。也就是说，Avro 不需要将二进制对象反序列化为对象即可实现比较，因为它可以直接对字节流进行操作。[①]在使用 StringPair 模式的情况下(没有 order 属性)，Avro 按以下方式实现二进制比较。

第一个字段(即 left 字段)使用 UTF-8 编码，由此 Avro 可以根据字母表顺序进行比较。如果它们不同，则判定其顺序，且 Avro 可以在该处停止比较。否则，如果这两个字节顺序是相同的，那么比较第二个字段(即 right 字段)，同样在字节尺度上使用字母表序排列，因为该字段同样也使用 UTF-8 编码。

① 该属性的一个有用结果是，我们可以根据对象或相应的二进制表示(后者在 BinaryData 对象上使用 hashCode()静态方法)算出 Avro 数据的哈希代码，两种情况下结果相同。

注意，这里描述的比较方法在逻辑上与 5.3.3 节所描述的二进制比较器相同。更重要的是 Avro 已经为我们提供了比较器，所以无需重写和维护这部分代码，同时我们也可以通过修改 reader 的模式来简单修改排列顺序。对于 *SortedStringPair.avsc* 或 *SwitchedStringPair.avsc* 模式来说，Avro 所使用的比较方法本质上与刚才所描述的是一致的，只不过要考虑比较哪个字段，考虑使用哪种顺序，是升序还是降序。

在本章稍后部分，我们会将 Avro 的排序逻辑与 MapReduce 联合使用，实现 Avro 数据文件的并行排序。

12.7 关于 Avro MapReduce

为了方便在 Avro 数据上运行 MapReduce 程序，Avro 提供了一些 API 类。我们将使用 `org.apache.avro.mapreduce` 包中的新 MapReduce API 类，你也可以从 `org.apache.avro.mapred` 包中找到(旧风格的)MapReduce API 类。

这次，我们使用 Avro MapReduce API 来重写找出天气数据集中每年最高温度的 MapReduce 程序。我们用下列模式来表征天气记录：

```
{
  "type": "record",
  "name": "WeatherRecord",
  "doc": "A weather reading.",
  "fields": [
    {"name": "year", "type": "int"},
    {"name": "temperature", "type": "int"},
    {"name": "stationId", "type": "string"}
  ]
}
```

范例 12-2 中的程序读取文本输入(文本格式在前面几章出现过)，并输出包含天气记录的 Avro 数据文件。

范例 12-2. 该 MapReduce 找出最高气温，输出的是 Avro 文件

```
public class AvroGenericMaxTemperature extends Configured implements Tool {

  private static final Schema SCHEMA = new Schema.Parser().parse(
      "{" +
      "  \"type\": \"record\"," +
      "  \"name\": \"WeatherRecord\"," +
      "  \"doc\": \"A weather reading.\"," +
      "  \"fields\": [" +
      "    {\"name\": \"year\", \"type\": \"int\"}," +
```

```
    "    {\"name\": \"temperature\", \"type\": \"int\"}," +
    "    {\"name\": \"stationId\", \"type\": \"string\"}" +
    "  ]" +
    "}"
);

public static class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, AvroKey<Integer>,
          AvroValue<GenericRecord>> {
  private NcdcRecordParser parser = new NcdcRecordParser();
  private GenericRecord record = new GenericData.Record(SCHEMA);

  @Override
  protected void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
    parser.parse(value.toString());
    if (parser.isValidTemperature()) {
      record.put("year", parser.getYearInt());
      record.put("temperature", parser.getAirTemperature());
      record.put("stationId", parser.getStationId());
      context.write(new AvroKey<Integer>(parser.getYearInt()),
          new AvroValue<GenericRecord>(record));
    }
  }
}

public static class MaxTemperatureReducer
    extends Reducer<AvroKey<Integer>, AvroValue<GenericRecord>,
          AvroKey<GenericRecord>, NullWritable> {

  @Override
  protected void reduce(AvroKey<Integer> key,
      Iterable<AvroValue<GenericRecord>> values, Context context)
      throws IOException, InterruptedException {
    GenericRecord max = null;
    for (AvroValue<GenericRecord> value : values) {
      GenericRecord record = value.datum();
      if (max == null ||
          (Integer) record.get("temperature") > (Integer)
          max.get("temperature")) {
      max = newWeatherRecord(record);
      }
    }
    context.write(new AvroKey(max), NullWritable.get());
  }
  private GenericRecord newWeatherRecord(GenericRecord value) {
    GenericRecord record = new GenericData.Record(SCHEMA);
    record.put("year", value.get("year"));
    record.put("temperature", value.get("temperature"));
    record.put("stationId", value.get("stationId"));
    return record;
  }
}
```

```
  @Override
  public int run(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.printf("Usage: %s [generic options] <input><output>\n",
          getClass().getSimpleName());
      ToolRunner.printGenericCommandUsage(System.err);
      return -1;
    }

    Job job = new Job(getConf(), "Max temperature");
    job.setJarByClass(getClass());

    job.getConfiguration().setBoolean(
        Job.MAPREDUCE_JOB_USER_CLASSPATH_FIRST, true);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    AvroJob.setMapOutputKeySchema(job, Schema.create(Schema.Type.INT));
    AvroJob.setMapOutputValueSchema(job, SCHEMA);
    AvroJob.setOutputKeySchema(job, SCHEMA);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(AvroKeyOutputFormat.class);

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setReducerClass(MaxTemperatureReducer.class);

    return job.waitForCompletion(true) ? 0 : 1;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new AvroGenericMaxTemperature(), args);
    System.exit(exitCode);
  }
}
```

这个程序使用的是通用 Avro 映射(Generic Avro mapping)。这样就不需要通过生成代码来表征数据记录，从而避免了损失类型安全性(通过字符串值来引用字段名称，例如“temperature”)的代价[①]。为了方便起见，天气记录的模式已加入到代码中(读取 SCHMA 常量)。不过在实际情况下，从驱动器本地文件中读取模式，并通过 Hadoop 作业配置将模式传递给 mapper 和 reducer，可以提高代码的可维护性。(具体技巧可以参见 9.4 节。)

该 API 与常规的 Hadoop MapReduce API 有两个较大不同之处。第一个不同是对 Avro Java 类型封装的使用。针对这个 MapReduce 程序，键是年份(一个整数)，值

① 通过生成类的方式使用指定映射(specific mapping)的例子，请参见示例代码中的 AvroSpecificiMaxTemperature 类。

是天气记录，由 Avro 的 GenericRecord 表征。在 map 输出(以及 reduce 输入)中，键类型被转型为 AvroKey<Integer>，值类型被转型为 AvroValue<GenericRecord>。

MaxTemperatureReducer 针对每个键(年份)所对应的所有记录执行迭代运算，并找到那条有最高温度的记录。有必要对目前找到的最高温度的记录做一个备份，因为该迭代运算为了达到高效的目的需要重用该实例，并且只更新相关字段。

与传统 MapReduce 的第二个差异是，它使用 AvroJob 来配置作业。AvroJob 类非常适用于为输入、map 输出以及最后输出数据指定 Avro 模式。在上述程序中没有设置输入模式，因为我们是从文本文件中读取数据。map 输出的键模式是 Avro int，值模式是天气记录模式。最后输出数据模式是天气记录模式，并且写入 Avro 数据文件中的输出格式是 AvroOutputFormat 格式，值被忽略，该值为 NullWritable。

下面的命令行代码显示了如何在一个小型采样数据集上运行该程序：

```
% export HADOOP_CLASSPATH=avro-examples.jar
% export HADOOP_USER_CLASSPATH_FIRST=true
    # override version of Avro in Hadoop
% hadoop jar avro-examples.jar AvroGenericMaxTemperature \
    input/ncdc/sample.txt output
```

执行完成之时，我们可以使用 Avro 工具 JAR 来查看输出结果，该结果是 JSON 格式的 Avro 数据文件，每行一条记录如下：

```
% java -jar $AVRO_HOME/avro-tools-*.jar tojson output/part-00000.avro
{"year":1949,"temperature":111,"stationId":"012650-99999"}
{"year":1950,"temperature":22,"stationId":"011990-99999"}
```

在上述例子中，我们读取的是一个文本文件，然后创建一个 Avro 数据文件，当然其他组合也是可行的，这有利于将数据格式在 Avro 格式和其他格式之间来回转换，例如 SequenceFile。详情参见 Avro MapReduce 包的说明文档。

12.8 使用 Avro MapReduce 进行排序

在本节中，我们利用 Avro 的排序能力，并结合使用 MapReduce，写一段对 Avro 数据文件进行排序的程序(范例 12-3)。

范例 12-3. 对 Avro 数据文件进行排序的 MapReduce 程序

```
public class AvroSort extends Configured implements Tool {
```

```
  static class SortMapper<K> extends Mapper<AvroKey<K>, NullWritable,
      AvroKey<K>, AvroValue<K>> {
    @Override
    protected void map(AvroKey<K> key, NullWritable value,
      Context context) throws IOException, InterruptedException {
      context.write(key, new AvroValue<K>(key.datum()));
    }
  }

  static class SortReducer<K> extends Reducer<AvroKey<K>, AvroValue<K>,
      AvroKey<K>, NullWritable> {
    @Override
    protected void reduce(AvroKey<K> key, Iterable<AvroValue<K>> values,
        Context context) throws IOException, InterruptedException {
      for (AvroValue<K> value : values) {
        context.write(new AvroKey(value.datum()), NullWritable.get());
      }
    }
  }

  @Override
  public int run(String[] args) throws Exception {

    if (args.length != 3) {
      System.err.printf(
        "Usage: %s [generic options] <input><output><schema-file>\n",
        getClass().getSimpleName());
      ToolRunner.printGenericCommandUsage(System.err);
      return -1;
    }

    String input = args[0];
    String output = args[1];
    String schemaFile = args[2];

    Job job = new Job(getConf(), "Avro sort");
    job.setJarByClass(getClass());

    job.getConfiguration().setBoolean(
        Job.MAPREDUCE_JOB_USER_CLASSPATH_FIRST, true);

    FileInputFormat.addInputPath(job, new Path(input));
    FileOutputFormat.setOutputPath(job, new Path(output));

    AvroJob.setDataModelClass(job, GenericData.class);

    Schema schema = new Schema.Parser().parse(new File(schemaFile));
    AvroJob.setInputKeySchema(job, schema);
    AvroJob.setMapOutputKeySchema(job, schema);
    AvroJob.setMapOutputValueSchema(job, schema);
    AvroJob.setOutputKeySchema(job, schema);

    job.setInputFormatClass(AvroKeyInputFormat.class);
```

```
  job.setOutputFormatClass(AvroKeyOutputFormat.class);

  job.setOutputKeyClass(AvroKey.class);
  job.setOutputValueClass(NullWritable.class);
  job.setMapperClass(SortMapper.class);
  job.setReducerClass(SortReducer.class);

  return job.waitForCompletion(true) ? 0 : 1;
}

public static void main(String[] args) throws Exception {
  int exitCode = ToolRunner.run(new AvroSort(), args);
  System.exit(exitCode);
 }
}
```

这个程序(使用了通用的 Avro 映射，因此无需生成任何代码)能够对由通用类型参数 K 表示的任何 Java 类型的 Avro 记录进行排序。我们选择值的类型与键的类型相同，以便在值按照键分组后，可以对有多值对应于一个键的情况(根据排序函数)输出同一个键对应的所有值，不丢失任何记录。[①]Mapper 输出了封装在 AvroKey 和 AvroValue 中的输入键。Reducer 作为一个识别器，将值作为输出键传递，并写入 Avro 数据文件。

排序发生在 MapReduce 的混洗过程中，排序函数由传入程序中的 Avro 模式确定。下面我们使用该程序对先前创建的 *pairs.avro* 文件进行排序，根据 *SortedStringPair.avsc* 模式，我们将按 right 字段的降序进行排序。首先，使用 Avro 工具 JAR 检查输入数据：

```
% java -jar $AVRO_HOME/avro-tools-*.jar tojson input/avro/pairs.avro
{"left":"a","right":"1"}
{"left":"c","right":"2"}
{"left":"b","right":"3"}
{"left":"b","right":"2"}
```

然后运行排序程序：

```
% hadoop jar avro-examples.jar AvroSort input/avro/pairs.avro output \
  ch12-avro/src/main/resources/SortedStringPair.avsc
```

最后，检查输出并查看是否正确排序。

```
% java -jar $AVRO_HOME/avro-tools-*.jar tojson output/part-r-00000.avro
{"left":"b","right":"3"}
{"left":"b","right":"2"}
```

① 如果此处我们使用 identity mapper 和 reducer，那么程序会在排序的同时删除重复的键。9.2.4 节遇到过在同一键情况下复制值对象来实现信息复制的想法。

```
{"left":"c","right":"2"}
{"left":"a","right":"1"}
```

12.9 其他语言的 Avro

除了 Java 语言之外，还有其他语言和框架也可以使用 Avro 数据。

`AvroAsTextInputFormat` 被设计用来允许 Hadoop Streaming 程序读取 Avro 数据文件。文件中的每条数据均被转化为一个字符串，通过 JSON 格式或者原始字节（如果是 `Avrobytes` 类型的话）来表示。另一方面，你可以指定 `AvroTextOutputFormat` 作为 Streaming 作业的输出格式，并按照 `bytes` 模式创建 Avro 数据文件，其中每条记录是从 Streaming 输出的、由制表符分隔的键-值对。这两个类均可以在 `org.apache.avro.mapred` 包中找到。

运用 Pig、Hive、Crunch 和 Spark 等框架来处理 Avro 数据文件也值得考虑，因为它们都可以通过指定适合的数据存储格式来读/写 Avro 数据文件。详情可以参见本书中的相关章节。

第 13 章

关于 Parquet

Apache Parquet (*http://parquet.incubator.apache.org/*)是一种能够有效存储嵌套数据的列式存储格式。

由于列式存储格式在文件大小和查询性能上表现优秀，因而受到人们的青睐。在列式存储格式下，同一列的数据连续保存。一般来说，这种做法可以允许更高效的编码方式，从而使列式存储格式的文件常常比行式存储格式的同等文件占用更少空间。例如，对于存储时间戳的列，采用的编码方式可以是存储第一个时间戳的值，尔后的值则只需要存储与前一个值之间的差，根据时间局部性原理(即同一时间前后的记录彼此相邻)，这种编码方式更倾向于占用较小的空间。另外，由于查询引擎能够跳过对本次查询无用的行，从而提高了查询的性能(参见图 5-4)。本章对 Parquet 进行了深入考察，不过需要注意的是，在 Hadoop 生态系统中还有其他一些列式存储格式，比如 Hive 项目著名的 ORCFile (Optimized Record Columnar File)。

Parquet 的突出贡献在于能够以真正的列式存储格式来保存具有深度嵌套结构的数据。在现实世界中，具有多级嵌套模式的系统比较普遍，所以这种能力非常重要。Parquet 脱胎于 Google 发表的一篇关于 Dremel 的论文[①]，它通过一种新颖的技术，以扁平的列式存储格式和很小的额外开销来存储嵌套的结构。有了这种技术，即使是嵌套的字段在读取时也不需要牵扯到其他字段，从而带来了性能上的极大提升。

① Sergey Melnik 等人在 2010 年第 36 届 VLDB 国际会议论文集中发表的论文，标题为“Dremel: Interactive Analysis of Web-Scale Datasets”，网址为 *http://research.google.com/pubs/pub36632.html*。

Parquet 的另一个特点是有很多工具都可支持这种格式。作为 Parquet 的缔造者，Twitter 和 Cloudera 的工程师们希望在尝试使用新工具来处理现有数据时能够更加简化。为了达成这一目标，他们将该项目划分为两个部分，其一是以语言无关的方式来定义文件格式的 Parquet 规范(即 parquet-format)，另一部分是不同语言(Java 和 C++)的规范实现，以便人们能够使用多种工具读/写 Parquet 文件。事实上，本书所涉及的大部分数据处理组件都支持 Parquet 格式(包括 MapReduce、Pig、Hive、Cascading、Crunch 和 Spark)。这种灵活性同样也延伸至内存中的表示法：Java 的实现并没有绑定某一种表示法，因而可以使用 Avro、Thrift 或 Protocol Buffers 等多种内存数据表示法来将数据写入 Parquet 文件或者从 Parquet 文件中读取数据。

13.1 数据模型

Parquet 定义了少数几个原子数据类型，如表 13-1 所示。

表 13-1. Parquet 的原子类型

类型	描述
boolean	二进制值
int32	32 位有符号整数
int64	64 位有符号整数
int96	96 位有符号整数
float	(32 位) IEEE 754 单精度浮点数
double	(64 位) IEEE 754 双精度浮点数
binary	8 位无符号字节序列
fixed_len_byte_array	固定数量的 8 位无符号字节

保存在 Parquet 文件中的数据通过模式进行描述，模式的根为 message，message 中包含一组字段，每个字段由一个重复数(required、optional 或 repeated)、一个数据类型和一个字段名称构成。下面是一个简单的气象记录的 Parquet 模式：

```
message WeatherRecord {
  required int32 year;
  required int32 temperature;
  required binary stationId (UTF8);
}
```

请注意，Parquet 的原子类型并不包括字符串类型。事实上，Parquet 定义了一些逻辑类型，这些逻辑类型指出应当如何对原子类型进行解读，从而使得序列化的表

示(即原子类型)与特定于应用的语义(即逻辑类型)相互独立。可以通过 UTF8 注解的 binary 原子类型表示字符串类型。表 13-2 列出了 Parquet 定义的一些逻辑类型，且每种逻辑类型都有一个具有代表性的模式范例。表中没有列出的类型包括有符号整数、无符号整数、其他一些日期或时间类型以及 JSON 和 BSON 文档类型。有关详情可以参见 Parquet 规范。

表 13-2. Parquet 的逻辑类型

逻辑类型注解	描述	模式示例

UTF8	由 UTF-8 字符组成的字符串，可用于注解 binary	`message m {`
		`  required binary a (UTF8);`
		`}`

ENUM	命名值的集合，可用于注解 binary	`message m {`
		`  required binary a (ENUM);`
		`}`

DECIMAL (precision,scale)	任意精度的有符号小数，可用于注解 int32、int64、binary 或 fixed_len_byte_array	`message m {`
		`  required int32 a (DECIMAL(5,2));`
		`}`

DATE	不带时间的日期值，可用于注解 int32。用 Unix 元年(1970 年 1 月 1 日)以来的天数表示	`message m {`
		`  required int32 a (DATE);`
		`}`

LIST	一组有序的值，可用于注解 group	`message m {`
		`  required group a (LIST) {`
		`  repeated group list {`
		`  required int32 element;`
		`  }`
		`  }`
		`}`

MAP	一组无序的键-值对，可用于注解 group	`message m {`
		`  required group a (MAP) {`
		`  repeated group key_value {`
		`  required binary key (UTF8);`
		`  optional int32 value;`
		`  }`
		`  }`
		`}`

Parquet 利用 group 类型来构造复杂类型，它可以增加一级嵌套。[1]没有注解的 group 就是一个简单的嵌套记录。

可以用一种特殊的两级嵌套 group 结构构造 list 和 map，如表 13-2 所示。list 是通过 LIST 注解的 group 来表示，其中又嵌套了一个重复的 group(命名为 list)，元素字段包含在这个内层 group 中。从表 13-2 的示例中可以看出，一个 32 位整数的 list 由数据类型为 int32 且重复数为 required(必须出现一次)的元素字段构成。对 map 来说，外层的 group a(使用 MAP 注解)嵌套了一个可重复的内层 group(命名为 key_value)，其中包含 key 和 value 两个字段。

嵌套编码

使用面向列式的存储格式时，同一列数据连续存储。对于气象记录模式这种既无嵌套也无重复的扁平表而言，事情非常简单。由于每一列都含有相同数量的值，因此可以直观地判断出每个值属于哪一行。

当遇到嵌套和重复时，比如 map 模式，事情一般会变得有些复杂，因为还需要对嵌套的结构进行编码。有些列式存储格式通过将嵌套结构扁平化来回避这个问题，使得只有位于最上层的列才能以列主(column-major)方式存储，例如 Hive 的 RCFile 就采取了这种方式。这样，具有嵌套列的 map 中的键和值将会交错存储，也就是说，虽然你只想读取键，却不得不把值也读取到内存中。

Parquet 使用的是 Dremel 编码方法，即模式中的每个原子类型的字段都单独存储为一列，且每个值都要通过使用两个整数来对其结构进行编码，这两个整数分别是列定义深度(definition level)和列元素重复次数(repetition level)。这种编码方式的细节错综复杂，[2]不过你可以把列定义深度和列元素重复次数的存储想像成类似于用一个位字段来为扁平记录的空值进行编码，而非空值则一个紧挨一个地存储。

这种编码方式带来的好处是对任意一列(即使是嵌套列)数据的读取都不需要涉及到其他列。例如，在读取 Parquet 的 map 键-值对中的键时，不需要访问任何值，从而使其性能得到显著提升，尤其是当值非常大的时候，比如，包含很多字段的嵌套记录。

① 这种做法基于 Protocol Buffers (*https://developers.google.com/protocol-buffers/*)中使用的模型，其中 group 用于定义像 list 和 map 之类的复杂类型。

② Julien Le Dem 对此有很好的阐述，网址为 *http://bit.ly/dremel_parquet*。

13.2 Parquet 文件格式

Parquet 文件由一个文件头(header)、一个或多个紧随其后的文件块(block)，以及一个用于结尾的文件尾(footer)构成。文件头中仅包含一个称为 PAR1 的 4 字节数字(Magic Number)，它用来识别整个 Parquet 文件格式。文件的所有元数据都被保存在文件尾中。文件尾中的元数据包括文件格式的版本信息、模式信息、额外的健值对以及所有块的元数据信息。文件尾的最后两个字段分别是一个 4 字节字段(其中包含了文件尾中元数据长度的编码)和一个 PAR1(与文件头中的相同)。

由于元数据保存在文件尾中，因此在读 Parquet 文件时，首先要做的就是找到文件的结尾，然后(减去 8 个字节)读取文件尾中的元数据长度，并根据元数据长度逆向读取文件尾中的元数据。顺序文件和 Avro 数据文件都是把元数据保存在文件头中，并且使用 sync marker 来分割文件块，而 Parquet 文件则不同，由于文件块之间的边界信息被保存在文件尾的元数据中，因此 Parquet 文件不需要使用 sync marker。这种做法之所以可行，正是因为元数据要等到最后才写入，此时所有文件块都已写完，只要文件没有关闭，writer 就能在内存中保留这些文件块的边界位置。综上所述，由于通过读取文件尾可以定位文件块，因此 Parquet 文件是可分割且可并行处理的(例如通过 MapReduce 处理)。

Parquet 文件中的每个文件块负责存储一个**行组**(row group)，行组由**列块**(column chunk)构成，且一个列块负责存储一列数据。每个列块中的数据以**页**(page)为单位存储，如图 13-1 所示。

图 13-1. Parquet 文件的内部结构

由于每页所包含的值都来自于同一列，因此极有可能这些值之间的差别并不大，那么使用页作为压缩单位是非常合适的。初级压缩来自编码方式，最简单的编码方式是无格式编码(plain encoding)，即原封不动地存储一个值(例如使用 4 字节的小端字节表示法来存储 `int32` 类型)，然而，这种编码方式并没有提供任何程度的压缩。

Parquet 会使用一些带有压缩效果的编码方式，包括差分编码(保存值与值之间的差)、游程长度编码(将一连串相同的值编码为一个值以及重复次数)、字典编码(创建一个字典，对字典本身进行编码，然后使用代表字典索引的一个整数来表示值)。在大多数情况下，Parquet 还会使用其他一些技术，比如位紧缩法(bit packing)，它将多个较小的值保存在一个字节中以节省空间。

在写文件时，Parquet 会根据列的类型自动选择适当的编码方式。例如，在保存布尔类型时，Parquet 会结合游程长度编码与位紧缩法。大部分数据类型的默认编码方式是字典编码，但如果字典太大，就要退回到无格式编码。触发退回的阈值称为字典页大小(dictionary page size)，其默认值等于页的大小(因此，倘若使用字典编码，那么这个字典页不得超过一页的范围)。请注意，实际采用的编码方式保存在文件的元数据中，这样才能确保 reader 在读取数据时使用正确的编码方式。

除编码外，还可以以页为单位，利用标准压缩算法对编码后的数据进行第二次压缩。Parquet 的默认设置是不使用任何压缩算法，但它可以支持 Snappy、gzip 和 LZO 等压缩工具。

对于嵌套数据来说，每一页还需要存储该页所包含的值的列定义深度和列元素重复次数。由于这两个数都是很小的整数(最大值取决于模式指定的嵌套深度)，因此使用位紧缩法与游程长度编码可以非常有效地进行编码。

13.3 Parquet 的配置

Parquet 文件的属性在写操作时设置。表 13-3 中列出的文件属性适用于 MapReduce(用 13.5 节中讨论的格式)、Crunch、Pig 或 Hive 创建的 Parquet 文件。

在设置文件块的大小时需要折衷考虑扫描效率与内存的使用。较大的文件块包含较多的行，因此扫描效率更高，同时也可以提高顺序 I/O 操作的效率(因为在设置列块时的额外开销比较少)。但是，每个文件块在读/写操作时都需要缓存在内存中，这个限制使得文件块不能太大。默认的文件块为 128 MB。

表 13-3. ParquetOutputFormat 的属性

属性名称	类型	默认值	描述
parquet.block.size	int	134217728 (128 MB)	文件块(行组)的大小，以字节为单位
parquet.page.size	int	1048576 (1 MB)	页的大小，以字节为单位
parquet.dictionary.page.size	int	1048576 (1 MB)	一个页可允许的最大的字典，以字节为单位，若字典超过这个尺寸，将退回到无格式编码
parquet.enable.dictionary	boolean	true	是否使用字典编码
parquet.compression	字符串	UNCOMPRESSED	Parquet 文件使用的压缩类型：UNCOMPRESSED、SNAPPY、GZIP 或 LZO。用于替代 mapreduce.output.fileoutputformat.compress

Parquet 文件块的大小不能超过其 HDFS 块大小，只有这样才能使每个 Parquet 文件块仅需读一个 HDFS 块(因而也只需要从一个数据结点上读)。比较常见的做法是为这两个属性设置相同的值，事实上，它们的默认值都是 128 MB。

页是 Parquet 文件的最小存储单元，因此，如果想要读取任意一行数据(为了方便举例，我们假设它只有一列数据)，就必须对包含这一行数据的页进行解压缩和解编码处理。所以，对于单行查找来说，页越小，在找到目标值之前需要读取的值就越少，因而效率越高。但是较小的页会带来较大的存储和处理额外开销，原因在于如果页越多，额外的元数据(偏移值、字典)也就越多。默认情况下，页的大小为 1 MB。

13.4 Parquet 文件的读/写

在大多数情况下，我们会使用高级工具来处理 Parquet 文件的读/写操作，比如 Pig、Hive 或 Impala。不过，有些时候也需要对其进行低级的顺序访问，而这正是本节所要讨论的。

Parquet 具有一个可插入式的内存数据模型，其作用是要让 Parquet 文件格式更好地与类型广泛的各种工具和组件集成。在 Java 中，这种集成体现在 ReadSupport 和 WriteSupport 上。这两个类负责完成工具或组件所使用的对象与代表了模式中各种 Parquet 数据类型的对象之间的转换。

在 parquet.example.data 和 parquet.example.data.simple 软件包中有一个

Parquet 附带的简单的内存数据模型，我们将利用它来举例说明。下一节我们还会通过 Avro 来完成相同的任务。

从名称即可看出，Parquet 所附带的这些示例类只是一些用来演示怎样处理 Parquet 文件的对象模型。对于正式产品来说，则应当使用某种支持性框架(如 Avro、Protocol Buffers 或 Thrift)。

在写 Parquet 文件之前，必须先定义其 Parquet 模式，该模式通过一个 parquet.schema.MessageType 实例来表示：

```
MessageType schema = MessageTypeParser.parseMessageType(
    "message Pair {\n" +
    " required binary left (UTF8);\n" +
    " required binary right (UTF8);\n" +
"}");
```

接下来，为即将写入文件的每个记录创建 Parquet message 实例。对 parquet.example.data 软件包来说，一个 message 被表示为一个 Group 实例，并通过 GroupFactory 来构造：

```
GroupFactory groupFactory = new SimpleGroupFactory(schema);
Group group = groupFactory.newGroup()
    .append("left", "L")
    .append("right", "R");
```

请注意，message 所包含的值的逻辑类型为 UTF8，而 Group 为我们提供了它与 Java String 之间的自然转换。

下面这段代码演示的是如何创建 Parquet 文件并向该文件写入一个 message。通常，为了向文件写入多个 message，应当在循环语句中调用 write()方法。不过，此处只需要写一个 message。

```
Configuration conf = new Configuration();
Path path = new Path("data.parquet");
GroupWriteSupport writeSupport = new GroupWriteSupport();
GroupWriteSupport.setSchema(schema, conf);
ParquetWriter<Group> writer = new ParquetWriter<Group>(path, writeSupport,
    ParquetWriter.DEFAULT_COMPRESSION_CODEC_NAME,
    ParquetWriter.DEFAULT_BLOCK_SIZE,
    ParquetWriter.DEFAULT_PAGE_SIZE,
    ParquetWriter.DEFAULT_PAGE_SIZE, /* dictionary page size */
    ParquetWriter.DEFAULT_IS_DICTIONARY_ENABLED,
    ParquetWriter.DEFAULT_IS_VALIDATING_ENABLED,
    ParquetProperties.WriterVersion.PARQUET_1_0, conf);
writer.write(group);
writer.close();
```

我们需要为构造函数 ParquetWriter 提供一个 WriteSupport 实例，该实例定义了如何将 message 类型转换为 Parquet 类型。在这个例子中，我们使用的是 Group 类型，因此需要用到 GroupWriteSupport 类。请注意，通过调用 GroupWriteSupport 中的 setSchema()静态方法可以将 Parquet 模式设置到 Configuration 对象上，然后再把 Configuration 对象传递给 ParquetWriter。从这个例子中还可以看出应当如何设置 Parquet 文件的属性，其中相应的属性可以参考表 13-3。

Parquet 文件的读操作比写操作更简单，因为在读操作时不需要保存 Parquet 文件模式，所以不需要定义模式。不过，如果想通过投影来返回文件中的列的子集，那么也可以设置一个读取模式。另外，读操作不需要设置文件属性，因为这是在写操作时设置的。

```
GroupReadSupport readSupport = new GroupReadSupport();
ParquetReader<Group> reader = new ParquetReader<Group>(path, readSupport);
```

ParquetReader 中的 read()方法可以读取下一个 message，一旦到达文件结尾，它就会返回空值。

```
roup result = reader.read();
assertNotNull(result);
assertThat(result.getString("left", 0), is("L"));
assertThat(result.getString("right", 0), is("R"));
assertNull(reader.read());
```

请注意，传递给 getString()方法的参数 0 指明了需要读取的字段的索引，因为字段中的值可能是重复的。

13.4.1 Avro、Protocol Buffers 和 Thrift

大多数应用程序更倾向于使用 Avro、Protocol Buffers 或 Thrift 这样的框架来定义数据模型，Parquet 则迎合了这些需求。此时我们使用的不再是 ParquetWriter 和 ParquetReader，而是 AvroParquetWriter、ProtoParquetWriter 或 ThriftParquetWriter 及其分别对应的 reader 的类。这些类负责完成 Avro、Protocol Buffers 或 Thrift 模式与 Parquet 模式之间的转换(还包括执行框架的数据类型与 Parquet 数据类型之间的等价映射)，这就意味着我们不需要直接与 Parquet 模式打交道。

仍然重复前一小节中的例子，不过这一次用的是 Avro Generic API(参见 12.2 节)。

Avro 模式如下：

```
{
  "type": "record",
  "name": "StringPair",
    "doc": "A pair of strings.",
"fields": [
    {"name": "left", "type": "string"},
    {"name": "right", "type": "string"}
  ]
}
```

通过下面这段代码可以创建一个模式实例和一个通用(generic)记录：

```
Schema.Parser parser = new Schema.Parser();
Schema schema =
    parser.parse(getClass().getResourceAsStream("StringPair.avsc"));

GenericRecord datum = new GenericData.Record(schema);
datum.put("left", "L");
datum.put("right", "R");
```

然后，写一个 Parquet 文件：

```
Path path = new Path("data.parquet");
AvroParquetWriter<GenericRecord> writer =
    new AvroParquetWriter<GenericRecord>(path, schema);
writer.write(datum);
writer.close();
```

AvroParquetWriter 将 Avro 模式转换为 Parquet 模式，同时还要将各个 Avro GenericRecord 实例转换为相应的 Parquet 数据类型以便写入 Parquet 文件。这个文件就是一个常规的 Parquet 文件，除了需要额外的元数据来保存 Avro 模式之外，它与上一小节利用 GroupWriteSupport 所写的文件完全一致。我们可以使用 Parquet 的命令行工具来查看该文件的元数据：[①]

```
% parquet-tools meta data.parquet
...
extra: avro.schema = {"type":"record","name":"StringPair", ...
...
```

类似地，可以使用下述语句查看由 Avro 模式生成的 Parquet 模式：

```
% parquet-tools schema data.parquet
message StringPair {
```

① Parquet 工具可以从 Parquet Maven 资源库中以二进制压缩包的形式下载。请访问 *http://search.maven.org* 网站并搜索 parquet-tools。

```
  required binary left (UTF8);
  required binary right (UTF8);
  }
```

反之则可以使用 AvroParquetReader 来读取 Parquet 文件，并重新得到 Avro 的 GenericRecord 对象：

```
AvroParquetReader<GenericRecord> reader =
    new AvroParquetReader<GenericRecord>(path);
GenericRecord result = reader.read();
assertNotNull(result);
assertThat(result.get("left").toString(), is("L"));
assertThat(result.get("right").toString(), is("R"));
assertNull(reader.read());
```

13.4.2 投影模式和读取模式

有些时候，我们只是希望想读取文件中的少数几列，这种情况并不少见，事实上，这正是像 Parquet 这样的列式存储格式存在的真正原因：节省时间并提升 I/O 操作的效率。利用投影模式可以选择所需读取的列，例如下面这个模式用于只读取 StringPair 中的 right 字段：

```
{
  "type": "record",
  "name": "StringPair",
  "doc": "The right field of a pair of strings.",
  "fields": [
    {"name": "right", "type": "string"}
  ]
}
```

利用 AvroReadSupport 中的便捷静态方法 setRequestedProjection()将该投影模式设置到配置中：

```
Schema projectionSchema = parser.parse(
    getClass().getResourceAsStream("ProjectedStringPair.avsc"));
Configuration conf = new Configuration();
AvroReadSupport.setRequestedProjection(conf, projectionSchema);
```

然后，将该配置传递给 AvroParquetReader 的构造函数：

```
AvroParquetReader<GenericRecord> reader =
    new AvroParquetReader<GenericRecord>(conf, path);
GenericRecord result = reader.read();
assertNull(result.get("left"));
assertThat(result.get("right").toString(), is("R"));
```

Protocol Buffers 和 Thrift 都能以同样的方式支持投影模式。另外，Avro 还允许通过调用 `AvroReadSupport` 中的 `setReadSchema()`方法指定一个读取模式。读取模式负责根据表 12-4 中列出的规则来解析 Avro 记录。

Avro 既有投影模式又有读取模式，其原因在于：投影模式必须是 Parquet 文件在写操作时所使用模式的一个子集，因而无法通过增加新的字段来对模式进行扩展。

这两种模式的用途不同，且两者可同时使用。投影模式被用来过滤想要从 Parquet 文件中读取的列。虽然它是通过 Avro 模式来表示的，但可以简单地将其视为一个列表，其中包含的是需要读取的 Parquet 列。另一方面，读取模式仅用于解析 Avro 记录，因为它与从 Parquet 文件中读取哪些列无关，所以它永远不会转换成一个 Parquet 模式。例如，如果我们为 12.5 节中的 Avro 模式增加一个 `description` 字段，并将其当作读取模式，那么即使 Parquet 文件中并没有这个字段，记录中也应当包含该字段的默认值。

13.5 Parquet MapReduce

MapReduce 的输入/输出格式中有一部分是用于通过 MapReduce 作业来读/写 Parquet 文件 Parquet 格式，包括与 Avro、Protocol Buffers 和 Thrift 模式和数据打交道的格式。

范例 13-1 中给出的程序是一个只有 map 阶段的作业，它用于读取文本文件并生成 Parquet 文件，其中每个记录包含两个字段：行的偏移量(类型为 `int64`，在 Avro 中被转换为 `long`)和行本身(一个字符串)。它使用 Avro Generic API 作为内存中的数据模型。

范例 13-1. MapReduce 程序用 AvroParquetOutputFormat 将文本文件转换为 Parquet 文件

```
public class TextToParquetWithAvro extends Configured implements Tool {

  private static final Schema SCHEMA = new Schema.Parser().parse(
      "{\n" +
      " \"type\": \"record\",\n" +
      " \"name\": \"Line\",\n" +
      " \"fields\": [\n" +
      " {\"name\": \"offset\", \"type\": \"long\"},\n" +
      " {\"name\": \"line\", \"type\": \"string\"}\n" +
      " ]\n" +
      "}");
```

```
  public static class TextToParquetMapper
      extends Mapper<LongWritable, Text, Void, GenericRecord> {

    private GenericRecord record = new GenericData.Record(SCHEMA);

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
      record.put("offset", key.get());
      record.put("line", value.toString());
      context.write(null, record);
    }
  }

  @Override
  public int run(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.printf("Usage: %s [generic options] <input> <output>\n",
        getClass().getSimpleName());
      ToolRunner.printGenericCommandUsage(System.err);
      return -1;
    }

    Job job = new Job(getConf(), "Text to Parquet");
    job.setJarByClass(getClass());

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(TextToParquetMapper.class);
    job.setNumReduceTasks(0);

    job.setOutputFormatClass(AvroParquetOutputFormat.class);
    AvroParquetOutputFormat.setSchema(job, SCHEMA);

    job.setOutputKeyClass(Void.class);
    job.setOutputValueClass(Group.class);

    return job.waitForCompletion(true) ? 0 : 1;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new TextToParquetWithAvro(), args);
    System.exit(exitCode);
  }
}
```

这个作业的输出格式设置为 AvroParquetOutputFormat，并且由于使用的是 Avro Generic API，所以为了进行匹配，我们将输出的键和值的数据类型分别设置为 Void 和 GenericRecord。Void 仅仅表示键总是被设置为 null。

与前一小节中的 AvroParquetWriter 类似，AvroParquetOutputFormat 自动将

Avro 模式转换为 Parquet 模式。将该 Avro 模式设置到 Job 实例上，从而使得 MapReduce 任务在写文件时能够找到该模式。

这段 mapper 程序很简单，它获取文件偏移量(键)和行数据(值)，构建一个 Avro `GenericRecord` 对象，并把这个对象作为值(键永远是空)写到 MapReduce 的 `Context` 类对象中。`AvroParquetOutputFormat` 负责将 Avro `GenericRecord` 转换为 Parquet 文件格式编码。

Parquet 是一种列式存储格式，因此它需要在内存中缓存多行数据。即使这个例子中的 mapper 所做的仅仅是传递少量的值，Parquet writer 也需要足够的内存来缓存每个文件块(行组)，请注意文件块的默认值是 128 MB。如果由于内存错误而导致作业失败，可以通过设置 `parquet.block.size`(参见表 13-3)为 writer 调整文件块大小。也可能需要使用 10.3.3 节中讨论的有关设置来修改 MapReduce 任务的内存分配(在读操作和写操作时)。

下述命令对一个包含四行内容的文本文件 *quangle.txt* 运行该程序：

```
% hadoop jar parquet-examples.jar TextToParquetWithAvro \
input/docs/quangle.txt output
```

我们可以使用 Parquet 命令行工具在屏幕上查看输出的 Parquet 文件：

```
% parquet-tools dump output/part-m-00000.parquet
INT64 offset
--------------------------------------------------------------------------------
*** row group 1 of 1, values 1 to 4 ***
value 1: R:0 D:0 V:0
value 2: R:0 D:0 V:33
value 3: R:0 D:0 V:57
value 4: R:0 D:0 V:89
BINARY line
--------------------------------------------------------------------------------
*** row group 1 of 1, values 1 to 4 ***
value 1: R:0 D:0 V:On the top of the Crumpetty Tree
value 2: R:0 D:0 V:The Quangle Wangle sat,
value 3: R:0 D:0 V:But his face you could not see,
value 4: R:0 D:0 V:On account of his Beaver Hat.
```

注意一个行组中的所有值是如何一起显示的，V 表示值，R 表示列元素重复次数，D 表示列定义深度。因为这个模式不存在嵌套，所以后两者的值都是零。

第 14 章

关于 Flume

Hadoop 的构建宗旨是处理大型数据集。通常，我们假设这些数据已经存储在 HDFS 中或者能够随时批量复制到 HDFS。然而，有许多系统并不符合此假设。这些系统生产出我们想要通过 Hadoop 来汇总、存储和分析的数据流。与这些系统打交道，Apache Flume (*http://flume.apache.org/*)再合适不过。

设计 Flume 的宗旨是向 Hadoop 批量导入基于事件的海量数据。一个典型的例子是利用 Flume 从一组 Web 服务器中收集日志文件，然后把这些文件中的日志事件转移到一个新的 HDFS 汇总文件中以做进一步处理，其终点(或者用 Flume 的术语称之为 sink)通常为 HDFS。不过，Flume 具有足够的灵活性，也可以将数据写到其他系统中，例如 HBase 或 Solr。

要想使用 Flume，就需要运行 Flume 代理。Flume 代理是由持续运行的 source(数据来源)、sink(数据目标)以及 channel(用于连接 source 和 sink)构成的 Java 进程。Flume 的 source 产生事件，并将其传送给 channel，channel 存储这些事件直至转发给 sink。可以把 source-channel-sink 的组合视为基本 Flume 构件。

Flume 由一组以分布式拓扑结构相互连接的代理构成。系统边缘的代理(例如，与 Web 服务器共存于同一台机器上的代理)负责采集数据，并把数据转发给负责汇总的代理，然后再将这些数据存储到其最终目的地。代理通过配置来运行一组特定的 source 和 sink。因此，使用 Flume 所要做的主要工作就是通过配置使得各个组件融接到一起。本章将展示如何构建用于收集数据的 Flume 拓扑结构，你也可以把它当作是自己的 Hadoop 管道的一部分。

14.1 安装 Flume

从下载页面(*http://flume.apache.org/download.html*)下载一个稳定版本的 Flume 二进制发行包，并在适当的位置解压缩该文件包：

```
% tar xzf apache-flume-x.y.z-bin.tar.gz
```

为了方便起见，可以把 Flume 的二进制文件路径添加到自己的路径中：

```
% export FLUME_HOME=~/sw/apache-flume-x.y.z-bin
% export PATH=$PATH:$FLUME_HOME/bin
```

然后，使用 `flume-ng` 命令启动 Flume 代理，正如我们下面将要看到的。

14.2 示例

为了演示 Flume 是如何工作的，我们首先从以下设置出发。

(1) 监视新增文本文件所在的本地目录。

(2) 每当有新增文件时，文件中的每一行都将被发往控制台。

本例中的新增文件是由手工添加的，不过也不难想像由一个 Web 服务器之类的进程不断创建我们希望利用 Flume 采集的新文件。此外，对于实际系统而言，我们需要的不仅仅是记录文件的内容，而是要将文件内容写入 HDFS 以待后续处理，本章稍后介绍。

在本例中，Flume 代理仅运行了一个 source-channel-sink 组合，并且它利用 Java 属性文件进行配置。这些配置控制着 source、sink 和 channel 的类型以及它们的连接方式。这个例子所使用的配置如范例 14-1 所示。

范例 14-1. 使用 spooling directory source 和 logger sink 的 Flume 配置

```
agent1.sources = source1
agent1.sinks = sink1
agent1.channels = channel1

agent1.sources.source1.channels = channel1
agent1.sinks.sink1.channel = channel1

agent1.sources.source1.type = spooldir
agent1.sources.source1.spoolDir = /tmp/spooldir
```

```
agent1.sinks.sink1.type = logger

agent1.channels.channel1.type = file
```

属性名称构成了一个分级结构，顶级为代理的名称。在本例中只有一个代理，其名称为 agent1。下一级定义的是代理中的不同组件的名称，因此，以 agent1.sources 为例，它列出在 agent1 中运行的 source 的名称(本例只有一个 source，即 source1)。同样，agent1 只有一个 sink(即 sink1)和一个 channel(即 channel1)。

再下一级是组件的属性。组件可用的属性取决于该组件的类型。在本例中，agent1.sources.source1.type 设置为 spooldir，它是一个 spooling directory source，用于监视缓冲目录中的新增文件。Spooling directory source 有一个称为 spoolDir 的属性定义，因此，对于 source1 来说，完整的键应当是 agent1.sources.source1.spoolDir。source 所使用的 channel 则通过 agent1.sources.source1.channels 属性来设置。

在本例中，sink 是一个 logger sink，用于将事件记录到控制台。该 sink 必须连接 channel1(通过 agent1.sinks.sink1.channel 属性来设置)。[①]由于此处的 channel 类型是 file channel，因此 channel 中的事件可以持久存储在磁盘上。整个系统如图 14-1 所示。

图 14-1. 具有通过 file channel 连接的 spooling directory source 和 logger sink 的 Flume 代理

在运行本示例之前，我们首先需要在本地文件系统上创建一个缓冲目录：

```
% mkdir /tmp/spooldir
```

① 注意，source 的属性是 channels(复数)，但 sink 的属性是 channel(单数)。这是因为一个 source 可以向一个以上的 channel 输送数据(参见 14.5 节)，而一个 sink 只能吸纳来自一个 channel 的数据。另外，一个 channel 也有可能向多个 sink 输送数据，详情可以参见 14.7 节。

然后，使用 Flume-ng 命令启动 Flume 代理：

```
% flume-ng agent \
--conf-file spool-to-logger.properties \
--name agent1 \
--conf $FLUME_HOME/conf \
-Dflume.root.logger=INFO,console
```

--conf-file 用于指定 Flume 的属性文件(即范例 14-1 所示的属性文件)，另外还要通过--name 来传递代理的名称(由于一个 Flume 属性文件可以定义多个代理，因此我们必须指明运行的是哪一个代理)。通过--conf 告诉 Flume 在哪里可以找到它的通用配置，例如环境设置。

在一台新终端上，我们在缓冲目录中新建一个文件。Spooling directory source 不允许文件被编辑改动，因此为了防止写了一半的文件被 source 读取，应当先把全部内容写到一个隐藏文件中，然后再执行文件重命名命令，从而使得 source 能够读取到完整的文件：[①]

```
% echo "Hello Flume" > /tmp/spooldir/.file1.txt
% mv /tmp/spooldir/.file1.txt /tmp/spooldir/file1.txt
```

回到代理终端，我们看到 Flume 已经检测到该文件并对其进行了处理：

```
Preparing to move file /tmp/spooldir/file1.txt to
  /tmp/spooldir/file1.txt.COMPLETED
Event: { headers:{} body: 48 65 6C 6C 6F 20 46 6C 75 6D 65 Hello Flume }
```

Spooling directory source 导入文件的方式是把文件按行拆分，并为每行创建一个 Flume 事件。事件由一个可选的 header 和一个二进制的 body 组成，其中 body 是 UTF-8 编码的文本行。Logger sink 使用十六进制和字符串两种形式来记录 body。由于该缓冲目录中的文件仅包含一行内容，因此被记录的事件也只有一个。我们还看到该文件被 source 重命名为 file1.txt.COMPLETED，这表明 Flume 已经完成文件的处理，并且对它不会再有任何动作。

14.3 事务和可靠性

Flume 使用两个独立的事务分别负责从 source 到 channel 以及从 channel 到 sink 的

① 对于连续追加记录的日志文件，可以使用定期滚动方式，并将旧文件转到缓冲目录下，供 Flume 读取。

事件传递。在上一节的例子中，spooling directory source 为文件的每一行创建一个事件。一旦事务中的所有事件全部传递到 channel 且提交成功，那么 source 就将该文件标记为完成。

事务以类似的方式应用于从 channel 到 sink 的事件传递过程。如果由于某种原因使得事件无法记录，那么事务将会回滚，而所有的事件仍然保留在 channel 中，等待重新传递。

本例使用的 channel 是 *file channel*，这种类型的 channel 具有持久性：只要事件被写入 channel，即使代理重新启动，事件也不会丢失。(Flume 还提供有 *memory channel*，由于它的事件缓存在存储器中，因此它不具有持久存储能力。采用 memory channel 时，如果代理重新启动，事件就会丢失。在某些时候这种情况是可以接受的，具体取决于应用。与 file channel 相比，memory channel 的优势在于具有较高的吞吐量。)

总的来说，source 产生的每个事件都会到达 sink。这里需要说明的是，每个事件到达 sink 至少一次(at least once)，也就是说，同一事件有可能会重复到达。不论 source 还是 sink 都有可能产生重复。例如，即使代理重启之前有部分或全部事件已经被提交到 channel，但是在代理重启之后，spooling directory source 还是会为所有未完成的文件重新传递事件。代理重新启动后，logger sink 也会重新记录那些已经记录但未提交的事件(如果代理在这两个操作之间关闭)。

“At-least-once”看起来好像是一个缺陷，而实际上它是一个可以接受的性能上的权衡。“exactly once”在含义上更加严格，但它需要使用一种两阶段的提交协议(two-phase commit protocol)，代价很高。这两种不同的选择所体现的正是作为高容量并行事件采集系统的 Flume(采用 at-least-once 方式)与较为传统的企业信息系统(采用 exactly-once 方式)之间的区别。采用 at-least-once 方式时，重复的事件可留待后续的处理流水线来移除。通常，这需要以特定于应用程序的重复数据删除作业的形式写在 MapReduce 或 Hive 之中。

批量处理

为了提高效率，Flume 在有可能的情况下尽量以事务为单位来批量处理事件，而不是逐个事件地进行处理。批量处理方式尤其有利于提高 file channel 的性能，因为每个事务只需要写一次本地磁盘和调用一次 fsync。

批量的大小取决于组件的类型，并且在大多数情况下是可配置的。例如，spooling

directory source 以 100 行文本作为一个批次来读取(可通过 BatchSize 属性设置)。同样，在通过 Avro RPC 发送之前，Avro sink(参见 14.6 节)试图从 channel 中批量地读取 100 个事件，当然，如果可用的事件不足 100 个也不会引起阻塞。

14.4 HDFS Sink

Flume 关注的重点是向 Hadoop 数据存储器传递海量数据，因此，接下来让我们看看应当如何配置 Flume 代理以向 HDFS sink 传递事件。范例 14-2 中的配置将上一个例子中的 sink 更改为 HDFS sink，其中必不可缺的属性设置只有两个：sink 的类型(即 hdfs)和 hdfs.path。hdfs.path 指定文件存放的位置(如果路径中没有指定文件系统，那么像往常一样使用由 Hadoop 的 fs.defaultFS 属性所指定的文件系统，正如本例所示)。另外，我们还指定了一个有意义的文件前缀和后缀，并指示 Flume 将事件以文本格式写入文件。

范例 14-2. 使用 spooling directory source 和 HDFS sink 的 Flume 配置

```
agent1.sources = source1
agent1.sinks = sink1
agent1.channels = channel1

agent1.sources.source1.channels = channel1
agent1.sinks.sink1.channel = channel1

agent1.sources.source1.type = spooldir
agent1.sources.source1.spoolDir = /tmp/spooldir

agent1.sinks.sink1.type = hdfs
agent1.sinks.sink1.hdfs.path = /tmp/flume
agent1.sinks.sink1.hdfs.filePrefix = events
agent1.sinks.sink1.hdfs.fileSuffix = .log
agent1.sinks.sink1.hdfs.inUsePrefix = _
agent1.sinks.sink1.hdfs.fileType = DataStream

agent1.channels.channel1.type = file
```

重新启动代理以应用 *spool-to-hdfs.properties* 的配置，并在文件缓冲目录中创建一个新文件：

```
% echo -e "Hello\nAgain" > /tmp/spooldir/.file2.txt
% mv /tmp/spooldir/.file2.txt /tmp/spooldir/file2.txt
```

这一次，事件被传递给 HDFS sink 并写到一个文件。对于正在进行写操作处理的文件，其文件名会添加一个后缀“.tmp”，以表明文件处理尚未完成。在本例

中，hdfs.inUsePrefix 属性被设置为下划线(默认值为空)，此举将导致正在进行写操作处理的文件的文件名还要添加一个_(下划线)作为前缀。这样做是因为 MapReduce 会忽略以下划线为前缀的文件。因此，一个典型的临时文件的文件名为“*_events.1399295780136.log.tmp*”，其中的数字是由 HDFS sink 生成的时间戳。

在超过给定的打开时间(默认值为 30 秒，由 hdfs.rollInterval 属性设置)或者达到给定的文件大小(默认值为 1024 个字节，由 hdfs.rollSize 属性设置)，再或者写满了给定数量的事件(默认值为 10，由 hdfs.rollCount 属性设置)之前，HDFS sink 保持文件的打开状态。如果以上三个条件中有任意一条得到满足，则关闭文件，并且体现其正在使用中的文件名的前缀和后缀都被移除。此后，新的事件将被写入一个新的文件(新文件的文件名将具有体现其正在使用中的前缀和后缀，直至该轮处理结束)。

在等待 30 秒后，我们可以肯定文件的处理已经结束，现在可以查看它的内容：

```
% hadoop fs -cat /tmp/flume/events.1399295780136.log
Hello
Again
```

HDFS sink 以运行 Flume 代理的用户的身份来写文件，除非设置了 hdfs.proxyUser 属性，那么 HDFS sink 将以指定用户的身份来写文件。

14.4.1 分区和拦截器

大型数据集常常被组织为分区(partition)，这样做的好处是，如果查询仅涉及到数据的某个子集，查询过程就可以被限制在特定的分区范围内。Flume 事件的数据通常按时间来分区，因此，我们可以定期运行某个进程来对已完成的分区进行各种数据转换处理(例如，删除重复的事件)。

我们可以很容易地将示例中的数据存储方式更改为分区存储方式，只需要对 hdfs.path 属性进行设置，使之具有使用时间格式转义序列的子目录：

```
agent1.sinks.sink1.hdfs.path = /tmp/flume/year=%Y/month=%m/day=%d
```

此处，我们选择天作为分区粒度，不过也可以使用其他级别的分区粒度，结果将导致不同的文件目录布局方案。(如果使用 Hive，可参考 17.6.2 节有关 Hive 如何在磁盘上布局分区的介绍。)在 Flume 用户指南(*http://flume.apache.org/FlumeUserGuide.html*)中，与 HDFS sink 相关的文件提供了格式转义序列的完整

列表。

一个 Flume 事件将被写入哪个分区是由事件的 header 中的 timestamp(时间戳)决定的。在默认情况下，事件 header 中并没有 timestamp，但是它可以通过 Flume **拦截器**(interceptor)来添加。拦截器是一种能够对事件流中的事件进行修改或删除的组件，它们连接 source 并在事件被传递到 channel 之前对事件进行处理。[①]下面的两行配置用于为 source1 增加一个时间戳拦截器，它将为 source 产生的每个事件添加一个 timestamp header：

```
agent1.sources.source1.interceptors = interceptor1
agent1.sources.source1.interceptors.interceptor1.type = timestamp
```

时间戳拦截器可以确保这些时间戳能够基本如实地反映事件创建的时间。对某些应用程序而言，在事件写入 HDFS 时使用一个时间戳已经足够了，但是必须要注意，如果存在多层 Flume 代理，那么事件的创建时间和写入时间之间可能存在明显差异，尤其是当代理出现停机的情况时(参见 14.6 节)。针对这种情况，我们可以对 HDFS sink 的 hdfs.useLocalTimeStamp 属性进行设置，以便使用由运行 HDFS sink 的 Flume 代理所产生的时间戳。

14.4.2 文件格式

一般而言，使用二进制格式来存储数据是一个好主意，因为它生成的文件比使用文本格式的文件更小。HDFS sink 使用的文件格式由 hdfs.fileType 属性及其他一些属性的组合控制。

默认的 Hdfs.fileType 文件格式为 SequenceFile，也就是把事件写入一个顺序文件。在这个顺序文件中，键的数据类型为 LongWritable，它包含的是事件的时间戳(如果事件的 header 中没有时间戳，就使用当前时间)。值的数据类型为 BytesWritable，其中包含的是事件的 body。如果 hdfs.writeFormat 被设置为 Text，那么顺序文件中的值就变成 Text 数据类型，而非 BytesWritable 类型。

针对 Avro 文件所使用的配置略有不同，它的 Hdfs.fileType 属性被设置为 DataStream，就像纯文本一样。此外，serializer(注意，没有前缀 hdfs.)必须设置为 avro_event。如果想要启用压缩，则需要设置 serializer.compressionCodec 属性。下面这个例子的 HDFS sink 配置为将事件写到一个 Snappy 压缩的 Avro 文

① 表 14-1 介绍了 Flume 提供的拦截器。

件中：

```
agent1.sinks.sink1.type = hdfs
agent1.sinks.sink1.hdfs.path = /tmp/flume
agent1.sinks.sink1.hdfs.filePrefix = events
agent1.sinks.sink1.hdfs.fileSuffix = .avro
agent1.sinks.sink1.hdfs.fileType = DataStream
agent1.sinks.sink1.serializer = avro_event
agent1.sinks.sink1.serializer.compressionCodec = snappy
```

每个事件用一个两字段的 Avro 记录来表示，这两个字段分别为 `header` 和 `body`，其中 `header` 对应于 Avro 映射(字符串值)，`body` 则对应于 Avro 字节字段。

若想使用定制的 Avro 模式，可以有多种选项。如果需要把一个 Avro 内存中的对象发送到 Flume，那么以使用 `Log4jAppender` 为宜。它允许通用 log4j 的 `Logger` 来记录 Avro 的通用映射、特殊映射或自反映射对象，并将其发送给 Flume 代理运行的 Avro source(参见 14.6 节)。在这种情况下，HDFS sink 的 `serializer` 属性应当设置为 `org.apache.flume.sink.hdfs.AvroEventSerializer$Builder`，并且其 Avro 模式在 header 中设置(参见相关类的文档)。

另一方面，如果事件最初不是从 Avro 对象导出的，那么可以写一个定制的 serializer，以便将 Flume 事件转换为具有定制模式的 Avro 对象。为此，`org.apache.flume.serialization` 包的 `AbstractAvroEventSerializer` 类可以作为一个很好的起点。

14.5 扇出

术语扇出(fan out)指的是从一个 source 向多个 channel，亦即向多个 sink 传递事件。例如，范例 14-3 中的配置就是将事件同时传递到一个 HDFS sink(经由 `channel1a` 到达 `sink1a`)和一个 logger sink(经由 `channel1b` 到达 `sink1b`)。

范例 14-3. 用 spooling directory source 且扇出到 HDFS sink 和 logger sink 的 Flume 配置

```
agent1.sources = source1
agent1.sinks = sink1a sink1b
agent1.channels = channel1a channel1b

agent1.sources.source1.channels = channel1a channel1b
agent1.sinks.sink1a.channel = channel1a
agent1.sinks.sink1b.channel = channel1b

agent1.sources.source1.type = spooldir
agent1.sources.source1.spoolDir = /tmp/spooldir
```

```
agent1.sinks.sink1a.type = hdfs
agent1.sinks.sink1a.hdfs.path = /tmp/flume
agent1.sinks.sink1a.hdfs.filePrefix = events
agent1.sinks.sink1a.hdfs.fileSuffix = .log
agent1.sinks.sink1a.hdfs.fileType = DataStream

agent1.sinks.sink1b.type = logger

agent1.channels.channel1a.type = file
agent1.channels.channel1b.type = memory
```

本例的主要变化在于将 agent1.sources.source1.channels 属性设置为一个由 channel1a 和 channel1b 组成的空格分隔列表，使得该 source 被配置为向多个 channel 传递事件。在本例中，连接 logger sink 的 channel(即 channel1b)是一个 memory channel，因为我们记录事件的目的只是为了调试，并不介意代理重启时事件的丢失。此外，与前面的例子相同，每个 channel 通过配置连接到一个 sink。图 14-2 描绘了完整的流程。

图 14-2. 有 spooling directory source 并扇出到一个 HDFS sink 和一个 logger sink 的 Flume agent

14.5.1 交付保证

Flume 使用独立的事务来负责从 spooling directory source 到每一个 channel 的每批事件的传递。在本例中，有一个事务负责从 source 到连接 HDFS sink 的 channel 的事件交付，另一个事务负责从 source 到连接 logger sink 的 channel 的同一批事件的交付。若其中有任何一个事务失败(例如 channel 满溢)，这些事件都不会从 source 中删除，而是等待稍后重试。

在本例中，由于我们并不介意是否有一些事件未能交付给 logger sink，因此可以指定该 channel 为 optional(可选的) channel。在这种情况下，如果与该 channel 相关联的事务失败，source 并不会保留事件以待再次尝试。(请注意，如果在两个

channel 的事务都提交之前代理出现故障，那么在代理重新启动后所有受影响的事件都会重新传递，即使未提交的 channel 被标记为 optional 也是这样)。为此，需要把 source 的属性 selector.optional 设置为一个空格分隔的 channel 的列表：

```
agent1.sources.source1.selector.optional = channel1b
```

近实时索引

对事件进行索引以便于搜索是扇出技术在实践应用中的一个很好的例子。来自 source 的事件既被发送到 HDFS sink(这是事件的主要信息库，因此使用 required channel)，同时也被发送到 Solr(或 Elasticsearch) sink，以建立搜索索引(使用 optional channel)。

MorphlineSolrSink 从 Flume 事件中提取字段，并把它们转换为 Solr 文件(通过使用 Morphline 配置文件)，然后再将其装载到一个活跃的 Solr 搜索服务器中。由于导入的数据在很短的时间内就可以呈现在搜索结果中，所以这个过程被称为近实时(near real time)过程。

14.5.2 复制和复用选择器

正常的扇出流是向所有 channel 复制事件，但有些时候，人们希望有更多的行为方式可以选择，从而使某些事件被发送到这一个 channel，而另一些事件被发送到另一个 channel。这可以通过在 source 上设置一个**复用选择器**(multiplexing selector)来实现，此时我们可以定义路由规则，使得事件 header 中的特定值与某个 channel 相匹配。详细的配置信息请参考 Flume 用户指南。

14.6 通过代理层分发

如何对一组 Flume 代理进行扩展？如果每个产生原始数据的节点上只运行一个代理，那么在目前为止我们所描述的各种配置情况下，不管任何时间，每一个写入 HDFS 的文件的内容都完全由来自同一节点的事件组成。要是能够把来自一组节点的事件汇聚到一个文件中，效果可能会更好，因为这样可以减少文件的数量，增加文件的大小(以减轻同时施加在 HDFS 身上的压力，并提高 MapReduce 的处理效率，参见 8.2.1 节)。另外，由于向文件输入数据的节点变多，因此文件可以更快地推陈出新，从而使得这些文件可用于分析的时间更接近于事件的创建时间，这在某些情况下是有必要的。

要想实现 Flume 事件的汇聚，就需要使用分层结构的 Flume 代理。第一层代理负责采集来自原始 source(如 Web 服务器)的事件，并将它们发送到第二层。第二层代理的数量比第一层少，这些代理先汇总来自第一层代理的事件，再把这些事件写入 HDFS(参见图 14-3)。如果 source 节点的数量庞大，那么最好使用多层代理。

图 14-3. 通过第二层代理汇聚来自第一层的 Flume 事件

要构建分层的结构，需要使用某种特殊的 sink 来通过网络发送事件，再加上相应的 source 来接收这些事件。Avro sink 可通过 Avro RPC 将事件发送给运行在另一个 Flume 代理上的其他的 Avro。另外也可以使用 Thrift sink，它通过 Thrift RPC 做同样的事情，并搭配了一个 Thrift source。[①]

不要被命名所迷惑：Avro 的 sink 和 source 不提供写入或读取 Avro 文件的能力，它们仅用于代理层的事件分发，并且为了做到这一点，它们需要通过 Avro RPC 进行通信(并因此而得名)。若想将事件写入 Avro 文件，应当使用 HDFS sink，如 14.4.2 节所述。

范例 14-4 描绘了一个两层 Flume 代理的配置。这个配置文件定义了两个代理，`agent1` 和 `agent2`。运行在第一层的代理称为 `agent1`，它具有 `spooldir` source 和

① Avro 的 sink-source 对比 Thrift 的 sink-source 对出现得更早，在写本书的时候，Thrift 的 sink-source 对还无法提供某些功能，例如加密。

Avro sink，并通过 file channel 连接。运行在第二层的代理是 agent2，它具有 Avro source，用于监听从 agent1 的 Avro sink 发送过来的事件所抵达的端口。Agent2 的 sink 所使用的配置与范例 14-2 中的 HDFS sink 相同。

请注意，由于在同一台机器上运行了两个 file channel，因此必须通过配置使得它们分别指向不同的数据以及检查点目录(默认情况下都在用户的主目录中)。只有这样做，两个 channel 的文件才不会彼此重叠。

范例 14-3. 使用 spooling directory source 和 HDFS sink 的两层 Flume 代理的配置

```
# First-tier agent

agent1.sources = source1
agent1.sinks = sink1
agent1.channels = channel1

agent1.sources.source1.channels = channel1
agent1.sinks.sink1.channel = channel1

agent1.sources.source1.type = spooldir
agent1.sources.source1.spoolDir = /tmp/spooldir

agent1.sinks.sink1.type = avro
agent1.sinks.sink1.hostname = localhost
agent1.sinks.sink1.port = 10000

agent1.channels.channel1.type = file
agent1.channels.channel1.checkpointDir=/tmp/agent1/file-channel/checkpoint
agent1.channels.channel1.dataDirs=/tmp/agent1/file-channel/data

# Second-tier agent

agent2.sources = source2
agent2.sinks = sink2
agent2.channels = channel2

agent2.sources.source2.channels = channel2
agent2.sinks.sink2.channel = channel2

agent2.sources.source2.type = avro
agent2.sources.source2.bind = localhost
agent2.sources.source2.port = 10000

agent2.sinks.sink2.type = hdfs
agent2.sinks.sink2.hdfs.path = /tmp/flume
agent2.sinks.sink2.hdfs.filePrefix = events
agent2.sinks.sink2.hdfs.fileSuffix = .log
agent2.sinks.sink2.hdfs.fileType = DataStream

agent2.channels.channel2.type = file
agent2.channels.channel2.checkpointDir=/tmp/agent2/file-channel/checkpoint
```

```
agent2.channels.channel2.dataDirs=/tmp/agent2/file-channel/data
```

该系统如图 14-4 所示。

图 14-4. 由 Avro sink-source 对连接的一个两层 Flume 代理

这两个代理需要分别运行，它们用的`--conf-file` 参数相同，但`--name` 参数不同，分别为

```
% flume-ng agent --conf-file spool-to-hdfs-tiered.properties --name agent1 ...
```

以及

```
% flume-ng agent --conf-file spool-to-hdfs-tiered.properties --name agent2 ...
```

交付保证

Flume 利用事务来确保每一批事件都能可靠地从 source 传递至 channel，以及从 channel 传递至 sink。在 Avro sink-source 连接的背景下，事务可用于确保将事件可靠地从一个代理传递到下一个代理。

在 `agent1` 中，Avro sink 从 file channel 读取一批事件的操作被封装在一个事务中。只有当 Avro sink 收到确认，表示它到 `agent2` 的 Avro source RPC 端点的写操作成功时该事务才会提交，而只有当 `agent2` 将事件批量写入其 file channel 操作的事务被成功提交后才会发送该确认。因此，Avro 的 sink-source 对可以保证事件

从一个 Flume 代理的 channel 成功传递到另一个 Flume 代理的 channel(至少一次)。

假如其中任何一个代理停止运行，显然这些事件将无法顺利传递到 HDFS。例如，若是 agent1 停止运行，那么文件将会积压在缓冲目录中，以等待 agent1 重新启动并处理。此外，在代理停止运行的那一刻，它自己的 file channel 中的所有事件在代理重新启动后仍然可用，因为 file channel 可以提供持久存储保证。

如果 agent2 停止运行，事件将被保留在 agent1 的 file channel 中，直至 agent2 重新启动。但是请注意，channel 的容量必然是有限的，如果 agent1 的 channel 已填满而 agent2 还没有恢复运行，那么任何新事件都将丢失。在默认情况下，file channel 能够恢复的事件数量不超过一百万条(可以通过 capacity 属性进行设置)，另外，当检查点目录中的可用磁盘空间低于 500 MB 时，也将停止接受事件(由 minimumRequiredSpace 属性控制)。

上述两种情况都假设代理最终能够恢复运行，但事实并非总是如此(例如正在运行的硬件出现了故障)。如果 agent1 无法恢复运行，损失仅限于其 file channel 中保存的那些在 agent1 停机前尚未来得及交付给 agent2 的事件。就此处所描述的架构而言，由于第一层有多个类似于 agent1 的代理，因此，可以利用这一层中的其他节点来接管故障节点。例如，如果该节点运行了负载均衡的 Web 服务器，那么其他节点将会消化掉故障 Web 服务器上的通信量，并且它们将生成新的 Flume 事件传递到 agent2。因此，没有新的事件会丢失。

但是，如果 agent2 无法恢复运行，带来的后果要严重得多。对上游的第一层代理(以 agent1 为例)而言，channel 中的所有事件都将丢失，并且产生的新事件也无法传递。这个问题的解决方案是让 agent1 具有多个冗余的 Avro sink，并把这些 sink 安排在同一个 sink 组中。因此，如果目标 agent2 的 Avro 端点不可用，agent1 还可以尝试使用同一个 sink 组中的其他 sink。我们将在下一小节讨论 sink 组。

14.7 Sink 组

sink 组允许将多个 sink 当作一个 sink 来处理，以实现故障转移或负载均衡(参见图 14-5)。若某个第二层代理不可用，事件将被传递给另一个第二层代理，从而使这些事件不中断地到达 HDFS。

图 14-5. 为了负载均衡或故障转移而使用多个 sink

要想配置 sink 组，首先要把代理的 `sinkgroups` 属性设置为该 sink 组的名称，然后为 sink 组列出组中的所有 sink 以及 sink 处理器的类型。Sink 处理器的类型用于设置 sink 的选择策略。范例 14-5 给出的是在两个 Avro 端点之间实现负载均衡的配置。

范例 14-5. 利用 sink 组在两个 Avro 端点之间实现负载均衡的 Flume 配置

```
agent1.sources = source1
agent1.sinks = sink1a sink1b
agent1.sinkgroups = sinkgroup1
agent1.channels = channel1

agent1.sources.source1.channels = channel1
agent1.sinks.sink1a.channel = channel1
agent1.sinks.sink1b.channel = channel1

agent1.sinkgroups.sinkgroup1.sinks = sink1a sink1b
agent1.sinkgroups.sinkgroup1.processor.type = load_balance
agent1.sinkgroups.sinkgroup1.processor.backoff = true

agent1.sources.source1.type = spooldir
agent1.sources.source1.spoolDir = /tmp/spooldir

agent1.sinks.sink1a.type = avro
agent1.sinks.sink1a.hostname = localhost
agent1.sinks.sink1a.port = 10000

agent1.sinks.sink1b.type = avro
agent1.sinks.sink1b.hostname = localhost
```

```
agent1.sinks.sink1b.port = 10001

agent1.channels.channel1.type = file
```

这个例子定义了两个 Avro sink：sink1a 和 sink1b，它们之间的区别仅在于所连接的 Avro 端点不同(由于我们是在本地主机上运行所有实例，所以这个区别体现在端口上，对分布式结构来说，是两个主机不同而端口相同)。我们还定义了 sinkgroup1，并设置 sink1a 和 sink1b 作为它的 sink 组成员。

处理器类型设置为 load_balance，它试图使用循环选择机制在组中的两个 sink 成员之间分发事件流(你可以通过 processor.selector 属性来改变此设置)。如果某个 sink 不可用，那么就尝试下一个 sink，如果两个 sink 都不可用，事件也不会从 channel 中移除，就像在单个 sink 的情况下一样。默认情况下，sink 处理器不会记住 sink 的不可用性，所以每一批被传递的事件都会重试故障的 sink。这样做有可能使效率变低，所以我们可以通过设置 processor.backoff 属性来改变这种行为，让故障 sink 在指数增加超时周期内被列入黑名单(最长周期可达 30 秒，由 processor.selector.maxTimeOut 属性控制)。

还有另一种类型的处理器：failover。这种处理器不是在 sink 之间进行事件负载均衡处理，而是在优选 sink 可用的情况下一直使用优选 sink，而当优选 sink 故障时则切换到另一个 sink。Failover 类型的处理器需要为 sink 组维护其 sink 成员的优先顺序，并按照优先顺序来尝试传递。如果具有最高优先级的 sink 不可用，那么就尝试下一个最高优先级的 sink，以此类推。故障的 sink 在不断增加的超时周期内被列入黑名单(最长周期可达 30 秒，由 processor.maxpenalty 属性控制)。

范例 14-6 给出了作为第二层代理的 agent2a 的配置。

范例 14-6. 在负载均衡的背景下的一个第二层代理的 Flume 配置

```
agent2a.sources = source2a
agent2a.sinks = sink2a
agent2a.channels = channel2a

agent2a.sources.source2a.channels = channel2a
agent2a.sinks.sink2a.channel = channel2a

agent2a.sources.source2a.type = avro
agent2a.sources.source2a.bind = localhost
agent2a.sources.source2a.port = 10000

agent2a.sinks.sink2a.type = hdfs
agent2a.sinks.sink2a.hdfs.path = /tmp/flume
agent2a.sinks.sink2a.hdfs.filePrefix = events-a
```

```
agent2a.sinks.sink2a.hdfs.fileSuffix = .log
agent2a.sinks.sink2a.hdfs.fileType = DataStream

agent2a.channels.channel2a.type = file
```

Agent2b 的配置基本相同，除了 Avro source 端口(因为我们是在本地主机上运行这个例子)以及 HDFS sink 创建的文件名的前缀不同之外。文件名前缀用于确保多个第二层代理同时创建的 HDFS 文件不会出现重名。

通常，Flume 代理运行在不同的机器上，在这种情况下，可以利用主机名来让文件名唯一，只需要配置一个主机拦截器(参见表 14-1)，并在文件路径中包含转义字符序列%{host}：

```
agent2.sinks.sink2.hdfs.filePrefix = events-%{host}
```

图 14-6 给出了整个系统的示意图。

图 14-6. 在两个代理之间实现负载均衡

14.8 Flume 与应用程序的集成

既然 Avro source 用来接收 Flume 事件的一个 RPC 端点，那么就有可能编写 RPC 客户端程序将事件发送至该端点，并且这个客户端可以嵌入到任何一个希望向 Flume 导入事件的应用程序中。

Flume SDK 模块提供的 Java `RpcClient` 类被用来发送 `Event` 对象到一个 Avro 端点(即运行在 Flume 代理上的一个 Avro source，通常这个 Flume 代理位于另一层)。通过配置，客户端程序可以在端点之间实现故障转移或负载均衡，并且也能支持 Thrift 端点(Thrift sources)。

Flume **嵌入式代理**(embedded agent)也提供了类似功能，它是一个运行在 Java 应用程序中的精简版的 Flume 代理。嵌入式代理有一个特殊的 source，应用程序通过调用 `EmbeddedAgent` 对象中的方法将 Flume 事件对象发送到该 source。它唯一支持的 sink 是 Avro sink，不过，我们可以通过配置使用多个 sink 来实现故障转移或负载均衡。

有关 Flume SDK 和嵌入式代理的更详细的描述，可以参考 Flume 开发者指南，网址为 *http://flume.apache.org/FlumeDeveloperGuide.html*。

14.9 组件编目

我们在本章只使用了少量的 Flume 组件。Flume 组件很多，表 14-1 对它们进行了简单介绍。关于这些组件的配置及使用的更多信息，请参考 Flume 用户指南，网址为 *http://flume.apache.org/FlumeUserGuide.html*。

表 14-1. Flume 组件

类别	组件	描述
Source	Avro	监听由 Avro sink 或 Flume SDK 通过 Avro RPC 发送的事件所抵达的端口
	Exec	运行一个 Unix 命令(例如 tail -F/path/to/file)，并把从标准输出上读取的行转换为事件。请注意，此 source 不能保证事件被传递到 channel，更好的选择可以参考 spooling directory source 或 Flume SDK
	HTTP	监听一个端口，并使用可插拔句柄(例如，JSON 处理程序或二进制数据处理程序)把 HTTP 请求转为事件

续表

类别	组件	描述
Source	JMS	读取来自 JMS Queue 或 Topic 的消息并将其转为事件
	Netcat	监听一个端口，并把每行文本转换为一个事件
	Sequence generator	依据增量计数器来生成事件。对测试有用
	Spooling directory	按行来读取保存在文件缓冲目录中的文件，并将其转换为事件
	Syslog	从日志中读取行，并将其转换为事件
	Thrift	监听由 Thrift sink 或 Flume SDK 通过 Thrift RPC 发送的事件所抵达的端口
	Twitter	连接 Twitter 的 streaming API(firehose 的 1%)，并将 tweet 转为事件
Sink	Avro	通过 Avro RPC 发送事件到一个 Avro source
	Elasticsearch	使用 Logstash 格式将事件写到 Elasticsearch 集群
	File roll	将事件写到本地文件系统
	HBase	使用某种序列化工具将事件写到 HBase
	HDFS	以文本、序列文件、Avro 或定制格式将事件写到 HDFS
	IRC	将事件发送给 IRC 通道
	Logger	使用 SLF4J 记录 INFO 级别的事件。对测试有用
	Morphline (Solr)	通过进程内的 Morphline 命令链来运行事件。通常用于将数据加载到 Solr
	Null	丢弃所有事件
	Thrift	通过 Thrift RPC 发送事件到 Thrift source
Channel	File	将事件存储在一个本地文件系统上的事务日志中
	JDBC	将事件存储在数据库中(嵌入式 Derby)
	Memory	将事件存储在一个内存队列中
Interceptor	Host	在所有事件上设置一个包含代理主机名或 IP 地址的主机 header
	Morphline	通过一个 Morphline 配置文件来过滤事件。用于有条件地丢弃事件或者基于模式匹配地添加 header 或内容提取
	Regex extractor	使用指定的正则表达式来设置从事件 body 中以文本形式提取的 header
	Regex filtering	通过将事件 body 以文本形式与指定的正则表达式进行匹配来决定包括或是不包括事件
	Static	在所有事件上设置一个固定的 header 及其值

续表

类别	组件	描述
Interceptor	Timestamp	设置一个时间戳 header，其中包含的是代理处理事件的时间，以毫秒为单位
	UUID	在所有事件上设置一个 ID header，它所包含的是一个全局唯一标识符。对将来删除重复数据有用

14.10 延伸阅读

本章简单概述了 Flume。更多细节可参考 O'Reilly 2014 年出版的 *Using Flume* (*http://shop.oreilly.com/product/0636920030348.do*)，作者 Hari Shreedharan。另外，O'Reilly 2014 年出版的 *Hadoop Application Architectures* 也有很多关于设计导入管道(以及构建普通 Hadoop 应用程序)的实用信息，作者为 Mark Grover、Ted Malaska、Jonathan Seidman 和 Gwen Shapira，网址为 *http://shop.oreilly.com/product/0636920033196.do*。

第 15 章

关于 Sqoop

(作者：Aaron Kimball)

Hadoop 平台的最大优势在于它支持使用不同形式的数据。HDFS 能够可靠地存储日志和来自不同渠道的其他数据，MapReduce 程序能够解析多种“特定的”(ad hoc)数据格式，抽取相关信息并将多个数据集组合成非常有用的结果。

但是为了能够和 HDFS 之外的数据存储库进行交互，MapReduce 程序需要使用外部 API 来访问数据。通常，一个组织中有价值的数据都存储在关系型数据库系统(RDBMS)等结构化存储器中。Apache Sqoop(*http://sqoop.apache.org/*)是一个开源工具，它允许用户将数据从结构化存储器抽取到 Hadoop 中，用于进一步的处理。抽取出的数据可以被 MapReduce 程序使用，也可以被其他类似于 Hive 的工具使用。(甚至可以使用 Sqoop 将数据从数据库转移到 HBase。)一旦生成最终的分析结果，Sqoop 便可以将这些结果导回数据存储器，供其他客户端使用。

在本章中，我们将了解 Sqoop 是如何工作的，学习如何在数据处理过程中使用它。

15.1 获取 Sqoop

在几个地方都可以获得 Sqoop。该项目的主要位置是在 Apache 软件基金会(*http://sqoop.apache.org/*)，这里有 Sqoop 的所有源代码和文档。在这个站点可以获得 Sqoop 的官方版本和当前正在开发的新版本的源代码，这里同时还提供项目编译说明。另外，也可从 Hadoop 供应商的发行版中获得 Sqoop。

如果是从 Apache 下载的版本，它将被放在一个类似于*/home/yourname/sqoop-x.y.z/*的目录中。我们称这个目录为$SQOOP_HOME。可以通过运行可执行脚本$SQOOP_HOME/bin/sqoop来启动 Sqoop。

如果使用供应商的版本，那么安装包会把 Sqoop 的脚本放在类似于*/usr/bin/sqoop*的标准位置。可以通过在命令行上简单地键入 sqoop 来运行它。无论通过何种方式安装了 Sqoop，从现在起，我们都用执行 sqoop 脚本来表示运行它。

Sqoop 2

Sqoop 2 对 Sqoop 进行了重写，以解决 Sqoop 1 架构上的局限性。例如，Sqoop1 是命令行工具，不提供 Java API，因此很难嵌入到其他程序中。另外，Sqoop 1 的所有连接器都必须掌握所有输出格式，为此编写新的连接器就需要做大量工作。Sqoop 2 具有用以运行作业的服务器组件和一整套客户端，包括命令行接口(CLI)、网站用户界面、REST API 和 Java API。Sqoop 2 还能使用其他执行引擎，例如 Spark。注意，Sqoop 2 的 CLI 与 Sqoop 1 的 CLI 并不兼容。

Sqoop 1 是目前比较稳定的发布版本，本章使用的也是 Sqoop 1。Sqoop 2 正处于开发期间，可能并不具备 Sqoop 1 的所有功能，因此，如果你想要在自己的产品中使用 Sqoop 2，首先应当检查一下它是否支持你的用例。

不带参数运行 Sqoop 是没有什么意义的：

```
% sqoop
Try sqoop help for usage.
```

Sqoop 组织成一组工具或命令。不选择工具，Sqoop 便无所适从。help 是其中一个工具的名称，它能够打印出可用工具的列表，如下所示：

```
% sqoop help
usage: sqoop COMMAND [ARGS]

Available commands:
  codegen            Generate code to interact with database records
  create-hive-table  Import a table definition into Hive
  eval               Evaluate a SQL statement and display the results
  export             Export an HDFS directory to a database table
  help               List available commands
  import             Import a table from a database to HDFS
  import-all-tables  Import tables from a database to HDFS
  job                Work with saved jobs
  list-databases     List available databases on a server
  list-tables        List available tables in a database
```

```
  merge                    Merge results of incremental imports
  metastore                Run a standalone Sqoop metastore
  version                  Display version information

See 'sqoop help COMMAND' for information on a specific command.
```

根据它的解释，通过将特定工具的名称作为参数，help 还能提供对该工具的使用说明：

```
% sqoop help import
usage: sqoop import [GENERIC-ARGS] [TOOL-ARGS]

Common arguments:
  --connect <jdbc-uri>  Specify JDBC connect string
  --driver <class-name>  Manually specify JDBC driver class to use
  --hadoop-home <dir>   Override $HADOOP_HOME
  --help                Print usage instructions
-P        Read password from console
  --password <password> Set authentication password
  --username <username> Set authentication username
  --verbose                  Print more information while working
...
```

运行 Sqoop 工具的另外一种方法是使用与之对应的特定脚本。这样的脚本一般被命名为 *sqoop-toolname*，例如，*sqoop-help* 和 *sqoop-import* 等。运行这两个脚本与运行 sqoop help 或 sqoop import 命令是一样的。

15.2 Sqoop 连接器

Sqoop 拥有一个可扩展的框架，使得它能够从(向)任何支持批量数据传输的外部存储系统导入(导出)数据。一个 Sqoop 连接器(connector)就是这个框架下的一个模块化组件，用于支持 Sqoop 的导入和导出操作。Sqoop 附带的连接器能够支持大多数常用的关系型数据库系统，包括 MySQL、PostgreSQL、Oracle、SQL Server、DB2 和 Netezza。同时还有一个通用的 JDBC 连接器，用于连接支持 JDBC 协议的数据库。Sqoop 所提供的 MySQL、PostgreSQL、Oracle 和 Netezza 连接器都是经过优化的，通过使用数据库特定的 API 来提供高效率的批量数据传输(第 15.5.4 小节中将会进行详细介绍)。

除了内置的 Sqoop 连接器外，还有很多针对各种数据存储器的第三方连接器可用，能够支持对企业级数据仓库(例如 Teradata)和 NoSQL 存储器(例如 Couchbase)的连接。这些连接器必须另外单独下载，可以根据连接器所附带的安装说明将其添加到已有的 Sqoop 安装中。

15.3 一个导入的例子

在安装了 Sqoop 之后，可以用它将数据导入(import)到 Hadoop。在本章的所有示例中，我们都使用了支持多种平台的易用数据库系统 MySQL 作为外部数据源。

安装和配置 MySQL 时，可以参考在线文档(*http://dev.mysql.com/doc*)，特别是其中的第 2 章应该很有帮助。基于 Debian 的 Linux 系统(如 Ubuntu)的用户可以通过键入 `sudo apt-get install mysql-client mysql-server` 进行安装；RedHat 的用户可以通过键入 `sudo yum install mysql mysql-server` 进行安装。

现在，MySQL 已经安装好，让我们先登录，然后创建一个数据库(范例 15-1)。

范例 15-1. 创建一个新的 MySQL 数据库模式

```
% mysql -u root -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 235
Server version: 5.6.21 MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.

mysql>CREATE DATABASE hadoopguide;
Query OK, 1 row affected (0.00 sec)

mysql>GRANT ALL PRIVILEGES ON hadoopguide.* TO '%'@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql>quit;
Bye
```

前面的密码提示要求输入 root 用户的密码，就像 root 用户通过密码进行 shell 登录一样。如果你正在使用 Ubuntu 或其他不支持 root 直接登录的 Linux 系统，则键入安装 MySQL 时设定的密码。如果没有设置密码，就直接键入回车。

在这个会话中，我们创建了一个新的名为 `hadoopguide` 的数据库模式，本章将一直使用这个数据库模式，然后我们允许所有本地用户查看和修改 `hadoopguide` 模式的内容，最后，关闭这个会话。[①]

现在让我们登录到数据库(这次不是作为 root，而是你自己)，然后创建一个将被导

① 当然，在生产环境中，我们需要更谨慎地对待访问控制，而这里只是用于演示的目的。上述的授权也假设你正在使用一个伪分布式 Hadoop 实例。如果使用的是一个真正的分布式 Hadoop 集群，至少需要一个启用了远程访问的用户，他的账户用于通过 Sqoop 执行导入和导出。

入 HDFS 的表(范例 15-2)。

范例 15-2. 填充数据库

```
% mysql hadoopguide
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 257
Server version: 5.6.21 MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> CREATE TABLE widgets(id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    -> widget_name VARCHAR(64) NOT NULL,
    -> price DECIMAL(10,2),
    -> design_date DATE,
    -> version INT,
    -> design_comment VARCHAR(100));
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO widgets VALUES (NULL, 'sprocket', 0.25, '2010-02-10',
    ->1, 'Connects two gizmos');
Query OK, 1 row affected (0.00 sec)

mysql>INSERT INTO widgets VALUES (NULL, 'gizmo', 4.00, '2009-11-30', 4,
    ->NULL);
Query OK, 1 row affected (0.00 sec)

mysql>INSERT INTO widgets VALUES (NULL, 'gadget', 99.99, '1983-08-13',
    -> 13, 'Our flagship product');
Query OK, 1 row affected (0.00 sec)

mysql> quit;
```

在前面的示例中，我们创建了一个名为 widgets 的新表。在本章后面的例子中我们都将使用这个虚构的产品数据库。widgets 表有几个不同数据类型的字段。

在进行下一步之前，还需要下载 MySQL 的 JDBC 驱动的 JAR 文件(Connector/J)，并将其存放在 Sqoop 的 lib 目录下，从而使之被添加到 Sqoop 的类路径中。

现在让我们使用 Sqoop 将这个表导入 HDFS：

```
% sqoop import --connect jdbc:mysql://localhost/hadoopguide \
> --table widgets -m 1
...
14/10/28 21:36:23 INFO tool.CodeGenTool: Beginning code generation
...
14/10/28 21:36:28 INFO mapreduce.Job: Running job: job_1413746845532_0008
14/10/28 21:36:35 INFO mapreduce.Job: Job job_1413746845532_0008 running in
uber mode : false
14/10/28 21:36:35 INFO mapreduce.Job: map 0% reduce 0%
14/10/28 21:36:41 INFO mapreduce.Job: map 100% reduce 0%
```

```
14/10/28 21:36:41 INFO mapreduce.Job: Job job_1413746845532_0008 completed
successfully
...
14/10/28 21:36:41 INFO mapreduce.ImportJobBase: Retrieved 3 records.
```

Sqoop 的 import 工具会运行一个 MapReduce 作业，该作业会连接 MySQL 数据库并读取表中的数据。默认情况下，该作业会并行使用 4 个 map 任务来加速导入过程。每个任务都会将其所导入的数据写到一个单独的文件，但所有 4 个文件都位于同一个目录中。在本例中，由于我们知道只有三行可以导入的数据，因此指定 Sqoop 只使用一个 map 任务(-m 1)，这样我们就只得到一个保存在 HDFS 中的文件。

我们可以检查这个文件的内容，如下所示：

```
% hadoop fs -cat widgets/part-m-00000
1,sprocket,0.25,2010-02-10,1,Connects two gizmos
2,gizmo,4.00,2009-11-30,4,null
3,gadget,99.99,1983-08-13,13,Our flagship product
```

本例中使用的连接字符串(*jdbc:mysql://localhost/hadoopguide*)表明需要从本地机器上的数据库中读取数据。如果使用了分布式 Hadoop 集群，则连接字符串中不能使用 localhost，否则与数据库不在同一台机器上运行的 map 任务都将无法连接到数据库。即使是从数据库服务器所在主机上运行 Sqoop，也需要为数据库服务器指定完整的主机名。

在默认情况下，Sqoop 会将我们导入的数据保存为逗号分隔的文本文件。如果导入数据的字段内容中存在分隔符，我们可以另外指定分隔符、字段包围字符和转义字符。使用命令行参数可以指定分隔符、文件格式、压缩方式以及对导入过程进行更细粒度的控制，Sqoop 自带的《Sqoop 使用指南》[①]或 Sqoop 的在线帮助(sqoop help import，或者在 CDH 中使用 man sqoop-import)找到对应于相关参数的描述。

文本和二进制文件格式

Sqoop 可以将数据导入成几种不同的格式。文本文件(默认)是一种人类可读的数据表示形式，并且是平台独立和最简单的数据格式。但文本文件不能保存二进制字段(例如数据库中类型为 VARBINARY 的列)，并且在区分 null 值和字符串 null 时可能会出现问题(尽管使用--null-string 选项可以控制空值的表示方式)。

① 可以从 Apache 软件基金会网站(*http://sqoop.apache.org/*)获取。

为了处理这些情况，应该使用 Sqoop 的 SequenceFile 格式、Avro 格式或 Parquet 文件。这些二进制格式能够为导入的数据提供最精确的表示方式，同时还允许对数据进行压缩，并支持 MapReduce 并行处理同一文件的不同部分。然而，Sqoop 的目前版本还不能将 Avro 或 SequenceFile 文件加载到 Hive 中(尽管可以手动地 Avro 数据文件加载到 Hive 中，而 Parquet 则可以通过 Sqoop 直接加载到 Hive 中)。SequenceFile 文件格式的另一个缺点是它只支持 Java 语言，而 Avro 和 Parquet 数据文件却可以被很多种语言处理。

15.4 生成代码

除了能够将数据库表的内容写到 HDFS，Sqoop 同时还生成了一个 Java 源文件(widgets.java)，保存在当前的本地目录中。在运行了前面的 sqoop import 命令之后，可以通过 ls widgets.java 命令看到这个文件。

在 15.5 节中，我们将看到 Sqoop 在把源数据库的表数据写到 HDFS 之前，会首先用生成的代码对其进行反序列化。

生成的类(widgets)中能够保存一条从被导入表中取出的记录。该类可以在 MapReduce 中使用这条记录，也可以将这条记录保存在 HDFS 中的一个 SequenceFile 文件中。在导入过程中，由 Sqoop 生成的类会将每一条被导入的行保存在 SequenceFile 文件的键-值对格式中“值”的位置上。

也许你不想将生成的类命名为 widgets，因为每一个类的实例只对应于一条记录。我们可以使用另外一个 Sqoop 工具来生成源代码，但并不真正执行导入操作；但生成的代码仍然会检查数据库表，以确定与每个字段相匹配的数据类型：

```
% sqoop codegen --connect jdbc:mysql://localhost/hadoopguide \
>--table widgets --class-name Widget
```

codegen 工具只是简单地生成代码，它不执行完整的导入操作。我们指定生成一个名为 Widget 的类，这个类将被写到 Widget.java 文件中。在之前执行的导入过程中，我们也可以指定类名--class-name 和其他代码生成参数。如果你意外地删除了生成的源代码，或希望使用不同于导入过程的设定来生成代码，都可以用这个工具来重新生成代码。

如果计划使用导入到 SequenceFile 文件中的记录，你将不可避免地用到生成的类(对 SequenceFile 文件中的数据进行反序列化)。在使用文本文件中的记录时不

需要用到生成的代码，但在 15.6 节中，我们将看到 Sqoop 生成的代码有助于解决数据处理过程中的一些繁琐问题。

其他序列化系统

最近的 Sqoop 版本支持基于 Avro 的序列化和模式生成(参见第 12 章)，允许在项目中使用 Sqoop，而无须集成生成的代码。

15.5 深入了解数据库导入

如前所述，Sqoop 是通过一个 MapReduce 作业从数据库中导入一个表，这个作业从表中抽取一行行记录，然后将记录写入 HDFS。MapReduce 是如何读取记录的呢？本节将解释 Sqoop 的底层工作机理。

图 15-1 粗略演示了 Sqoop 是如何与源数据库及 Hadoop 进行交互的。像 Hadoop 一样，Sqoop 是用 Java 语言编写的。Java 提供了一个称为 JDBC(Java Database Connectivity)的 API，应用程序可以使用这个 API 来访问存储在 RDBMS 中的数据，同时可以检查数据类型。大多数数据库厂商都提供了 JDBC 驱动程序，其中实现了 JDBC API 并包含用于连接其数据库服务器的必要代码。

图 15-1. Sqoop 的导入过程

根据用于访问数据库的连接字符串中的 URL，Sqoop 尝试预测应该加载哪个驱动程序，而你要事先下载所需的 JDBC 驱动并且将其安装在 Sqoop 客户端。在 Sqoop 无法判定使用哪个 JDBC 驱动程序时，用户可以明确指定如何将 JDBC 驱动程序加载到 Sqoop，这样一来，Sqoop 就能够支持大多数的数据库平台。

在导入开始之前，Sqoop 使用 JDBC 来检查将要导入的表。它检索出表中所有的列以及列的 SQL 数据类型。这些 SQL 类型(`VARCHAR`、`INTEGER` 等)被映射为 Java 数据类型(`String`、`Integer` 等)，在 MapReduce 应用中将使用这些对应的 Java 类型来保存字段的值。Sqoop 的代码生成器使用这些信息来创建对应表的类，用于保存从表中抽取的记录。

例如，之前提到的 `Widget` 类包含下列方法，这些方法用于从抽取的记录中检索所有的列：

```
public Integer get_id();
public String get_widget_name();
public java.math.BigDecimal get_price();
public java.sql.Date get_design_date();
public Integer get_version();
public String get_design_comment();
```

不过，对于导入来说，更关键的是 `DBWritable` 接口的序列化方法，这些方法能使 `Widget` 类和 JDBC 进行交互：

```
public void readFields(ResultSet __dbResults) throws SQLException;
public void write(PreparedStatement __dbStmt) throws SQLException;
```

JDBC 的 `ResultSet` 接口提供了一个用于从查询结果中检索记录的游标；这里的 `readFields()`方法将用 `ResultSet` 中一行数据的列来填充 `Widget` 对象的字段。这里的 `write()`方法允许 Sqoop 将新的 `Widget` 行插入表，这个过程称为**导出**(exporting)。15.8 节将对此进行讨论。

Sqoop 启动的 MapReduce 作业用到一个 `InputFormat`，它可以通过 JDBC 从一个数据库表中读取部分内容。Hadoop 提供的 `DataDrivenDBInputFormat` 能够为几个 map 任务对查询结果进行划分。

使用一个简单的查询通常就可以读取一张表的内容，例如：

```
SELECT col1,col2,col3,... FROM tableName
```

但是，为了获得更好的导入性能，人们经常将这样的查询划分到多个节点上执行。查询是根据一个**划分列**(splitting column)来进行划分的。根据表的元数据，

Sqoop 会选择一个合适的列作为划分列(如果主键存在的话，通常是表的主键)。主键列中的最小值和最大值会被读出，与目标任务数一起用来确定每个 map 任务要执行的查询。

例如，假设 widgets 表中有 100 000 条记录，其 id 列的值为 0～99 999。在导入这张表时，Sqoop 会判断出 id 是表的主键列。启动 MapReduce 作业时，用来执行导入的 DataDrivenDBInputFormat 便会发出一条类似于 SELECT MIN(id), MAX(id) FROM widgets 的查询语句。检索出的数据将用于对整个数据集进行划分。假设我们指定并行运行 5 个 map 任务(使用-m 5)，这样便可以确定每个 map 任务要执行的查询分别为：SELECT id, widget_name, ... FROM widgets WHERE id >= 0 AND id < 20000，SELECT id, widget_name, ... FROM widgets WHERE id >= 20000 AND id < 40000，…，以此类推。

划分列的选择是影响并行执行效率的重要因素。如果 id 列的值不是均匀分布的(也许在 id 值 50 000 到 75 000 的范围内没有记录)，那么有一部分 map 任务可能只有很少或没有工作要做，而其他任务则有很多工作要做。在运行一个导入作业时，用户可以指定一个列作为划分列(使用--split-参数)，从而调整作业的划分使其符合数据的真实分布。如果使用-m 1 参数来让一个任务执行导入作业，就不再需要这个划分过程。

在生成反序列化代码和配置 InputFormat 之后，Sqoop 将作业发送到 MapReduce 集群。map 任务执行查询并且将 ResultSet 中的数据反序列化到生成类的实例，这些数据要么直接保存在 SequenceFile 文件中，要么在写到 HDFS 之前被转换成分隔的文本。

15.5.1 导入控制

Sqoop 不需要每次都导入整张表。例如，可以指定仅导入表的部分列。用户也可以在查询中加入 WHERE 子句(使用--where 参数)，以此来限定需要导入的记录。例如，如果上个月已经将 id 为 0～99 999 的记录导入，而本月供应商的产品目录中增加了 1000 种新部件，那么导入时在查询中加入子句 WHERE id >= 100000，就可以实现只导入所有新增的记录。用户提供的 WHERE 子句会在任务分解之前执行，并且被下推至每个任务所执行的查询中。

通过指定--query 参数可以实现更多控制，例如执行列变换。

15.5.2 导入和一致性

在向 HDFS 导入数据时，重要的是要确保访问的是数据源的一致性快照。从一个数据库中并行读取数据的 map 任务分别运行在不同的进程中，因此它们不可能共享同一个数据库事务。保证一致性的最好方法是在导入时不允许运行任何对表中现有数据进行更新的进程。

15.5.3 增量导入

定期运行导入是一种很常见的方式，这样做可以使 HDFS 的数据与数据库的数据保持同步。为此，需要识别哪些是新数据。对于某一行来说，只有当特定列(由`--check-column`参数指定)的值大于指定值(通过`--last-value`设置)时，Sqoop 才会导入该行数据。

通过`--last-value`所指定的值可以是严格递增的行号，例如在 MySQL 中有`AUTO_INCREMENT`属性的主键。这种模式很适用于数据库中的表只有新行添加，而不存在对现有行更新的情况。这称为 append 模式，它通过`--incremental append`来激活。另一种情况是基于时间的增量导入(通过`--incremental lastmodified`激活)，它适用于现有行也有可能被更新的情况，此时需要指定列(通过`--check-column`参数指定)以记录最近一次更新的时间。

增量导入结束时，程序显示在下次导入时将被指定为`--last-value`的值，这对于手工运行的增量导入来说非常重要，但对定期运行的增量导入，最好使用 Sqoop 的 saved job 工具，它可以自动保存最近一次的值并在下次作业运行时使用。要想了解 saved job 的更多用法，可以输入`sqoop job -help`命令。

15.5.4 直接模式导入

Sqoop 的架构支持它在多种可用的导入方法中进行选择，而大多数数据库都使用上述基于`DataDrivenDBInputFormat`的方法。一些数据库提供了能够快速抽取数据的特定工具，例如 MySQL 的`mysqldump`能够以大于 JDBC 的吞吐率从表中读取数据。在 Sqoop 的文档中将这种使用外部工具的方法称为**直接模式**(direct mode)。由于直接模式并不像 JDBC 方法那样通用，(例如，MySQL 的直接模式不能处理大对象数据，类型为`CLOB`或`BLOB`的列，Sqoop 需要使用 JDBC 专用的 API 将这些列载入 HDFS。)所以使用直接模式导入时必须由用户明确地启动(通

过--direct 参数)。

对于那些提供了此类特定工具的数据库，Sqoop 使用这些工具就能够得到很好的效果。采用直接模式从 MySQL 中导入数据通常比基于 JDBC 的导入更加高效(就 map 任务和所需时间而言)。Sqoop 仍然并行启动多个 map 任务，接着这些任务将分别创建 mysqldump 程序的实例并且读取它们的运行结果。Sqoop 也支持采用直接模式从 PostgreSQL、Oracle 和 Netezz 中导入数据。

即使用直接模式来访问数据库的内容，元数据的查询仍然要通过 JDBC 来实现。

15.6 使用导入的数据

一旦数据导入 HDFS，就可以供定制的 MapReduce 程序使用。导入的文本格式数据可以供 Hadoop Streaming 中的脚本或以 TextInputFormat 为默认格式运行的 MapReduce 作业使用。

为了使用导入记录的个别字段，必须对字段分隔符(以及转义/包围字符)进行解析，抽出字段的值并转换为相应的数据类型。例如，在文本文件中，“sprocket” widget 的 id 表示成字符串“1”，但必须被解析为 Java 的 Integer 或 int 类型的变量。Sqoop 生成的表类能够自动完成这个过程，使你可以将精力集中在真正要运行的 MacReduce 作业上。每个自动生成的类都有几个名为 parse()的重载方法，这些方法可以对表示为 Text、CharSequence、char[]或其他常见类型的数据进行操作。

名为 MaxWidgetId 的 MapReduce 应用(在示例代码中)可以找到具有最大 ID 的部件。使用示例代码附带的 Maven POM，这个类可以和 *Widget.java* 一起编译成一个 JAR 文件。该 JAR 文件名为 *sqoop-examples.jar*，并且像下面这样运行：

```
% HADOOP_CLASSPATH=$SQOOP_HOME/sqoop-version.jar hadoop jar \
> sqoop-examples.jar MaxWidgetId -libjars $SQOOP_HOME/sqoop-version.jar
```

MaxWidgetId.run()方法在运行时以及 map 任务在集群上运行时(通过-libjars 参数)，该命令确保了 Sqoop 位于本地的类路径中(通过$HADOOP_CLASSPATH)。

运行之后，HDFS 的 *maxwidgets* 路径中便有一个名为 *part-r-00000* 的文件，其内容如下：

```
3,gadget,99.99,1983-08-13,13,Our flagship product
```

注意，在这个 MapReduce 示例程序中，一个 Widget 对象从 mapper 被发送到 reducer，这个自动生成的 Widget 类实现了 Hadoop 提供的 Writable 接口，该接口允许通过 Hadoop 的序列化机制来发送对象以及写到 SequenceFile 文件或从 SequenceFile 文件读出对象。

这个 MaxWidgetID 的例子建立在新的 MapReduce API 之上。虽然某些高级功能(例如使用大对象数据)只有在新的 API 中使用起来才更方便，但无论新旧 API，都可以用来构建依赖于 Sqoop 生成代码的 MapReduce 应用。

前面 12.7 节介绍了用于处理 Avro 格式导入的 API。采用 Avro 通用映射时，MapReduce 程序不需要使用针对数据表的模式所生成的代码(尽管使用 Avro 的特定编译器时这也是其中一个选项；在这种情况下，Sqoop 不会生成代码)。示例代码中包含有一个名为 MaxWidgetIdGenericAvro 的程序，用于找出具有最大 ID 的部件并将结果写入一个 Avro 数据文件。

导入的数据与 Hive

我们将在第 17 章看到，对于很多类型的分析任务来说，使用类似于 Hive 的系统来处理关系操作有利于加快分析任务的开发。特别是对于那些来自于关系数据源的数据，使用 Hive 是非常有帮助的。Hive 和 Sqoop 共同构成了一个强大的服务于分析任务的工具链。

假设在我们的系统中有另外一组数据记录，来自一个基于 Web 的零部件采购系统。这个系统返回的记录文件中包含部件 ID、数量、送货地址和订单日期。

下面是此类记录的例子：

```
1,15,120 Any St.,Los Angeles,CA,90210,2010-08-01
3,4,120 Any St.,Los Angeles,CA,90210,2010-08-01
2,5,400 Some Pl.,Cupertino,CA,95014,2010-07-30
2,7,88 Mile Rd.,Manhattan,NY,10005,2010-07-18
```

通过使用 Hadoop 来分析这组采购记录，我们可以深入了解我们的销售业务。将这些数据与来自关系数据源(widgets 表)的数据相结合，可以使我们做得更好。在这个例子中，我们将计算哪个邮政编码区域的销售业绩最好，便可以让我们的销售团队更加关注该区域。为了做到这一点，我们同时需要来自销售记录和 widgets 表的数据。

上述销售记录数据保存在一个名为 *sales.log* 的本地文件中。

首先，让我们将销售数据载入 Hive：

```
hive>CREATE TABLE sales(widget_id INT, qty INT,
>street STRING, city STRING, state STRING,
>zip INT, sale_date STRING)
>ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
OK
Time taken: 5.248 seconds
hive> LOAD DATA LOCAL INPATH "ch15-sqoop/sales.log" INTO TABLE sales;
...
Loading data to table default.sales
Table default.sales stats: [numFiles=1, numRows=0, totalSize=189, rawDataSize=0]
OK
Time taken: 0.6 seconds
```

Sqoop 能够根据一个关系数据源中的表来生成一个 Hive 表。既然我们已经将 widgets 表的数据导入到 HDFS，那么我们就直接生成相应 Hive 表的定义，然后加载保存在 HDFS 中的数据：

```
% sqoop create-hive-table --connect jdbc:mysql://localhost/hadoopguide \
> --table widgets --fields-terminated-by ','
...
14/10/29 11:54:52 INFO hive.HiveImport: OK
14/10/29 11:54:52 INFO hive.HiveImport: Time taken: 1.098 seconds
14/10/29 11:54:52 INFO hive.HiveImport: Hive import complete.
% hive
hive> LOAD DATA INPATH "widgets" INTO TABLE widgets;
Loading data to table widgets
OK
Time taken: 3.265 seconds
```

在为一个特定的已导入数据集创建相应的 Hive 表定义时，我们需要指定该数据集所使用的分隔符。否则，Sqoop 将允许 Hive 使用它自己的默认分隔符(与 Sqoop 的默认分隔符不同)。

Hive 的数据类型不如大多数 SQL 系统的丰富。很多 SQL 类型在 Hive 中都没有直接对应的类型。当 Sqoop 为导入操作生成 Hive 表定义时，它会为数据列选择最合适的 Hive 类型，这样可能会导致数据精度的下降。一旦出现这种情况，Sqoop 就会提供一条警告信息，如下所示：

```
14/10/29 11:54:43 WARN hive.TableDefWriter:
Column design_date had to be
cast to a less precise type in Hive
```

如果想直接从数据库将数据导入到 Hive，可以将上述的三个步骤(将数据导入 HDFS；创建 Hive 表；将 HDFS 中的数据导入 Hive)缩短为一个步骤。在进行导入时，Sqoop 可以生成 Hive 表的定义，然后直接将数据导入 Hive 表。如果我们还没

有执行过导入操作，就可以使用下面的命令，根据 MySQL 中的数据直接创建 Hive 中的 widgets 表：

```
% sqoop import --connect jdbc:mysql://localhost/hadoopguide \
>--table widgets -m 1 --hive-import
```

*使用--hive-import 参数来运行 sqoop import 工具，可以从源数据库中直接*将数据载入 Hive；它自动根据源数据库中表的模式来推断 Hive 表的模式。这样，只需要一条命令，你就可以在 Hive 中来使用自己的数据。

无论选择哪一种数据导入的方式，现在我们都可以使用 widgets 数据集和 sales 数据集来计算最赚钱的邮政编码地区。让我们来做这件事，并且把查询的结果保存在另外一张表中，供将来使用：

```
hive> CREATE TABLE zip_profits
> AS
> SELECT SUM(w.price * s.qty) AS sales_vol, s.zip FROM SALES s
> JOIN widgets w ON (s.widget_id = w.id) GROUP BY s.zip;
...
Moving data to: hdfs://localhost/user/hive/warehouse/zip_profits
...
OK
hive> SELECT * FROM zip_profits ORDER BY sales_vol DESC;
...
OK
403.71 90210
28.0 10005
20.0 95014
```

15.7 导入大对象

很多数据库都支持在一个字段中保存大量的数据，根据数据是文本还是二进制类型，通常将其保存在表中 CLOB 或 BLOB 类型的列中。数据库一般会对这些“大对象”进行特殊处理。大多数的表在磁盘上的物理存储都如图 15-2 所示。通过行扫描来确定哪些行匹配特定的查询条件时，通常需要从磁盘上读出每一行的所有列。如果也是以这种方式“内联”(inline)存储大对象，它们会严重影响扫描的性能。因此，一般将大对象与它们的行分开存储，如图 15-3 所示。在访问大对象时，需要通过行中包含的引用来“打开”它。

图 15-2. 数据库通常以行数组的方式来存储表，行中所有列存储在相邻的位置

图 15-3. 大对象通常保存在单独的存储区域，行的主存储区域包含指向大对象的间接引用

在数据库中使用大对象的困难表明，像 Hadoop 这样的系统更适合于处理大型的、复杂的数据对象，也是存储此类信息的理想选择。Sqoop 能够从表中抽取大对象数据，并且将它们保存在 HDFS 中供进一步处理。

同在数据库中一样，MapReduce 在将每条记录传递给 mapper 之前一般要对其进行**物化**(materialize)。如果单条记录真的很大，物化操作将非常低效。

如前所示，Sqoop 所导入的记录在磁盘上的存储格式与数据库的内部数据结构非常相似：将每条记录的所有字段放在一起组成的一个记录数组。在导入的记录上运行一个 MapReduce 程序时，每个 map 任务必须将所读记录的所有字段完全物化。如果 MapReduce 程序所处理的输入记录中仅有很小一部分的大对象字段的内容，那么将所有记录完全物化将导致程序效率低下。此外，从大对象的大小来看，在内存中进行完全物化也许是无法实现的。

为了克服这些困难，当导入的大对象数据大于阈值 16 M 时(通过 sqoop.inline.lob.length.max 设置，以字节为单位)，Sqoop 将导入的大对象数据存储在 LobFile 格式的单独文件中。LobFile 格式能够存储非常大的单条记录(使用了 64 位的地址空间)，每条记录保存一个大对象。LobFile 格式允许客户端持有对记录的引用，而不访问记录内容，对记录的访问是通过 java.io.InputStream(用于二进制对象)或 java.io.Reader(用于字符对象)来实现的。

在导入一条记录时，所有的“正常”字段会在一个文本文件中一起被物化，同时还生成一个指向保存 CLOB 或 BLOB 列的 LobFile 文件的引用。例如，假设我们的 widgets 表有一个名为 schematic 的 BLOB 字段，该字段用于保存每个部件的原理图。

导入的记录可能像下面这样：

```
2,gizmo,4.00,2009-11-30,4,null,externalLob(lf,lobfile0,100,5011714)
```

externalLob(…)是对外部存储大对象的一个引用，这个大对象以 LobFile 格式(lf)存储在 *lobfile0* 文件中，同时还给出了该对象在文件中的字节位移和长度。

在使用这条记录时，Widget.get_schematic()方法会返回一个为 BlobRef 类型的对象，用于引用 schematic 列，但这个对象并不真正包含记录的内容。BlobRef.getDataStream() 方法实际会打开 LobFile 文件并返回一个 InputStream，用于访问 schematic 字段的内容。

在使用一个 MapReduce 作业来处理许多 Widget 记录时，可能你只需要访问少数几条记录的 schematic 字段。单个原理图数据可能有几兆大小或更大，使用这种方式时，只需要承担访问所需大对象的 I/O 开销。

在一个 map 任务中，BlobRef 和 ClobRef 类会缓存对底层 LobFile 文件的引用。如果你访问几个顺序排列记录的 schematic 字段，就可以利用现有文件指针来定位下一条记录。

15.8 执行导出

在 Sqoop 中，导入(import)是指将数据从数据库系统移动到 HDFS。与之相反，导出(export)是将 HDFS 作为数据源，而将一个远程数据库作为目标。在前面的几个

小节中，我们导入了一些数据并且使用 Hive 对数据进行了分析。我们可以将分析的结果导出到一个数据库中，供其他工具使用。

将一张表从 HDFS 导出到数据库时，必须在数据库中创建一张用于接收数据的目标表。虽然 Sqoop 可以推断出哪个 Java 类型适合存储 SQL 数据类型，但反过来却是行不通的(例如，有几种 SQL 列的定义都可以用来存储 Java 的 `String` 类型，如 `CHAR(64)`、`VARCHAR(200)`或其他一些类似定义)。因此，必须由用户来确定哪些类型是最合适的。

我们打算从 Hive 中导出 `zip_profits` 表。首先需要在 MySQL 中创建一个具有相同列顺序及合适 SQL 类型的目标表：

```
% mysql hadoopguide
mysql>CREATE TABLE sales_by_zip (volume DECIMAL(8,2), zip INTEGER);
Query OK, 0 rows affected (0.01 sec)
```

接着我们运行导出命令：

```
% sqoop export --connect jdbc:mysql://localhost/hadoopguide -m 1 \
>--table sales_by_zip --export-dir /user/hive/warehouse/zip_profits \
>--input-fields-terminated-by '\0001'
...
14/10/29 12:05:08 INFO mapreduce.ExportJobBase: Transferred 176 bytes in 13.5373
seconds (13.0011 bytes/sec)
14/10/29 12:05:08 INFO mapreduce.ExportJobBase: Exported 3 records.
```

最后，可以通过检查 MySQL 来确认导出成功：

```
% mysql hadoopguide -e 'SELECT * FROM sales_by_zip'
+--------+-------+
| volume | zip   |
+--------+-------+
| 28.00  | 10005 |
| 403.71 | 90210 |
| 20.00  | 95014 |
+--------+-------+
```

在 Hive 中创建 `zip_profits` 表时，我们没有指定任何分隔符。因此 Hive 使用了自己的默认分隔符：字段之间使用 Ctrl-A 字符(Unicode 编码 `0x0001`)分隔，每条记录末尾使用一个换行符分隔。当我们使用 Hive 来访问这张表的内容时(`SELECT` 语句)，Hive 将数据转换为制表符分隔的形式，用于在控制台上显示。但是直接从文件中读取这张表时，我们要将所使用的分隔符告知 Sqoop。Sqoop 默认记录是以换行符作为分隔符，但还需要将字段分隔符 Ctrl-A 告之 Sqoop。可以在 `sqoop export` 命令中使用`--input-fields-terminated-by` 参数来指定字段分隔符。

Sqoop 在指定分隔符时支持几种转义序列(以字符'\'开始)。

在示例语法中，所用的转义序列被包围在'单引号'中，以确保 shell 会按字面意义处理它。如果不使用引号，前导的反斜杠就需要转义处理(例如，`--input-fields-terminated-by \\0001`)。表 15-1 列出了 Sqoop 所支持的转义序列。

表 15-1. 转义序列可以用于指定非打印字符作为 Sqoop 中字段和记录的分隔符

转义序列	描述
`\b`	退格
`\n`	换行
`\r`	回车
`\t`	制表符
`\'`	单引号
`\"`	双引号
`\\`	反斜杠
`\0`	NUL。用于在字段或行之间插入 NUL 字符或在`--enclosed-by`、`--optionally-enclosed-by`和`--escaped-by`参数中使用时表示禁用包围/转义
`\0ooo`	一个 Unicode 字符代码点的八进制表示，实际字符由八进制值 *ooo* 指定
`\0xhhh`	一个 Unicode 字符代码点的十六进制表示，采用`\0xhhh`的形式，其中 *hhh* 是十六进制值。例如，`--fields-terminated-by '\0x10'`指定的是回车符

15.9 深入了解导出功能

Sqoop 导出功能的架构与其导入功能的非常相似(参见图 15-4)。在执行导出操作之前，Sqoop 会根据数据库连接字符串来选择一个导出方法。对于大多数系统来说，Sqoop 都会选择 JDBC。然后，Sqoop 会根据目标表的定义生成一个 Java 类。这个生成的类能够从文本文件中解析出记录，并能够向表中插入类型合适的值(除了能够读取 `ResultSet` 中的列)。接着，会启动一个 MapReduce 作业，从 HDFS 中读取源数据文件，使用生成的类解析出记录，并且执行选定的导出方法。

基于 JDBC 的导出方法会产生一批 `INSERT` 语句，每条语句会向目标表中插入多条记录。在大部分的数据库系统中，通过一条语句插入多条记录的执行效率要高于多次执行插入单条记录的 `INSERT` 语句。多个独立的线程被用于从 HDFS 读取数据并与数据库进行通信，以确保涉及不同系统的 I/O 操作尽量能够重叠执行。

图 15-4. 使用 MapReduce 并行执行导出

对于 MySQL 数据库来说，Sqoop 可以采取使用 `mysqlimport` 的直接模式方法。每个 map 任务会生成一个 `mysqlimport` 进程，该进程通过本地文件系统上的一个命名 FIFO 通道进行通信，数据通过这个 FIFO 通道流入 `mysqlimport`，然后再被写入数据库。

虽然从 HDFS 读取数据的 MapReduce 作业大多根据所处理文件的数量和大小来选择并行度(map 任务的数量)，但 Sqoop 的导出工具允许用户明确设定任务的数量。由于导出性能会受并行的数据库写入线程数量的影响，所以 Sqoop 使用 `CombineFileInputFormat` 类将输入文件分组分配给少数几个 map 任务去执行。

15.9.1 导出与事务

进程的并行特性决定了导出操作往往不是原子操作。Sqoop 会生成多个并行执行的任务，分别导出数据的一部分。这些任务的完成时间各不相同，即使在每个任务内部都使用事务，不同任务的执行结果也不可能同时提交。此外，数据库系统经常使用固定大小的缓冲区来存储事务数据，这使得一个任务中的所有操作不可能在一个事务中完成。Sqoop 每导入几千条记录便会执行一次提交，以确保不会

出现内存不足的情况。在导出操作进行过程中，提交过的中间结果都是可见的。因此在导出过程完成前，不要启动那些要使用导出结果的应用程序，否则这些应用会看到不完整的导出结果。

为了解决这个问题，Sqoop 可以先将数据导出到一个临时阶段表中，然后在导出任务完成前(即导出操作成功执行后)，在一个事务中将临时阶段表中的数据全部移动到目标表中。可以通过`--staging-table` 选项来指定一个临时阶段表，这个临时阶段表必须是已经存在的，并且和目标表具有相同的模式定义。如果不使用`--clear-staging-table` 选项，则这个临时阶段表必须是空表。

使用临时阶段表会降低执行速度，因为它需要写两次数据：先写入临时阶段表，再写入目标表。整个导出过程运行所使用的空间也更多，因为从临时阶段表向目标表导出时存在数据的两个复本。

15.9.2 导出和 SequenceFile

之前的导出示例是从一个 Hive 表中读取源数据，该 Hive 表以分隔文本文件形式保存在 HDFS 中。Sqoop 也可以从非 Hive 表的分隔文本文件中导出数据。例如，Sqoop 可以导出 MapReduce 作业结果的文本文件。

Sqoop 还可以将存储在 `SequenceFile` 中的记录导出到输出表，不过有一些限制。`SequenceFile` 中不能保存任意类型的记录。Sqoop 的导出工具从 `SequenceFile` 中读取对象后直接发送到 `OutputCollector`，由它将这些对象传递给数据库导出 `OutputFormat`，因此为了能让 Sqoop 使用，记录必须被保存在 `SequenceFile` 键-值对格式的“值”部分，并且必须继承抽象类 `com.cloudera.sqoop.lib.SqoopRecord`(像 Sqoop 生成的所有类那样)。

如果基于导出目标表，使用 codegen 工具(`sqoop-codegen`)为记录生成一个 `SqoopRecord` 的实现，那就可以写一个 MapReduce 程序，填充这个类的实例并将它们写入 `SequenceFile`，接着 `sqoop-export` 就可以将这些 `SequenceFile` 文件导出到表中。还有另外一种方法，即将数据放入 `SqoopRecord` 实例中，然后保存到 `SequenceFile` 中。如果数据是从数据库表导入 HDFS 的，那么在经过某种形式的修改后，可以将结果保存在持有相同数据类型记录的 `SequenceFile` 中。

在这种情况下，Sqoop 应当利用现有的类定义从 `SequenceFile` 中读取数据，而不是像导出文本记录时所做的那样，为执行导出生成一个新的(临时的)记录容器类。通过为 Sqoop 提供`--class-name` 和`--jar-file` 参数，可以禁止它生成代

码，而使用现有的记录类和 jar 包。在导出记录时，Sqoop 将使用指定 jar 包中指定的类。

在下面的例子中，我们将重新导入 widgets 表到 SequenceFile 中，然后再将其导出到另外一个数据库表：

```
% sqoop import --connect jdbc:mysql://localhost/hadoopguide \
>--table widgets -m 1 --class-name WidgetHolder --as-sequencefile \
>--target-dir widget_sequence_files --bindir .
...
14/10/29 12:25:03 INFO mapreduce.ImportJobBase: Retrieved 3 records.

% mysql hadoopguide
mysql>CREATE TABLE widgets2(id INT, widget_name VARCHAR(100),
    ->price DOUBLE, designed DATE, version INT, notes VARCHAR(200));
Query OK, 0 rows affected (0.03 sec)

mysql> exit;

% sqoop export --connect jdbc:mysql://localhost/hadoopguide \
>--table widgets2 -m 1 --class-name WidgetHolder \
>--jar-file WidgetHolder.jar --export-dir widget_sequence_files
...
14/10/29 12:28:17 INFO mapreduce.ExportJobBase: Exported 3 records.
```

在导入过程中，我们指定使用 SequenceFile 格式，并且将 JAR 文件放入当前目录(使用--bindir)，这样将来便可以重用它，否则它将会被保存在一个临时目录中。然后我们创建一个用于导出的目标表，该表在模式上稍有不同(但与源数据兼容)。最后，运行导出操作，使用现有的生成代码从 SequenceFile 中读取记录并将它们写入数据库。

15.10 延伸阅读

有关 Sqoop 的更多使用信息，请参阅 O'Reilly 在 2013 年出版的 *Apache Sqoop Cookbook* (*http://shop.oreilly.com/product/0636920029519.do*)，作者 Kathleen Ting 和 JarekJarcec Cecho。

第 16 章

关于 Pig

Apache Pig(*http://pig.apache.org/*)为大型数据集的处理提供了更高层次的抽象。MapReduce 使作为程序员的你能够自己定义一个 map 函数和一个紧跟其后的 reduce 函数。但是，你必须使数据处理过程与这一连续的 map 和 reduce 模式相匹配。很多时候，数据处理需要多个 MapReduce 过程才能实现。而使得数据处理过程与该模式匹配可能很困难。有了 Pig，就能使用更为丰富的数据结构。这些数据结构往往都是多值和嵌套的。Pig 还提供了一套更强大的数据变换操作，包括在 MapReduce 中被忽视的连接(join)操作。

Pig 包括两部分。

(1) 用于描述数据流的语言，称为 Pig Latin。

(2) 用于运行 Pig Latin 程序的执行环境。当前有两个环境：单 JVM 中的本地执行环境和 Hadoop 集群上的分布式执行环境。

Pig Latin 程序由一系列的“操作”(operation)或“变换”(transformation)组成。每个操作或变换对输入进行数据处理，并产生输出结果。从整体上看，这些操作描述了一个数据流。Pig 执行环境把数据流翻译为可执行的内部表示并运行它。在 Pig 内部，这些变换操作被转换成一系列 MapReduce 作业。但作为程序员，多数情况下你并不需要知道这些转换是如何进行的，如此便可以将精力集中在数据上，而非执行细节上。

Pig 是一种探索大规模数据集的脚本语言。MapReduce 的一个缺点是开发周期太长。写 mapper 和 reducer，对代码进行编译和打包，提交作业，获取结果，这整

个过程非常耗时。即便使用 Streaming 能在这一过程中去除代码的编译和打包步骤，仍不能改善这一情况。Pig 的诱人之处在于仅用控制台上的五六行 Pig Latin 代码就能够处理 TB 级的数据。事实上，正是由于雅虎公司想让科研人员和工程师能够更便捷地挖掘大规模数据集，才设计开发了 Pig。Pig 提供了多个命令来检查和处理程序中已有的数据结构。因此，它能够很好地支持程序员写查询。Pig 的一个更有用的特性是它支持在输入数据的一个有代表性的子集上试运行。这样一来，用户可以在处理整个数据集前检查程序执行时是否会有错误。

Pig 被设计为可扩展的。处理路径中的几乎每个部分，包括载入、存储、过滤、分组、连接都可以定制。这些操作都可以使用用户定义函数(user-defined function，UDF)进行修改。这些用户定义函数作用于 Pig 的嵌套数据模型。因此，它们可以在底层与 Pig 的操作集成。UDF 的另一个好处是，相较于为了写 MapReduce 程序而开发的代码库，它们更易于重用。

在有些情况下，Pig 的表现不如 MapReduce 程序。但随着新版本的发布，Pig 的开发团队使用了复杂、精巧的算法来实现 Pig 的关系型操作，二者的差距在不断缩小。公平地说，除非你愿意花大量时间来优化 Java MapReduce 程序，否则用 Pig Latin 来写查询的确能够帮你节约时间。

16.1 安装与运行 Pig

Pig 是作为一个客户端应用程序运行的。即使准备在 Hadoop 集群上运行 Pig，也无需在集群上额外安装什么东西：Pig 从工作站上发出作业，并在工作站上和 HDFS(或其他 Hadoop 文件系统)进行交互。

Pig 的安装很简单。从 *http://pig.apache.org/releases.html* 下载一个稳定版本，然后把 tar 压缩包解压到工作站上的合适路径：

```
% tar xzf pig-x.y.z.tar.gz
```

把 Pig 的二进制文件路径添加到命令行路径也很方便，例如：

```
% export PIG_HOME=~/sw/pig-x.y.z
% export PATH=$PATH:$PIG_HOME/bin
```

还需要设置 JAVA_HOME 环境变量，指明 Java 的安装路径。

输入 pig -help 可获得使用帮助。

16.1.1 执行类型

Pig 有两种执行类型或称为模式(mode)：本地模式(local mode)和 MapReduce 模式(MapReduce mode)。在本书写作期间，用于 Apache Tez 和 Spark(参见第 19 章)的执行模式尚在开发之中。这两种模式都承诺将在性能上远远超出 MapReduce 模式，因此，如果你正在使用的 Pig 版本中这两种模式是可用的，不妨一试。

1. 本地模式

在本地模式下，Pig 运行在单个 JVM 中，访问本地文件系统。该模式只适用于处理小规模数据集或试用 Pig 时。

执行类型可用`-x`或`-exectype`选项进行设置。如果要使用本地模式运行，应把该选项设置为`local`：

```
% pig -x local
grunt>
```

这样就能够启动 Grunt。Grunt 是与 Pig 进行交互的 shell 环境。稍后我们要对它进行详细讨论。

2. MapReduce 模式

在 MapReduce 模式下，Pig 将查询翻译为 MapReduce 作业，然后在 Hadoop 集群上执行。集群可以是伪分布的，也可以是全分布的。如果要用 Pig 处理大规模数据集，应该使用(全分布集群上的)MapReduce 模式。

要使用 MapReduce 模式，首先需要检查下载的 Pig 版本是否与正在使用的 Hadoop 版本兼容。Pig 发布版本只和特定的 Hadoop 版本对应。发行说明中记录了版本的对应关系。

Pig 根据`HADOOP_HOME`环境变量来寻找并运行对应的 Hadoop 客户端。但是，如果该环境变量没有被设置过，那么 Pig 会运行捆绑在 Pig 中的 Hadoop 库。注意，捆绑的库的版本可能和集群上的 Hadoop 版本不一样。所以，最好明确指定`HADOOP_HOME`。

然后，需要将 Pig 指向集群的 namenode 和资源管理器。如果安装在`HADOOP_HOME`下的 Hadoop 已经进行了设置，那么不需要进行额外的配置。否则，可以把`HADOOP_CONF_DIR`设为包含`fs.defaultFS`、`yarn.resourcemanager.address`和

mapreduce.framework.name 定义的 Hadoop 站点文件(后者应被设置为 yarn)的目录。

另一种办法是，在 Pig 的 *conf* 目录(或由 PIG_CONF_DIR 指定的目录)下的 *pig.properties* 文件中设置这些属性。下面是为一个伪分布集群所做的设置：

```
fs.defaultFS=hdfs://localhost/
mapreduce.framework.name=yarn
yarn.resourcemanager.address=localhost:8032
```

一旦设置好 Pig 到 Hadoop 集群的连接，就可以设置-x 选项为 mapreduce 或忽略该选项来运行 Pig。可以忽略该选项的原因是 MapReduce 模式是 Pig 的默认执行模式。使用-brief 选项可以阻止时间戳的生成：

```
% pig -brief
Logging error messages to: /Users/tom/pig_1414246949680.log
Default bootup file /Users/tom/.pigbootup not found
Connecting to hadoop file system at: hdfs://localhost/
grunt>
```

从输出中可以看到，Pig 报告了它所连接的文件系统(而并非 YARN 资源管理器)。

在 MapReduce 模式下，通过选项设置可启用 *auto-local mode*(将 pig.auto.local.enabled 选项值设置为 true)，当输入数据小于 100 MB 时(通过 pig.auto.local.input.maxbytes 选项设置，默认值为 100 000 000)，运行过程被优化为数个本地运行的小作业，并且所使用的 reducer 的数量不超过 1 个。

16.1.2 运行 Pig 程序

有三种执行 Pig 程序的方法。它们在本地和 MapReduce 模式下都适用。

1. 脚本

Pig 可以运行包含 Pig 命令的脚本文件。例如，pig script.pig 将运行在本地文件 *script.pig* 中的命令，或者，对于很短的脚本，也可以使用-e 选项直接在命令行中以字符串形式输入脚本。

2. Grunt

Grunt 是运行 Pig 命令的交互式 shell 环境。如果没有指明 Pig 要运行的文件，而且

也没有使用-e 选项，Pig 就会启动 Grunt。在 Grunt 环境中，可以通过 run 和 exec 命令运行 Pig 脚本。

3. 嵌入式方法

还可以在 Java 中通过 PigServer 类来运行 Pig 程序。这就像能够在 Java 中使用 JDBC 运行 SQL 程序一样。如果要以编程的方式访问 Grunt，则需要使用 PigRunner。

16.1.3 Grunt

Grunt 包含的行编辑功能类似于 GNU Readline(用在 bash shell 环境等命令行应用中)。例如，按组合键 Ctrl+E 可以将光标移到行末。Grunt 也记录过去执行过的命令。[①]可以使用 Ctrl-P 和 Ctrl-N 或上下键来回显命令历史缓存中的上或下一行。

Grunt 的另一个有用的特色是自动补全机制。它能够在你按下 Tab 键时试图自动补全 Pig Latin 的关键词和函数。例如，如果有如下未完成的命令行：

```
grunt> a = foreach b ge
```

此时如果按下 Tab 键，那么 ge 会自动扩展成 Pig Latin 的关键词 generate：

```
grunt> a =foreach b generate
```

可以创建一个名为 *autocomplete* 的文件，并将其放置在 Pig 的类路径(如 Pig 的 *install* 目录的 *conf* 目录)或者启动 Grunt 的目录下，以此来定制自动补全的单词。这个文件中每个单词占一行，且单词中不能出现空白字符。自动补全的匹配是大小写敏感的。在这个文件中列出一些常用的文件路径特别有用(因为 Pig 并不提供文件名自动补全)。在该文件中列出你创建的用户自定义函数也能带来很多便利。

可以使用 help 命令来列出命令列表。结束一个 Grunt 会话时，可以使用 quit 命令或等价的快捷键\q 退出。

16.1.4 Pig Latin 编辑器

有多种编辑器都提供了对 Pig Latin 语法的高亮显示，包括 Eclipse、IntelliJ IDEA、Vim、Emacs 和 TextMate。详情请参见维基百科关于 Pig 的描述，网址为

① 历史记录保存在 *home* 目录下的.*pig_history* 文件中。

https://cwiki.apache.org/confluence/display/PIG/PigTools。

不少 Hadoop 发行版都附有 Hue web interface (*http://gethue.com/*)，其中包含 Pig 脚本编辑器和启动程序。

16.2 示例

现在，让我们用 Pig Latin 写一个计算天气数据集中年度最高气温的程序(与第 2 章中用 MapReduce 写的程序功能相同)，作为示例。这个程序只需要很少几行代码：

```
-- max_temp.pig: Finds the maximum temperature by year
records = LOAD 'input/ncdc/micro-tab/sample.txt'
AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
quality IN (0, 1, 4, 5, 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
MAX(filtered_records.temperature);
DUMP max_temp;
```

为了了解这个程序都做了些什么事情，我们将使用 Pig 的 Grunt 解释器。它让我们能够输入几行代码，然后通过交互来理解程序在做什么。在本地模式下启动 Grunt，然后输入 Pig 脚本的第一行：

```
grunt>records =LOAD 'input/ncdc/micro-tab/sample.txt'
>>AS (year:chararray, temperature:int, quality:int);
```

为了简单起见，程序假设输入是由制表符分割的文本，每行只包含年度、气温和质量三个字段。(事实上，如后文所述，Pig 的输入格式处理能力要比这个灵活得多)。这行代码描述的是我们要处理的输入数据。`year:chararray` 给出了字段的名称和类型。`chararray` 和 Java 字符串类似，`int` 和 Java 的 `int` 类似。`LOAD` 操作接受一个 URI 参数作为输入。在这个示例中，我们只使用了一个本地文件，不过我们也可以引用一个 HDFS URI。`AS` 子句(可选的)设定了字段的名称，以便在后面的语句中更方便地引用它们。

`LOAD` 操作的结果和 Pig Latin 其他所有操作的结果一样，都是一个关系(relation)，即一个元组集合。元组类似于数据库表中的一行数据，包含按照特定顺序排列的多个字段。在这个示例中，`LOAD` 函数根据输入文件，生成一个(年份，气温，质量)元组的集合。我们把关系写出来就是每个元组一行，每行由括号括起，每项字段由逗号分隔：

```
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
```

关系被赋予一个名称或**别名**(alias)以便于引用。这个关系的别名是 records。我们可以使用 DUMP 操作来查看别名所对应的关系的内容：

```
grunt>DUMP records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

我们还可以把 DESCRIBE 操作作用于一个关系的别名，来查看该关系的结构，即关系的**模式**(schema)：

```
grunt>DESCRIBE records;
records: {year: chararray,temperature: int,quality: int}
```

从输出中可以知道，records 有三个字段，别名分别为 year、temperature 和 quality。字段的别名是我们在 AS 子句中指定的。同样，字段的类型也是在 AS 子句中指定的。我们在后面会对 Pig 中的数据类型进行更深入的介绍。

第二条语句去除了没有气温的记录(即用值 9999 表示气温的记录)以及读数质量不令人满意的记录。在这个很小的数据集中，没有记录被过滤掉：

```
grunt> filtered_records = FILTER records BY temperature != 9999 AND
>> quality IN (0, 1, 4, 5, 9);
grunt>DUMP filtered_records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

第三条语句使用 GROUP 函数把 records 关系中的记录按照 year 字段分组。让我们用 DUMP 来查看 GROUP 的结果：

```
grunt> grouped_records = GROUP filtered_records BY year;
grunt> DUMP grouped_records;
(1949,{(1949,78,1),(1949,111,1)})
(1950,{(1950,-11,1),(1950,22,1),(1950,0,1)})
```

这样，我们就有了两行，或者称为两个元组。每个元组对应于输入数据中的一个年度。每个元组的第一个字段是用于分组的字段(即年度)。第二个字段是该年度的

元组的包(bag)。包是一个元组的无序集。在 Pig Latin 里，包用大括号来表示。

通过上述分组方式，我们已经为每个年度创建了一行。剩下的事情就是在每个包中找到包含最高气温的那个元组。在此之前，让我们先来理解 grouped_records 这一关系的结构：

```
grunt>DESCRIBE grouped_records;
grouped_records: {group: chararray,filtered_records: {year: chararray,
temperature: int,quality: int}}
```

从输出结果可以看到，Pig 给分组字段起了个别名 group。第二个字段和被分组的 filtered_records 关系的结构相同。根据这些信息，我们可以试着执行第四条语句对数据进行变换：

```
grunt>max_temp =FOREACH grouped_records GENERATE group,
>> MAX(filtered_records.temperature);
```

FOREACH 对每一行数据进行处理，并生成一组导出的行。导出的行的字段由 GENERATE 子句定义。在这个示例中，第一个字段是 group，也就是年度。第二个字段稍微复杂一点。filtered_records.temperature 引用了 grouped_records 关系中的 filtered_records 包中的 temperature 字段。MAX 是计算包中字段的最大值的内置函数。在这个例子中，它计算了 filtered_records 包中的最高温度。我们看一下它的结果：

```
grunt> DUMP max_temp;
(1949,111)
(1950,22)
```

这样，我们便已成功计算出每年的最高气温。

生成示例

在前面这个示例中，我们使用了一个仅包含少数行的抽样数据集，以简化数据流跟踪和调试。创建一个精简的数据集是一门艺术。理想情况下，这个数据集的内容应该足够丰富，能够覆盖查询中可能碰到的各种情况(满足完备性[completeness]条件)，同时，这个数据集应该足够小，能够被程序员直观理解(满足简明性[conciseness]条件)。通常情况下，使用随机取样并不能满足要求，因为连接和过滤这两个操作往往会去除掉所有的随机取样的数据，而导致产生一个空的结果集。这样是无法描述典型的数据流的。

Pig 通过 ILLUSTRATE 操作提供了生成相对完备和简明的数据集的工具。下面列出

的是运行 ILLUSTRATE 后的输出(进行了少许格式重排)：

```
grunt>ILLUSTRATE max_temp;
-------------------------------------------------------------------------------
| records     | year:chararray      | temperature:int      | quality:int      |
-------------------------------------------------------------------------------
|             | 1949                | 78                   | 1                |
|             | 1949                | 111                  | 1                |
|             | 1949                | 9999                 | 1                |
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
| filtered_records   | year: chararray | temperature: int | quality: int     |
-------------------------------------------------------------------------------
|                    | 1949            | 78               | 1                |
|                    | 1949            | 111              | 1                |
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
| grouped_records  | group:chararray | filtered_records:bag{:tuple(year: chararray,|
                                       temperature: int,quality: int)}          |
-------------------------------------------------------------------------------
|                  | 1949            | {(1949, 78, 1), (1949, 111, 1)}          |
-------------------------------------------------------------------------------
-----------------------------------------------
| max_temp     | group:chararray      | :int     |
-----------------------------------------------
|              | 1949                 | 111      |
-----------------------------------------------
```

注意，Pig 既使用了部分的原始数据(这对于保持生成数据集的真实性很重要)，也创建了一些新的数据。Pig 注意到查询中 9999 这一值，所以创建了一个包含该值的元组来测试 FILTER 语句。

综上所述，ILLUSTRATE 的输出易于跟踪，而且也能帮助理解查询的执行过程。

16.3 与数据库进行比较

我们已经演示了如何运行 Pig。看上去，Pig Latin 和 SQL 很相似。GROUP BY 和 DESCRIBE 之类的操作更加强了这种感觉。但是，这两种语言之间，以及 Pig 和关系型数据库管理系统(RDBMS)之间，有几个方面是不同的。

它们之间最显著的不同是：Pig Latin 是一种数据流编程语言，而 SQL 是一种声明式编程语言。换句话说，一个 Pig Latin 程序是一组针对输入关系的一步步操作。其中每一步都是对数据的简单变换。相反，SQL 语句是一个约束的集合。这些约束结合在一起，定义了输出。从很多方面看，用 Pig Latin 编程更像在 RDBMS 中查询规划器(query planner)这一层对数据进行操作。查询规划器决定了如何将声明式语句转化为一系列系统化的执行步骤。

RDBMS 把数据存储在严格定义了模式的表内。Pig 对它所处理的数据要求则宽松得多：你可以在运行时定义模式，而且这是可选的。本质上，Pig 可以在任何来源的元组上进行操作(当然，数据源必须支持并行的读操作，例如存放在多个文件中)。它使用 UDF 从原始格式中读取元组。[1]最常用的输入格式是用制表符分隔的字段组成的文本文件。Pig 为这种输入格式提供了内置加载函数。和传统的数据库不同，Pig 并不提供专门的数据导入过程将数据加载到 RDBMS。从文件系统(通常是 HDFS)中加载数据是处理的第一个步骤。

Pig 对复杂的嵌套数据结构的支持也使其不同于只能处理平面数据类型的 SQL。Pig 的语言能够和 UDF 以及流式操作紧密集成。Pig Latin 的这一能力及其嵌套数据结构，使它比大多数 SQL 的变种具有更强的定制能力。

RDBMS 具有一些支持在线和低延迟查询的特性，如事务和索引。然而，这些特性 Pig 都没有。Pig 并不支持随机读和几十毫秒级别的查询。它也不支持针对一小部分数据的随机写。同 MapReduce 一样，所有的写都是批量的、流式的写操作。

Hive(参见第 17 章)介于 Pig 和传统的 RDBMS 之间。同 Pig 一样，Hive 也被设计为用 HDFS 作为存储，但是它们之间有着显著的区别。Hive 的查询语言 HiveQL 是基于 SQL 的。任何熟悉 SQL 的人都可以轻松使用 HiveQL 写查询。和 RDBMS 相同，Hive 要求所有数据必须存储在表中，而表必须有模式，且模式由 Hive 进行管理。但是，Hive 允许为预先存储在于 HDFS 的数据关联一个模式。所以，数据的加载步骤是可选的。通过使用 HCatalog，Pig 也可以处理 Hive 表，详情参见 16.4.5 节。

16.4 PigLatin

本节对 Pig Latin 编程语言的语法和语义进行非正式的介绍。[2]本小节并不是完整的编程语言参考，[3]但是这里的内容足以帮助大家很好地理解 Pig Latin 的组成。

① 或像“Pig 哲学”(*http://pig.apache.org/philosophy.html*)所说：“猪什么都吃”(Pigs eat anything)。

② 不要把 Pig Latin 编程语言和语言游戏 Pig Latin 混淆。Pig Latin 游戏就是把单词开始的声母移到单词最后，并加上“ay”的尾音。例如，“pig”变为“ig-pay”，而“Hadoop”则变为“Adoop-hay”。

③ Pig Latin 并没有正式的语言定义。但通过 Pig 网站(*http://pig.apache.org/*)的链接，可以找到 Pig Latin 语言的完整指南。

16.4.1 结构

一个 Pig Latin 程序由一组语句构成。一个语句可以理解为一个操作，或一个命令。[1]例如，GROUP 操作是这样一种语句：

```
grouped_records = GROUP records BY year;
```

另一个语句的例子是列出 Hadoop 文件系统中文件的命令：

```
ls /
```

如前面的 GROUP 语句所示，一条语句通常用分号结束。实际上，那是一条必须用分号表示结束的语句。如果省略了分号，它会产生一个语法错误。而另一方面，ls 命令则可以不使用分号结束。一般的规则是：在 Grunt 中，交互使用的语句或命令不需要表示结束的分号。这包括交互式的 Hadoop 命令以及用于诊断的操作，例如 DESCRIBE。加上表示结束的分号总是不会错。因此，如果不确定是否需要分号，把它加上是最简单的解决办法。

必须用分号表示结束的语句可以分成多行以便于阅读：

```
records = LOAD 'input/ncdc/micro-tab/sample.txt'
  AS (year:chararray, temperature:int, quality:int);
```

Pig Latin 有两种注释方法。双减号表示单行注释。Pig Latin 解释器会忽略从第一个减号开始一直到行尾的所有内容：

```
-- My program
DUMP A; -- What's in A?
```

C 语言风格的注释更灵活。这是因为它使用/*和*/符号表示注释块的开始和结束。这样，注释既可以跨多行，也可以内嵌在某一行内：

```
/*
 * Description of my program spanning
 * multiple lines.
 */
A  = LOAD 'input/pig/join/A';
B  = LOAD 'input/pig/join/B';
C  = JOIN A BY $0, /* ignored */ B BY $1;
DUMP C;
```

① 在 Pig Latin 的文档里，有时这些术语是可以相互替换的。例如，“GROUP 命令”“GROUP 操作”和“GROUP 语句”的含义相同。

Pig Latin 有一个关键词列表。其中的单词在 Pig Latin 中有特殊含义，不能用作标识符。这些单词包括操作(LOAD，ILLUSTRATE)、命令(cat，ls)、表达式(matches，FLATTEN)以及函数(DIFF，MAX)等。它们会在随后几个小节介绍。

Pig Latin 采用混合的大小写敏感规则。操作和命令是大小写无关的(这样能使得交互式操作更“宽容”)，而别名和函数名则是大小写敏感的。

16.4.2 语句

在 Pig Latin 程序执行时，每个命令按顺序进行解析。如果遇到句法错误或其他(语义)错误，例如未定义的别名，解释器会终止运行，并显示一条错误消息。解释器会给每个关系操作建立一个逻辑计划(logical plan)。逻辑计划构成了 Pig Latin 程序的核心。解释器把为一个语句创建的逻辑计划加到到目前为止已经解析完的程序的逻辑计划上，然后继续处理下一条语句。

特别要注意，在构造整个程序的逻辑计划时，Pig 并不处理数据。我们仍然以前面的 Pig Latin 程序为例：

```
-- max_temp.pig: Finds the maximum temperature by year
records = LOAD 'input/ncdc/micro-tab/sample.txt'
AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
quality IN (0, 1, 4, 5, 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
MAX(filtered_records.temperature);
DUMP max_temp;
```

Pig Latin 解释器看到第一行 LOAD 语句时，首先确认它在语法和语义上是正确的，然后再把这个操作加入逻辑计划。但是，解释器并不是真的从文件加载数据(它甚至不去检查该文件是否存在)。Pig 到底要把文件加载到哪里呢？数据是加载到内存吗？即使这些数据可以放入内存，Pig 又如何处理数据呢？我们可能并不需要所有的数据(因为后续的语句可能会过滤数据)，因此加载数据没有意义。关键问题在于，在没有定义整个数据流之前，开始任何处理都是没有意义的。与此类似，Pig 验证 GROUP 和 FOREACH…GENERATE 语句，并把它们加入逻辑计划中，但并不执行这两条语句。让 Pig 开始真正执行的是 DUMP 语句。此时，逻辑计划被编译成物理计划，并执行。

多查询执行

由于 DUMP 是一个诊断工具，因此它总是会触发语句的执行。STORE 命令与 DUMP 不同。在交互模式下，STORE 和 DUMP 一样，总是会触发语句的执行(这一过程包含了 run 命令)。但是，在批处理模式下，它不会触发执行(此时包含了 exec 命令)。这是为了从性能角度考虑而进行的设计。在批处理模式下，Pig 会解析整个脚本，看看是否能够为减少写或读磁盘的数据量进行优化。考虑如下的简单示例：

```
A  = LOAD 'input/pig/multiquery/A';
B  = FILTER A BY $1 == 'banana';
C  = FILTER A BY $1 != 'banana';
STORE B INTO 'output/b';
STORE C INTO 'output/c';
```

关系 B 和 C 都是从 A 导出的。因此，为了防止读两遍 A，Pig 可以用一个 MapReduce 作业从 A 读取数据，并把结果写到两个输出文件中去：一个给 B，一个给 C。这一特性称为**多查询执行**(multiquery execution)。

Pig 的以前版本不包括这一特性：批处理模式下脚本中的每个 STORE 语句都会触发语句的执行，从而使每个 STORE 语句都有一个对应的作业。可以在执行时对 pig 使用-M 或-no_multiquery 选项来禁用多查询执行，恢复使用以前的设置。

Pig 的物理计划是一系列的 MapReduce 作业。在本地模式下，这些作业在本地 JVM 中运行，而在 MapReduce 模式下，它们在 Hadoop 集群上运行。

可以用 EXPLAIN 命令对一个关系查看 Pig 所创建的逻辑和物理计划(例如 EXPLAIN max_temp;)。

EXPLAIN 也显示 MapReduce 计划，即显示物理操作是如何组成 MapReduce 作业的。这是查看 Pig 为查询运行多少个 MapReduce 作业的好办法。

表 16-1 概括了能够作为 Pig 逻辑计划一部分的关系操作。16.6 节将详细介绍这些操作。

有些种类的语句并不会被加到逻辑计划中去。例如，诊断操作 DESCRIBE、EXPLAIN 以及 ILLUSTRATE。这些操作是用来让用户能够与逻辑计划进行交互以进行调试的(参见表 16-2)。DUMP 也是一种诊断操作，它只能用于与很小的结果集进行交互调试，或与 LIMIT 结合使用，来获得某个较大的关系的一小部分行。当

输出包含的行比较多的时候，应该使用 STORE 语句，这是因为 STORE 语句将结果存入文件中，而不是在控制台上显示。

表 16-1. Pig Latin 的关系操作

类型	操作	描述
加载与存储	LOAD	从文件系统或其他存储加载数据，存入关系
	STORE	将一个关系存放到文件系统或其他存储中
	DUMP	将关系打印到控制台
过滤	FILTER	从关系中删除不需要的行
	DISTINCT	从关系中删除重复的行
	FOREACH...GENERATE	在关系中增加或删除字段
	MAPREDUCE	以一个关系作为输入运行某个 MapReduce 作业
	STREAM	使用外部程序对一个关系进行变换
	SAMPLE	对一个关系进行随机取样
	ASSERT	确保一个关系中的所有行就某个条件而言都是真的，否则失败
分组与连接	JOIN	连接两个或多个关系
	COGROUP	对两个或更多关系中的数据进行分组
	GROUP	在一个关系中对数据进行分组
	CROSS	创建两个或更多关系的乘积(叉乘)
	CUBE	为一个关系中指定列的所有组合生成聚集数据
排序	ORDER	根据一个或多个字段对某个关系进行排序
	RANK	为一个关系中的每个元组分配一个顺序号，可以选择先根据字段排序再分配。
	LIMIT	将一个关系的元组个数限定在一定数量内
组合和切分	UNION	合并两个或多个关系为一个关系
	SPLIT	把某个关系切分两个或多个关系

表 16-2. Pig Latin 的诊断操作

操作	描述
DESCRIBE	打印关系的模式
EXPLAIN	打印逻辑和物理计划
ILLUSTRATE	使用生成的输入子集显示逻辑计划的试运行结果

为了在 Pig 脚本中使用宏和用户自定义函数，PigLatin 还提供了 REGISTER、DEFINE 和 IMPORT 这三个语句(参见表 16-3)。

表 16-3. Pig Latin 的宏和 UDF 语句

语句	描述
REGISTER	在 Pig 运行时环境中注册一个 JAR 文件
DEFINE	为宏、UDF、流式脚本或命令规范新建别名
IMPORT	把在另一个文件中定义的宏导入脚本

因为这些命令并不处理关系，所以它们不会被加入逻辑计划。相反，这些命令会被立即执行。Pig 提供了与 Hadoop 文件系统和 MapReduce 进行交互的命令及其他一些工具命令(参见表 16-4)。与 Hadoop 文件系统进行交互的命令对在 Pig 处理前和处理后进行的数据移动非常有用。

表 16-4. PigLatin 命令

类别	命令	描述
Hadoop 文件系统	cat	打印一个或多个文件的内容
	cd	改变当前目录
	copyFromLocal	复制本地文件或目录
	copyToLocal	将一个文件或目录从 Hadoop 文件系统复制到本地文件系统
	cp	把一个文件或目录复制到另一个目录
	fs	访问 Hadoop 文件系统 shell
	ls	打印文件列表信息
	mkdir	创建新目录
	mv	将一个文件或目录移动到另一个目录
	pwd	打印当前工作目录的路径
	rm	删除一个文件或目录
	rmf	强制删除文件或目录(即使文件或目录不存在也不会失败)
Hadoop MapReduce 工具	kill	终止某个 MapReduce 作业
	exec	在新的 Grunt shell 中以批处理模式运行脚本
	help	显示可用的命令和选项
	history	打印当前 Grunt 会话中运行的查询语句
	quit(\q)	退出解释器
	run	在当前 Grunt shell 中运行脚本
	set	设置 Pig 选项和 MapReduce 作业属性
	sh	在 Grunt 中运行 shell 命令

文件系统相关的命令可以对任何 Hadoop 文件系统的文件或目录进行操作。这些命

令和 hadoopfs 命令很像(这是意料之中的，因为两者都是 Hadoop FileSystem 接口的简单封装)。可以使用 Pig 的 fs 命令访问所有的 Hadoop 文件系统的 shell 命令。例如，fs -ls 显示文件列表，fs -help 则显示所有可用命令的帮助信息。

准确地说，使用哪个 Hadoop 文件系统是由 Hadoop Core 站点文件中的 fs.default.name 属性决定的。3.3 节详细介绍了如何设置这个属性。

除了 set 命令以外，这些命令的含义大都不言自明。set 命令用于设置控制 Pig 行为的选项，包括所有 MapReduce 作业的属性。debug 选项用于在脚本中打开或关闭调试日志(也可以在启动 Pig 时使用-d 或-debug 选项控制日志的级别)：

```
grunt>set debug on
```

另一个很有用的选项是 job.name。它为 Pig 作业设定一个有意义的名称。这样，即可方便地知道在共享的 Hadoop 集群上有哪些 Pig MapReduce 作业是自己的。如果 Pig 正在运行某个脚本(而不是通过 Grunt 运行交互式查询)，默认作业名称是基于脚本的名称的。

表 16-4 中有两个命令可以运行 Pig 脚本：exec 和 run。它们的区别是 exec 在一个新的 Grunt shell 环境中以批处理方式运行脚本。因此，所有脚本中定义的别名在脚本运行结束后不能在 shell 中再被访问。另一方面，如果使用 run 运行脚本，那么效果就和在 shell 中手工输入脚本的内容是一样的。因此，运行该脚本的 shell 命令历史中包含脚本的所有语句。只能用 exec 进行多查询执行，即 Pig 以批处理方式一次执行一批语句(详情参见 16.4.2 节的补充内容“多查询执行”)，而不能使用 run 命令进行多查询执行。

控制流

Pig Latin 的设计中缺少原生的控制流语句。如果想撰写需要条件逻辑或循环结构的程序，建议把 Pig Latin 嵌入到其他语言之中，如 Python、JavaScript 或 Java，由该语言管理控制流。在这一模型下，宿主脚本使用 compile-bind-run API 来执行 Pig 脚本，并获得脚本的状态。要想获得 API 的详细信息，请参考 Pig 的帮助文档。

嵌入式 Pig 程序总是在 JVM 中运行。如果要执行 Python 或 JavaScript 程序，应使用 pig 命令，后面跟脚本名。合适的 Java 脚本引擎(对 Python 而言是 Jython，对 JavaScript 而言是 Rhino)会被自动选择来执行脚本。

16.4.3 表达式

在 Pig 中，可以通过计算表达式得到某个值。表达式可以作为包含关系操作的语句的一部分。Pig 可以使用丰富的表达式类型。Pig 中的很多表达式类型和其他编程语言中的表达式相像。表 16-5 列出了各种表达式及其简要说明和示例。本章将有很多这样的表达式。

表 16-5. Pig Latin 表达式

类别	表达式	描述	示例
常数	文字	常量值(参见表 16-6 中“文字示例”一栏)	1.0, 'a'
字段(位置指定)	$n	第 n 个字段(以 0 为基数)	$0
字段(名字指定)	f	字段名 f	year
字段(消除歧义)	r::f	分组或连接后关系 r 中的名为 f 的字段	A::year
投影	c.$n,c.f	在容器 c(关系、包或元组)中的字段(按位置或名称指定)	records.$0, records.year
Map 查找	m#k	在映射 m 中键 k 所对应的值	items#'Coat'
类型转换	(t) f	将字段 f 转换为类型 t	(int) year
算术	x + y,x -y	加法和减法	$1 + $2,$1 - $2
	x *y,x / y	乘法和除法	$2 * $2,$2 / $2
	x % y	取模运算，即 x 除以 y 后的余数	$1 % $2
	+x,-x	正和负	+1,-1
条件	x ? y : z	二值条件三元运算符，如果 x 为真，则 y，否则为 z	quality == 0 ? 0 : 1
	CASE	多条件	CASE q WHEN 0 THEN 'good' ELSE 'bad' END
比较	x == y,x != y	相等和不等	quality == 0, temperature != 9999
	x >y,x <y	大于和小于	quality > 0, quality < 10
	x >= y,x <= y	大于等于和小于等于	quality >= 1, quality <= 9
	xmatchesy	正则表达式匹配	quality matches '[01459]'
	xisnull	是空值	temperature is null
	xisnotnull	不是空值	temperature is not null
布尔型	xory	逻辑或	q == 0 or q == 1
	xandy	逻辑与	q == 0 and r == 0
	notx	逻辑非	not q matches '[01459]'
	IN x	设定成员	q IN (0, 1, 4, 5, 9)
函数型	fn(f1,f2,...)	在 f1,f2 等字段上应用函数 fn	isGood(quality)
平面化	FLATTEN(f)	从包和元组中去除嵌套	FLATTEN(group)

16.4.4 类型

前面出现过 Pig 的一些简单数据类型，例如 int 和 chararray。本节将更详细地讨论 Pig 的内置数据类型。

Pig 有六种数值类型：int、long、float、double、biginteger 和 bigdecimal。它们和 Java 中对应的数值类型相同。此外，Pig 还有 bytearray 类型，它类似于 Java 用来表示二进制大对象的 byte 数组；chararray 类型则类似于用 UTF-16 格式表示文本数据的 java.lang.String。chararray 也可以被加载或存储为 UTF-8 格式；datetime 类型为短日期时间格式，精确度到毫秒，并包含时区。

Pig 没有任何一种数据类型对应于 Java 的 byte、short 或 char。这些数据类型都能使用 Pig 的 int 类型或 chararray 类型(针对 char)来方便地表示。

布尔、数值、文本、二进制与时间类型都是原子类型。PigLatin 有三种用于表示嵌套结构的复杂类型：tuple(元组)、bag(包)和 map(映射)。表 16-6 列出了 PigLatin 的所有数据类型。

表 16-6. Pig Latin 数据类型

类别	数据类型	描述	文字示例
布尔	boolean	true/false 值	true
数值	int	32 位有符号整数	1
	long	64 位有符号整数	1L
	float	32 位浮点数	1.0F
	double	64 位浮点数	1.0
	biginteger	任意精度整数	'10000000000'
	bigdecimal	任意精度带符号小数	'0.11000100000000000000001'
文本	chararray	UTF-16 格式的字符数组	'a'
二进制	bytearray	字节数组	不支持
时间	datetime	带时区的日期和时间	不支持，使用内置的 ToDate 函数
复杂类型	tuple	任何类型的字段序列	(1,'pomegranate')
	bag	元组的无序多重集合(允许重复元组)	{(1,'pomegranate'),(2)}
	map	一个键-值对的集合。键必须是字符数组，值可以是任何类型的数据	['a'#'pomegranate']

复杂类型通常从文件加载或者使用关系操作进行构建。但是要注意，表 16-6 中的文字示例只是在 PigLatin 程序中表示常数值时使用的形式，当使用 PigStorage 加载器从文件加载时，数据的原始形式往往与之不同。例如，在文件中如表 16-6 所示的包的数据可能形如{(1,pomegranate),(2)}，注意，此时没有单引号。如果有合适的模式，该数据可以加载到一个关系。该关系只有一个仅有一个字段的行，而该字段的值是包。

Pig 提供了内置的 TOTUPLE、TOBAG 以及 TOMAP 函数。它们被用来将表达式转化为元组、包以及映射。

虽然关系和包在概念上是相同的(本质上它们都是元组的无序多重集合)，但实际上 Pig 对它们的处理稍有不同。关系是顶层构造结构，而包必须在某个关系中。正常情况下，不必操心它们的区别，但是它们在使用时会有一些限制。对于新手，这些限制仍然可能导致错误。例如，不能根据包文字直接创建一个关系。因此，如下语句会运行失败：

```
A = {(1,2),(3,4)}; --Error
```

针对这种情况，最简单的解决办法是使用 LOAD 语句从文件加载数据。

另一个例子是，对待关系不能像处理包那样把一个字段投影为一个新的关系(例如，使用位置符号，用$0 指向 A 的第一个字段)：

```
B = A.$0;
```

要达到这个目的，必须使用关系操作将一个关系 A 转换为另一个关系 B：

```
B = FOREACH A GENERATE $0;
```

Pig Latin 将来的版本可能采用相同的方法来处理关系和包，旨在消除这种不一致性。

16.4.5 模式

Pig 中的一个关系可以关联一个模式。模式为关系中的字段指定名称和类型。我们前面已经介绍过 LOAD 语句的 AS 子句如何在关系上附以模式：

```
grunt>records = LOAD 'input/ncdc/micro-tab/sample.txt'
>>   AS (year:int, temperature:int, quality:int);
grunt>DESCRIBE records;
records: {year: int,temperature: int,quality: int}
```

这次，虽然加载的是和上次一样的文件，但年份已声明为整数类型，而不是 chararray 类型。如果要对年份这一字段进行算术操作(例如将它变为一个时间戳)，那么用整数类型更为合适。相反，如果只是想把它作为一个简单的标识符，那么表示为 chararray 类型则更合适。Pig 的这种模式声明方式提供了很大的灵活性。这和传统 SQL 数据库要求在数据加载前必须先声明模式的方式截然不同。Pig 的设计目的是用它来分析不包含数据类型信息的纯文本输入文件，因此，它为字段确定类型的时机比 RDBMS 要晚也是理所当然的。

我们也可以完全忽略类型声明：

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>>   AS (year, temperature, quality);
grunt>DESCRIBE records;
records: {year: bytearray,temperature: bytearray,quality: bytearray}
```

在这个例子中，我们在模式中只确定了字段的名称：year、temperature 和 quality。默认数据类型为最通用的 bytearray，即二进制串。

不必为每一个字段都给出类型。你可以让某些字段的类型为默认的 bytearray，就像如下模式声明示例中的 year 字段：

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>>   AS (year, temperature:int, quality:int);
grunt> DESCRIBE records;
records: {year: bytearray,temperature: int,quality: int}
```

但是，如果要用这种方式来确定模式，必须在模式中定义每一个字段。同样，不能只确定字段的类型而不给出其名称。另一方面，模式本身是可选的。可以省略 AS 子句，如下所示：

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt';
grunt> DESCRIBE records;
Schema for records unknown.
```

对于没有对应模式的关系中的字段，只能使用位置符号进行引用：$0 表示关系中的第一个字段，$1 表示第二个，依此类推。它们的类型都是默认的 bytearray：

```
grunt>projected_records = FOREACH records GENERATE $0, $1, $2;
grunt>DUMP projected_records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
grunt> DESCRIBE projected_records;
projected_records: {bytearray,bytearray,bytearray}
```

虽然不为字段指明类型很省事(特别是在撰写查询的开始阶段)，但如果指定了字段的类型，我们可以使 PigLatin 程序更清晰，也使程序运行得更高效。因此，在一般情况下，建议指明字段的数据类型。

1. 通过 HCatalog 使用 Hive

虽然在查询中声明模式的方式是灵活的，但这种方式并不利于模式重用。处理相同输入数据的一组 Pig 查询常常使用相同的模式。如果一个查询要处理很多字段，那么在每个查询中维护重复出现的模式会很困难。

HCatalog(Hive 的一个组件)通过提供基于 Hive 的 metastore 表的元数据服务解决了这一问题。这样，Pig 查询就可以通过名称引用模式，而不必每次都得指明整个模式。例如，通过运行 17.2 节的示例将数据加载到一个名为 records 的 Hive 表后，Pig 可以用下述方式访问该表的模式和数据：

```
% pig -useHCatalog
grunt> records = LOAD 'records' USING org.apache.hcatalog.pig.HCatLoader();
grunt> DESCRIBE records;
records: {year: chararray,temperature: int,quality: int}
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

2. 验证与空值

SQL 数据库在加载数据时，会强制检查表模式中的约束。例如，试图将一个字符串加载到声明为数值型的列会失败。在 Pig 中，如果一个值无法被强制转换为模式中声明的类型，Pig 会用空值 null 替代。如果有如下天气数据输入，在定义为整数型的地方出现了一个字符“e”，我们来看一下这一验证机制是如何工作的：

```
1950 0   1
1950 22  1
1950 e   1
1949 111 1
1949 78  1
```

Pig 在处理损坏的行时会为违例的值产生一个 null。在输出到屏幕(或使用 STORE 存储)时，空值 null 被显示(或存储)为一个空位：

```
grunt>records = LOAD 'input/ncdc/micro-tab/sample_corrupt.txt'
```

```
>>AS (year:chararray, temperature:int, quality:int);
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,,1)
(1949,111,1)
(1949,78,1)
```

Pig 会为非法字段产生警告(在此没有显示)，但是它不会终止处理。大的数据集普遍都有损坏值、无效值或意料之外的值，因而逐步修正每一条无法解析的记录一般都不太现实。作为一种替代方法，我们可以一次性地把所有的非法记录都找出来，然后再一起处理它们。我们可以修正我们的程序(因为这些记录表示我们写程序时犯了错误)，或把这些记录过滤掉(因为这些数据无法使用)：

```
grunt>corrupt_records = FILTER records BY temperature is null;
grunt> DUMP corrupt_records;
(1950,,1)
```

请注意，这里对 isnull 操作的使用和 SQL 中的类似。事实上，我们可以在原始记录中获得更多的信息(例如标识符和无法被解析的值等)，帮助分析问题数据。

可以用对关系中的行进行计数的常用语句来获得损坏记录的条数，如下所示：

```
grunt> grouped = GROUP corrupt_records ALL;
grunt> all_grouped = FOREACH grouped GENERATE group, COUNT(corrupt_records);
grunt> DUMP all_grouped;
(all,1)
```

16.6.3 节在介绍 GROUP 时详细解释了分组和 ALL 操作。

另一个有用的技巧是使用 SPLIT 操作把数据划分成“好”和“坏”两个关系，然后再分别对它们进行分析：

```
grunt>SPLIT records INTO good_records IF temperature is not null,
>> bad_records OTHERWISE;
grunt>DUMP good_records;
(1950,0,1)
(1950,22,1)
(1949,111,1)
(1949,78,1)
grunt>DUMP bad_records;
(1950,,1)
```

让我们回到前面 temperature 数据类型未声明的情况，此时无法轻松检测到损坏的数据，因为它没有被当作 null 值：

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample_corrupt.txt'
>> AS (year:chararray, temperature, quality:int);
```

```
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,e,1)
(1949,111,1)
(1949,78,1)
grunt> filtered_records = FILTER records BY temperature != 9999 AND
>> quality IN (0, 1, 4, 5, 9);
grunt> grouped_records = GROUP filtered_records BY year;
grunt> max_temp = FOREACH grouped_records GENERATE group,
>> MAX(filtered_records.temperature);
grunt> DUMP max_temp;
(1949,111.0)
(1950,22.0)
```

在这种情况下，temperature 字段被解释为 bytearray，因此在数据加载时，损坏的字段并没有被检测出来。在传输给 MAX 函数时，因为 MAX 只能处理数值类型，所以 temperature 字段被强制转换为 double 类型。损坏的字段不能被表示为 double，所以它被当作 null 处理，MAX 则会忽略这个值。通常，最好的解决办法是在加载数据时声明数据类型，并在进行主要的处理前查看关系中缺失的值或损坏的值。

有时，因为有些字段缺失，损坏的数据被显示为比较短的元组。可以用 SIZE 函数对它们进行过滤，如下所示：

```
grunt> A = LOAD 'input/pig/corrupt/missing_fields';
grunt> DUMP A;
(2,Tie)
(4,Coat)
(3)
(1,Scarf)
grunt> B = FILTER A BY SIZE(TOTUPLE(*)) > 1;
grunt> DUMP B;
(2,Tie)
(4,Coat)
(1,Scarf)
```

3. 模式合并

在 Pig 中，不用为数据流中每一个新产生的关系声明模式。在大多数情况下，Pig 能够根据输入关系的模式来确定关系操作的输出结果的模式。

那么模式是如何传播到新关系的呢？有些关系操作不会改变模式。因此，LIMIT 操作(对一个关系的最大元组数进行限制)产生的关系就和它所处理的关系具有相同的模式。对于其他操作，情况可能更复杂一些。例如，UNION 操作将两个或多个关系合并成一个，并试图同时合并输入关系的模式。如果这些模式由于数据类型

或字段个数不同而不兼容，那么 UNION 产生的模式是未知的。

针对数据流中的任何关系，都可以使用 DESCRIBE 操作来获取它们的模式。如果要重新定义一个关系的模式，可以使用带 AS 子句的 FOREACH…GENERATE 操作来定义输入关系的一部分或全部字段的模式。

16.5 节在讨论用户自定义函数时将进一步讨论模式。

16.4.6 函数

Pig 中的函数有四种类型。

1. 计算函数(Evalfunction)

计算函数获取一个或多个表达式作为输入，并返回另一个表达式。MAX 就是一个内置计算函数的例子，它返回一个包内所有项中的最大值。有些计算函数是**聚集函数**(aggregate function)，这意味着它们作用于包，并产生一个“标量值”(scalar value)。MAX 就是一个聚集函数。此外，很多聚集函数是**代数的**(algebraic)。也就是说这些函数的结果可以增量计算。如果用 MapReduce 的术语来表示，就是通过使用 combiner 进行计算，代数函数的计算效率可以提高很多(参见 2.4.2 节对 combiner 函数的讨论)。MAX 是一个代数函数，而计算一组值的“中位数”(median)的函数则不是代数函数。

2. 过滤函数(Filterfunction)

过滤函数是一类特殊的计算函数。这类函数返回的是逻辑布尔值。正如其名，过滤函数被 FILTER 操作用于移除不需要的行。它们也可以用于其他以布尔条件作为输入的关系操作或用于使用布尔或条件表达式的表达式。IsEmpty 就是一个内置过滤函数。它测试一个包或映射是否含有元素。

3. 加载函数(Loadfunction)

加载函数指明如何从外部存储加载数据到一个关系。

4. 存储函数(Storefunction)

存储函数指明如何把一个关系中的内容存到外部存储。通常，加载和存储函数由相同的类型实现。例如，PigStorage 从分隔的文本文件中加载数据，也能以相同

的格式存储数据。

Pig 有很多内置的函数，表 16-7 列出了其中的一部分。完整的内置函数列表包括很多标准数学、字符串、日期/时间和集合函数。Pig 的每个发布版本文档都包含这些函数列表。

表 16-7. Pig 的部分内置函数

类别	函数名称	描述
计算	AVG	计算包中项的平均值
	CONCAT	把两个字节数组或字符数组连接成一个
	COUNT	计算一个包中非空值的项的个数
	COUNT_STAR	计算一个包中的项的个数，包括空值
	DIFF	计算两个包的差。如果两个参数不是包，那么如果它们相同，则返回一个包含这两个参数的包；否则返回一个空的包
	MAX	计算一个包中项的最大值
	MIN	计算一个包中项的最小值
	SIZE	计算一个类型的大小。数值型的大小总是 1；对于字符数组，它返回字符的个数；对于字节数组，它返回字节的个数；对于容器(container，包括元组、包、映射)，它返回其中项的个数
	SUM	计算一个包中项的值的总和
	TOBAG	把一个或多个表达式转换为单独的元组，然后把这些元组放入包，它是()的同义词
	TOKENIZE	对一个字符数组进行标记解析，并把结果词放入一个包
	TOMAP	将偶数个表达式转换为一个键-值对的映射，它是[]的同义词
	TOP	计算包中最前面的 *n* 个元组
	TOTUPLE	将一个或多个表达式转换为一个元组，它是{}的同义词
过滤	IsEmpty	判断一个包或映射是否为空
加载/存储	PigStorage	用字段分隔文本格式加载或存储关系。用一个可设置的分隔符(默认为一个制表符)把每一行分隔为字段后，将它们分别存储于元组的各个字段。这是不指定加载/存储方式时的默认存储函数[①]
	TextLoader	从纯文本格式加载一个关系。每一行对应于一个元组。每个元组只包含一个字段，即该行文本
	JsonLoader, JsonStorage	从(Pig 定义的)JSON 格式加载关系，或将关系存储为 JSON 格式。每个元组存储为一行

① 通过设置 pig.default.load.func 和 pig.default.store.func 可以更改默认存储为完全限定的加载和存储函数类名。

续表

类别	函数名称	描述
加载/存储	AvroStorage	从 Avro 数据文件中加载关系，或将关系存储至 Avro 数据文件中
	ParquetLoader, ParquetStorer	从 Parquet 文件中加载关系，或将关系存储至 Parquet 文件中
	OrcStorage	从 Hive ORCFiles 中加载关系，或将关系存储至 Hive ORCFiles 中
	HBaseStroage	从 HBase 表中加载关系，或将关系存储至 HBase 表中

5. 一些其他库

如果没有找到你需要的函数，你可以撰写自己的用户自定义函数(或简称为 UDF)，如 16.5 节所述。但在此之前，可以先查看一下 Piggy Bank (*https://cwiki.apache.org/confluence/display/PIG/PiggyBank?*)。这是一个 Pig 社区共享 Pig 函数的库。例如，在 Piggy Bank 中有 CSV 文件、Hive RCFiles、SequenceFiles，以及 XML 文件的加载和存储函数。Pig 网站中有关于 Piggy Bank 的 JAR 文件，不需要配置就可直接使用。Pig 的 API 文档中包含 Piggy Bank 提供的函数列表。

Apache DataFu (*http://datafu.incubator.apache.org/*)是另一个蕴含丰富 Pig UDF 的库。除了常用函数之外，它还包括计算基本统计数据的函数、执行取样与评估的函数、实现散列的函数以及与 web 数据(会话流程、链路分析)相关的函数。

16.4.7 宏

宏提供了在 Pig Latin 内部对可重用的 Pig Latin 代码进行打包的功能。例如，我们可以把对关系进行分组并在每一组内查找最大值的 Pig Latin 程序抽出来，定义成如下的宏：

```
DEFINE max_by_group(X, group_key, max_field) RETURNS Y {
A = GROUP $X by $group_key;
$Y = FOREACH A GENERATE group, MAX($X.$max_field);
};
```

这个宏的名字为 max_by_group。它包含三个参数：一个关系 X 以及两个字段名：group_key 和 max_field。它返回一个关系 Y。在宏的体内，参数和返回别名通过使用$前缀进行引用，例如$X。

宏的使用方法如下：

```
records = LOAD 'input/ncdc/micro-tab/sample.txt'
AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
quality IN (0, 1, 4, 5, 9);
max_temp = max_by_group(filtered_records, year, temperature);
DUMP max_temp
```

在运行时，Pig 将用宏的定义展开宏。展开后的程序如下所示，被展开的部分用粗体表示：

```
records = LOAD 'input/ncdc/micro-tab/sample.txt'
AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
quality IN (0, 1, 4, 5, 9);
macro_max_by_group_A_0 = GROUP filtered_records by (year);
max_temp = FOREACH macro_max_by_group_A_0 GENERATE group,
MAX(filtered_records.(temperature));
DUMP max_temp
```

一般情况下，你看不到程序展开后的形式，这是因为 Pig 是在内部进行这一操作的。但是，有些情况下，在撰写和调试宏时，如果能看到展开后的形式会比较有益，那么可以通过给 pig 命令传递-dryrun 参数让 Pig 只进行宏扩展(而不执行脚本)。

注意，传递给宏的参数(filtered_records、year 以及 temperature)已经取代了宏定义中的参数名。宏定义中不使用$前缀的别名，例如示例中的 A，仅在宏定义的局部有效。在展开时，它们被重写，以避免和程序的其他部分的别名冲突。在这个例子中，A在展开后的形式为macro_max_by_group_A_0。

为了方便重用，也可以在 Pig 脚本以外的文件中定义宏。这种情况下，需要在使用宏的脚本中导入这些文件。导入语句的形式如下：

```
IMPORT './ch16-pig/src/main/pig/max_temp.macro';
```

16.5 用户自定义函数

Pig 设计者认识到以插件形式提供使用用户定制代码的能力是一个关键问题，但这也是数据处理中最琐碎的工作。因此，他们的设计简化了用户自定义函数的定义与使用。我们在本节中只介绍 Java UDF。你也可以使用 Python、JavaScript、Ruby 或 Groovy 来写 UDF，它们都使用 Java Scripting API 来运行。

16.5.1 过滤 UDF

我们通过编写一个过滤不满足气温质量读数要求的天气记录的函数来演示如何写过滤函数。我们的基本思路是修改下面的代码：

```
filtered_records = FILTER records BY temperature != 9999 AND
quality IN (0, 1, 4, 5, 9);
```

修改后的代码如下：

```
filtered_records = FILTER records BY temperature != 9999 AND isGood(quality);
```

这样写有两个好处：使 Pig 脚本更精简；而且还封装了处理逻辑，以便轻松重用于其他脚本。如果只是写一个即时查询，我们可能不需要麻烦地为此而写一个 UDF，只有在需要不断做相同的处理时，才需要如此写可重用的 UDF。

所有的过滤函数都是 FilterFunc 的子类，而 FilterFunc 本身是 EvalFunc 的子类。我们后面会对 EvalFunc 进行更详细的介绍。在这里，我们只需要知道 EvalFunc 本质上就像下面的类这样：

```
public abstract class EvalFunc<T> {
  public abstract T exec(Tuple input) throws IOException;
}
```

EvalFunc 只有一个抽象方法 exec()。它的输入是一个元组，输出则只有一个值，其(参数化)类型为 T。输入元组的字段包含传递给函数的表达式，在这个例子里，它是一个整数。对于 FilterFunc，T 是 Boolean 类型的，对于那些不应该被过滤掉的元组，该方法应该返回 true。

对于本例中的质量过滤器，我们要写一个 IsGoodQuality 类，扩展 FilterFunc 并实现 exec()，参见范例 16-1。Tuple 类本质上是一个与某个类型关联的对象列表。在这里，我们只关心第一个字段(因为函数只有一个参数)。我们用 Tuple 的 get()方法，根据序号来获取这个字段。该字段的类型是整型。因此，如果它非空，我们就对它进行类型转换，检查它是否表示气温读数是正确的，并根据检查结果返回相应的值：true 或者 false。

范例 16-1. 该 FiterFunc UDF 删除包含不符合质量要求的气温读数记录

```
package com.hadoopbook.pig;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
```

```
import org.apache.pig.FilterFunc;

import org.apache.pig.backend.executionengine.ExecException;
import org.apache.pig.data.DataType;
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.logicalLayer.FrontendException;

public class IsGoodQuality extends FilterFunc {

  @Override
  public Boolean exec(Tuple tuple) throws IOException {
    if (tuple == null || tuple.size() == 0) {
      return false;
    }
    try {
      Object object = tuple.get(0);
      if (object == null) {
        return false;
      }
      int i = (Integer) object;
      return i == 0 || i == 1 || i == 4 || i == 5 || i == 9;
    } catch (ExecException e) {
      throw new IOException(e);
    }
  }
}
```

为了使用新函数，我们首先进行编译，并把它打包到一个 JAR 文件(在本书所附的示例代码中，包含如何进行打包的指导)。然后，我们通过 REGISTER 操作指定文件的本地路径(不带引号)，告诉 Pig 这个 JAR 文件的信息：

```
grunt>REGISTER pig-examples.jar;
```

最后，我们就可以调用这个函数：

```
grunt>filtered_records = FILTER records BY temperature != 9999 AND
>> com.hadoopbook.pig.IsGoodQuality(quality);
```

Pig 把函数名作为 Java 类名并试图用该类名来加载类以完成函数调用。(这也就是为什么函数名是大小写敏感的，因为 Java 类名是大小写敏感的。)在搜索类的时候，Pig 会使用包含已注册 JAR 文件的类加载器。运行于分布式模式时，Pig 会确保将 JAR 文件传输到集群。

对于本例中的 UDF，Pig 使用 com.hadoopbook.pig.IsGoodQuality 名称进行查找，能在我们注册的 JAR 文件中找到它。

内置函数的解析也使用同样的方式处理。内置函数和 UDF 的处理有一个区别：

Pig 会搜索一组内置包。因此，对于内置函数的调用并不一定要提供完整的名称。例如，函数 MAX 实际上是由包 org.apache.pig.builtin 中的类 MAX 实现的。这也是 Pig 搜索的内置包，所以我们在 Pig 程序中可以使用 MAX 而不需要用 org.apache.pig.builtin.MAX。

我们可以通过使用以下命令行参数唤醒 Grunt，从而把我们的包加入搜索路径中：

```
-Dudf.import.list=com.hadoopbook.pig
```

或者，我们也可以使用 DEFINE 操作为函数定义别名，以缩短函数名：

```
grunt> DEFINE isGood com.hadoopbook.pig.IsGoodQuality();
grunt>filtered_records = FILTER records BY temperature != 9999 AND isGood(quality);
```

需要在一个脚本里多次使用同一个函数时，为该函数定义别名是一个好办法。如果要向 UDF 的实现类传递参数，必须定义别名。

如果注册了 JAR 文件并且将函数别名写入主目录下的*.pigbootup* 文件中，那么只要启动 Pig，它们就会运行。

使用类型

只有在质量字段的类型定义为 int 时，前面定义的过滤器才能正常工作。如果没有类型信息，这个 UDF 就不能正常处理，因为此时该字段的类型是默认类型 bytearray，表示为 DataByteArray 类，而 DataByteArray 不是整型，因此类型转换失败。

修正这一问题最直接的办法是在 exec()方法中把该字段转换成整型。但更好的办法是告诉 Pig 该函数所期望的各个字段的类型。EvalFunc 为此提供了 getArgToFuncMapping()方法。我们可以重载这个方法来告诉 Pig 第一个字段应该是整型。

```
@Override
public List<FuncSpec> getArgToFuncMapping() throws FrontendException {
  List<FuncSpec> funcSpecs = new ArrayList<FuncSpec>();
  funcSpecs.add(new FuncSpec(this.getClass().getName(),
    new Schema(new Schema.FieldSchema(null, DataType.INTEGER))));

  return funcSpecs;
}
```

这个方法为传递给 exec()方法的那个元组的每个字段返回一个 FuncSpec 对象。

在这个例子里，只有一个字段。我们构造一个匿名 FieldSchema (因为 Pig 在进行类型转换时忽略其名称，因此其名称以 null 传递)。该类型使用 Pig 的 DataType 类的常量 INTEGER 进行指定。

使用这个修改后的函数，Pig 将尝试把传递给函数的参数转换成整型。如果无法转换这个字段，则把这个字段传递为 null。如果字段为 null，exec()方法返回的结果总是 false。在这个应用中，因为我们想要在过滤掉质量字段无法理解的记录，所以这样做很合适。

16.5.2 计算 UDF

写计算函数比写过滤函数的步骤要稍微多一些。让我们考虑范例 16-2 中的 UDF。它类似于 java.lang.String 中 trim()方法，可以从 chararray 值中去掉开头和结尾的空白符。[①]

范例 16-2. 该 EvalFuncUDF 从 chararray 值中去除开头和结尾的空白符

```
public class Trim extends PrimitiveEvalFunc<String, String> {
  @Override
  public String exec(String input) {
    return input.trim();
  }
}
```

上述示例利用了 PrimitiveEvalFunc 函数，它是 EvalFunc 函数在输入数据类型为一个原子类型时的“特化”(specialization)。Trim UDF 的输入和输出数据类型皆为 String[②]。

在写计算函数的时候，需要考虑输出模式。在下面的语句中，B 的模式由函数 udf 决定：

```
B = FOREACH A GENERATE udf($0);
```

如果 udf 创建了有标量字段的元组，那么 Pig 可以通过“反射”(reflection)来确定 B 的模式。对于复杂数据类型，例如包、元组或映射，Pig 需要更多的信息。此时需要实现 outputSchema()方法将输出模式的相关信息告诉 Pig。

① 实际上 Pig 中有一个等价的内置函数，称为 TRIM。

② 虽然和这个示例无关，但如果计算函数要处理一个包(bag)，可能还需要另外实现 Pig 的 Algebraic 或 Accumulator 接口来提高以块(chunk)方式处理包的效率。

Trim UDF 返回一个字符串。Pig 把返回值翻译为 chararray 类型，如以下会话所示：

```
grunt>DUMP A;
( pomegranate)
(banana )
(apple)
( lychee )
grunt> DESCRIBE A;
A: {fruit: chararray}
grunt>B = FOREACH A GENERATE com.hadoopbook.pig.Trim(fruit);
grunt>DUMP B;
(pomegranate)
(banana)
(apple)
(lychee)
grunt>DESCRIBE B;
B: {chararray}
```

A 包含的 chararray 字段中有开头和结尾的空白符。我们将 Trim 函数应用于 A 的第一个字段(名为 fruit)，从而创建了 B。B 的字段被正确推断为 chararray。

动态调用

有时你希望使用由某个 Java 库提供的函数，而不是自己撰写 UDF。动态调用器(invoker)使你可以在 Pig 脚本中直接调用 Java 方法。使用这种方法的代价是，对于方法的调用是通过 Java 的反射(reflection)机制进行的。如果要为一个很大的数据集的每行记录进行调用，则会导致巨大的处理开销。所以，对于需要重复运行的脚本，最好使用专用的 UDF。

下面的代码片段展示如何定义和使用基于 Apache Commons Lang 的 StringUtils 类的 trim UDF：

```
grunt>DEFINE trim
InvokeForString('org.apache.commons.lang.StringUtils.trim','String');
grunt>B = FOREACH A GENERATE trim(fruit);
grunt>DUMP B;
(pomegranate)
(banana)
(apple)
(lychee)
```

使用调用器 InvokeForString 的原因是该方法的返回值类型是 String。此外还有 InvokeForInt、InvokeForLong、InvokeForDouble、InvokeForFload。调用器构造函数的第一个参数应当是可被调用的方法。第二个参数为以空格分隔的方法参数类的列表。

16.5.3 加载 UDF

我们将演示一个定制的加载函数，该函数可以通过指定纯文本的列的区域定义字段，和 Unix 的 cut 命令非常类似。[①]它的用法如下：

```
grunt>records = LOAD 'input/ncdc/micro/sample.txt'
>>USING com.hadoopbook.pig.CutLoadFunc('16-19,88-92,93-93')
>>AS (year:int, temperature:int, quality:int);
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

传递给 CutLoadFunc 的字符串是对列的说明：每一个由逗号分隔的区域定义了一个字段。字段的名称和数据类型通过 AS 子句进行定义。让我们来看范例 16-3 给出的 CutLoadFunc 的实现：

范例 11-3. 该加载函数以列区域作为字段加载元组

```
public class CutLoadFunc extends LoadFunc {

  private static final Log LOG = LogFactory.getLog(CutLoadFunc.class);

  private final List<Range> ranges;
  private final TupleFactory tupleFactory = TupleFactory.getInstance();
  private RecordReader reader;

  public CutLoadFunc(String cutPattern) {
    ranges = Range.parse(cutPattern);
  }

  @Override
  public void setLocation(String location, Job job)
      throws IOException {
    FileInputFormat.setInputPaths(job, location);
  }

  @Override
  public InputFormat getInputFormat() {
    return new TextInputFormat();
  }

  @Override
  public void prepareToRead(RecordReader reader, PigSplit split) {
    this.reader = reader;
  }
```

① 在 Piggy Bank 中有一个功能更全面的 UDF 可以完成同样的工作，称为 FixedWidthLoader。

```
@Override
public Tuple getNext() throws IOException {
  try {
    if (!reader.nextKeyValue()) {
      return null;
    }
    Text value = (Text) reader.getCurrentValue();
    String line = value.toString();
    Tuple tuple = tupleFactory.newTuple(ranges.size());
    for (int i = 0; i < ranges.size(); i++) {
      Range range = ranges.get(i);
      if (range.getEnd() > line.length()) {
        LOG.warn(String.format(
          " Range end (%s) is longer than line length (%s)" ,
          range.getEnd(), line.length()));
        continue;
      }
      tuple.set(i, new DataByteArray(range.getSubstring(line)));

    }
    return tuple;
  } catch (InterruptedException e) {
    throw new ExecException(e);
  }
 }
}
```

和 Hadoop 类似，Pig 的数据加载先于 mapper 运行，所以保证数据可以被分割成能被各个 mapper 独立处理的部分非常重要，8.2.1 节在讨论输入分片和记录时，介绍了更多背景知识。LoadFunc 一般使用底层已有的 InputFormat 来创建记录，而 LoadFunc 自身则提供把返回记录变为 Pig 元组的程序逻辑。

CutLoadFunc 类使用说明了每个字段列区域的字符串作为参数进行构造。解析该字符串并创建内部 Range 对象列表以封装这些区域的程序逻辑包含在 Range 类中。这里没有列出这些代码(可在本书所附的示例代码中找到)。

Pig 调用 LoadFunc 的 setLocation()把输入位置传输给加载器。因为 CutLoadFunc 使用 TextInputFormat 把输入切分成行，因此我们只用 FileInputFormat 的一个静态方法传递设置输入路径的位置信息。

Pig 使用了新的 MapReduceAPI。因此，我们使用的是 org.apache.hadoop.mapreduce 包的输入和输出格式及其关联的类。

然后，和在 MapReduce 中一样，Pig 调用 getInputFormat()方法为每一个分片

新建一个 RecordReader。Pig 把每个 RecordReader 传递给 CutLoadFunc 的 prepareToRead()方法以便通过引用来进行传递，这样，我们就可以在 getNext()方法中用它遍历记录。

Pig 运行时环境会反复调用 getNext()，然后加载函数从 reader 中读取元组直到 reader 读到分片中的最后一条记录。此时，加载函数返回空值 null 以报告已经没有可读的元组。

负责把输入文件的行转换为 Tuple 对象是 getNext()的任务。它利用 Pig 用于创建 Tuple 实例的类 TupleFactory 来完成这一工作。newTuple()方法新建一个包含指定字段数的元组。字段数就是 Range 类的个数，而这些字段使用 Range 对象所确定的输入行中的子串填充。

我们还需要考虑输入行比设定范围短的情况。一种选择是抛出异常并停止进一步的处理。如果你的应用不准备在碰到不完整或损坏的记录时继续工作，这样处理当然没有问题。在很多情况下，另一种更好的选择是返回一个有 null 字段的元组，然后让 Pig 脚本根据情况来处理不完整的数据。我们这里采取的是后一种方法：当区域超出行尾时，通过终止 for 循环把元组随后的字段都设成默认值 null。

使用模式

现在让我们来考虑加载的字段数据类型。如果用户指定模式，那么字段就需要转换成相应的数据类型。但在 Pig 中，这是在加载后进行的。因此，加载器应该始终用类型 DataByteArray 来构造包含 bytearray 字段的元组。当然，我们也可以让加载器函数来完成类型转换。这时需要重载 getLoadCaster()，以返回包含一组类型转换方法的定制 LoadCaster 接口实现。

CutLoadFunc 并没有重载 getLoadCaster()。因为默认的 getLoadCaster()实现返回了 Utf8StorageConverter。它提供了 UTF-8 编码数据到 Pig 数据类型的标准的转换功能。

在有些情况下，加载函数本身可以确定模式。例如，如果我们在加载 XML 或 JSON 这样的自描述数据，则可以为 Pig 创建一个模式来处理这些数据。此外，加载函数可以使用其他方法来确定模式，例如使用外部文件，或通过传递模式信息给构造函数。为了满足这些需要，加载函数应该(在实现 LoadFunc 接口之外)实现 LoadMetadata 接口，向 Pig 运行时环境提供模式。但是请注意，如果用户通过

LOAD 的 AS 子句定义模式，那么它的优先级将高于通过 LoadMetadata 接口定义的模式。

加载函数还可以实现 LoadPushDown 接口，了解查询需要哪些列。因为此时加载器可以只加载查询需要的列，因此这可能有助于按列存储的优化。在示例中，CutLoadFunc 需要读取元组的整行，所以只加载部分列不容易实现，鉴于此，我们在这里不使用这种优化技术。

16.6 数据处理操作

16.6.1 数据的加载和存储

在本章中，我们已经看过 Pig 如何从外部存储加载数据来进行处理。与之相似，存储处理结果也是非常直观的。下面的例子使用 PigStorage 将元组存储为以冒号分隔的纯文本值：

```
grunt> STORE A INTO 'out' USING PigStorage(':');
grunt> cat out
Joe:cherry:2
Ali:apple:3
Joe:banana:2
Eve:apple:7
```

其他内置存储函数参见前面的表 16-7。

16.6.2 数据的过滤

如果你已经把数据加载到关系中，那么下一步往往是对这些数据进行过滤，移除你不感兴趣的数据。通过在整个数据处理流水线的早期对数据进行过滤，可以使系统数据处理总量最小化，从而提升处理性能。

1. FOREACH…GENERATE 操作

我们已经介绍了如何使用带有简单表达式和 UDF 的 FILTER 操作从一个关系中移除行。FOREACH…GENERATE 操作用于逐个处理一个关系中的行。它可用于移除字段或创建新的字段。在这个示例里，我们既要删除字段，也要创建字段：

```
grunt> DUMP A;
(Joe,cherry,2)
(Ali,apple,3)
```

```
(Joe,banana,2)
(Eve,apple,7)
grunt>B = FOREACH A GENERATE $0, $2+1, 'Constant';
grunt>DUMP B;
(Joe,3,Constant)
(Ali,4,Constant)
(Joe,3,Constant)
(Eve,8,Constant)
```

在这里，我们已经创建了一个有三个字段的新关系 B。它的第一个字段是 A 的第一个字段($0)的投影。B 的第二个字段是 A 的第三个字段($2)加 1。B 的第三个字段是一个常量字段(即 B 中每一行在第三个字段的取值都相同)，其类型为 chararray，取值为 Constant。

FOREACH…GENERATE 操作可以使用嵌套形式以支持更复杂的处理。在如下示例中，我们计算天气数据集的多个统计值：

```
--year_stats.pig
REGISTER pig-examples.jar;
DEFINE isGood com.hadoopbook.pig.IsGoodQuality();
records = LOAD 'input/ncdc/all/19{1,2,3,4,5}0*'
  USING com.hadoopbook.pig.CutLoadFunc('5-10,11-15,16-19,88-92,93-93')
  AS (usaf:chararray, wban:chararray, year:int, temperature:int, quality:int);

grouped_records = GROUP records BY year PARALLEL 30;

year_stats = FOREACH grouped_records {
uniq_stations = DISTINCT records.usaf;
good_records = FILTER records BY isGood(quality);
GENERATE FLATTEN(group), COUNT(uniq_stations) AS station_count,
    COUNT(good_records) AS good_record_count, COUNT(records) AS record_count;
}

DUMP year_stats;
```

通过使用我们前面开发的 cut UDF，我们从输入数据集加载多个字段到 records 关系中。接下来，我们根据年份对 records 进行分组。请注意，我们使用关键字 PARALLEL 来设置要使用多少个 reducer。这在使用集群进行处理时非常重要。然后，我们使用嵌套的 FOREACH…GENERATE 操作对每个组分别进行处理。第一重嵌套语句使用 DISTINCT 操作为每一个气象观测站的 USAF 标识创建一个关系。第二层嵌套语句使用 FILTER 操作和一个 UDF 为包含“好”的读数的记录创建一个关系。最后一层嵌套语句是 GENERATE 语句(嵌套 FOREACH…GENERATE 语句必须以 GENERATE 语句作为最后一层嵌套语句)。该语句使用分组后的记录和嵌套语句块创建的关系生成了需要的汇总字段。

在若干年数据上运行以上程序，我们得到如下结果：

```
(1920,8L,8595L,8595L)
(1950,1988L,8635452L,8641353L)
(1930,121L,89245L,89262L)
(1910,7L,7650L,7650L)
(1940,732L,1052333L,1052976L)
```

这些字段分别表示年份、不同气象观测站的个数、好的读数的总数和总的读数。从中我们可以看到气象观测站个数和读数个数是如何随着时间变化而增长的。

2. STREAM 操作

STREAM 操作让你可以用外部程序或脚本对关系中的数据进行变换。这一操作的命名对应于 Hadoop 的 Streaming，后者为 MapReduce 的提供类似能力(参见 2.5 节对 Hadoop Streaming 的讨论)。

STREAM 可以使用内置命令作为参数。下面的例子使用 Unix cut 命令从 A 中每个元组抽取第二个字段。注意，命令及其参数要用反向撇号引用：

```
grunt>C = STREAM A THROUGH `cut -f 2`;
grunt> DUMP C;
(cherry)
(apple)
(banana)
(apple)
```

STREAM 操作使用 PigStorage 来序列化/反序列化关系，输出为程序的标准输出流或从标准输入流读入。A 中的元组变换成由制表符分隔的行，然后传递给脚本。脚本的输出结果被逐行读入，并根据制表符来划分以创建新的元组，然后输出到关系 C。也可以使用 DEFINE 操作，通过实现 PigToStream 和 StreamToPig(两者都包含在 org.apache.pig 包中)来提供定制的序列化和反序列化程序。

在编写定制的处理脚本时，Pig 流式处理是最有用的。以下的 Python 脚本用于筛选气温记录：

```
#!/usr/bin/env python

import re
import sys

for line in sys.stdin:
  (year, temp, q) = line.strip().split()
  if (temp != "9999" and re.match("[01459]", q)):
    print "%s\t%s" % (year, temp)
```

要使用这一脚本，需要把脚本传输到集群上。这可以通过 DEFINE 子句来完成。该子句还为 STREAM 命令创建了一个别名。然后，便可以像下面的 Pig 脚本那样在 STREAM 语句中使用该别名：

```
--max_temp_filter_stream.pig
DEFINE is_good_quality `is_good_quality.py`
  SHIP ('ch16-pig/src/main/python/is_good_quality.py');
records = LOAD 'input/ncdc/micro-tab/sample.txt'
  AS (year:chararray, temperature:int, quality:int);
filtered_records = STREAM records THROUGH is_good_quality
  AS (year:chararray, temperature:int);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
  MAX(filtered_records.temperature);
DUMP max_temp;
```

16.6.3 数据的分组与连接

在 MapReduce 中对数据集进行连接操作需要程序员写不少程序，详情参见 9.3 节对连接的讨论。Pig 为连接操作提供很好的内置支持，简化了数据集的连接。因为只有非规范化的大规模数据集才最适宜使用 Pig(或 MapReduce)这样的工具进行分析，因此连接在 Pig 中的使用频率远小于在 SQL 中的使用频率。

1. 关于 JOIN 语句

让我们来看一个内连接的示例。考虑有如下关系 A 和 B：

```
grunt> DUMP A;
(2,Tie)
(4,Coat)
(3,Hat)
(1,Scarf)
grunt> DUMP B;
(Joe,2)
(Hank,4)
(Ali,0)
(Eve,3)
(Hank,2)
```

我们可以在两个关系的数值型(标识符)属性上对它们进行连接操作：

```
grunt>C = JOIN A BY $0, B BY $1;
grunt>DUMP C;
(2,Tie,Joe,2)
(2,Tie,Hank,2)
(3,Hat,Eve,3)
```

```
(4,Coat,Hank,4)
```

这是一个典型的内连接(inner join)操作：两个关系元组的每次匹配都和结果中的一行相对应。因为连接谓词(join predicate)为相等，所以这其实是一个等值连接(equijoin)。结果中的字段由所有输入关系的所有字段组成。

如果要进行连接的关系太大，不能全部放在内存中，则应该使用通用的连接操作。如果有一个关系小到能够全部放在内存中，则可以使用一种特殊的连接操作，即**分段复制连接**(fragment replicate join)，它把小的输入关系发送到所有mapper，并在 map 端使用内存查找表对(分段的)较大的关系进行连接。要使用特殊的语法让 Pig 使用分段复制连接[①]：

```
grunt>C = JOIN A BY $0, B BY $1 USING "replicated";
```

这里，第一个关系必须是大的关系，后面则是一个或多个相对较小的关系(能够全部存放在内存中)。

Pig 也支持通过使用类似于 SQL 的语法(17.7.3 节在讨论外连接时将介绍 Hive 相关语法)进行外连接(outer join)。例如：

```
grunt>C = JOIN A BY $0 LEFT OUTER, B BY $1;
grunt>DUMP C;
(1,Scarf,,)
(2,Tie,Joe,2)
(2,Tie,Hank,2)
(3,Hat,Eve,3)
(4,Coat,Hank,4)
```

2. 关于 COGROUP 语句

JOIN 结果的结构总是“平面”的，即一组元组。COGROUP 语句和 JOIN 类似，但是不同点在于，它会创建一组嵌套的输出元组集合。如果你希望利用如下语句中输出结果那样的结构，那么 COGROUP 将会有用：

```
grunt>D = COGROUP A BY $0, B BY $1;
grunt>DUMP D;
(0,{},{(Ali,0)})
(1,{(1,Scarf)},{})
(2,{(2,Tie)},{(Hank,2),(Joe,2)})
(3,{(3,Hat)},{(Eve,3)})
(4,{(4,Coat)},{(Hank,4)})
```

① 在 USING 子句中还可以使用其他关键词，包括“skewed”(为包含偏斜的键值空间的大规模数据集使用)、“merge”(为在要连接的键上已经进行了排序的输入关系上使用合并连接)和“merge-sparse”(小于等于 1%的数据是匹配的)。具体如何使用这些特殊的连接操作，请参见 Pig 的帮助文档。

COGROUP 为每个不同的分组键值生成一个元组。每个元组的第一个字段就是那个键值。其他字段是各个关系中匹配该键值的元组所组成的包。第一个包中包含关系 A 中有该键值的匹配元组。同样，第二个包中包含关系 B 中有该键值的匹配元组。

如果某个键值在一个关系中没有匹配的元组，那么对应于这个关系的包就为空。在前面的示例中，因为没有人购买围巾(ID 为 1)，所以对应元组的第二个包就为空。这是一个外连接的例子。COGROUP 的默认类型是外连接。可以使用关键词 OUTER 来显式指明使用外连接，COGROUP 产生的结果和前一个语句相同：

```
D = COGROUP A BY $0 OUTER, B BY $1 OUTER;
```

也可以使用关键词 INNER 让 COGROUP 使用内连接的语义，剔除包含空包的行。INNER 关键词是针对关系进行使用的，因此如下语句只去除关系 A 中不匹配的行(在这个示例中就是去掉未知商品 0 对应的行)：

```
grunt>E = COGROUP A BY $0 INNER, B BY $1;
grunt>DUMP E;
(1,{(1,Scarf)},{})
(2,{(2,Tie)},{(Hank,2),(Joe,2)})
(3,{(3,Hat)},{(Eve,3)})
(4,{(4,Coat)},{(Hank,4)})
```

我们可以把这个结构平面化，从 A 找出买了每一项商品的人。

```
grunt>F = FOREACH E GENERATE FLATTEN(A), B.$0;
grunt>DUMP F;
(1,Scarf,{})
(2,Tie,{(Hank),(Joe)})
(3,Hat,{(Eve)})
(4,Coat,{(Hank)})
```

把 COGROUP、INNER 和 FLATTEN(消除嵌套)组合起来使用相当于实现了(内)连接：

```
grunt>G = COGROUP A BY $0 INNER, B BY $1 INNER;
grunt>H = FOREACH G GENERATE FLATTEN($1), FLATTEN($2);
grunt>DUMP H;
(2,Tie,Joe,2)
(2,Tie,Hank,2)
(3,Hat,Eve,3)
(4,Coat,Hank,4)
```

这和 JOIN A BY $0,B BY $1 的结果是一样的。

如果要连接的键由多个字段组成，则可以在 JOIN 或 COGROUP 语句的 BY 子句中把

它们都列出来。这时要保证每个 BY 子句中的字段个数相同。

下面是如何在 Pig 中进行连接的另一个示例。该脚本计算输入的时间段内每个观测站报告的最高气温：

```
--max_temp_station_name.pig
REGISTER pig-examples.jar;
DEFINE isGood com.hadoopbook.pig.IsGoodQuality();

stations = LOAD 'input/ncdc/metadata/stations-fixed-width.txt'
  USING com.hadoopbook.pig.CutLoadFunc('1-6,8-12,14-42')
  AS (usaf:chararray, wban:chararray, name:chararray);

trimmed_stations = FOREACH stations GENERATE usaf, wban, TRIM(name);

records = LOAD 'input/ncdc/all/191*'
  USING com.hadoopbook.pig.CutLoadFunc('5-10,11-15,88-92,93-93')
  AS (usaf:chararray, wban:chararray, temperature:int, quality:int);

filtered_records = FILTER records BY temperature != 9999 AND isGood(quality);
grouped_records = GROUP filtered_records BY (usaf, wban) PARALLEL 30;
max_temp = FOREACH grouped_records GENERATE FLATTEN(group),
  MAX(filtered_records.temperature);
max_temp_named = JOIN max_temp BY (usaf, wban), trimmed_stations BY (usaf, wban)
  PARALLEL 30;
max_temp_result = FOREACH max_temp_named GENERATE $0, $1, $5, $2;

STORE max_temp_result INTO 'max_temp_by_station';
```

我们使用先前开发的 cut UDF 来加载包括气象观测站 ID(USAF 和 WBAN 标识)、名称的关系以及包含所有气象记录且以观测站 ID 为键的关系。我们在根据气象观测站进行连接之前，先根据观测站 ID 对气象记录进行分组和过滤，并计算最高气温的聚集值。最后，在进行连接之后，我们把所需要的字段——即 USAF、WBAN、观测站名称和最高气温——投影到最终结果。

下面是 20 世纪头 10 年的结果：

```
228020 99999 SORTAVALA      322
029110 99999 VAASAAIRPORT   300
040650 99999 GRIMSEY        378
```

因为观测站的元数据较少，所以这个查询可以通过使用分段复制连接来进一步提升运行效率。

3. 关于 CROSS 语句

PigLatin 包含**叉乘**(cross-product，也称“笛卡儿积”)操作。这一操作把一个关系

中的每个元组和第二个中的所有元组进行连接(如果有更多的关系，那么这个操作就进一步把结果逐一和这些关系的每一个元组进行连接)。这个操作的输出结果的大小是输入关系的大小的乘积。输出结果可能会非常大：

```
grunt>I = CROSS A, B;
grunt>DUMP I;
(2,Tie,Joe,2)
(2,Tie,Hank,4)
(2,Tie,Ali,0)
(2,Tie,Eve,3)
(2,Tie,Hank,2)
(4,Coat,Joe,2)
(4,Coat,Hank,4)
(4,Coat,Ali,0)
(4,Coat,Eve,3)
(4,Coat,Hank,2)
(3,Hat,Joe,2)
(3,Hat,Hank,4)
(3,Hat,Ali,0)
(3,Hat,Eve,3)
(3,Hat,Hank,2)
(1,Scarf,Joe,2)
(1,Scarf,Hank,4)
(1,Scarf,Ali,0)
(1,Scarf,Eve,3)
(1,Scarf,Hank,2)
```

在处理大规模数据集时，应该尽量避免会产生平方(或更差)级中间结果的操作。只有在极少数情况下，才需要对整个输入数据集计算叉乘。

例如，一开始，用户可能觉得必须生成文档集合中所有文档的两两配对组合才能计算文档两两之间的相似度。但是，随着对数据和应用的深入了解，他会发现大多数文档配对的相似度为零(即它们之间没有关系)。于是，我们就能找到一种更好的算法来计算相似度。

在此，解决这一问题的主要思路是把计算聚焦于用于计算相似度的实体，如文档中的关键词(term)，让它们成为算法的核心。事实上，我们还要删去对区分文档没有帮助的词，即停用词(stop-word)，进一步缩减问题的搜索空间。使用这一技术，分析近一百万个(10^6)文档大约会产生约十亿个(10^9)中间结果文档配对。[①]而如果用朴素的方法(即生成输入集合的叉乘)或不消除停用词，会产生一万亿个(10^{12})

① 引文出自 Tamer Elsayed, Jimmy Lin 和 Douglas Woard 的文章，标题为 "Pairwise Document Similarity in Large Collections with MapReduce", Proceedings of the 46th Annual Meeting of the Association of Computational Linguistics, June 2008，网址为 *http://bit.ly/doc_similarity*。

个文档配对。

4. 关于 GROUP 语句

COGROUP 用于把两个或多个关系中的数据放到一起，而 GROUP 语句则对一个关系中的数据进行分组。GROUP 不仅支持对键值进行分组(即把键值相同的元组放到一起)，你还可以使用表达式或用户自定义函数作为分组键。例如，有如下关系 A：

```
grunt> DUMP A;
(Joe,cherry)
(Ali,apple)
(Joe,banana)
(Eve,apple)
```

我们根据这个关系的第二个字段的字符个数进行分组：

```
grunt>B = GROUP A BY SIZE($1);
grunt>DUMP B;
(5,{(Eve,apple),(Ali,apple)})
(6,{(Joe,banana),(Joe,cherry)})
```

GROUP 会创建一个关系，它的第一个字段是分组字段，其别名为 group。第二个字段是包含与原关系(在本示例中就是 A)模式相同的被分组字段的包。

有两种特殊的分组操作：ALL 和 ANY。ALL 把一个关系中的所有元组放入一个包。这和使用某个常量函数作为分组函数所获得的结果一样：

```
grunt> C = GROUP A ALL;
grunt> DUMP C;
(all,{(Eve,apple),(Joe,banana),(Ali,apple),(Joe,cherry)})
```

注意，在这种 GROUP 语句中，没有关键词 BY。ALL 分组常用于计算关系中的元组个数(参见 16.4.5 节对验证与空值的讨论)。

关键词 ANY 用于对关系中的元组随机分组。它对于取样非常有用。

16.6.4 数据的排序

Pig 中的关系是无序的。考虑如下关系 A：

```
grunt> DUMP A;
(2,3)
(1,2)
(2,4)
```

Pig 按什么顺序来处理这个关系中的行是不一定的。特别是在使用 DUMP 或 STORE

检索 A 中的内容时，Pig 可能以任何顺序输出结果。如果想设置输出的顺序，可以使用 ORDER 操作按照某个或某几个字段对关系中的数据进行排序。默认的排序方式是对具有相同类型的字段值使用自然序进行排序，而不同类型字段值之间的排序则是任意的、确定的(例如，一个元组总是小于一个包)。

如下示例对 A 中元组根据第一个字段的升序和第二个字段的降序进行排序：

```
grunt> B = ORDER A BY $0, $1 DESC;
grunt> DUMP B;
(1,2)
(2,4)
(2,3)
```

对排序后关系的后续处理并不保证能够维持已排好的顺序。例如：

```
grunt> C = FOREACH B GENERATE *;
```

即使关系 C 和关系 B 有相同的内容，关系 C 用 DUMP 或 STORE 仍然可能产生以任意顺序排列的输出结果。正是由于这样，通常只在获取结果前一步才使用 ORDER 操作。

LIMIT 语句对于限制结果的大小以快速获得一个关系样本，非常有用。(用于随机取样的 SAMPLE 操作或者使用 ILLUSTRATE 命令的“取原型化”(prototyping)操作则更适用于根据数据产生有代表性的样本。)LIMIT 语句可紧跟 ORDER 语句使用，来获得排在最前面的 *n* 个元组。通常，LIMIT 会随意选择一个关系中的 *n* 个元组。但是，当它紧跟 ORDER 语句使用时，ORDER 产生的次序会继续保持(这和其他操作不保持输入关系数据顺序的规则不同，是一个例外)：

```
grunt>D = LIMIT B 2;
grunt>DUMP D;
(1,2)
(2,4)
```

如果所给的限制值远远大于关系中所有元组个数的总数，则返回所有元组(LIMIT 操作没有作用)。

使用 LIMIT 能够提升系统的性能。因为 Pig 会在处理流水线中尽早使用限制操作，以最小化需要处理的数据总量。因此，如果不需要所有输出数据，就应该用 LIMIT 操作。

16.6.5 数据的组合和切分

有时，你希望把几个关系组合在一起。为此，可以使用 UNION 语句。例如：

```
grunt>DUMP A;
(2,3)
(1,2)
(2,4)
grunt>DUMP B;
(z,x,8)
(w,y,1)
grunt> C = UNION A, B;
grunt> DUMP C;
(2,3)
(z,x,8)
(1,2)
(w,y,1)
(2,4)
```

C 是关系 A 和 B 的“并”(union)。因为关系本身是无序的，因此 C 中元组的顺序是不确定的。另外，如示例中那样，我们可以对两个模式不同或字段个数不同的关系进行并操作。Pig 会试图合并正在执行 UNION 操作的两个关系的模式。在这个例子中，两个模式是不兼容的，因此 C 没有模式：

```
grunt>DESCRIBE A;
A: {f0: int,f1: int}
grunt>DESCRIBE B;
B: {f0: chararray,f1: chararray,f2: int}
grunt>DESCRIBE C;
Schema for C unknown.
```

如果输出关系没有模式，就需要脚本能处理字段个数和数据类型都不同的元组。

SPLIT 操作是 UNION 操作的反操作。它把一个关系划分成两个或多个关系。可以参考 16.4.5 节讨论验证与空值时所提供的示例，了解如何使用 SPLIT。

16.7 Pig 实战

在开发和运行 Pig 应用程序时，知道一些实用技术是非常有帮助的。本小节将介绍一些这样的技术。

16.7.1 并行处理

运行在 MapReduce 模式时，一件重要的事情是使得处理的并行度与所处理的数据

集大小相匹配。默认情况下，Pig 将根据输入数据的大小设置 reducer 的个数：每 1 GB 输入使用一个 reducer，且 reducer 的个数不超过 999。可以通过设置 pig.exec.reducers.bytes.per.reducer(默认为 1 000 000 000 字节)和 pig.exec.reducers.max(默认为 999)来修改这一设置。

为了告诉 Pig 每个作业要用多少个 reducer，可以在 reduce 阶段的操作中使用 PARALLEL 子句。在 reduce 阶段使用的操作包括所有的“分组”(grouping)和“连接”(joining)操作(GROUP、COGROUP、JOIN 和 CROSS)，以及 DISCTINCT 和 ORDER。下面的代码将 GROUP 的 reducer 个数设为 30：

```
grouped_records = GROUP records BY year PARALLEL 30;
```

也可以通过设置 default_parallel 选项来达到同样的目的。修改的选项将作用于所有后续的作业：

```
grunt>set default_parallel 30
```

设置 reduce 任务个数的一种比较好的方式是把该参数设为稍小于集群中的 reduce 任务的时隙(slot)的个数。对于这个问题的详细讨论，可以参见 8.1.1 节，了解如何选择 reducer 的个数。

map 任务的个数由输入的大小决定(每个 HDFS 块一个 map)，不受 PARALLEL 子句的影响。

16.7.2 匿名关系

在刚刚定义了一个关系之后，通常都会紧跟着 DUMP 或 DESCRIBE 之类的诊断的操作。由于这种情况非常普遍，因此 Pig 使用@来指向前一个关系，这是一种快捷方式。同样，在使用解释器时需要为每个关系取名也是件烦人的事。Pig 允许使用特殊语法=>来创建没有别名的关系，而此类关系就只能通过@来引用。示例如下：

```
grunt>=> LOAD 'input/ncdc/micro-tab/sample.txt';
grunt>DUMP @
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

16.7.3 参数代换

如果有定期运行的 Pig 脚本，你可能希望让这个脚本能够在不同参数设置下运行。例如，一个每天运行一次的脚本可能要根据日期来决定它要处理哪些输入文件。Pig 支持**参数代换**(parametersubstitution)，即用运行时提供的值替换脚本中的参数。参数由前缀为$字符的标识符来表示。例如，在以下脚本中，$input 和 $output 用来指定输入和输出路径：

```
--max_temp_param.pig
records = LOAD '$input' AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
  quality IN (0, 1, 4, 5, 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
  MAX(filtered_records.temperature);
STORE max_temp into '$output';
```

参数值可以在启动 Pig 时使用-param 选项指定，每个参数一个：

```
% pig-param input=/user/tom/input/ncdc/micro-tab/sample.txt \
> -param output=/tmp/out \
> ch16-pig/src/main/pig/max_temp_param.pig
```

也可以把参数值放在文件中，通过-param_file 选项把参数传递给 Pig。例如，把参数定义放在文件中，也可以获得相同的结果：

```
# Input file
input=/user/tom/input/ncdc/micro-tab/sample.txt
# Output file
output=/tmp/out
```

对 Pig 的调用相应进行如下调整：

```
% pig -param_file ch16-pig/src/main/pig/max_temp_param.param \
> ch16-pig/src/main/pig/max_temp_param.pig
```

可以重复使用-param_file 来指定多个参数文件，还可以同时使用-param 和-param_file 选项。如果同一个参数在参数文件和命令行中都有定义，那么命令行中最后出现的参数值优先级最高。

1. 动态参数

针对使用-param 选项来提供的参数，很容易使其值变成动态的，运行命令或脚本即可变为动态的。很多 Unix 的 shell 环境中都用反引号引用的命令来替换实际值。我们可以使用这一功能实现根据日期来确定输出目录：

```
% pig -param input=/user/tom/input/ncdc/micro-tab/sample.txt \
> -param output=/tmp/`date "+%Y-%m-%d" `/out \
> ch16-pig/src/main/pig/max_temp_param.pig
```

Pig 也支持在参数文件中的反引号，在 shell 中执行用反引号引用的命令，并使用 shell 的输出结果作为替换值。如果命令或脚本返回一个非零的退出状态并退出，Pig 会报告错误消息并终止执行。在参数文件中使用反引号是一种很有用的特性，它意味着可以使用完全相同的方法在文件中或命令行中定义参数。

2. 参数代换处理

参数代换是脚本运行前的一个预处理步骤。可以使用-dryrun 选项运行 Pig 来查看预处理器所进行的代换。在-dryrun 模式下，Pig 对参数进行代换(以及宏扩展)并生成一个使用了代换值的原来脚本的副本，但并不执行该脚本。在普通模式下，可以在运行之前查看生成的脚本，检查参数代换是否合理(例如，在动态生成代换的情况下)。

16.8 延伸阅读

本章介绍了如何使用 Pig 的基本情况。若想了解更详细的内容，请参考 O'Reilly 在 2011 年出版的 *Programming Pig*，网址为(*http://shop.oreilly.com/product/0636920018087.do*，作者 Alan Gates。

第 17 章

关于 Hive

在标题为“Information Platforms and the Rise of the Data Scientist”的文章一文中，Jeff Hammerbacher[1]把“信息平台”描述为“企业摄取(ingest)、处理(process)、生成(generate)信息的行为”与“帮助加速从经验数据中学习”的“中心”。

在 Facebook，Jeff 团队所构建的信息平台中，最庞大的组成部分是 Apache Hive (*https://hive.apache.org/*)。Hive 是一个构建在 Hadoop 上的数据仓库框架，是应 Facebook 每天产生的海量新兴社会网络数据进行管理和(机器)学习的需求而产生和发展的。在尝试了不同系统之后，Jeff 团队选择 Hadoop 来存储和处理数据，因为 Hadoop 的性价比高，同时还能够满足他们的可伸缩性要求。

Hive 的设计目的是让精通 SQL 技能但 Java 编程技能相对较弱的分析师能够对 Facebook 存放在 HDFS 中的大规模数据集执行查询。今天，Hive 已经是一个成功的 Apache 项目，很多组织把它用作一个通用的、可伸缩的数据处理平台。

当然，SQL 并不是所有大数据问题的理想工具。例如，它并不适合用来开发复杂的机器学习算法。但它对很多分析任务非常有用，而且它的另一个优势是业内人士都非常熟悉它。此外，SQL 是商业智能工具的“通用语言”(可以通过 ODBC 这一桥梁来用)，Hive 有条件和这些产品进行集成。

① 编注：最早提出“数据科学家”这个头衔的人，也是 Facebook 数据科学团队的负责人，Cloudera 的联合创始人。这位出生于 1983 年的数学天才有一句名言：“我们这一代最杰出的头脑都在拼命思考如何吸引人们点击更多的广告，这个感觉糟透了。”他后来加入纽约著名的西奈山医院，成为一名医学研究者，运用自己的数据分析才能去攻克癌症、老年痴呆症、糖尿病及其他慢性疾病。

本章介绍如何使用 Hive。我们假设你用过 SQL 和常见的数据库体系结构。在介绍 Hive 特性的同时，我们会经常将这些特性与其传统 RDBMS 对应部分进行比较。

17.1 安装 Hive

Hive 一般在工作站上运行。它把 SQL 查询转换为一系列在 Hadoop 集群上运行的作业。Hive 把数据组织为表，通过这种方式为存储在 HDFS 上的数据赋予结构。元数据(如表模式)存储在 metastore 数据库中。

刚开始使用 Hive 时，为了方便，可以让 metastore 运行在本地机器上。这一设置是默认设置。此时，创建的 Hive 表的定义是在本地机器上，所以无法和其他用户共享这些定义。17.3.3 节将介绍如何设置生产环境中常用的远程共享 metastore。

安装 Hive 的过程非常简单。首先必须在本地安装和集群上相同版本的 Hadoop。[①] 当然，在刚开始使用 Hive 时，你可能会选择在本地以独立模式或伪分布模式运行 Hadoop。对于这些选项的介绍，可参见附录 A。

Hive 能和哪些版本的 Hadoop 共同工作？

每个 Hive 的发布版本都被设计为能够和多个版本的 Hadoop 共同工作。一般而言，Hive 支持 Hadoop 最新发布的稳定版本以及之前的老版本。这些信息列在发布说明中。只要确保 hadoop 可执行文件在相应的路径中或设置 HADOOP_HOME 环境变量，就不必另行告诉 Hive 当前正在使用哪个版本的 Hadoop。

下载 Hive 的一个发布版本(*http://hive.apache.org/downloads.html*)，然后把压缩包解压到工作站上合适的位置：

```
% tar xzf apache-hive-x.y.z-bin.tar.gz
```

把 Hive 放在你自己的路径下以便于访问：

```
% export HIVE_HOME=~/sw/apache-hive-x.y.z-bin
% export PATH=$PATH:$HIVE_HOME/bin
```

现在，键入 hive 启动 Hive 的 shell 环境：

① 假设工作站和 Hadoop 集群之间已有网络连接。可以在运行 Hive 之前在本地安装 Hadoop，并通过 hadoop fs 命令执行一些 HDFS 操作，测试两者的版本是否相同。

```
% hive
hive>
```

Hive 的 shell 环境

Hive 的 shell 环境是我们和 Hive 交互、发出 HiveQL 命令的主要方式。HiveQL 是 Hive 的查询语言。它是 SQL 的一种“方言”。它的设计在很大程度上深受 MySQL 的影响。因此，如果熟悉 MySQL，你会觉得 Hive 很亲切。

第一次启动 Hive 时，我们可以通过列出 Hive 的表来检查 Hive 是否正常工作，此时应该没有任何表。命令必须以分号结束，告诉 Hive 立即执行该命令：

```
hive> SHOW TABLES;
OK
Time taken: 0.473 seconds
```

和 SQL 类似，HiveQL 一般是大小写不敏感的(除了字符串比较以外)，因此 `show tables;`和上面的命令效果相同。制表符(Tab)会自动补全 Hive 的关键字和函数。

对于全新安装，这个命令会花几秒钟来执行。因为系统采用“延迟”(lazy)策略，所以直到此时才在机器上创建 metastore 数据库。(该数据库把相关文件放在运行 `hive` 命令那个位置下的 *metastore_db* 目录中。)

也可以以非交互式模式运行 Hive 的 shell 环境。使用`-f` 选项可以运行指定文件中的命令。在这个示例中，我们运行脚本文件 *script.q*：

```
% hive -f script.q
```

对于较短的脚本，可用`-e` 选项在行内嵌入命令。此时不需要表示结束的分号：

```
% hive -e 'SELECT * FROM dummy'
OK
X
Time taken: 1.22 seconds, Fetched: 1 row(s)
```

有一个较小的数据表用于测试查询是很有用的。例如，我们可以用文本数据测试 SELECT 表达式中的函数(参见 17.5.2 节对操作符和函数的讨论)。下面是一个生成一个单行表的方法：

```
% echo 'X' > /tmp/dummy.txt
% hive -e " CREATE TABLE dummy (value STRING); \
LOAD DATA LOCAL INPATH '/tmp/dummy.txt' \
OVERWRITE INTO TABLE dummy"
```

无论是在交互式还是非交互式模式下，Hive 都会把操作运行时的信息打印输出到

标准错误输出(standard error)，例如运行一个查询所花的时间。可以在启动程序的时候使用-S 选项强制不显示这些消息，只输出查询结果：

```
% hive -S -e 'SELECT * FROM dummy'
X
```

其他比较有用的 Hive Shell 的特性包括：使用!前缀来运行宿主操作系统的命令；使用 dfs 命令来访问 Hadoop 文件系统。

17.2 示例

让我们看一下如何用 Hive 查询我们在前面几章使用的气象数据集。第一个步骤是把数据加载到 Hive 管理的存储。在这里，我们将让 Hive 把数据存储在本地文件系统。稍后我们会介绍如何把表存储到 HDFS。

和 RDBMS 一样，Hive 把数据组织成表。我们使用 CREATE TABLE 语句为气象数据新建一个表：

```
CREATE TABLE records (year STRING, temperature INT, quality INT)
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '\t';
```

第一行声明一个 records 表，包含三列：year，temperature 和 quality。还必须指明每一列的数据类型，在这里，年为字符串类型，另外两列为整数型。

到目前为止，所用的 SQL 都是我们所熟悉的。但是接下来的 ROW FORMAT 子句是 HiveQL 所特有的。这个子句所声明的是数据文件的每一行是由制表符分隔的文本。Hive 按照这一格式读取数据：每行三个字段，分别对应于表中的三列，字段间以制表符分隔，每行以换行符分隔。

接下来，可以向 Hive 输入数据。这里出于探索的目的，只用一个很小的样本数据集：

```
LOAD DATA LOCAL INPATH 'input/ncdc/micro-tab/sample.txt'
OVERWRITE INTO TABLE records;
```

这一命令告诉 Hive 把指定的本地文件放入其仓库目录中。这只是一个简单的文件系统操作。这个操作并不解析文件或把它存储为内部数据库格式，因为 Hive 并不强制使用任何特定文件格式。文件以原样逐字存储，Hive 不会对文件进行修改。

在这个示例中，我们把 Hive 表存储在本地文件系统中(fs.default.name 设为默

认值 file:///)。在 Hive 的仓库目录中，表存储为目录。仓库目录由选项 hive.metastore.warehouse.dir 控制，默认值为*/usr/hive/warehouse*。

这样，records 表的文件便可以在本地文件系统的*/usr/hive/warehouse/records* 目录中找到：

```
% ls /user/hive/warehouse/record/
sample.txt
```

在这个示例中，我们现在只有一个文件 *sample.txt*，但是在一般情况下，可以有多个文件，而且 Hive 会在查询表的时候读入所有这些文件。

LOAD DATA 语句中的 OVERWRITE 关键字告诉 Hive 删除表对应目录中已有的所有文件。如果省去这一关键字，Hive 就简单地把新的文件加入目录(除非目录下正好有同名的文件，此时将替换掉原有的同名文件)。

数据现在已经在 Hive 中，我们可以对它运行一个查询：

```
hive> SELECT year, MAX(temperature)
    > FROM records
    > WHERE temperature != 9999 AND quality IN (0, 1, 4, 5, 9)
    > GROUP BY year;
1949 111
1950 22
```

这个 SQL 查询没什么特别的：它是一个带 GROUP BY 子句的 SELECT 语句。这个查询根据年份对行进行分组，然后使用 MAX()聚集函数在每个年份组中找到最高气温。Hive 的优势在于把这个查询转化为一个作业并为我们执行这个作业，然后把结果打印输出到控制台。虽然 Hive 和其他数据库有一些细微的差别，例如 Hive 支持的 SQL 的结构以及可以查询中数据的格式等(我们在本章中将探究差别)但能够在原始数据上执行 SQL 查询，才能彰显出 Hive 的强大功能。

17.3 运行 Hive

这一节介绍运行 Hive 的一些更实用的技术，包括如何设置 Hive 使其能运行在 Hadoop 集群和共享的 metastore 上。为此，我们会深入介绍 Hive 体系结构。

17.3.1 配置 Hive

和 Hadoop 类似，Hive 用 XML 配置文件进行设置。配置文件为 *hivesite.xml*，它在

Hive 的 *conf* 目录下。通过这个文件，可以设置每次运行 Hive 时希望 Hive 使用的选项。该目录下还包括 *hive-default.xml*(其中记录 Hive 使用的选项及其默认值)。

传递`--config` 选项参数给 `hive` 命令，可以通过这种方式重新定义 Hive 查找 *hive-site.xml* 文件的目录：

```
% hive --config /Users/tom/dev/hive-conf
```

注意，这个选项指定的是包含配置文件的目录，而不是配置文件 *hive-site.xml* 本身。这对于有(对应于多个集群的)多个站点文件时很有用，可以方便地在这些站点文件之间进行切换。还有另一种方法，可以设置 `HIVE_CONF_DIR` 环境变量来指定配置文件目录，效果相同。

hive-site.xml 文件最适合存放详细的集群连接信息，因为可以使用 Hadoop 属性 `fs.defaultFS` 和 `yarn.resourcemanager.address` 来指定文件系统和资源管理器(关于配置 Hadoop 的详细信息，请参见附录 A)。如果没有设定这两个参数，它们就像在 Hadoop 中一样，被设为默认值，也就是使用本地文件系统和本地(正在运行的)“作业运行器”(job runner)，这对于试着用 Hive 来处理测试数据集非常方便。metastore 的配置选项(参见 17.3.3 节对 metastore 的讨论)一般也能在 *hive-site.xml* 中找到。

Hive 还允许向 `hive` 命令传递`-hiveconf` 选项来为单个会话(per-session)设置属性。例如，下面的命令设定在会话中使用一个(伪分布)集群：

```
% hive -hiveconf fs.defaultFS=hdfs://localhost \
  -hiveconf mapreduce.framework.name=yarn \
  -hiveconf yarn.resourcemanager.address=localhost:8032
```

如果准备让多个 Hive 用户共享一个 Hadoop 集群，则需要使 Hive 所用的目录对所有用户可写。以下命令将创建目录，并设置合适的权限：

```
% hadoop fs -mkdir /tmp
% hadoop fs -chmod a+w /tmp
% hadoop fs -mkdir -p /user/hive/warehouse
% hadoop fs -chmod a+w /user/hive/warehouse
```

如果所有用户在同一个用户组中，把仓库目录的权限设为 `g+w` 就够了。

还可以在一个会话中使用 `SET` 命令更改设置。这对于为某个特定的查询修改 Hive 设置非常有用。例如，以下命令确保表的定义中都使用“桶”(bucket)，详情可以参见 17.6.2 节：

```
hive> SET hive.enforce.bucketing=true;
```

可以用只带属性名的 SET 命令查看任何属性的当前值：

```
hive> SET hive.enforce.bucketing;
hive.enforce.bucketing=true
```

不带参数的 SET 命令会列出 Hive 所设置的所有属性(及其取值)。注意，这个列表中不包含 Hadoop 的默认值，除非这个值用本节中介绍的某个方法重写了。使用 SET-v 可以列出系统中的所有属性，包括 Hadoop 的默认值。

设置属性有一个优先级层次。在下面的列表中，越小的值表示优先级越高。

(1) Hive SET 命令。

(2) 命令行-hiveconf 选项。

(3) *hive-site.xml* 和 Hadoop 站点文件(*core-site.xml*、*hdfs-site.xml*、*mapredsite.xml*、*yarn-site.xml*)。

(4) Hive 默认值和 Hadoop 默认文件(*core-default.xml*、*hdfs-default.xml*、*mapred-default.xml*、*yarn-default.xml*)。

本书 6.2.2 节介绍了可以设置哪些属性，可以从中了解更详细地描述了有关 Hadoop 属性的配置问题。

1. 执行引擎

Hive 的原始设计是以 MapReduce 作为执行引擎(目前仍然是默认的执行引擎)。目前，Hive 的执行引擎还包括 Apache Tez (*http://tez.apache.org/*)，另外 Hive 对 Spark(参见第 19 章)的支持也正在开发中。Tez 和 Spark 都是通用有向无环图(DAG)引擎，它们比 MapReduce 更加灵活，性能也更优越。例如，在使用 MapReduce 时，中间作业的输出会被“物化”(materialize)存储到 HDFS 上， Tez 和 Spark 则不同，它们可以根据 Hive 规划器的请求，把中间结果写到本地磁盘上，甚至在内存中缓存，以避免额外的复制开销。

具体使用哪种执行引擎由属性 hive.execution.engine 来控制，其默认值为 mr (即 MapReduce)。基于每查询的执行引擎的切换操作非常简单，因此我们可以观察不同引擎对特定查询的不同使用效果。下述语句设置 Tez 为 Hive 的执行引擎：

```
hive> SET hive.execution.engine=tez;
```

注意，首先必须在 Hadoop 集群上安装 Tez，Hive 说明文档中包含了详细的最新安装的最新步骤。

2. 日志记录

可以在本地文件系统的*${java.io.tmpdir}/${user.name}/hive.log* 中找到 Hive 的错误日志。错误日志对于诊断配置问题和其他错误非常有用。Hadoop 的 MapReduce 任务日志对于调试也非常有帮助，相关信息请参见 6.5.6 节对 Hadoop 用户日志的讨论。

在很多系统中，`${java.io.tmpdir}`就是*/tmp*，如果不是，或者你希望将日志目录指定到其他位置，可以使用下述命令：

```
% hive -hiveconf hive.log.dir='/tmp/${user.name}'
```

日志的配置存放在 *conf/hive-log4j.properties* 中。可以通过编辑这个文件来修改日志的级别和其他日志相关设置。但是，更方便的办法是在会话中对日志配置进行设置。例如，下面的语句可以方便地将调试消息发送到控制台：

```
% hive -hiveconf hive.root.logger=DEBUG,console
```

17.3.2 Hive 服务

Hive 的 shell 环境只是 `hive` 命令提供的其中一项服务。我们可以在运行时使用`--service` 选项指明要使用哪种服务。键入 `hive --service help` 可以获得可用服务列表。下面介绍一些最有用的服务。

- cli　Hive 的命令行接口(shell 环境)。这是默认的服务。
- hiveserver2　让 Hive 以提供 Thrift 服务的服务器形式运行，允许用不同语言编写的客户端进行访问。hiveserver2 在支持认证和多用户并发方面比原始的 hiveserver 有很大改进。使用 Thrift、JDBC 和 ODBC 连接器的客户端需要运行 Hive 服务器来和 Hive 进行通信。通过设置 `hive.server2.thrift.port` 配置属性来指明服务器所监听的端口号(默认为 10000)。
- beeline　以嵌入方式工作的 Hive 命令行接口(类似于常规的 CLI)，或者使用 JDBC 连接到一个 HiveServer 2 进程。
- hwi　Hive 的 Web 接口。在没有安装任何客户端软件的情况下，这个简单的 Web 接口可以代替 CLI。另外，Hue 是一个功能更全面的 Hadoop

Web 接口，其中包括运行 Hive 查询和浏览 Hive metastore 的应用程序。

- jar　与 `hadoop jar` 等价。这是运行类路径中同时包含 Hadoop 和 Hive 类 Java 应用程序的简便方法。
- metastore　默认情况下，metastore 和 Hive 服务运行在同一个进程里。使用这个服务，可以让 metastore 作为一个单独的(远程)进程运行。通过设置 `METASTORE_PORT` 环境变量(或者使用-p 命令行选项)可以指定服务器监听的端口号(默认为 9083)。

Hive 客户端

如果以服务器方式运行 Hive (`hive --service hiveserver2`)，可以在应用程序中以不同机制连接到服务器。Hive 客户端和服务之间的联系如图 17-1 所示。

- **Thrift 客户端**　Hive 服务器提供 Thrift 服务的运行，因此任何支持 Thrift 的编程语言都可与之交互。有些第三方项目还提供 Python 和 Ruby 客户端。详情可访问 Hive 的英文维基页面(*http://bit.ly/hive_server*)。
- **JDBC 驱动**　Hive 提供了 Type 4(纯 Java)的 JDBC 驱动，定义在 `org.apache.hadoop.hive.jdbc.HiveDriver` 类中。在以 `jdbc:hive2://host:port/dbname` 形式配置 JDBC URI 以后，Java 应用程序可以在指定的主机和端口连接到在另一个进程中运行的 Hive 服务器。驱动使用 Java 的 Thrift 绑定来调用由 Hive Thrift 客户端实现的接口。

你可能还希望通过 URI `jdbc:hive2://`，用 JDBC 内嵌模式来连接 Hive。在这个模式下，Hive 和发出调用的应用程序在同一个 JVM 中运行。这时不需要以独立服务器方式运行 Hive，这是因为此时应用程序并不使用 Thrift 服务或 Hive 的 Thrift 客户端。

Beeline CLI 使用 JDBC 驱动与 Hive 通信。

- **ODBC 驱动**　Hive 的 ODBC 驱动允许支持 ODBC 协议的应用程序(比如商业情报软件)连接到 Hive。Apache Hive 的发布版本中没有 ODBC 驱动，不过有些厂商会提供一个免费版的 ODBC 驱动。(和 JDBC 驱动类似，ODBC 驱动使用 Thrift 和 Hive 服务器进行通信。)

图 17-1. Hive 体系结构

17.3.3 Metastore

metastore 是 Hive 元数据的集中存放地。metastore 包括两部分：服务和后台数据的存储。默认情况下，metastore 服务和 Hive 服务运行在同一个 JVM 中，它包含一个内嵌的以本地磁盘作为存储的 Derby 数据库实例。这称为内嵌 metastore 配置(embedded metastore configuration)，参见图 17-2。

图 17-2. Metastore 的配置

使用内嵌 metastore 是 Hive 入门最简单的方法。但是，每次只有一个内嵌 Derby 数据库可以访问某个磁盘上的数据库文件，这就意味着一次只能为每个 metastore 打开一个 Hive 会话。如果要试着启动第二个会话，在它试图连接 metastore 时，会得到错误信息。

如果要支持多会话(以及多用户)，需要使用一个独立的数据库。这种配置称为**本地 metastore 配置**(local metastore)，因为 metastore 服务仍然和 Hive 服务运行在同一个进程中，但连接的却是在另一个进程中运行的数据库，在同一台机器上或在远程机器上。任何 JDBC 兼容的数据库都可以通过表 17-1 列出的 javax.jdo.option.* 配置属性来供 metastore 使用。[①]

表 17-1. 重要的 metastore 配置属性

属性名称	类型	默认值	描述
hive.metastore.warehouse.dir	URI	/user/hive/warehouse	相对于 fs.default.name 的目录，托管表就存储在这里
hive.metastore.uris	逗号分隔的URI	未设定	如果未设置(默认值)，则使用当前的 metastore，否则连接到由 URI 列表指定要连接的远程 metastore 服务器。如果有多个远程服务器，则客户端便以轮询(round robin)方式连接
javax.jdo.option.ConnectionURL	URI	jdbc:derby:;databaseName=metastored b;create=true	metastore 数据库的 JDBC URL
javax.jdo.option.ConnectionDriverName	字符串	org.apache.derby.jdbc.EmbeddedDriver	JDBC 驱动器的类名
javax.jdo.option.ConnectionUserName	字符串	APP	JDBC 用户名
javax.jdo.option.ConnectionPassword	字符串	mine	JDBC 密码

对于独立的 metastore，MySQL 是一种很受欢迎的选择。此时 javax.jdo.option.ConnectionURL 应该设为 jdbc:mysql://host/dbname? createDatabaseIfNotExist=true，而 javax.jdo.option.ConnectionDriverName 则设为 com.mysql.jdbc.Driver。当然，还需要设置用户名和密码。MySQL 的 JDBC 驱动的 JAR 文件(Connector/J)

① 这些属性以 javax.jdo 作为前缀，因为 metastore 的实现针对持久化 Java 对象使用了 Java Data Objects (JDO) API。它使用了 JDO 的 DataNucleus 实现。

必须在 Hive 的类路径中，把这个文件放入 Hive 的 *lib* 目录即可。

更进一步，还有一种 metastore 配置称为**远程 metastore 配置**(remote metastore)。在这种配置下，一个或多个 metastore 服务器和 Hive 服务运行在不同的进程内。这样一来，数据库层可以完全置于防火墙后，客户端则不需要数据库凭据(用户名和密码)，从而提供了更好的可管理性和安全。

可以通过把 `hive.metastore.uris` 设为 metastore 服务器 URI(如果有多个服务器，各个 URI 之间用逗号分隔)，把 Hive 服务设为使用远程 metastore。metastore 服务器 URI 的形式为 `thrift://host:port`。这里，端口号对应于启动 metastore 服务器时所设定的 `METASTORE_PORT` 值，详情可以参见 17.3.2 节对 Hive 服务的讨论。

17.4 Hive 与传统数据库相比

Hive 在很多方面和传统数据库类似(例如支持 SQL 接口)，但是其起初对 HDFS 和 MapReduce 底层的依赖意味着它的体系结构有别于传统数据库，而这些区别又影响着 Hive 所支持的特性。不过，随着时间的推移，这些局限性已逐渐消失，导致的结果就是 Hive 看上去和用起来都越来越像传统的数据库。

17.4.1 读时模式 vs.写时模式

在传统数据库里，表的模式是在数据加载时强制确定的。如果在加载时发现数据不符合模式，则拒绝加载数据。因为数据是在写入数据库时对照模式进行检查，因此这一设计有时被称为“**写时模式**”(schema on write)。

在另一方面，Hive 对数据的验证并不在加载数据时进行，而在查询时进行，这称为“**读时模式**”(schema on read)。

用户需要在这两种方法之间进行权衡。读时模式可以使数据加载非常迅速。这是因为它不需要读取数据来进行“解析”(parse)，再进行序列化并以数据库内部格式存入磁盘。数据加载操作仅仅是文件复制或移动。这一方法也更为灵活，试想，针对不同的分析任务，同一个数据可能会有两个模式。Hive 使用“外部表”(external table)时，这种情况是可能发生的，参见 17.6.1 节对托管表和外部表的讨论。

写时模式有利于提升查询性能。因为数据库可以对列进行索引，并对数据进行压缩。但是作为权衡，此时加载数据会花更多时间。此外，在很多加载时模式未知

的情况下，因为查询尚未确定，因此不能决定使用何种索引。这些情况正是 Hive “长袖善舞”的地方。

17.4.2 更新、事务和索引

更新、事务和索引都是传统数据库最重要的特性。但是，直到最近，Hive 也还没有考虑支持这些特性，因为 Hive 被设计为用 MapReduce 操作 HDFS 数据。在这样的环境下，“全表扫描”(full-table scan)是常态操作，而表更新则是通过把数据变换后放入新表实现的。对于在大规模数据集上运行的数据仓库应用，这一方式很见效。

早前的 Hive 能够利用 `INSERT INTO` 语句为表添加新的数据文件，以实现向现有表中批量增加新行。自 0.14.0 版开始，Hive 可以对表做一些更细粒度的更新。也就是说，可以使用 `INSERT INTO TABLE...VALUES` 插入少量通过 SQL 计算出来的值。另外，还可以对表中的行执行 `UPDATE` 和 `DELETE` 操作。

HDFS 不提供就地文件更新，因此，插入、更新和删除操作引起的一切变化都被保存在一个较小的增量文件中。由 metastore 在后台运行的 MapReduce 作业会定期将这些增量文件合并到“基表”(base table)文件中。上述功能只有在事务(由 Hive 的 0.13.0 版引入)的背景环境下才能发挥作用，因此这些正在使用的表必须启用了事务，以保证对这些表进行读操作的查询可以看到表的一致性快照。

Hive 的 0.7.0 发布版本还引入了表级(table-level)和分区级(partition-level)的锁。有了锁，就可以防止一个进程删除正在被另一个进程读取的表。锁由 ZooKeeper 透明管理，因此用户不必执行获得和释放锁的操作，但仍然可以通过 `SHOW LOCKS` 语句获取已经获得了哪些锁的信息。默认情况下，并未启用锁的功能。

在某些情况下，Hive 的索引能加快查询的速度。对于 `SELECT * from t WHERE x = a` 这样的查询，因为只需要扫描表文件的一小部分，因此可以利用在列 `x` 上的索引。目前 Hive 的索引分成两类：**紧凑**(compact)索引和**位图**(bitmap)索引。(索引的实现被设计为可插拔的，所以为了其他目的而设计的索引实现会陆续出现。)

紧凑索引存储每个值的 HDFS 块号，而不是存储文件内偏移量。因此存储不会占用过多的磁盘空间，且对于值被聚簇(clustered)存储于相近行的情况，索引仍然有效。位图索引使用压缩的位集合(bitset)来高效存储具有某个特殊值的行。这种索引一般适合于具有较少取值可能(low-cardinality)的列(如性别或国别)。

17.4.3 其他 SQL-on-Hadoop 技术

自从 Hive 诞生以来，针对 Hive 的局限性，这些年涌现出许多其他的 SQL-on-Hadoop 引擎技术，这方面的先锋之一是 Cloudera Impala (*http://impala.io/*)，它是开源交互式 SQL 引擎，Impala 在性能上要比基于 MapReduce 的 Hive 高一个数量级。Impala 使用专用的守护进程，这些守护进程运行在集群中的每个数据节点上。当客户端发起查询时，它首先会联系任意一个运行了 Impala 守护进程的节点，这个节点被当作是该查询的协调(coordinator)节点。协调节点向集群中的其他 Impala 守护进程分发工作，并收集结果以形成该查询的完整结果集。Impala 使用 Hive 的 metastore 并支持 Hive 格式和绝大多数的 HiveQL 结构(再加上 SQL-92)，因此，在实际操作中这两个系统可以直观地相互移植，或者运行在同一个集群上。

当然，Hive 也并非止步不前，随着 Cloudera 的 Impala 项目启动，Hortonworks 也发起了 Stinger 计划，它通过支持 Tez 作为执行引擎，再加上矢量化查询引擎等其他一些改进技术，使 Hive 在性能上得到很大的提升。

其他几个著名的开源 Hive 替代技术包括 Facebook Presto (*http://prestodb.io/*)、Apache Drill (*http://drill.apache.org/*)与 Spark SQL (*https://spark.apache.org/sql/*)。Presto 和 Drill 的架构类似于 Impala，只不过 Drill 的目标是 SQL:2011，而非 HiveQL。Spark SQL 使用 Spark 作为其底层引擎，并允许在 Spark 程序中嵌入 SQL 查询。

Spark SQL 与在 Hive 中使用 Spark 执行引擎(参见 17.3.1 节对执行引擎的介绍)并不是一回事。基于 Spark 的 Hive 能够提供 Hive 的所有功能，因为它本身隶属于 Hive 项目。而另一方面，Spark SQL 则是一种新兴的 SQL 引擎，它只在某种程度上提供与 Hive 的兼容。

Apache Phoenix (*http://phoenix.apache.org/*)则采取了另一种完全不同的方式，它提供的是基于 HBase 的 SQL。通过 JDBC 驱动实现 SQL 访问，JDBC 驱动将查询转换为 HBase 扫描，并利用 HBase 协同处理器来执行服务器端的聚合。另外，其元数据也存储在 HBase 中。

17.5 HiveQL

Hive 的 SQL“方言”称为 HiveQL，它是 SQL-92、MySQL 和 Oracle SQL 语言的混合体。它对 SQL-92 标准的支持已经变得越来越完善，并且有希望做得更好。HiveQL 也能支持一些由最新的 SQL 标准提供的功能，例如 SQL:2003 的窗口函数(也称为分析函数)。Hive 对 SQL-92 的有些扩展是受 MapReduce 启发而来的，如多表插入(详情可以参见 17.6.4 节)和 TRANSFORM，MAP 和 REDUCE 子句(详情可以参见 17.7.2 节)。

本章并不提供 HiveQL 的完整介绍，完整参考手册可参见 Hive 文档(*http://bit.ly/languagemanual*)。我们只聚焦于常用特性，并特别关注那些不同于 SQL-92 或像 MySQL 这样常用数据库的特性。表 17-2 给出了 SQL 和 HiveQL 特性的较高层次的比较。

表 17-2. SQL 和 HiveQL 的概要比较

特性	SQL	HiveQL	参考
更新	UPDATE，INSERT，DELETE	UPDATE，INSERT，DELETE	17.6.4 节和 17.4.2 节
事务	支持	有限支持	
索引	支持	支持	
延迟	亚秒级	分钟级	
数据类型	整数、浮点数、定点数、文本和二进制串、时间	布尔型、整数、浮点数、文本和二进制串、时间戳、数组、映射、结构	17.5.1 节
函数	数百个内置函数	数百个内置函数	17.5.2 节
多表插入	不支持	支持	17.6.4 节
CREATE TABLE AS SELECT	SQL-92 中不支持，但有些数据库支持	支持	17.6.4 节
SELECT	SQL-92	SQL-92。支持偏序的 SORT BY。可限制返回行数量的 LIMIT	17.7 节
连接	SQL-92 支持或变相支持(FROM 子句中列出连接表，在 WHERE 子句中列出连接条件)	内连接、外连接、半连接、映射连接、交叉连接	17.7.3 节

续表

特性	SQL	HiveQL	参考
子查询	在任何子句中支持“相关”的(correlated)或不相关的(noncorrelated)	只能在 FROM、WHERE 或 HAVING 子句中(不支持非相关子查询)	17.7.4 节
视图	可更新(是物化的或非物化的)	只读(不支持物化视图)	17.7.5 节
扩展点	用户定义函数	用户定义函数	17.8 节和 17.7.2 节
	存储过程	MapReduce 脚本	

17.5.1 数据类型

Hive 支持原子和复杂数据类型。原子数据类型包括数值型、布尔型、字符串类型和时间戳类型。复杂数据类型包括数组、映射和结构。Hive 的数据类型在表 17-3 中列出。注意，列出的是它们在 HiveQL 中使用的形式而不是它们在表中序列化存储的格式(参见 17.6.3 节)。

表 17-3. Hive 的数据类型

类别	类型	描述	文字示例
基本数据类型	BOOLEAN	true/false	TRUE
	TINYINT	1 字节(8 位)有符号整数，从-128 到 127	1Y
	SMALLINT	2 字节(16 位)有符号整数，从-32 768 到 32 767	1S
	INT	4 字节(32 位)有符号整数，从-2 147 483 648 到 2 147 483 647	1
	BIGINT	8 字节(64 位)有符号整数，从-9 223 372 036 854 775 808 到 9 223 372 036 854 775 807	1L
	FLOAT	4 字节(32 位)单精度浮点数	1.0
	DOUBLE	8 字节(64 位)双精度浮点数	1.0
	DECIMAL	任意精度有符号小数	1.0
	STRING	无上限可变长度字符串	'a', "a"
	VARCHAR	可变长度字符串	'a', "a"
	CHAR	固定长度字符串	'a', "a"
	BINARY	字节数组	不支持
	TIMESTAMP	精度到纳秒的时间戳	1325502245000, 2012-01-02 03:04: 05.123456789'
	DATE	日期	'2012-01-02'

续表

类别	类型	描述	文字示例
复杂数据类型	ARRAY	一组有序字段。字段的类型必须相同	array(1,2)[①]
	MAP	一组无序的键-值对。键的类型必须是原子的；值可以是任何类型。同一个映射的键的类型必须相同，值的类型也必须相同	map('a',1,'b',2)
	STRUCT	一组命名的字段。字段的类型可以不同	struct('a',1,1.0)，[②] named_struct('col1', 'a', 'col2', 1, 'col3', 1.0)
	UNION	值的数据类型可以是多个被定义的数据类型中的任意一个，这个值通过一个整数(零索引)来标记其为联合类型中的哪个数据类型	create_union(1, 'a', 63)

① 数组、映射和结构的文字形式可以通过函数得到：array()、map()、struct()三个函数都是 Hive 的内置函数。

② 列命名为 col1、col2、col3 等。

1. 原子类型

虽然有些 Hive 的原子数据类型的命名受到 MySQL 数据类型名称的影响(其中有些又和 SQL-92 相同)，但这些数据类型基本对应于 Java 中的类型。BOOLEAN 类型用于存储真值(true)和假值(false)。有四种有符号整数类型：TINYINT、SMALLINT、INT 以及 BIGINT，分别等价于 Java 的 byte、short、int 和 long 原子数据类型。它们分别为 1 字节、2 字节、4 字节和 8 字节有符号整数。

Hive 的浮点数据类型 FLOAT 和 DOUBLE 对应于 Java 的 float 和 double 类型，分别为 32 位和 64 位浮点数。

DECIMAL 类型用于表示任意精度的小数，类似于 Java 的 BigDecimal 类型，常常被用来表示货币值。DECIMAL 的值以整数非标度值的形式保存。**精度**(precision)指明非标度值的位数，而**标度**(scale)就是指小数点右侧的位数。因此 DECIMAL(5, 2)保存的是从-999.99 到 999.99 之间的数。如果标度省略，则其默认值为 0，因此 DECIMAL(5)保存的就是从-99,999 到 99,999 范围之间的整数。如果精度省略，则其默认值为 10，因此 DECIMAL 等价于 DECIMAL(10,0)。精度的最大允许值为 38，而标度值不能超过精度值。

在 Hive 中，存储文本的数据类型有三种。STRING 是一个无最大长度声明的变长字符串。(理论上其中最多能存储 2GB 的字符数，但如果真要物化存储那么大的值，效率肯定很低。Sqoop 提供了大对象的支持，详见 15.7 节对导入大对象的讨

论。) VARCHAR 与 STRING 相似，只不过它需要声明最大长度(允许长度范围在 1 到 65355 之间)，例如 VARCHAR(100)。CHAR 类型是固定长度的字符串，如有必要则以空格填充尾部，例如 CHAR(100)。当 CHAR 值被用于字符串比较操作时，忽略尾部空格。

BINARY 数据类型用于存储变长的二进制数据。

TIMESTAMP 数据类型存储精度为纳秒的时间戳。Hive 提供了在 Hive 时间戳、Unix 时间戳(从 UNIX 纪元开始的秒数)、字符串之间进行转换的 UDF，这样就使得常用的日期操作较为容易进行处理。但是 TIMESTAMP 中并未封装时区信息。可以使用 to_utc_timestamp 和 from_utc_timestamp 函数来进行时区转换。

DATE 类型保存的日期，包括年、月、日三个组成部分。

2. 复杂类型

Hive 有四种复杂数据类型：ARRAY、MAP、STRUCT 和 UNION。ARRAY 和 MAP 与 Java 中的同名数据类型类似，而 STRUCT 是一种记录类型，它封装了一个命名的字段集合。UNION 是从几种数据类型中指明选择一种，UNION 的值必须与这些数据类型之一完全匹配。

复杂数据类型允许任意层次的嵌套。复杂数据类型声明必须用尖括号符号指明其中数据字段的类型。如下所示的表定义有三列，每一列对应一种复杂数据类型：

```
CREATE TABLE complex (
c1 ARRAY<INT>,
c2 MAP<STRING, INT>,
c3 STRUCT<a:STRING, b:INT, c:DOUBLE>,
c4 UNIONTYPE<STRING, INT>
);
```

如果把表 17-3 中“文字示例”列中所示 ARRAY、MAP、STRCUT 和 UNION 类型的数据加载到表中(17.6.3 节将要介绍需要什么格式的文件)，则如下的查询展示了每种类型的字段访问操作：

```
hive> SELECT c1[0], c2['b'], c3.c, c4 FROM complex;
1 2 1.0 {1:63}
```

17.5.2 操作与函数

Hive 提供了普通 SQL 操作符，包括：关系操作符(例如等值判断：x='a'；空值判

断：x IS NULL；模式匹配：x LIKE 'a%')，算术操作符(例如加法：x + 1)，以及逻辑操作符(例如逻辑或：x OR y))。这些操作符和 MySQL 的操作符一样，而和 SQL-92 不同：||是逻辑或(OR)操作符，而不是字符串“连接”(concatenation)操作符。在 MySQL 和 Hive 中，字符串连接应该用 concat 函数。

Hive 提供的内置函数太多，以至于这里无法一一列举。这些函数分成几个大类，包括数学和统计函数、字符串函数、日期函数(用于操作表示日期的字符串)、条件函数、聚集函数以及处理 XML(使用 xpath 函数)和 JSON 的函数。

可以在 Hive 的 shell 环境中输入 SHOW FUNCTIONS 以获取函数列表。[1]要想了解某个特定函数的使用帮助，可以使用 DESCRIBE 命令：

```
hive> DESCRIBE FUNCTION length;
length(str | binary) - Returns the length of str or number of bytes in binary data
```

如果没有你需要的内置函数，那么可以自己动手写，详情参见 17.8 节。

类型转换

原子数据类型形成了一个 Hive 函数和操作符表达式进行隐式类型转换的层次。例如，如果某个表达式要使用 INT，那么 TINYINT 会被转换为 INT。但是，Hive 不会进行反向转换，它会返回错误，除非使用 CAST 操作。

隐式类型转换规则概述如下：任何数值类型都可以隐式地转换为一个范围更广的类型或者文本类型(STRING、VARCHAR、CHAR)。所有文本类型都可以隐式地转换为另一种文本类型。可能令人惊讶的是文本类型都能隐式转换为 DOUBLE 或 DECIMAL。BOOLEAN 类型不能转换为其他任何数据类型，同样也不能在表达式中隐式地转换为任何数据类型。TIMESTAMP 和 DATE 可以被隐式转换为文本类型。

你可以使用 CAST 操作显式进行数据类型转换。例如，CAST('1' AS INT)将把字符串'1'转换成整数值 1。如果强制类型转换失败，例如执行 CAST('X' AS INT)，表达式就会返回空值 NULL。

① 或查阅 Hive 函数参考手册(*http://bit.ly/languagemanual_udf*)。

17.6 表

Hive 的表在逻辑上由存储的数据和描述表中数据形式的相关元数据组成。数据一般存放在 HDFS 中，但它也可以放在其他任何 Hadoop 文件系统中，包括本地文件系统或 S3。Hive 把元数据存放在关系型数据库中，而不是放在 HDFS 中，详情参见 17.3.3 节对 metastore 的讨论。

在这一节中，我们将进一步了解如何创建表格、Hive 提供的不同物理存储格式以及如何导入这些不同格式的数据。

多数据库/模式支持

很多关系型数据库提供了多个“命名空间”(namespace)的支持。这样，用户和应用就可以被隔离到不同的数据库或模式中。Hive 也对此提供了同样的支持，可用的命令包括 CREATE DATABASE *dbname*、USE *dbname* 以及 DROP DATABASE *dbname* 这样的语句。可以通过 *dbname.tablename* 来完全限定某一张表。如果没有指明数据库，那么所指的是在 default 数据库中的表。

17.6.1 托管表和外部表

在 Hive 中创建表时，默认情况下 Hive 负责管理数据。这意味着 Hive 把数据移入它的“仓库目录”(warehouse directory)。另一种选择是创建一个**外部表**(external table)。这会让 Hive 到仓库目录以外的位置访问数据。

这两种表的区别表现在 LOAD 和 DROP 命令的语义上。我们先来看**托管表**(managed table)。

加载数据到托管表时，Hive 把数据移到仓库目录。例如：

```
CREATE TABLE managed_table (dummy STRING);
LOAD DATA INPATH '/user/tom/data.txt' INTO table managed_table;
```

把文件 hdfs://user/tom/data.txt 移动到 Hive 的 managed_table 表的仓库目录中，即 hdfs://user/hive/warehouse/managed_table。[1]

① 只有源和目标文件在同一个文件系统中移动才会成功。当然，作为特例，如果用了 LOCAL 关键字，Hive 会把本地文件系统的数据复制到 Hive 的仓库目录(即使它们在同一个文件系统中)。在其他所有情况下，最好把 LOAD 视为一个移动操作。

由于加载操作就是文件系统中的文件移动或文件重命名，因此它的执行速度很快。但记住，即使是托管表，Hive 也并不检查表目录中的文件是否符合为表所声明的模式。如果有数据和模式不匹配，只有在查询时才会知道。我们通常要通过查询为缺失字段返回的空值 NULL 才知道存在不匹配的行。可以发出一个简单的 SELECT 语句来查询表中的若干行数据，从而检查数据是否能够被正确解析。

如果随后要丢弃一个表，可使用以下语句：

```
DROP TABLE managed_table;
```

这个表，包括它的元数据和数据，会被一起删除。在此我们要重复强调，因为最初的 LOAD 是一个移动操作，而 DROP 是一个删除操作，所以数据会彻底消失。这就是 Hive 所谓的“托管数据”的含义。

对于外部表而言，这两个操作的结果就不一样了：由你来控制数据的创建和删除。外部数据的位置需要在创建表的时候指明：

```
CREATE EXTERNAL TABLE external_table (dummy STRING)
  LOCATION '/user/tom/external_table';
LOAD DATA INPATH '/user/tom/data.txt' INTO TABLE external_table;
```

使用 EXTERNAL 关键字以后，Hive 知道数据并不由自己管理，因此不会把数据移到自己的仓库目录。事实上，在定义时，它甚至不会检查这一外部位置是否存在。这是一个非常有用的特性，因为这意味着你可以把创建数据推迟到创建表之后才进行。

丢弃外部表时，Hive 不会碰数据，而只会删除元数据。

那么，应该如何选择使用哪种表呢？在多数情况下，这两种方式没有太大的区别(当然 DROP 语义除外)，因此这只是个人喜好问题。作为一个经验法则，如果所有处理都由 Hive 完成，应该使用托管表。但如果要用 Hive 和其他工具来处理同一个数据集，应该使用外部表。普遍的用法是把存放在 HDFS(由其他进程创建)的初始数据集用作外部表进行使用，然后用 Hive 的变换功能把数据移到托管的 Hive 表。这一方法反之也成立，外部表(未必在 HDFS 中)可以用于从 Hive 导出数据供其他应用程序使用。[①]

需要使用外部表的另一个原因是你想为同一个数据集关联不同的模式。

① 也可以用 INSERT OVERWRITE DIRECTORY 把数据导出到 Hadoop 文件系统中。

17.6.2 分区和桶

Hive 把表组织成**分区**(partition)。这是一种根据**分区列**(partition column，如日期)的值对表进行粗略划分的机制。使用分区可以加快数据分片(slice)的查询速度。

表或分区可以进一步分为**桶**(bucket)。它会为数据提供额外的结构以获得更高效的查询处理。例如，通过根据用户 ID 来划分桶，我们可以在所有用户集合的随机样本上快速计算基于用户的查询。

1. 分区

以分区的常用情况为例。考虑日志文件，其中每条记录包含一个时间戳。如果我们根据日期来对它进行分区，那么同一天的记录就会被存放在同一个分区中。这样做的优点是：对于限制到某个或某些特定日期的查询，它们的处理可以变得非常高效。因为它们只需要扫描查询范围内分区中的文件。注意，使用分区并不会影响大范围查询的执行，我们仍然可以查询跨多个分区的整个数据集。

一个表可以以多个维度来进行分区。例如，在根据日期对日志进行分区以外，我们可能还要进一步根据国家对每个分区进行**子分区**(subpartition)，以加速根据地理位置进行的查询。

分区是在创建表的时候用 PARTITIONED BY 子句定义的[1]。该子句需要定义列的列表。例如，对前面提到的假想的日志文件，我们可能要把表记录定义为由时间戳和日志行构成：

```
CREATE TABLE logs (ts BIGINT, line STRING)
PARTITIONED  BY (dt STRING, country STRING);
```

在我们把数据加载到分区表的时候，要显式指定分区值：

```
LOAD DATA LOCAL INPATH 'input/hive/partitions/file1'
INTO TABLE logs
PARTITION (dt='2001-01-01', country='GB');
```

在文件系统级别，分区只是表目录下嵌套的子目录。把更多文件加载到 logs 表以后，目录结构可能像下面这样：

① 在创建表后可以使用 ALTER TABLE 语句来增加或移除分区。

```
/user/hive/warehouse/logs
├──dt=2001-01-01/
│   ├──country=GB/
│   │   ├──file1
│   │   └──file2
│   └──country=US/
│       └──file3
└──dt=2001-01-02/
    ├──country=GB/
    │   └──file4
    └──country=US/
        ├──file5
        └──file6
```

日志表有两个日期分区：2001-01-01 和 2001-01-02，分别对应于子目录 *dt=2001-01-01* 和 *dt=2001-01-02*；和两个国家分区：GB 和 US，分别对应于嵌套子目录 country=GB 和 country=US。数据文件则存放在底层目录中。

可以用 SHOW PARTITIONS 命令让 Hive 告诉我们表中有哪些分区：

```
hive> SHOW PARTITIONS logs;
dt=2001-01-01/country=GB
dt=2001-01-01/country=US
dt=2001-01-02/country=GB
dt=2001-01-02/country=US
```

记住，PARTITIONED BY 子句中的列定义是表中正式的列，称为**分区列**(partition column)，但是，数据文件并不包含这些列的值，因为它们源于目录名。

可以在 SELECT 语句中以通常的方式使用分区列。Hive 会对输入进行修剪，从而只扫描相关的分区。例如：

```
SELECT ts, dt, line
FROM logs
WHERE country='GB';
```

将只扫描 file1、file2 和 file4。还要注意，这个查询返回 dt 分区列的值。这个值是 Hive 从目录名中读取的，因为它们在数据文件中并不存在。

2. 桶

把表(或分区)组织成桶(bucket)有两个理由。第一个理由是获得更高的查询处理效率。桶为表加上了额外的结构。Hive 在处理有些查询时能够利用这个结构。具体而言，连接两个在(包含连接列的)相同列上划分了桶的表，可以使用 map 端连接(map-side join)高效地实现。

把表划分成桶的第二个理由是使“取样”或者说“采样”(sampling)更高效。在处理大规模数据集时，在开发和修改查询的阶段，如果能在数据集的一小部分数据上试运行查询，会带来很多方便。我们在本节的最后将看看如何高效地进行取样。

首先，我们来看如何告诉 Hive 一个表应该被划分成桶。我们使用 CLUSTERED BY 子句来指定划分桶所用的列和要划分的桶的个数：

```
CREATE TABLE bucketed_users (id INT, name STRING)
CLUSTERED BY (id) INTO 4 BUCKETS;
```

在这里，我们使用用户 ID 来确定如何划分桶(Hive 对值进行哈希并将结果除以桶的个数取余数。这样，任何一桶里都会有一个随机的用户集合。

对于 map 端连接的情况，首先两个表以相同方式划分桶，处理左边表内某个桶的 mapper 知道右边表内相匹配的行在对应的桶内，这样，mapper 只需要获取那个桶(这只是右边表内存储数据的一小部分)即可进行连接。这一优化方法并不一定要求两个表必须具有相同的桶的个数，两个表的桶个数是倍数关系也可以。用 HiveQL 对两个划分了桶的表进行连接，更多细节可参见 17.7.3 节对 map 连接的讨论。

桶中的数据可以根据一个或多个列另外进行排序。由于这样对每个桶的连接变成了高效的归并排序(merge-sort)，因此可以进一步提升 map 端连接的效率。以下语法声明一个表使其使用排序桶：

```
CREATE TABLE bucketed_users (id INT, name STRING)
CLUSTERED BY (id) SORTED BY (id ASC) INTO 4 BUCKETS;
```

我们如何保证表中的数据都划分成桶了呢？把在 Hive 外生成的数据加载到划分成桶的表中，当然是可以的。其实让 Hive 来划分桶更容易。这一操作通常针对已有的表。

Hive 并不检查数据文件中的桶是否和表定义中的桶一致(无论是对于桶的数量或用于划分桶的列)。如果两者不匹配，在查询时可能会碰到错误或未定义的结果。因此，建议让 Hive 来进行划分桶的操作。

有一个没有划分桶的用户表：

```
hive>    SELECT * FROM users;
0        Nat
2        Joe
3        Kay
4        Ann
```

要向分桶后的表中填充成员，需要将 hive.enforce.bucketing 属性设置为 true。这样，Hive 就知道用表定义中声明的数量来创建桶，然后使用 INSERT 命令即可：

```
INSERT OVERWRITE TABLE bucketed_users
SELECT * FROM users;
```

物理上，每个桶就是表(或分区)目录里的一个文件。它的文件名并不重要，但是桶 *n* 是按照字典序排列的第 *n* 个文件。事实上，桶对应于 MapReduce 的输出文件分区：一个作业产生的桶(输出文件)和 reduce 任务个数相同。我们可以通过查看刚才创建的 bucketed_users 表的布局来了解这一情况。运行如下命令：

```
hive> dfs -ls /user/hive/warehouse/bucketed_users;
```

将显示有 4 个新建的文件。文件名如下(文件名由 Hive 产生)：

```
000000_0
000001_0
000002_0
000003_0
```

第一个桶里包括用户 ID 0 和 4，因为一个 INT 的哈希值就是这个整数本身，在这里即除以桶数(4)以后的余数：[①]

```
hive> dfs -cat /user/hive/warehouse/bucketed_users/000000_0;
0Nat
4Ann
```

用 TABLESAMPLE 子句对表进行取样，我们可以获得相同的结果。这个子句会将查询限定在表的一部分桶内，而不是使用整个表：

```
hive> SELECT * FROM bucketed_users
    > TABLESAMPLE(BUCKET 1 OUT OF 4 ON id);
4 Ann
0 Nat
```

桶的个数从 1 开始计数。因此，前面的查询从 4 个桶的第一个中获取所有的用户。对于一个大规模的、均匀分布的数据集，这会返回表中约 1/4 的数据行。我们也可以用其他比例对若干个桶进行取样(因为取样并不是一个精确的操作，因此这个比例不一定是桶数的整数倍)。例如，下面的查询返回一半的桶：

① 显示原始文件时，因为分隔字符是一个不能打印的控制字符，因此字段都挤在一起。它所使用的控制字符将在下一节介绍。

```
hive> SELECT * FROM bucketed_users
    > TABLESAMPLE(BUCKET 1 OUT OF 2 ON id);
4 Ann
0 Nat
2 Joe
```

因为查询只需要读取和 TABLESAMPLE 子句匹配的桶，所以取样分桶表是非常高效的操作。如果使用 rand()函数对没有划分成桶的表进行取样，即使只需要读取很小一部分样本，也要扫描整个输入数据集：

```
hive> SELECT * FROM users
    > TABLESAMPLE(BUCKET 1 OUT OF 4 ON rand());
2    Joe
```

17.6.3 存储格式

Hive 从两个维度对表的存储进行管理，分别是**行格式**(row format)和**文件格式**(file format)。行格式指行和一行中的字段如何存储。按照 Hive 的术语，行格式的定义由 SerDe 定义。SerDe 是“序列化和反序列化工具”(Serializer-Deserializer)的合成词。

当作为反序列化工具进行使用时，也就是查询表时，SerDe 将把文件中字节形式的数据行反序列化为 Hive 内部操作数据行时所使用的对象形式。使用序列化工具时，也就是执行 INSERT 或 CTAS(参见 17.6.4 节)时，表的 SerDe 会把 Hive 的数据行内部表示形式序列化成字节形式并写到输出文件中。

文件格式指一行中字段容器的格式。最简单的格式是纯文本文件，但是也可以使用面向行的和面向列的二进制格式。

1. 默认存储格式：分隔的文本

如果在创建表时没有用 ROW FORMAT 或 STORED AS 子句，那么 Hive 所使用的默认格式是分隔的文本，每行(line)存储一个数据行(row)。[①]

默认的行内分隔符不是制表符，而是 ASCII 控制码集合中的 Control-A(它的 ASCII 码为 1)。选择 Control-A(在文档中有时记作^A)作为分隔符是因为和制表符相比，它出现在字段文本中的可能性比较小。在 Hive 中无法对分隔符进行转义，因此，挑选一个不会在数据字段中用到的字符作为分隔符非常重要。

① 默认格式可以通过 hive.default.fileformat 属性设置。

集合类元素的默认分隔符为字符 Control-B。它用于分隔 ARRAY 或 STRUCT 或 MAP 的键-值对中的元素。默认的映射键(map key)分隔符为字符 Control-C。它用于分隔 MAP 的键和值。表中各行之间用换行符分隔。

前面对分隔符的描述对一般情况下的平面数据结构——即只包含原子数据类型的复杂数据类型——都是没有问题的。但是，对于嵌套数据类型，这还不够。事实上，嵌套的层次(level)决定了使用哪种分隔符。

例如，对于数组的数组，外层数组的分隔符如前所述是 Control-B 字符，但内层数组则使用分隔符列表中的下一项(Control-C 字符)作为分隔符。如果不确定 Hive 使用哪个字符作为某个嵌套结构的分隔符，可以运行以下命令：

```
CREATE TABLE nested
AS
SELECT array(array(1, 2), array(3, 4))
FROM dummy;
```

然后再使用 hexdump 或类似的命令来查看输出文件的分隔符。

实际上，Hive 支持 8 级分隔符，分别对应于 ASCII 编码的 1，2，……，8。但是你只能重载其中的前三个。

因此，以下语句：

```
CREATE TABLE ...;
```

等价于下面显式说明的语句：

```
CREATE TABLE ...
ROW  FORMAT   DELIMITED
  FIELDS TERMINATED BY '\001'
  COLLECTION ITEMS TERMINATED BY '\002'
  MAP KEYS TERMINATED BY '\003'
  LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

注意，我们可以使用八进制形式来表示分隔符，例如，001 表示 Control-A。

Hive 在内部使用一个名为 LazySimpleSerDe 的 SerDe 来处理这种分隔格式以及我们在第 8 章看到的面向行的 MapReduce 文本输入和输出格式。这里使用前缀“lazy”的原因是这个 SerDe 对字段的反序列化是延迟处理的，只有在访问字段时才进行反序列化。但是，由于文本以冗长的形式进行存放，所以这种存储格式并不紧凑。比如，一个布尔值事实上是以文本字符串 true 或 false 的形式存放的。

这种简单的格式有很多好处，例如，使用其他工具(包括 MapReduce 程序或 Streaming)来处理这样的格式非常容易。但是还可以选择一些更紧凑和高效的二进制 SerDe。这方面的问题稍后再讨论。

2. 二进制存储格式：顺序文件、Avro 数据文件、Parquet 文件、RCFile 与 ORCFile

二进制格式的使用方法非常简单，只需要通过 CREATE TABLE 语句中的 STORED AS 子句做相应声明。这里不需要指定 ROW FORMAT，因为其格式由底层的二进制文件格式来控制。

二进制格式可划分为两大类：面向行的格式和面向列的格式。一般来说，面向列的存储格式对于那些只访问表中一小部分列的查询比较有效。相反，面向行的存储格式适合同时处理一行中很多列的情况。

Hive 本身支持两种面向行的格式：Avro 数据文件(参见第 12 章)和顺序文件(参见 5.4.1 节)，它们都是通用的可分割、可压缩的格式。另外，Avro 还支持模式演化以及多种编程语言的绑定。在 Hive 0.14.0 之后的版本中，使用下述语句可以将表存储为 Avro 格式：

```
SET hive.exec.compress.output=true;
SET avro.output.codec=snappy;
CREATE TABLE ... STORED AS AVRO;
```

请注意，通过设置相应的属性即可支持表压缩。

类似地，Hive 利用 CREATE TABLE 语句中的 STORED AS SEQUENCEFILE 子句声明，将顺序文件作为存储格式。与压缩相关的属性可参考 5.2.3 节。

Hive 本身可支持的面向列的格式(参见 5.4.3 节)包括：Parquet(第 13 章)、RCFile 和 ORCFile。下面这个示例使用 CREATE TABLE...AS SELECT 语句(参见 17.6.4 节)来创建一个表的 Parquet 格式的复本。

```
CREATE TABLE users_parquet STORED AS PARQUET
AS
SELECT * FROM users;
```

3. 使用定制的 SerDe：RegexSerDe

现在让我们看看如何使用定制的 SerDe 来加载数据。我们将使用一个用户贡献的 SerDe，它采用一个正则表达式从一个文本文件中读取定长的观测站元数据：

```
CREATE TABLE stations (usaf STRING, wban STRING, name STRING)
```

```
ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
  "input.regex" = "(\\d{6}) (\\d{5}) (.{29}) .*"
);
```

在前面的示例中，我们在 ROW FORMAT 子句中使用 DELIMITED 关键字来说明文本是如何分隔的。在这个示例中，我们用另一种方式，用 SERDE 关键字和实现 SerDe 的 Java 类的完整类名(即 org.apache.hadoop.hive.contrib.serde2.RegexSerDe)，来指明使用哪个 SerDe。

SerDe 可以用 WITH SERDEPROPERTIES 子句来设置额外的属性。在这里，我们要设置 RegexSerDe 特有的 input.regex 属性。

input.regex 是在反序列化期间将要使用的正则表达式模式，用来将数据行(row)中的部分文本转化为列的集合。正则表达式匹配时使用 Java 的正则表达式语法(*http://bit.ly/java_regex*)。通过识别一组一组的括号来确定列(称为捕获组，即 capturing group)。[①]在这个示例中，有三个捕获组：usaf(六位数的标识符)、wban(五位数的标识符)以及 name(29 个字符的定长列)。

我们像以前一样，用如下 LOAD DATA 语句向表中输入数据：

```
LOAD DATA LOCAL INPATH "input/ncdc/metadata/stations-fixed-width.txt" INTO
TABLE stations;
```

回想一下，LOAD DATA 把文件复制或移动到 Hive 的仓库目录中。在这里，由于源在本地文件系统中，所以使用的是复制操作。加载操作并不使用表的 SerDe。

从文件中检索数据时，如下面这个简单查询所示，反序列化会调用 SerDe，从而使每一行的字段都能被正确解析出来：

```
hive> SELECT * FROM stations LIMIT 4;
10000       99999    BOGUS NORWAY
010003      99999    BOGUS NORWAY
010010      99999    JAN  MAYEN
010013      99999    ROST
```

从上面的示例中可以看出 RegexSerDe 能够帮助 Hive 获取数据，但是它的效率很

① 有时，可能要在正则表达式中使用括号，而不希望这些括号被当作捕获所用的符号。例如，模式(ab)+可用于匹配一个或多个连续的 ab 字符串。这时的解决办法是在左括号后加问号?表示“非捕获组”(noncapturing group)。有多种表示非捕获组的构造结构(参见 Java 文档)，但在这个示例中，我们可以用(?ab)+来避免某个文本序列被捕获。

低，因此一般不用于通用存储格式，应当考虑把数据复制为二进制存储格式。

4. 存储句柄

存储句柄(Storage handler)用于 Hive 自身无法访问的存储系统，比如 HBase。存储句柄使用 STORED BY 子句指定。它代替了 ROW FORMAT 和 STORED AS 子句。有关 HBase 集成的详细信息可以参考 Hive 的英文维基页面(*http://bit.ly/hbase_int*)。

17.6.4 导入数据

我们已经见过如何使用 LOAD DATA 操作，通过把文件复制或移到表的目录中，从而把数据导入 Hive 的表(或分区)。也可以用 INSERT 语句把数据从一个 Hive 表填充到另一个，或在新建表的时候使用 *CTAS* 结构，*CTAS* 是 CREATE TABLE…AS SELECT 的缩写。

如果想把数据从一个关系型数据库直接导入 Hive，可以看一下 Sqoop。详情参见 15.6.1 节。

1. INSERT 语句

下面是 INSERT 语句的一个示例：

```
INSERT OVERWRITE TABLE target
SELECT col1, col2
  FROM source;
```

对于分区的表，可以使用 PARTITION 子句来指明数据要插入哪个分区：

```
INSERT OVERWRITE TABLE target
PARTITION (dt='2001-01-01')
SELECT col1, col2
  FROM source;
```

OVERWRITE 关键字意味着目标表(对于前面的第一个例子)或 2001-01-01 分区(对于第二个例子)中的内容会被 SELECT 语句的结果替换掉。如果要向已经填充了内容的非分区表或分区添加记录，那么可以使用 INSERT INTO TABLE。

可以在 SELECT 语句中通过使用分区值来动态指明分区：

```
INSERT OVERWRITE TABLE target
PARTITION (dt)
SELECT col1, col2, dt
  FROM source;
```

这种方法称为**动态分区插入**(dynamic-partition insert)。

Hive 从 0.14.0 版开始允许使用 INSERT INTO TABLE...VALUES 语句来插入一小撮以文字形式指明的记录。

2. 多表插入

在 HiveQL 中，可以把 INSERT 语句倒过来，把 FROM 子句放在最前面，查询的效果是一样的：

```
FROM source
INSERT OVERWRITE  TABLE target
  SELECT col1, col2;
```

可以在同一个查询中使用多个 INSERT 子句。此时，这样的语法会让查询的含义更清楚。这种**多表插入**(multitable insert)方法比使用多个单独的 INSERT 语句效率更高，因为只需要扫描一遍源表就可以生成多个不相交的输出。

下面的例子根据气象数据集来计算多种不同的统计数据：

```
FROM records2
INSERT OVERWRITE TABLE stations_by_year
SELECT year, COUNT(DISTINCT station)
GROUP BY year
INSERT OVERWRITE TABLE records_by_year
SELECT year, COUNT(1)
GROUP BY year
INSERT OVERWRITE TABLE good_records_by_year
SELECT year, COUNT(1)
WHERE temperature != 9999 AND quality IN (0, 1, 4, 5, 9)
GROUP BY year;
```

这里只有一个源表(records2)，但有三个表用于存放针对这个源表的三个不同查询所产生的结果。

3. CREATE TABLE...AS SELECT 语句

把 Hive 查询的输出结果存放到一个新的表往往非常方便，这可能是因为输出结果太多，不适宜于显示在控制台上或基于输出结果还有其他后续处理。

新表的列的定义是从 SELECT 子句所检索的列导出的。在下面的查询中，target 表有两列，分别名为 col1 和 col2，它们的数据类型和源表中对应的列相同：

```
CREATE TABLE target
AS
SELECT col1, col2
FROM source;
```

CTAS 操作是原子的，因此，如果 SELECT 查询由于某种原因失败，是不会创建新表的。

17.6.5 表的修改

由于 Hive 使用读时模式(schema on read)，所以在创建表以后，它可以非常灵活地支持对表定义的修改。但一般需要警惕，在很多情况下，要由你来确保修改数据以符合新的结构。

可以使用 ALTER TABLE 语句来重命名表：

```
ALTER TABLE source RENAME TO target;
```

在更新表的元数据以外，ALTER TABLE 语句还把表目录移到新名称所对应的目录下。在这个示例中，*/user/hive/warehouse/source* 被重命名为 */user/hive/warehouse/target*。对于外部表，这个操作只更新元数据，而不会移动目录。

Hive 允许修改列的定义，添加新的列，甚至用一组新的列替换表内已有的列。

例如，考虑添加一个新列：

```
ALTER TABLE target ADD COLUMNS (col3 STRING);
```

新的列 col3 添加在已有(非分区)列的后面。数据文件并没有被更新，因此原来的查询会为 col3 的所有值返回空值 null(当然，除非文件中原来就已经有额外的字段)。因为 Hive 并不允许更新已有的记录，所以需要使用其他机制来更新底层的文件。为此，更常用的做法是创建一个定义了新列的新表，然后使用 SELECT 语句把数据填充进去。

如果我们设想原来的数据类型可以用新的数据类型来进行解释，那么修改一个表的元数据(如列名或数据类型)就变得更为直观。

要想更进一步了解如何修改表的结构，包括添加或丢弃分区、修改和替换列，修改表和 SerDe 的属性，可访问 Hive 的英文维基页面(*http://bit.ly/data_def_lang*)。

17.6.6 表的丢弃

DROP TABLE 语句用于删除表的数据和元数据。如果是外部表，就只删除元数据，数据不会受到影响。

如果要删除表内的所有数据，但要保留表的定义，可使用 TRUNCATE TABLE 语句。例如：

```
TRUNCATE TABLE my_table;
```

上述语句对外部表不起作用，这种情况需要在 Hive 的 shell 环境中使用 dfs -rmr 命令来直接删除外部表目录。

另一种达到类似目的的方法是使用 LIKE 关键字创建一个与第一个表模式相同的新表：

```
CREATE TABLE new_table LIKE existing_table;
```

17.7 查询数据

这一节讨论如何使用 SELECT 语句的各种形式从 Hive 中检索数据。

17.7.1 排序和聚集

在 Hive 中可以使用标准的 ORDER BY 子句对数据进行排序。ORDER BY 将对输入执行并行全排序(类似 8.2.3 节中描述的全排序)。在很多情况下，并不需要结果是全局排序的。此时，可以换用 Hive 的非标准的扩展 SORT BY。SORT BY 为每个 reducer 产生一个排序文件。

在有些情况下，你需要控制某个特定行应该到哪个 reducer，其目的通常是为了进行后续的聚集操作。这就是 Hive 的 DISTRIBUTE BY 子句所做的事情。下面的例子根据年份和气温对气象数据集进行排序，以确保所有具有相同年份的行最终都在同一个 reducer 分区中：[①]

```
hive> FROM records2
    > SELECT year, temperature
    > DISTRIBUTE BY year
```

① 这是在 Hive 中对 9.2.4 节所讨论内容的另一种实现。

```
    > SORT BY year ASC, temperature DESC;
1949      111
1949      78
1950      22
1950      0
1950      -11
```

后续的查询(或把这个查询作为内嵌子查询的某个查询，详情参见 17.7.4 节)可以利用在同一文件中已经分好组并(以降序)排好序的年份气温。

如果 SORT BY 和 DISTRIBUTE BY 中所用的列相同，可以缩写为 CLUSTER BY 以便同时指定两者所用的列。

17.7.2 MapReduce 脚本

和 Hadoop Streaming 类似，TRANSFORM、MAP 和 REDUCE 子句可以在 Hive 中调用外部脚本或程序。假设我们像范例 17-1 那样，用一个脚本来过滤不符合某个条件的行，即删除低质量的气温读数。

范例 17-1. 过滤低质量气象记录的 Python 脚本

```
#!/usr/bin/env python

import re
import sys

for line in sys.stdin:
  (year, temp, q) = line.strip().split()
  if (temp != "9999" and re.match("[01459]", q)):
    print "%s\t%s" % (year, temp)
```

我们可以这样使用这个脚本:

```
hive> ADD FILE /Users/tom/book-workspace/hadoop-book/ch17-hive/
src/main/python/is_good_quality.py;
hive> FROM records2
    > SELECT TRANSFORM(year, temperature, quality)
    > USING 'is_good_quality.py'
    > AS year, temperature;
1950 0
1950 22
1950 -11
1949 111
1949 78
```

在运行查询之前，我们需要在 Hive 中注册脚本。通过这一操作，Hive 知道需要把脚本文件传输到 Hadoop 集群上(参见 9.4.2 节对分布式缓存的讨论)。

查询本身把 year，temperature 和 quality 这些字段以制表符分隔的行的形式流式传递给脚本 *is_good_quality.py*，并把制表符分隔的输出解析为 year 和 temperature 字段，最终形成查询的输出。

这一示例并不使用 reducer。如果要用查询的嵌套形式，我们可以指定 map 和 reduce 函数。这一次我们用 MAP 和 REDUCE 关键字。但在这两个地方用 SELECT TRANSFORM 也能达到同样的效果。max_temperature_reduce.py 脚本的内容参见范例 2-10：

```
FROM (
  FROM records2
  MAP year, temperature, quality
  USING 'is_good_quality.py'
  AS year, temperature) map_output
REDUCE year, temperature
USING 'max_temperature_reduce.py'
AS year, temperature;
```

17.7.3 连接

和直接使用 MapReduce 相比，使用 Hive 的一个好处在于 Hive 简化了常用操作。对比在 MapReduce 中实现连接(join)要做的事情(参见 9.3 节)，在 Hive 中进行连接操作就能充分体现这个好处。

1. 内连接

内连接是最简单的一种连接。输入表之间的每次匹配都会在输出表里生成一行。让我们来考虑两个演示用的小表：sales 列出了人名及其所购商品的 ID，things 列出商品的 ID 和名称：

```
hive> SELECT * FROM sales;
Joe       2
Hank      4
Ali       0
Eve       3
Hank      2
hive> SELECT * FROM things;
2    Tie
4    Coat
3    Hat
1    Scarf
```

我们可以像下面这样对两个表进行内连接：

```
hive> SELECT sales.*, things.*
    > FROM sales JOIN things ON (sales.id = things.id);
Joe     2       2       Tie
Hank    4       4       Coat
Eve     3       3       Hat
Hank    2       2       Tie
```

FROM 子句中的表 sales 和 JOIN 子句中的表 things 用 ON 子句中的谓词进行连接。Hive 只支持等值连接(equijoin)，这意味着在连接谓词中只能使用等号。在这个示例中，等值条件是两个表的 id 列必须相同。

在 Hive 中，可以在连接谓词中使用 AND 关键字分隔的一系列表达式来连接多个列。还可以在查询中使用多个 JOIN…ON…子句来连接多个表。Hive 会智能地以最少 MapReduce 作业数来执行连接。

Hive 与 MySQL 和 Oracle 一样，允许在 SELECT 语句的 FROM 子句中列出要连接的表，而在 WHERE 子句中指定连接条件。例如，下面的语句是我们刚才举例的查询的另一种表示方法：：

```
SELECT sales.*, things.*
FROM sales, things
WHERE sales.id = things.id;
```

单个的连接用一个 MapReduce 作业实现。但是，如果多个连接的连接条件中使用了相同的列，那么平均每个连接可以用少于一个 MapReduce 作业来实现。[①]你可以在查询前使用 EXPLAIN 关键字来查看 Hive 将为某个查询使用多少个 MapReduce 作业：

```
EXPLAIN
SELECT sales.*, things.*
FROM sales JOIN things ON (sales.id = things.id);
```

EXPLAIN 的输出中有很多查询执行计划的详细信息，包括抽象语法树、Hive 执行各阶段之间的依赖图以及每个阶段的信息。一个阶段可能是像 MapReduce 作业文件移动这样的一个操作。如果要查看更详细的信息，可以在查询前使用 EXPLAIN EXTENDED。

Hive 目前使用基于规则的查询优化器来确定查询是如何执行的。但自 Hive 0.14.0 开始，Hive 也可以使用基于代价的优化器。

① JOIN 子句中表的顺序很重要：一般最好将最大的表放在最后。详细的信息，包括如何为 Hive 的查询规划器给出提示，可访问 Hive 的英文维基页面。

2. 外连接

外连接可以让你找到连接表中不能匹配的数据行。在前面的示例里，我们在进行内连接时，Ali 那一行没有出现在输出中。因为她所购商品的 ID 没有在 things 表中出现。如果我们把连接的类型改为 LEFT OUTER JOIN，即使左侧表(sales)中的有些行无法与所要连接的表(things)中的任何数据行对应，查询还是会返回这个表中的每一个数据行：

```
hive> SELECT sales.*, things.*
    > FROM sales LEFT OUTER JOIN things ON (sales.id = things.id);
Joe 2 2 Tie
Hank 4 4 Coat
Ali 0 NULL NULL
Eve 3 3 Hat
Hank 2 2 Tie
```

注意，此时返回了 Ali 所在的数据行，但因为这一行无匹配，所以 things 表的对应列为空值 NULL。

Hive 也支持右外连接(right outer join)，即和左连接相比，交换两个表的角色。在这里，things 表中的所有商品，即使没有任何人购买它们(scarf)，也都会被返回：

```
hive> SELECT sales.*, things.*
    > FROM sales RIGHT OUTER JOIN things ON (sales.id = things.id);
Joe 2 2 Tie
Hank 2 2 Tie
Hank 4 4 Coat
Eve 3 3 Hat
NULL NULL 1 Scarf
```

最后，还有一种全外连接(full outer join)，即两个连接表中的所有行在输出中都有对应的行：

```
hive> SELECT sales.*, things.*
    > FROM sales FULL OUTER JOIN things ON (sales.id = things.id);

Ali     0       NULL    NULL
NULL    NULL    1       Scarf
Joe     2       2       Tie
Hank    2       2       Tie
Eve     3       3       Hat
Hank    4       4       Coat
```

3. 半连接

我们来考虑如下 IN 子查询，它能够查找 things 表中在 sales 表中出现过的所有

商品：

```
SELECT *
FROM things
WHERE things.id IN (SELECT id from sales);
```

我们可以像下面这样重写这个查询：

```
hive> SELECT *
    > FROM things LEFT SEMI JOIN sales ON (sales.id = things.id);
2    Tie
4    Coat
3    Hat
```

写 LEFT SEMI JOIN 查询时必须遵循一个限制：右表(sales)只能在 ON 子句中出现。例如，我们不能在 SELECT 表达式中引用右表。

4. map 连接

回顾最初的内连接示例：

```
SELECT sales.*, things.*
FROM sales JOIN things ON (sales.id = things.id);
```

如果有一个连接表小到足以放入内存，例如示例中的 things，Hive 就可以把较小的表放入每个 mapper 的内存来执行连接操作，这就称为 map 连接。

执行这个查询不使用 reducer，因此这个查询对 RIGHT 或 FULL OUTER JOIN 无效，因为只有在对所有输入上进行聚集的步骤(即 reduce)才能检测到哪个数据行无法匹配。

map 连接可以利用分桶的表(参见 17.6.2 节)，因为作用于左侧表的桶的 mapper 加载右侧表中对应的桶即可执行连接。这时使用的语法和前面提到的在内存中进行连接是一样的，只不过还需要用下面的语法启用优化选项：

```
SET hive.optimize.bucketmapjoin=true;
```

17.7.4 子查询

子查询是内嵌在另一个 SQL 语句中的 SELECT 语句。Hive 对子查询的支持很有限，它只允许子查询出现在 SELECT 语句的 FROM 子句中，或某些特殊情况下的 WHERE 子句中。

Hive 允许不相关子查询，即子查询是在 WHERE 子句中通过 IN 或 EXISTS 引用的自包含的查询。而相关子查询(即由外部查询引用的子查询)目前还不支持。

下面的查询可以找到每年每个气象站最高气温的均值：

```
SELECT station, year, AVG(max_temperature)
FROM (
SELECT station, year, MAX(temperature) AS max_temperature
FROM records2
WHERE temperature != 9999 AND quality IN (0, 1, 4, 5, 9)
GROUP BY station, year
) mt
GROUP BY station, year;
```

这里 FROM 中的子查询用于计算每个气象站/日期组合中的最高气温，然后外层查询使用 AVG 聚集函数计算这些最高读数的均值。

外层查询像访问表那样访问子查询的结果，所以我们必须为子查询赋予一个别名(mt)的原因。子查询中的列必须有唯一的名称，以便外层查询可以引用这些列。

17.7.5 视图

视图是一种用 SELECT 语句定义的**虚表**(virtual table)。视图可以用来以一种不同于磁盘实际存储的形式把数据呈现给用户。现有表中的数据常常需要以一种特殊的方式进行简化和聚集以便于后期处理。视图也可以用来限制用户，使其只能访问被授权可以看到的表的子集。

在 Hive 中，创建视图时并不把视图物化存储到磁盘上。相反，视图的 SELECT 语句只是在执行引用视图的语句时才执行。如果一个视图要对基表进行大规模的变换，或视图的查询会频繁执行，你可以选择新建一个表，并把视图的内容存储到新表中，以此来手工物化它，可以参见 17.6.4 节对 CREATE TABLE…AS SELECT 语句的讨论。

我们可以用视图重写前一节中的查询，它用于查找每年各个气象站气温最大值的均值。首先，让我们为有效记录(即有特定 quality 值的记录)创建一个视图：

```
CREATE VIEW valid_records
AS
SELECT *
FROM records2
WHERE temperature != 9999 AND quality IN (0, 1, 4, 5, 9);
```

创建视图时并不执行查询，查询只是存储在 metastore 中。SHOW TABLES 命令的输出结果里包括视图。可以使用 DESCRIBE EXTENDED *view_name* 命令来查看某个视图的详细信息，包括用于定义它的那个查询。

接下来，让我们为每个观测站每年的最高气温创建第二个视图。这个视图基于 valid_record 视图：

```
CREATE VIEW max_temperatures (station, year, max_temperature)
AS
SELECT station, year, MAX(temperature) FROM valid_records
GROUP BY station, year;
```

在这个视图定义中，我们显式地列出了列的名称。我们这么做是因为最高气温列是一个聚集表达式，如果不指明，Hive 会自己创建一个别名(例如_c2)。我们也可以在 SELECT 语句中使用 AS 子句来为列命名。

有了这两个视图，现在我们就可以执行查询了：

```
SELECT station, year, AVG(max_temperature)
FROM max_temperatures
GROUP BY station, year;
```

这个查询的结果和前面使用子查询的查询是一样的。特别地，Hive 为它们所创建的 MapReduce 作业的个数也是一样的：都是两个，每个 GROUP BY 使用一个。从这个例子可以看到，Hive 可以把使用视图的查询组织成一系列作业，效果与不使用视图的查询一样。换句话说，即使在执行时，Hive 也不会在不必要的情况下物化视图。

Hive 中的视图是只读的，所以无法通过视图为基表加载或插入数据。

17.8 用户定义函数

你要写的查询有时无法轻松(或根本不能)使用 Hive 提供的内置函数来表示。通过编写**用户定义函数**(user-defined function，UDF)，Hive 可以方便地插入用户写的处理代码并在查询中调用它们。

UDF 必须用 Java 语言编写。Hive 本身也是用 Java 写的。对于其他编程语言，可以考虑使用 SELECT TRANSFORM 查询，有了它，可以让数据流经用户定义的脚本(参见 17.7.2 节)。

Hive 中有三种 UDF：(普通)UDF、用户定义聚集函数(user-defined aggregate function，UDAF)以及用户定义表生成函数(user-defined table-generating function，UDTF)。它们所接受的输入和产生的输出的数据行的数量不同。

- UDF 操作作用于单个数据行，且产生一个数据行作为输出。大多数函数(例如数学函数和字符串函数)都属于这一类。
- UDAF 接受多个输入数据行，并产生一个输出数据行。像 COUNT 和 MAX 这样的函数都是聚集函数。
- UDTF 操作作用于单个数据行，且产生多个数据行(即一个表)作为输出。

和其他两种类型相比，表生成函数的知名度较低。所以让我们来看一个示例。考虑这样一个表，它只有一列 x，包含的是字符串数组。回头看看表的定义和填充方式是很有启发的：

```
CREATE TABLE arrays (x ARRAY<STRING>) ROW
FORMAT DELIMITED
  FIELDS TERMINATED BY '\001'
  COLLECTION ITEMS TERMINATED BY '\002';
```

注意，ROW FORMAT 子句指定数组中的项用 Control-B 字符分隔。我们要加载的示例文件内容如下，为了显示方便，用^B 表示 Control-B 字符：

```
a^Bb
c^Bd^Be
```

在运行 LOAD DATA 命令以后，可以通过下面的查询确认数据已正确加载：

```
hive> SELECT * FROM arrays;
["a","b"]
["c","d","e"]
```

接下来，我们可以使用 explode UDTF 对表进行变换。这个函数为数组中的每一项输出一行。因此，在这里，输出的列 y 的数据类型为 STRING。其结果是，表被“平面化”(flattened)成五行：

```
hive> SELECT explode(x) AS y FROM arrays;
a
b
c
d
e
```

带 UDTF 的 SELECT 语句在使用时有一些限制(例如，它们不能检索额外的列表达

式)，使实际使用时这些语句的用处并不大。为此，Hive 支持 LATERAL VIEW 查询。这一语句的功能更强大。这里不介绍 LATERAL VIEW 查询。相关详情可以访问 Hive 的英文维基页面(*http://bit.ly/lateral_view*)。

17.8.1 写 UDF

为了演示如何写和使用 UDF，我们将写一个简单的剪除字符串尾字符的 UDF。Hive 已经有一个内置的名为 trim 的函数，所以我们把自己的函数称为 strip。Strip Java 类的代码如范例 17-2 所示。

范例 17-2. 剪除字符串尾字符的 UDF

```
package com.hadoopbook.hive;

import org.apache.commons.lang.StringUtils;
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;

public class Strip extends UDF {
  private Text result = new Text();

  public Text evaluate(Text str) {
    if (str == null) {
      return null;
    }

    result.set(StringUtils.strip(str.toString()));
    return result;
  }
    public Text evaluate(Text str, String stripChars) {
    if (str == null) {
      return null;
    }
    result.set(StringUtils.strip(str.toString(), stripChars));
    return result;
  }
}
```

一个 UDF 必须满足下面两个条件：

- 一个 UDF 必须是 org.apache.hadoop.hive.ql.exec.UDF 的子类
- 一个 UDF 必须至少实现了 evaluate()方法

evaluate()方法不是由接口定义的，因为它可接受的参数的个数、它们的数据类型及其返回值的数据类型都是不确定的。Hive 会检查 UDF，看能否找到和函数调用相匹配的 evaluate()方法。

这个 Strip 类有两个 evaluate()方法。第一个方法去除输入的前导和结束的空白字符，而第二个方法则去除字符串尾出现在指定字符集合中的任何字符。实际的字符处理工作交由 Apache Commons 项目里的 StringUtils 类来完成。所以代码中唯一值得提的是对 Hadoop Writable 库中 Text 的使用。实际上，Hive 支持在 UDF 中使用 Java 的基本类型(以及其他一些像 java.util.List 和 java.util.Map 这样的类型)，所以，下面这样函数签名(signature)的效果是一样的：

```
public String evaluate(String str)
```

但是，通过使用 Text，我们可以利用对象重用的优势，增效节支。因此一般推荐使用这种方法。

为了在 Hive 中使用 UDF，我们需要把编译后的 Java 类打包成一个 JAR 文件(可以用本书所附代码输入 mvn package 来完成)，接下来，在 metastore 中注册这个函数并使用 CREATE FUNCTION 语句为它起名：

```
CREATE FUNCTION strip AS 'com.hadoopbook.hive.Strip'
USING JAR '/path/to/hive-examples.jar';
```

本地使用 Hive 只需要一个本地文件路径就足够了，但是在集群上，则应当将 JAR 文件复制到 HDFS 中，并在 USING JAR 子句中使用 HDFS 的 URI。

现在，可以像使用内置函数一样使用 UDF：

```
hive> SELECT strip(' bee ') FROM dummy;
bee
hive> SELECT strip('banana', 'ab') FROM dummy;
nan
```

注意，UDF 名对大小写不敏感：

```
hive> SELECT STRIP(' bee ') FROM dummy;
bee
```

如果想要删除这个函数，可使用 DROP FUNCTION 语句：

```
DROP FUNCTION strip;
```

利用 TEMPORARY 关键字可以创建一个仅在 Hive 会话期间有效的函数，也就是说这个函数并没有在 metastore 中持久化存储。

```
ADD JAR /path/to/hive-examples.jar;
CREATE TEMPORARY FUNCTION strip AS 'com.hadoopbook.hive.Strip';
```

使用临时函数时，最好在主目录中创建一个*.hiverc* 文件，以包含定义这些 UDF 的

命令，而这个文件会在每个 Hive 会话开始时自动运行。

要想在开始时调用 ADD JAR，还可以在 Hive 启动时指定查找附加 JAR 文件的路径，这个路径会被加入 Hive 的类路径(也包括任务的类路径)。这种技术对于每次运行 Hive 时自动添加你的 UDF 库是很有用的。

有两种指明路径的办法：在 *hive* 命令后传递 `--auxpath` 选项：

```
% hive --auxpath /path/to/hive-examples.jar
```

或在运行 Hive 前设置 HIVE_AUX_JARS_PATH 环境变量。附加路径可以是一个用逗号分隔的 JAR 文件路径列表或包含 JAR 文件的目录。

17.8.2 写 UDAF

聚集函数比普通的 UDF 难写。因为值是在块内进行聚集的(这些块可能分布在很多任务中)，从而实现时要能够把部分的聚集值组合成最终结果。实现此功能的代码最好用示例来进行解释。让我们来看一个简单的 UDAF 的实现，它用于计算一组整数的最大值(范例 17-3)。

范例 17-3. 计算一组整数中最大值的 UDAF

```
package com.hadoopbook.hive;

import org.apache.hadoop.hive.ql.exec.UDAF;
import org.apache.hadoop.hive.ql.exec.UDAFEvaluator;
import org.apache.hadoop.io.IntWritable;

public class Maximum extends UDAF {

  public static class MaximumIntUDAFEvaluator implements UDAFEvaluator {

    private IntWritable result;

    public void init() {
      result =  null;
    }

  public boolean iterate(IntWritable value) {

    if (value == null) {
      return true;
    }
    if (result == null) {
      result = new IntWritable(value.get());
    } else {
      result.set(Math.max(result.get(), value.get()));
    }
      return true;
```

```
    }

    public IntWritable terminatePartial()  {
      return result;
    }

    public boolean merge(IntWritable other) {
      return iterate(other);
    }

    public IntWritable terminate {
      return result;
    }
  }
}
```

这个类的结构和 UDF 的稍有不同。UDAF 必须是 org.apache.hadoop.hive.ql.exec.UDAF(注意 UDAF 中的“A”)的子类，且包含一个或多个嵌套的、实现了 org.apache.hadoop.hive.ql.UDAFEvaluator 的静态类。在这个示例中，只有一个嵌套类，MaximumIntUDAFEvaluator。但是我们也可以添加更多的计算函数(如 MaximumLongUDAFEvaluator 和 MaximumFloatUDAFEvaluator)来提供计算长整型、浮点型等类型数最大值的 UDAF 的重载。

一个计算函数必须实现下面 5 个方法。

- **init()方法**　init()方法负责初始化计算函数并重设它的内部状态。在 MaximumIntUDAFEvaluator 中，我们把存放最终结果的 IntWritable 对象设为 null。我们使用 null 来表示目前还没有对任何值进行聚集计算，这和对空集 NULL 计算最大值应有的结果是一致的。

- **iterate()方法**　每次对一个新值进行聚集计算时都会调用 iterate()方法。计算函数要根据聚集计算的结果更新其内部状态。iterate()接受的参数和 Hive 中被调用函数的参数是对应的。在这个示例中，只有一个参数。方法首先检查参数值是否为 null，如果是，则将其忽略。否则，result 变量实例就被设为 value 的整数值(如果这是方法第一次接受输入)，或设为当前值和 value 值中的较大值(如果已经接受一些值)。如果输入值合法，我们就让方法返回 true。

- **terminatePartial()方法**　Hive 需要部分聚集结果时会调用 terminatePartial()方法。这个方法必须返回一个封装了聚集计算当前状态的对象。在这里，因为只需要对已知的最大值或在没有值时的空值 null 进行封装，所以使用一个 IntWritable 即可。

- **merge()方法** 在 Hive 决定要合并一个部分聚集值和另一个部分聚集值时会调用 merge()方法。该方法接受一个对象作为输入。这个对象的类型必须和 terminatePartial()方法的返回类型一致。在这个示例里，merge()方法可以直接使用 iterate()方法，因为部分结果的聚集和原始值的聚集的表达方法是相同的。但一般情况下不能这样做(我们后面会看到更普遍的示例)，这个方法实现的逻辑会合并计算函数和部分聚集的状态。

- **terminate()方法** Hive 需要最终聚集结果时会调用 terminate()方法。计算函数需要把状态作为一个值返回。在这里，我们返回实例变量 result。

现在，让我们来执行这个新写的函数：

```
hive> CREATE TEMPORARY FUNCTION maximum AS 'com.hadoopbook.hive.Maximum';
hive> SELECT maximum(temperature) FROM records;
111
```

图 17-3 显示了计算函数的处理流程。

图 17-3. 包含 UDAF 部分结果的数据流

1. 一个更复杂的 UDAF

前面的示例有一个特别的现象：部分聚集结果可以使用和最终结果相同的类型(IntWritable)来表示。对于更复杂的聚集函数，情况并非如此。考虑一个计算一组 double 类型值均值的 UDAF，就可以看出这一点。从数学角度来看，要把两个部分的均值合并成最终的均值是不可能的(参见 2.4.2 节对 combiner 函数的讨论)。作为替代，我们可以用一个数对(目前已经处理过的 double 值的累积和以及目前已经处理过的数的个数)来表示部分聚集结果。

这个思路在 UDAF 中的实现如范例 17-4 所示。注意，部分聚集结果用一个嵌套的静态类 struct 实现，类名是 PartialResult，由于我们使用了 Hive 能够处理的字段类型(Java 原子数据类型)，所以 Hive 足够“聪明”，能够自己对这个类进行序列化和反序列化。

在这个示例中，merge()方法和 iterate()方法不同，因为它把“部分和”(partial sum)和“部分计数值”(partial count)分别进行成对的加法合并。此外，terminatePartial()的返回类型为 PartialResult，这个类型当然不会给调用函数的用户看到，terminate()的返回类型则是最终用户可以看到的 DoubleWritable。

范例 17-4. 计算一组 double 值均值的 UDAF

```
package com.hadoopbook.hive;

import org.apache.hadoop.hive.ql.exec.UDAF;
import org.apache.hadoop.hive.ql.exec.UDAFEvaluator;
import org.apache.hadoop.hive.serde2.io.DoubleWritable;

public class Mean extends UDAF {

  public static class MeanDoubleUDAFEvaluator implements UDAFEvaluator {
    public static class PartialResult {
      double sum;
      long count;
    }

    private PartialResult partial;

    public void init() {
      partial = null;
    }

    public boolean iterate(DoubleWritable value)  {
```

```
    if (value == null) {
      return true;
    }
    if (partial == null) {
      partial = new PartialResult();
    }
    partial.sum += value.get();
    partial.count++;
    return true;
  }

  public PartialResult terminatePartial() {
    return partial;
  }

  public boolean merge(PartialResult other) {
    if (other == null) {
      return true;
    }
    if (partial == null) {
      partial = new PartialResult();
    }
    partial.sum += other.sum;
    partial.count += other.count;
    return true;
  }

  public DoubleWritable terminate() {
    if (partial == null) {
      return null;
    }
    return new DoubleWritable(partial.sum / partial.count);
  }
 }
}
```

17.9 延伸阅读

有关 Hive 的更多信息，请参考 O'Reilly 在 2012 年出版的 *Programming Hive*，网址为 *http://shop.oreilly.com/product/0636920023555.do*，作者 Edward Capriolo、Dean Wampler 和 Jason Rutherglen。

第 18 章

关于 Crunch

Apache Crunch (*https://crunch.apache.org/*)是用来写 MapReduce 管线的高层 API。与普通的 MapReduce 相比，Crunch 的主要优势体现在它注重程序员友好的 Java 类型(如 String)以及旧式的纯 Java 对象，另外还有一组丰富的数据变换操作和多级管线(不需要显式地管理 MapReduce 工作流中的每个作业)。

从上述角度来看，Crunch 与 Java 版的 Pig 很像。在使用 Pig 时，经常会遇到的问题是写用户自定义函数所使用的语言(Java 或 Python)与写 Pig 脚本所使用的语言(Pig Latin)不同，因此程序员需要在两种不同的表示和语言之间切换，这使得他们的开发体验缺乏连贯性，而这一点正是 Crunch 所回避的。Crunch 与 Pig 不同，其程序和用户自定义函数(UDF)使用的都是同一种语言(Java 或 Scala)，并且 UDF 可以很好地嵌入到程序中。Crunch 的整体编程体验非常像是在写一个非分布式的程序。虽然 Crunch 与 Pig 有很多相似之处，但 Crunch 的灵感却是来自于 FlumeJava，一个由谷歌开发的用于构建 MapReduce 管线的 Java 库。

请不要把 FlumeJava 与第 14 章介绍的 Apache Flume 混淆，后者是用于收集流式事件数据的系统。要想更详细地了解 FlumeJava，可参考由 Craig Chambers 等人合著的 *FlumeJava: Easy, Efficient Data-Parallel Pipelines*，网址为 *http://bit.ly/data-parallel_pipelines*。

因为 Crunch 位于上层，所以 Crunch 管线是高度可组合的，并且可以把常用功能提取到库中以提供给其他程序重用，这一点与 MapReduce 不同，想要重用 MapReduce 的代码是很困难的，因为除了一些简单的情况(例如调用恒等函数或者像 `LongSumReducer` 这样的简单求和函数)之外，大多数程序都有自定义的 mapper 和 reducer 实现。在 MapReduce 中，要想为各式各样的变换操作(如排序和

连接)写 mapper 和 reducer 库并非易事，但在 Crunch 中这则是一件很自然的事情。例如，`org.apache.crunch. lib.Sort` 类中的 `sort()`方法可以对任何输入的 Crunch 集合进行排序。

虽然 Crunch 最初的设计是使用 Hadoop MapReduce 执行引擎来运行的，但 Crunch 并不依赖于 MapReduce，实际上也可以使用 Apache Spark(参见第 19 章)作为分布式执行引擎来运行 Crunch 管线。不同的引擎具有不同的特点。例如，倘若在作业与作业之间需要传递大量的中间数据，那么使用 Spark 就要比使用 MapReduce 效果更好，因为 Spark 可以把数据缓存在内存中，而不是像 MapReduce 那样把数据物化到磁盘上。毋须重写程序就能够在不同的引擎上试运行 Crunch 管线是一个非常强大的特色，因为这使得程序的功能性与程序的执行效率可以被区别对待，一般来说，随着执行引擎的不断调整，程序的执行效率也不断得到提高。

本章将介绍如何利用 Crunch 来编写数据处理程序。更详细的内容可以参考 Crunch 用户指南，网址为 *http://crunch.apache.org/user-guide.html*。

18.1 示例

我们先通过一个简单的 Crunch 管线示例来说明一些基本概念。范例 18-1 中给出的是按年份计算气象数据集中最高温度的 Crunch 版程序，这个程序就是我们在第 2 章中首次遇到的程序。

范例 18-1. 使用 Crunch 找出最高温度的应用程序

```
public class MaxTemperatureCrunch {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperatureCrunch <input path> <output path>");
      System.exit(-1);
    }

    Pipeline pipeline = new MRPipeline(getClass());
    PCollection<String> records = pipeline.readTextFile(args[0]);

    PTable<String, Integer> yearTemperatures = records
        .parallelDo(toYearTempPairsFn(), tableOf(strings(), ints()));
    PTable<String, Integer> maxTemps = yearTemperatures
        .groupByKey()
        .combineValues(Aggregators.MAX_INTS());

    maxTemps.write(To.textFile(args[1]));
    PipelineResult result = pipeline.done();
```

```
  System.exit(result.succeeded() ? 0 : 1);
}

static DoFn<String, Pair<String, Integer>> toYearTempPairsFn() {
  return new DoFn<String, Pair<String, Integer>>() {
    NcdcRecordParser parser = new NcdcRecordParser();

    @Override
    public void process(String input, Emitter<Pair<String, Integer>> emitter) {
      parser.parse(input);
      if (parser.isValidTemperature()) {
        emitter.emit(Pair.of(parser.getYear(), parser.getAirTemperature()));
      }
    }
  };
}
}
```

按照惯例，在检查命令行参数之后，程序首先构建一个 Crunch Pipeline 对象，它代表了我们希望运行的计算。顾名思义，管线可以有多个阶段，也就是说具有多个输入、输出、分枝和迭代的管线都是有可能的，尽管本例从一个单阶段管线入手。我们将使用 MapReduce 来运行这个管线，因此这里创建的是一个 MRPipeline，不过，你也可以选择使用 MemPipeline (在内存中运行管线以用于测试)或者 SparkPipeline(利用 Spark 引擎执行相同的计算)。

管线接受来自一个或多个输入源的数据。本例中的输入源是一个文本文件，文件名由命令行的第一个参数 args[0]指定。Pipeline 类有一个 readTextFile()方法，它可以把文本文件转换成 String 类型的 PCollection 对象，其中每个 String 表示一行文本。PCollection<S>是 Crunch 的最基本的数据类型，它表示由 S 类型的元素组成的不可修改且无序的分布式集合。PCollection<S>也可被视为非物化的 java.util. Collection，之所以称为非物化，是因为它的元素并没有被读取到内存中。本例的输入是由文本文件中的文本行构成的一个分布式的集合，用 PCollection<String>来表示。

Crunch 对 PCollection 进行各种计算操作并产生一个新的 PCollection。我们需要做的第一件事是解析输入文件中的每一行文本并过滤掉任何不良记录。这项任务使用 PCollection 的 parallelDo()方法来完成，它为 PCollection 中的每个元素都执行某种操作，并返回一个新的 PCollection。parallelDo()的方法签名如下：

```
<T> PCollection<T> parallelDo(DoFn<S,T> doFn, PType<T> type);
```

基本思路是，编写能够把类型 S 的实例变换为一个或多个类型 T 实例的 DoFn

实现，然后 Crunch 将该函数应用于 PCollection 的每个元素。显然，这个操作可以在 MapReduce 作业的 map 任务中并行执行。parallelDo()方法的第二个参数是 PType<T>对象，它向 Crunch 传递有关 T 的 Java 类型以及如何序列化该类型的信息。

实际上，本例使用的是重载形式的 parallelDo()，它创建的是一个扩展的 PCollection，称为 PTable<K, V>。PTable<K, V>是由键-值对构成的分布式 multi-map。(multi-map 是可以具有重复键-值对的 map。)因此，我们可以把年份作为键，把温度作为值来表示，从而在稍后的管线中能够对它进行分组及聚合操作。这个方法的签名是：

```
<K, V> PTable<K, V> parallelDo(DoFn<S, Pair<K, V>> doFn, PTableType<K, V> type);
```

在本例中，DoFn 解析输入文本文件中的文本行，并输出年份/温度对：

```
static DoFn<String, Pair<String, Integer>> toYearTempPairsFn() {
  return new DoFn<String, Pair<String, Integer>>() {
    NcdcRecordParser parser = new NcdcRecordParser();
    @Override
    public void process(String input, Emitter<Pair<String, Integer>> emitter)
      {parser.parse(input);
       if (parser.isValidTemperature()) {
         emitter.emit(Pair.of(parser.getYear(), parser.getAirTemperature()));
      }
    }
  };
}
```

在应用该函数后，得到一个年份/温度对的列表：

```
PTable<String, Integer> yearTemperatures = records
    .parallelDo(toYearTempPairsFn(), tableOf(strings(), ints()));
```

parallelDo()方法的第二个参数是通过 Crunch Writables 类的静态方法构造的 PTableType<K, V>实例(之所以使用 Writables 类的静态方法，是因为我们选择了使用 Hadoop 的 Writable 序列化框架来处理由 Crunch 写入的任何中间数据)。tableOf()方法则根据指定的键/值类型创建一个 PTableType。strings()方法声明内存中的键使用 Java String 对象来表示，并序列化为 Hadoop Text。值的类型是 Java int，并序列化为 Hadoop IntWritables。

此刻，我们的数据表示已经变得更加结构化，但是记录数量并没有改变，这是因为输入文件中的每一行对应于 yearTemperatures 表中的一个表项。为了计算数据集中每年最大的温度读数，我们需要对这些表项按年份来分组，然后找出每年

的最高温度值。幸运的是，Crunch 恰好提供了此类操作，它们是 PTable 的 API 的一部分。groupByKey()方法执行的是 MapReduce 的混洗(shuffle)操作，它按键对表项进行分组，并返回第三种 PCollection，我们称之为 PGroupedTable<K, V>，其 combineValues()方法能够把一个键的所有值聚合起来，就像 MapReduce 的 reducer 所做的那样：

```
PTable<String, Integer> maxTemps = yearTemperatures
    .groupByKey()
    .combineValues(Aggregators.MAX_INTS());
```

combineValues()方法的输入是一个 Crunch Aggregator 实例，它是一个简单的接口，用于表示一组任意类型值的聚合。在这里，我们可以充分利用 Aggregators 类中被称为 MAX_INTS 的内置聚合程序，其作用是找出一组整数值中的最大值。

此管线的最后一步是调用 write()把 maxTemps 表写入一个文件，write()的输入就是通过静态工厂 To 的 textFile 方法创建的一个文本文件目标对象。Crunch 实际上使用了 Hadoop 的 TextOutputFormat 格式来完成这个操作，这就意味着输出的每一行由制表符分隔的键和值组成：

```
maxTemps.write(To.textFile(args[1]));
```

目前为止，上述程序仅仅考虑管线的构造。为了执行这个管线，我们必须调用 done()方法，此时，程序进入阻塞状态，直至管线执行完毕。Crunch 返回一个 PipelineResult 对象，该对象封装了管线中运行的各种作业的统计数据以及管线运行成功与否的标志，后者被我们用来设置适当的程序退出代码。

当我们在样本数据集上运行该程序后，得到以下结果：

```
% hadoop jar crunch-examples.jar crunch.MaxTemperatureCrunch \
  input/ncdc/sample.txt output
% cat output/part-r-00000
1949 111
1950 22
```

18.2 Crunch 核心 API

本节将介绍 Crunch 的核心接口。从设计上来说，Crunch API 是一种高层的 API，因此程序员可以集中精力考虑计算的逻辑操作，而不是具体的执行细节。

18.2.1 基本操作

Crunch 的核心数据结构是 PCollection<S>，它是一个不可修改且无序的分布式 S 类型元素的集合。本节将讨论基于 PCollection 的基本操作及其派生出的类型：PTable 和 PGroupedTable。

1. union()

Crunch 的最简单的基本操作是 union()，它返回一个 PCollection，在这个返回的 PCollection 中包含了调用该函数的 PCollection 以及作为输入参数的 PCollection 的所有元素。例如：

```
PCollection<Integer> a = MemPipeline.collectionOf(1, 3);
PCollection<Integer> b = MemPipeline.collectionOf(2);
PCollection<Integer> c = a.union(b);
assertEquals("{2,1,3}", dump(c));
```

MemPipeline 的 collectionOf()方法用于创建一个由少量元素构成的 PCollection，通常这种 PCollection 被用于测试或演示。dump()方法是此处引入的另一个实用方法，它把一个小型的 PCollection 中的内容以字符串的形式呈现出来。dump()不属于 Crunch，但可以在本书随附示例代码的 PCollections 类中找到它的实现。由于 PCollection 是无序的，因此 c 中的元素的排序顺序没有定义。

在对两个 PCollection 进行并集操作时，它们必须是由同一管线创建的(否则该操作在运行时会失败)，并且必须具有相同的类型。后一个条件在编译时强制执行，这是因为 PCollection 是一种参数化的类型，且进行并集操作的两个 PCollection 的类型参数必须匹配。

2. parallelDo()

第二个基本操作是 parallelDo()，它为输入的 PCollection<S>中的每个元素调用某个函数，并返回包含该调用结果的一个新的输出 PCollection <T>。parallelDo()的最简单的形式包含两个参数：DoFn<S, T>实现和 PType<T>实例。DoFn<S, T>实现定义了一个用于把类型为 S 的元素变换为类型为 T 的元素的函数，PType<T>实例则用于描述输出类型 T，详情参见 18.2.2 节对 PTypes 的详细介绍。

下面这段代码演示的是如何利用 parallelDo()来为 String 类型的 PCollection 应用一个计算字符串长度的函数：

```
PCollection<String> a = MemPipeline.collectionOf("cherry", "apple", "banana");
PCollection<Integer> b = a.parallelDo(new DoFn<String, Integer>() {
  @Override
  public void process(String input, Emitter<Integer> emitter) {
    emitter.emit(input.length());
  }
}, ints());
assertEquals("{6,5,6}", dump(b));
```

本例输出一个整数型的 PCollection，其元素数目与输入 PCollection 的元素数目相同，因此也可以使用 DoFn 中的 MapFn 子类来实现这种一对一的映射：

```
PCollection<Integer> b = a.parallelDo(new MapFn<String, Integer>() {
  @Override
  public Integer map(String input) {
    return input.length();
  }
}, ints());
assertEquals("{6,5,6}", dump(b));
```

parallelDo()经常被用于滤除那些在后续处理步骤中不需要的数据。Crunch 为此专门提供了 filter()方法，其参数是一个特殊的 DoFn，称为 FilterFn。在这段实现中只需要实现 accept()方法来指示一个元素是否应当出现在输出中。例如，下面这段代码用于仅保留有偶数个字符的字符串：

```
PCollection<String> b = a.filter(new FilterFn<String>() {
  @Override
  public boolean accept(String input) {
    return input.length() % 2 == 0; // even
  }
});
assertEquals("{cherry,banana}", dump(b));
```

请注意，在 filter()方法的签名中没有 PType，这是因为输出 PCollection 与输入 PCollection 的类型相同。

如果 DoFn 明显地改变了它所操作的 PCollection 中的元素数目，则应当重写其 scaleFactor()方法，以给 Crunch 规划器(planner)一个提示，用以估计输出 PCollection 的相对大小，这样做可以提高效率。

FilterFn 的 scaleFactor()方法的返回值为 0.5，换句话说就是假设 FilterFn 将会过滤掉大约一半的 PCollection 元素。如果自己的过滤函数将要过滤掉的元素明显多于或少于一半，可以考虑重写 scaleFactor()方法。

有一种重载形式的 parallelDo()可以从 PCollection 产生一个 PTable。回顾 18.1 节中的示例，PTable<K, V>是一个由键-值对构成的多重映射，或者用 Java 语言来说，PTable<K, V>就是 PCollection<Pair<K, V>>，其中 Pair<K, V>是 Crunch 的键-值对类。

下面这段代码通过使用 DoFn 来创建一个 PTable，其中 DoFn 的功能是把输入字符串变换为键-值对，以字符串的长度作为键，字符串本身作为值：

```
PTable<Integer, String> b = a.parallelDo(
  new DoFn<String, Pair<Integer, String>>() {
  @Override
  public void process(String input, Emitter<Pair<Integer, String>> emitter) {
    emitter.emit(Pair.of(input.length(), input));
  }
}, tableOf(ints(), strings()));
assertEquals("{(6,cherry),(5,apple),(6,banana)}", dump(b));
```

从由某些值构成的 PCollection 中提取键以形成一个 PTable，此类任务很常见，因此 Crunch 专门提供了一种方法，称为 by()。by()方法的参数是 MapFn<S, K>，它的功能就是把输入值 S 映射为键 K：

```
PTable<Integer, String> b = a.by(new MapFn<String, Integer>() {
  @Override
  public Integer map(String input) {
    return input.length();
  }
}, ints());
assertEquals("{(6,cherry),(5,apple),(6,banana)}", dump(b));
```

3. groupByKey()

第三个基本操作是 groupByKey()，用于把 PTable<K, V>中具有相同键的所有值聚合起来。这个操作可以看作是 MapReduce 的混洗(shuffle)操作，事实上，在使用 MapReduce 执行引擎的情况下，它就是这样实现的。groupByKey()返回的 Crunch 类型是 PGroupedTable<K, V>，即 PCollection<Pair<K, Iterable<V>>>，或者说是一个 multi-map，其中每个键根据它的值与一个可迭代(iterable)集合配对。

继续讨论前面的代码，如果我们按照键来对这个由“长度-字符串”(length-string)映射构成的 PTable 进行分组，那么将会得到如下结果(其中方括号中的项表示一个 iterable 集合)：

```
PGroupedTable<Integer, String> c = b.groupByKey();
assertEquals("{(5,[apple]),(6,[banana,cherry])}", dump(c));
```

Crunch 可以根据表的大小为 groupByKey()操作设置分区数量(以减少 MapReduce 中的任务数)。在大多数情况下，分区数量使用默认设置即可，不过，如果有必要，也可以使用重载形式的 groupByKey(int)来明确地设置分区数量。

4. combineValues()

尽管从命名上看，PGroupedTable 好像是 PTable 的一个子类，但实际上它不是，所以你不能为 PGroupedTable 调用像 groupByKey()这样的方法。这是因为没有理由对一个已经按键分过组的 PTable 再次按键分组。你也可以把 PGroupedTable 视为产生另一个 PTable 之前的中间表示。归根结底，之所以我们需要按键来分组，是因为这样做可以针对每个键的值执行一些操作，而这正是第四个基本操作的基础。

combineValues()的最常见的形式是以组合函数 CombineFn<K,V>作为输入。CombineFn<K,V>就是 DoFn<Pair<K, Iterable<V>>, Pair<K, V>>的简写，它返回一个 PTable<K, V>。为了方便观察程序执行，可以考虑下面这个组合函数，它把一个键对应的所有字符串值连接起来，并以分号作为分隔符：

```
PTable<Integer, String> d = c.combineValues(new CombineFn<Integer, String>() {
  @Override
  public void process(Pair<Integer, Iterable<String>> input,
      Emitter<Pair<Integer, String>> emitter) {
    StringBuilder sb = new StringBuilder();
    for (Iterator i = input.second().iterator(); i.hasNext(); ) {
      sb.append(i.next());
      if (i.hasNext()) { sb.append(";"); }
    }
    emitter.emit(Pair.of(input.first(), sb.toString()));
  }
});
assertEquals("{(5,apple),(6,banana;cherry)}", dump(d));
```

字符串的连接操作是不可交换的，因此得到的结果没有确定性。这对于你的应用程序来说，可能非常重要，也可能无关紧要。

由于在 process()方法中使用了 Pair 对象，因而使得这段代码显得有些杂乱。Pair 对象必须通过 first()和 second()调用展开，并创建一个新的 Pair 对象以传出新的键-值对。这个组合函数对键来说没有做任何改变，因此我们可以使用一种重载形式的 combineValues()，它以 Aggregator 对象作为输入，并且仅对

值进行操作，而对键则不做任何改变地直接传递。更妙的是，我们可以使用 Aggregators 类的内置的 Aggregator 实现来对所有字符串实施连接操作。于是，该代码变为：

```
PTable<Integer, String> e = c.combineValues(Aggregators.STRING_CONCAT(";",
    false));
assertEquals("{(5,apple),(6,banana;cherry)}", dump(e));
```

有的时候，你可能想聚合 PGroupedTable 中的值并返回一个与被分组的值的类型不同的结果。这可以通过 mapValues()方法来实现，它的 MapFn 可以把 iterable 集合变换为另一种对象。例如，下面这段代码用于计算每个键所对应的值的个数：

```
PTable<Integer, Integer> f = c.mapValues(new MapFn<Iterable<String>, Integer>() {
  @Override
  public Integer map(Iterable<String> input) {
    return Iterables.size(input);
  }
}, ints());
assertEquals("{(5,1),(6,2)}", dump(f));
```

注意，虽然值都是字符串类型，但是应用这个 map 函数得到的结果却是整数类型。这段代码利用了 Guava 的 Iterables 类来计算 iterable 集合的大小。

你可能想知道，既然已经有强大的 mapValues()方法，为什么还需要有 combineValues()方法呢？原因是 combineValues()可以被当作 MapReduce 的 combiner 来运行，因此，通过在 map 端运行 combineValues()方法可减少混洗操作需要传输的数据量，从而使性能得到提升(参见 2.4.2 节)。由于 mapValues()方法被解释为 parallelDo()操作，因此在这种情况下，它只能在 reduce 端运行，所以不可能通过 combiner 来提高其性能。

最后，PGroupedTable 还有一种方法，称为 ungroup()，它用于把 PGroupedTable<K, V> 还原为 PTable<K, V>，即 groupByKey()操作的逆操作。但是，ungroup()不属于基本操作，因为它是通过 parallelDo()来实现的。对一个 PTable 来说，先调用 groupByKey()，再调用 ungroup()的效果相当于对该表执行了按键的不完全排序，但一般来说还是利用 Sort 库更加方便，因为它可以实现完全排序(通常这才是你所需要的)，并且还提供了一些排序选项。

18.2.2 类型

每个 PCollection<S>都有一个关联的类，称为 PType<S>，它用于封装有关 PCollection 中的元素类型的信息。PType<S>指明了 PCollection 中的元素的 Java 类型为 S，同时也给出从持久存储器读取到 PCollection 的序列化格式，以及相反方向从 PCollection 写入持久存储器的序列化格式。

Crunch 有两个 PType 家族：Hadoop Writables 和 Avro。大致来说，具体选择哪一个 PType 家族取决于管线使用的文件格式：Writables 用于顺序文件，Avro 用于 Avro 数据文件。这两个 PType 家族都可用于文本文件。管线也可以使用来自不同家族的 PTypes 的混合体(因为与 PType 关联的是 PCollection，而非管线)，但这种做法通常没有必要，除非你眼前需要跨家族，比如文件格式转换。

一般情况下，Crunch 会尽可能地隐藏不同序列化格式之间的差异，从而使编程人员可以在代码中使用自己熟悉的 Java 的类型。(这样做的另一个好处是能够方便地编写有关 Crunch 集合的库和工具程序，而不需要考虑它们属于哪个序列化家族。)例如，我们可以用普通的 Java String 对象来表示从文本文件中读取的文本行，而不需要使用 Writable Text 变量或 Avro 的 UTF8 对象。

PCollection 所使用的 PType 在 PCollection 创建时指定，尽管有些时候它是隐式指定的。例如，在读取文本文件时使用的是默认的 Writables，如下面这段测试代码所示：

```
PCollection<String> lines = pipeline.read(From.textFile(inputPath));
assertEquals(WritableTypeFamily.getInstance(), lines.getPType().getFamily());
```

不过，也可以显式地指定使用 Avro 序列化，只需要把恰当的 PType 传递给 textFile()方法即可。此处，我们利用 Avros 类的静态工厂方法来创建一个 PType<String>的 Avro 序列化表示：

```
PCollection<String> lines = pipeline.read(From.textFile(inputPath,
    Avros.strings()));
```

同理，对于那些将会创建新的 PCollection 的操作来说，必须为该操作指定 PType，并且要求它与 PCollection 的类型参数匹配。[1]例如，在前面的例子

① 某些操作不要求指定 PType，因为它们可以从相应的 PCollection 中推断出 PType。例如，filter()返回的 PCollection 应当与输入的 PType 相同。

中，parallelDo()操作从 PCollection<String>中提取整数型的键，并将其变换为 PTable<Integer, String>，且指明匹配的 PType 为

```
tableOf(ints(), strings())
```

上述三种方法都是从 Writables 中静态导入的。

1. 记录和元组

在处理由多个字段构成的复杂对象时，可以选择使用 Crunch 的记录(record)或者元组(tuple)。记录就是通过名称来访问字段的类，比如 Avro 的 GenericRecord 对象、旧式的纯 Java 对象(对应于 Avro 的 Specific 或 Reflect 对象)或者是定制的 Writable。另一方面，元组是通过位置来访问字段的类。Crunch 为此提供了 Tuple 接口，另外还有一些可用于少量元素组成元组的便捷类：Pair<K, V> 、Tuple3<V1, V2, V3>、Tuple4<V1, V2, V3, V4>，以及具有任意固定数量的值的元组 TupleN。

用记录写的 Crunch 程序更加易于阅读和理解，因此，我们应当尽可能使用记录而非元组。如果气象记录用由年份、温度和站点 ID 字段构成的 WeatherRecord 类来表示，那么使用

```
Emitter<Pair<Integer, WeatherRecord>>
```

将会比

```
Emitter<Pair<Integer, Tuple3<Integer, Integer, String>>
```

更简单。

WeatherRecord 清楚地表达了它是什么，元组则不同，它的类型名称没有传达任何有意义的信息。

正如这个例子所暗示的，我们不可能做到完全避免使用 Crunch 的 Pair 对象，因为它是 Crunch 用来表示一个表集合的基本方法之一，回想一下，PTable<K, V> 其实就是 PCollection<Pair<K, V>>。不过，在很多情况下还是可以想办法限制 Pair 对象的使用，这样做可以让代码更加易读。例如，在创建一个表时，如果表中的值与 PCollection 的值相同，那么使用 PCollection 的 by()方法要比使用 parallelDo()方法更好(参见 18.2.1 节对 parallelDo()的讨论)，或者使用以 Aggregator 对象作为输入的 PGroupedTable 的 combineValues()方法(参见 18.2.1 节中对 combineValues()的讨论)，而不是使用 CombineFn 方法。

在 Crunch 管线中使用记录的最便捷的方式是定义一个其字段能够被 Avro Reflect 序列化的 Java 类和一个无参数的构造函数，比如说下面的 WeatherRecord 类：

```
public class WeatherRecord {
  private int year;
  private int temperature;
  private String stationId;

  public WeatherRecord() {
  }

  public WeatherRecord(int year, int temperature, String stationId) {
    this.year = year;
    this.temperature = temperature;
    this.stationId = stationId;
  }

// ... getters elided
}
```

有了上述定义，就可以直观地从 PCollection<String>生成 PCollection<WeatherRecord>，并利用 parallelDo()方法把每一行文本解析到 WeatherRecord 对象中：

```
PCollection<String> lines = pipeline.read(From.textFile(inputPath));
PCollection<WeatherRecord> records = lines.parallelDo(
  new DoFn<String, WeatherRecord>() {
  NcdcRecordParser parser = new NcdcRecordParser();
  @Override
  public void process(String input, Emitter<WeatherRecord> emitter) {
    parser.parse(input);
    if (parser.isValidTemperature()) {
      emitter.emit(new WeatherRecord(parser.getYearInt(),
        parser.getAirTemperature(), parser.getStationId()));
    }
  }
}, Avros.records(WeatherRecord.class));
```

如前面这段代码所示，records()方法为 Avro Reflect 数据模型返回一个 Crunch PType，不过，它也支持的 Avro 的 Specific 和 Generic 数据模型。如果要使用 Avro Specific，那么应当先利用 Avro 模式文件来定义一个定制类型并生成相应的 Java 类，然后用生成的 Java 类调用 records()。对于 Avro Generic，则应当声明一个 GenericRecord 类。

Writables 也提供了 records()方法以用于定制类型，不过，它们的定义更加繁琐，因为你必须编写自己的序列化逻辑(参见 5.3.3 节)。

有了记录的集合，就可以用 Crunch 类库或者我们自己的处理函数对其进行计算。例如，下面这段代码按照字段的定义顺序来执行对气象记录的完全排序，按照先年份，再温度，最后站点 ID 的顺序：

```
PCollection<WeatherRecord> sortedRecords = Sort.sort(records);
```

18.2.3 源和目标

本节将介绍 Crunch 的各种数据源和目标，以及这些源和目标的用法。

1. 读取源

Crunch 管线的起点是一个或多个 Source<T>实例，它们指明输入数据的存储位置以及 PType<T>。对于读取文本文件这种简单情况而言，使用 Pipeline 的 readTextFile()方法效果就很好，但是对其他类型的数据源则需要使用 read()方法，它以 Source<T>对象为输入。实际上，下面的代码：

```
PCollection<String> lines = pipeline.readTextFile(inputPath);
```

简写形式如下：

```
PCollection<String> lines = pipeline.read(From.textFile(inputPath));
```

在 org.apache.crunch.io 包中的 From 类就像是各种文件源静态工厂方法的集合，而文本文件只是其中的一个例子。

另一种常见情况是读取由 Writable 键-值对所构成的顺序文件。此时，数据源是内容为键-值对的 TableSource<K, V>，并返回一个 PTable<K, V>。例如，由 IntWritable 类型的键和 Text 类型的值构成的顺序文件所产生的是一个 PTable<Integer, String>：

```
TableSource<Integer, String> source =
    From.sequenceFile(inputPath, Writables.ints(), Writables.strings());
PTable<Integer, String> table = pipeline.read(source);
```

同样，也可以把 Avro 数据文件读取到 PCollection 中，如下所示：

```
Source<WeatherRecord> source =
    From.avroFile(inputPath, Avros.records(WeatherRecord.class));
PCollection<WeatherRecord> records = pipeline.read(source);
```

通过 From 类的 formattedFile()方法，任何 MapReduce(在新 MapReduce API 中的)FileInputFormat 都可被用作 TableSource，这使得 Crunch 能够访问大量

Hadoop 支持的不同文件格式。Crunch 的数据源并不止 From 类中给出的这几种，另外还包括以下几种。

- AvroParquetFileSource，用于以 Avro PType 读取 Parquet 文件。
- FromHBase 的 table()方法，用于把 HBase 表中的行读取到 PTable<ImmutableBytesWritable, Result>中。ImmutableBytesWritable 是 HBase 的一个类，用于字节形式表示的行键(row key)，并在 Result 中包含行扫描得到的单元格，且通过配置能够仅返回某个特定列或列族的单元格。

2. 写入目标

要想把 PCollection 写入目标非常简单，只需在调用 PCollection 的 write()方法时使用所期望的 Target 即可。最常见的目标是一个文件，并且可以通过 To 类的静态工厂方法来选择文件类型。例如，下面这段代码用于在默认文件系统的 *output* 目录下写一个 Avro 文件：

```
collection.write(To.avroFile("output"));
```

而上述代码即为以下代码的简写：

```
pipeline.write(collection, To.avroFile("output"));
```

由于 PCollection 被写入的是 Avro 文件，因此必须使用 Avro 家族的 PType，否则管线运行将会失败。

在 To 类工厂中有一些方法可用于创建文本文件、顺序文件以及任何 MapReduce FileOutputFormat 格式的文件。对于 Parquet 文件格式(AvroParquetFileTarget)和 HBase(ToHBase)而言，Crunch 也有内置的 Target 实现。

Crunch 试图以顺其自然的方式向目标文件写入集合的类型。例如，使用 Pair 记录模式把 PTable 写入 Avro 文件，其中键和值的字段与 PTable 匹配。同样，PCollection 的值作为顺序文件的值写入(键为空)，而 PTable 写入文本文件时使用制表符分隔的键和值。

3. 输出已存在

如果基于文件的目标已经存在，那么在调用 write()方法时 Crunch 将会引发 CrunchRuntimeException 错误。这种行为保留了 MapReduce 的做法，这是一种保守行为，也就是说，除非用户明确指示，否则不会覆盖已存在的输出(参见

2.3.3 节)。

我们可以向 write()方法传递一个标志，用于指示覆盖已存在的输出，示例如下：

```
collection.write(To.avroFile("output"), Target.WriteMode.OVERWRITE);
```

倘若 *output* 已存在，那么在管线运行之前将其删除。

另外还有一种写模式是 APPEND 模式，它在输出目录下添加一个新文件[①]，并保留上一次运行产生的任何现有文件完好无损。Crunch 负责在文件名中利用唯一标识符来避免新产生的文件覆盖先前已产生的文件的可能性。[②]

最后一种写模式是 CHECKPOINT 模式，它用于把当前工作保存在一个文件中，从而使新的管线可以从检查点而不是管线的起点开始执行。这种模式的详情将在 18.3.5 节介绍。

4. 组合的源和目标

有的时候，你可能希望一边写入目标，一边又将其作为源来读取(即在同一个程序的另一个管线中)。对于这种情况，Crunch 提供了一种既是 Source<T>又是 Target 的 SourceTarget<T>接口。在 At 类中有一些静态工厂方法可用于为文本文件、顺序文件和 Avro 文件创建 SourceTarget 实例。

18.2.4 函数

不论什么样的 Crunch 程序，其关键部位一定是那些能够把一种 PCollection 变换为另一种 PCollection 的函数(用 DoFn 来表示)。本节将讨论在写自定义函数时需要注意的一些事项。

1. 函数的序列化

在写 MapReduce 程序时，需要把 mapper 和 reducer 代码打包到作业的 JAR 文件中，从而使 Hadoop 能够通过搜索类路径找到用户代码(参见 6.5.1 节)。Crunch 则采取了不同的方式。在管线执行时，所有 DoFn 实例都被序列化到一个文件中，并通过 Hadoop 的分布式缓存机制把该文件分发给各任务节点(参见 9.4.2 节)，然后

① 虽然名为 APPEND，但这种模式并不会把内容直接附加到已存在的输出文件中。
② HBaseTarget 不检查已存在的输出，因此它的行为就如同使用 APPEND 模式一样。

任务本身反序列化，从而使得这些 DoFn 能够被调用。

其结果是，作为用户的你不需要做任何打包工作，只需要确保自己的 DoFn 实现能够通过标准的 Java 序列化机制进行序列化。[①]

在大多数情况下，没有什么额外工作需要做，因为 DoFn 的基类被声明为 java.io.Serializable 接口实现。如果你的函数是无状态的，那么也就是说没有字段需要序列化，因此在序列化时不会出现问题。

不过，还是有一些需要注意的事项。假如 DoFn 作为一个内部类(也称为非静态嵌套类，如匿名类)被定义在一个没有实现 Serializable 的外部类中，那么就会出现问题：

```
public class NonSerializableOuterClass {

  public void runPipeline() throws IOException {
    // ...
    PCollection<String> lines = pipeline.readTextFile(inputPath);
    PCollection<String> lower = lines.parallelDo(new DoFn<String, String>() {
      @Override
      public void process(String input, Emitter<String> emitter) {
        emitter.emit(input.toLowerCase());
      }
    }, strings());
    // ...
  }
}
```

由于内部类可隐式地引用自己的类实例，因此，如果封装该函数的类没有被序列化，那么该函数也没有序列化，于是管线执行将会出现 CrunchRuntimeException 错误。要想解决这个问题很简单，可以把该函数定义为一个(命名的)静态嵌套类或者一个顶级类，也可以让封装该函数的类实现 Serializable 接口。

另一个问题是，如果函数依赖于一个以实例变量形式表示的非序列化的状态，并且它的类没有 Serializable，那么，在这种情况下可以将非序列化的实例变量标记为 transient(瞬态)，从而使 Java 不去尝试对其进行序列化，然后在 DoFn 的 initialize()方法中设置这个变量。Crunch 会在首次调用 process()方法之前先来调用 initialize()方法：

① 参见文档，网址为 *http://bit.ly/interface_serializable*。

```
public class CustomDoFn<S, T> extends DoFn<S, T> {

  transient NonSerializableHelper helper;

  @Override
  public void initialize() {
    helper = new NonSerializableHelper();
  }

  @Override
  public void process(S input, Emitter<T> emitter) {
    // use helper here
  }
}
```

虽然我们在这里没有举例，不过，也可以通过其他非瞬态实例变量(比如字符串)来传递状态，以初始化瞬态的实例变量。

2. 对象重用

在使用 MapReduce 时，reducer 的值迭代器中的对象是可重用的，其目的是为了提高效率(避免对象分配的开销)。对于 `PGroupedTable` 的 `combineValues()`和 `mapValues()`方法的迭代器来说，Crunch 具有相同的行为。因此，如果你想要在迭代器的外部引用一个对象，那么就应当复制该对象，以避免对象标识错误。

为了说明这个问题，我们编写了一段通用的实用程序，它为 PTable 中的每个键查找唯一值的集合，如范例 18-2 所示。

范例 18-2. 为 PTable 中的每个键查找唯一值集合

```
public static <K, V> PTable<K, Collection<V>> uniqueValues(PTable<K, V> table) {
  PTypeFamily tf = table.getTypeFamily();
  final PType<V> valueType = table.getValueType();
  return table.groupByKey().mapValues("unique",
      new MapFn<Iterable<V>, Collection<V>>() {
    @Override
    public void initialize() {
      valueType.initialize(getConfiguration());
    }

    @Override
    public Set<V> map(Iterable<V> values) {
      Set<V> collected = new HashSet<V>();
      for (V value : values) {
        collected.add(valueType.getDetachedValue(value));
      }
      return collected;
    }
  }, tf.collections(table.getValueType()));
```

```
}
```

其基本思路是，先按键分组，再遍历与每个键关联的所有值，并把唯一值收集到一个可以自动删除重复值的 Set 中。由于我们希望在迭代器之外保存这些值，因此在把这些值放入 Set 之前，需要先复制每个值。

幸运的是，我们并不需要编写代码来复制每一种可能出现的 Java 类，实际上，本例使用的是 Crunch 提供的 getDetachedValue()方法。根据表中的值的类型可以得到 PType，有了 PType 就可以通过 getDetachedValue()方法对 Java 类进行复制。请注意，我们还是需要在 DoFn 的 initialize()方法中初始化 PType，从而使这个 PType 能够访问配置以执行复制操作。

对于像 Strings 或 Integers 这样的不可变对象，调用 getDetachedValue()方法实际上是一个空操作，但是对于可变的 Avro 或 Writable 对象，调用 getDetachedValue()方法则会为每个值做深层复制。

18.2.5 物化

物化(Materialization)是让 PCollection 中的值变得可用的过程，只有物化后的值才能够被程序读取。例如，你可能希望读取一个(通常很小的)PCollection 的所有值并将其显示出来，或者把这些值发送给程序的其他部分，而不是把它们写入 Crunch 目标。物化 PCollection 的另一个理由是为了将 PCollection 的内容作为下一个处理步骤的判断基础。例如，测试一个迭代算法的收敛性，详情参见 18.3.4 节。

有多种方法可以物化 PCollection，其中最直接的方法是调用 materialize()，它返回一个包含了 PCollection 的值的 Iterable 集合。如果 PCollection 还没有被物化，那么 Crunch 必须执行管线以确保 PCollection 中的对象都已经被计算并存储到一个临时文件中，这样才有可能对这些值进行遍历操作。[①]

下面这段程序用于使文本文件中的文本行小写字母化：

```
Pipeline pipeline = new MRPipeline(getClass());
PCollection<String> lines = pipeline.readTextFile(inputPath);
PCollection<String> lower = lines.parallelDo(new ToLowerFn(), strings());
```

① 在这个例子中，并没有显式地调用 run()或 done()来运行管线，不过在管线运行完成后调用 done()仍然不失为好的做法，这样可以使中间文件得到处理。

```
Iterable<String> materialized = lower.materialize();
for (String s : materialized) { // pipeline is run
    System.out.println(s);
}
pipeline.done();
```

上面这段程序利用 ToLowerFn 函数对文本文件中的行进行变换操作，ToLowerFn 函数是单独定义的，因此它可以重复使用：

```
public class ToLowerFn extends DoFn<String, String> {
  @Override
  public void process(String input, Emitter<String> emitter) {
      emitter.emit(input.toLowerCase());
  }
}
```

变量 lower 调用 materialize()并返回一个 Iterable<String>。但是，调用 materialize()并不会导致管线运行，只有从 Iterable 中创建出一个 Iterator 后(通过每个 for each 循环语句隐式地创建)，Crunch 才会执行管线。当管线执行完毕，就可以对已物化的 PCollection 执行迭代过程，在本例中就是把小写字母化的文本行打印到控制台。

你可能会认为 PTable 的 materializeToMap()方法在行为上类似于 materialize()方法，不过，这两种方法之间存在两个重要区别。首先，materializeToMap()返回的是 Map<K, V>，而不是迭代器(iterator)，也就是说整张表会被立刻加载到内存中，而这种行为对于较大的集合来说是应当避免的。第二，虽然 PTable 是多映射(multi-map)的，但 Map 接口并不支持单个键具有多个值，因此，如果这张表中的一个键有多个值，那么在返回的 Map 中只能保留一个值，其余值将丢失。

为了避免上述问题，我们只需要简单地为一个表调用 materialize()方法以获得 Iterable<Pair<K, V>>即可。

1. PObject

物化 PCollection 的另一种方式是使用 PObject。PObject<T>是一个被标志为 *future* 的对象，也就是说程序在创建 PObject 时，类型为 T 的值的计算可能还没有完成。计算结果可以通过调用 PObject 的 getValue()方法获取，而在计算完成(通过 Crunch 管线的执行)并返回值之前，getValue()方法将处于阻塞状态。

调用 PObject 的 getValue()方法类似于调用 PCollection 的 materialize()

方法，因为它们都会触发管线运行来物化所需的集合。事实上，我们可以使用 PObject 来重写文本文件行小写字母化的程序，如下所示：

```
Pipeline pipeline = new MRPipeline(getClass());
PCollection<String> lines = pipeline.readTextFile(inputPath);
PCollection<String> lower = lines.parallelDo(new ToLowerFn(), strings());

PObject<Collection<String>> po = lower.asCollection();
for (String s : po.getValue()) { // pipeline is run
    System.out.println(s);
}
pipeline.done();
```

asCollection()方法把 PCollection<T>变换为普通的 Java Collection<T>。[①] 由于本例使用的是 PObject 的方式，因此，如果有必要，这个变换操作可以被推迟到程序的稍后某刻再执行。本例在获得 PObject 后立即调用了 PObject 的 getValue()方法，以便对 getValue()方法返回的 Collection 进行遍历操作。

asCollection()会把 PCollection 中的所有对象都物化在内存中，因而此方法只适用于较小的 PCollection，比如计算结果中只包含少数几个对象时。而 materialize()的使用则没有这样的限制，它对 Collection 进行遍历操作，而不是把整个集合一次性存储在内存中。

在写作本章内容时，Crunch 还没有提供任何方法用于在管线执行当中检查 PObject，比如在 DoFn 内部检查 PObject。只有在管线执行完毕后，才能对 PObject 进行检查。

18.3 管线执行

在管线构建期间，Crunch 会建立一个内部执行计划。这个执行计划要么由用户显式地运行，要么由 Crunch 隐式地运行(参见 18.2.5 节对物化的讨论)。这个执行计划是由 PCollection 操作构成的一个有向无环图，凡在计划内的每个 PCollection 都与产生它的操作之间存在引用关系，并且除了这些 PCollection 之外，还包含各种操作的参数。此外，每个 PCollection 都有一个内部状态，用于记录它是否已被物化。

① PTable<K, V>的 asMap()方法也能返回一个类型为 PObject<Map<K, V>>的对象。

18.3.1 运行管线

通过调用 Pipeline 的 run()方法可显式地执行管线操作，步骤如下。

首先进行优化处理，把执行计划分为若干阶段。优化的细节取决于执行引擎。也就是说，同样的计划，针对 MapReduce 的优化与针对 Spark 的优化是不同的。

接下来，执行优化计划中的各阶段(在有可能的情况下并行执行)，从而使结果得到的 PCollection 被物化。将要写入 Target 的 PCollection 被物化为目标本身，有可能是 HDFS 的一个输出文件，也可能是 HBase 的一张表。而中间 PCollection 的物化则是通过把该集合的序列化对象写入一个 HDFS 的临时中间文件来实现的。

最后，run()方法向调用者返回一个 PipelineResult 对象，其中包含了刚才运行的每个阶段的信息(持续时间以及 MapReduce 计数器[①])，以及管线运行是否成功(使用 succeeded()方法)。

clean()方法用于清除物化 PCollection 时创建的所有临时中间文件。clean()方法的调用应当在管线执行完毕后，以释放 HDFS 的磁盘空间。clean()方法有一个布尔参数，用于指示是否需要强行删除临时文件。如果这个参数设置为 false，那么临时文件将在管线的所有目标均已创建完毕后才能删除。

与其先调用 run()，再调用 clean(false)，还不如直接调用 done()更方便，它们具有相同的效果，包括指示管线运行启动，并在不需要临时文件时进行清理。

1. 异步执行

run()方法是一种阻塞调用，也就是说在返回之前，它会等待直至管线执行完毕。runAsync()方法是 run()的配对方法，只不过它在管线运行启动后即刻返回。run()方法也可用以下方法来实现：

```
public PipelineResult run() {
  PipelineExecution execution = runAsync();
  execution.waitUntilDone();
  return execution.getResult();
}
```

有些时候，你可能希望直接使用 runAsync()方法。最典型的例子是当你需要在等

① 可以利用 DoFn 的 increment()方法来增加一些定制的 Crunch 计数器。

待管线执行的同时运行其他代码，并且希望利用 PipelineExecution 提供的一些方法(例如检查执行计划、发现执行状态或者在中途停止管线执行)时。

PipelineExecution 实现了 Future<PipelineResult>（来自 *java.util.concurrent*），它提供下述简单代码来执行后台工作：

```
PipelineExecution execution = pipeline.runAsync();
// meanwhile, do other things here
PipelineResult result = execution.get(); // blocks
```

2. 调试

当发生故障时，通过调用 Pipeline 的 enableDebug()方法可以获取 MapReduce 任务日志中的详细调试信息。

另一个有用的设置是配置属性 crunch.log.job.progress。假如把这个属性设置为 true，那么 MapReduce 作业处理过程的每个阶段都会被记录到控制台：

```
pipeline.getConfiguration().setBoolean("crunch.log.job.progress", true);
```

18.3.2 停止管线

有时候，你可能需要在管线完成之前停止它。这也许是因为管线已经启动后，你才突然意识到代码中存在一个编程错误，所以希望停止管线的运行，等问题解决之后再重新启动。

如果管线是通过调用阻塞进程 run()或 done()来完成的，那么使用标准的 Java 线程中断机制可以让 run()或 done()返回。但是，在集群上运行的所有作业仍将继续运行，因为它们不会被 Crunch 杀死。

事实上，为了正确停止管线，需要异步启动该管线，以保留对 PipelineExecution 对象的引用：

```
PipelineExecution execution = pipeline.runAsync();
```

在这种情况下，停止管线及其作业就只需要调用 PipelineExecution 上的 kill()方法，并等待管线完成：

```
execution.kill();
execution.waitUntilDone();
```

此时，PipelineExecution 的状态是 PipelineExecution.Status. KILLED，且管线在此之前运行在这个集群上的所有作业都将被杀死。此模式的一种有效的应用方式是将其应用于 java 虚拟机的关机 hook 中，于是在使用组合键 Ctrl+C 来关闭 java 应用程序时就可以安全地终止当前正在运行的管线。

PipelineExecution 实现的是 Future<PipelineResult>，因此，调用 kill()可以获得与调用 cancel(true)相同的效果。

18.3.3 查看 Crunch 计划

在某些时候，查看优化的执行计划是非常有必要的，或者至少是启发性的。下面这段代码给出了如何获得一个以字符串形式表示的管线操作图的 DOT 文件，并将其写入一个文件(使用 Guava 的 Files 类)。这段代码的重点就在于它能够访问正在异步运行的管线所返回的 PipelineExecution：

```
PipelineExecution execution = pipeline.runAsync();
String dot = execution.getPlanDotFile();
Files.write(dot, new File("pipeline.dot"), Charsets.UTF_8);
execution.waitUntilDone();
pipeline.done();
```

为了便于查看，需要使用 dot 命令行工具把 DOT 文件转换成图形格式，如 PNG。例如，调用下面的命令可以把当前目录下的所有 DOT 文件都转换为 PNG 格式，因此 *pipeline.dot* 被转换为一个名为 *pipeline.dot.png* 的文件：

```
% dot -Tpng -O *.dot
```

有一个技巧可以在不具备 PipelineExecution 对象时获取 DOT 文件，比如同步或隐式地运行管线时(参见 18.2.5 节)。Crunch 可以把 DOT 文件的表示存储在作业配置中，以便在管线执行完成后检索：

```
PipelineResult result = pipeline.done();
String dot = pipeline.getConfiguration().get("crunch.planner.dotfile");
Files.write(dot, new File("pipeline.dot"), Charsets.UTF_8);
```

下面来看一个重要管线的计划，它用于为存储在 inputPath 目录下的文本文件计算词频统计直方图(参见范例 18-3)。这个管线若作为产品，可以变得很长，包含几十个 MapReduce 作业，但是下面这段代码虽然不长，却说明了 Crunch 计划的一些特点。

范例 18-3. 一个用于计算词频统计直方图的 Crunch 管线

```
PCollection<String> lines = pipeline.readTextFile(inputPath);
PCollection<String> lower = lines.parallelDo("lower", new ToLowerFn(), strings());
PTable<String, Long> counts = lower.count();
PTable<Long, String> inverseCounts = counts.parallelDo("inverse",
    new InversePairFn<String, Long>(), tableOf(longs(), strings()));
PTable<Long, Integer> hist = inverseCounts
    .groupByKey()
    .mapValues("count values", new CountValuesFn<String>(), ints());
hist.write(To.textFile(outputPath), Target.WriteMode.OVERWRITE);
pipeline.done();
```

这个管线计划的框图如图 18-1 所示。

图 18-1. 用于计算字数的直方图的 Crunch 管线的计划图

源和目标以文件夹图标的形式在图中呈现。图的顶部显示了输入源，底部显示了输出目标。我们可以看到，图中共有两个 MapReduce 作业(分别被标记为 *Crunch Job 1* 和 *Crunch Job 2*)，其中一个作业的输出被写入到由 Crunch 生成的临时顺序文件中，而另一个作业则把该文件作为输入读取。管线执行完毕后，调用 `clean()` 删除临时文件。

Crunch Job 2(实际上，它才是先运行的作业)由 map 阶段和 reduce 阶段组成，如图中标注框所示。每个 map 和 reduce 都被分解为更小的操作，它们在图中所标记的名称对应于代码中 Crunch 基本操作的名称。例如，map 阶段的第一次 `parallelDo()`操作被标记为 `lower`，它的功能就是小写字母化 `PCollection` 中的字符串。

为管线所使用的 `PCollection` 重载方法以及相关类命名的时候，应当取一些对操作来说有意义的名称。这样可以增强计划框图的可读性。

在小写字母化操作之后，程序接下来所做的变换是通过内置的 `count()`方法产生一个 `PTable`，其中包含每个唯一字符串出现的次数。`count()`方法实际上执行了三个 Crunch 基本操作：一个是称为 *Aggregate.count* 的 `parallelDo()`操作；另一个是在图中标记为 *GBK* 的 `groupByKey()`操作；还有一个是被标记为 *combine* 的 `combineValues()`操作。

每个 *GBK* 操作都将作为 MapReduce 的混洗(shuffle)步骤实现，而 `groupByKey()` 和 `combineValues()` 操作则在 reduce 阶段运行。*Aggregate.count* 的 `parallelDo()`操作在 map 阶段运行，但是请注意，它与 `lower` 操作处于同一个 map 中，因为 Crunch planner 试图最小化管线需要运行的 MapReduce 作业数量。同样，*inverse* `parallelDo()`操作属于前一个 reduce 阶段。[①]

最后的变换操作是对 `PTable` 进行逆向计数，以找出每个计数出现的频率。例如，如果字符串 `apple` 和 `orange` 都出现了三次，那么计数 3 的频率为 2。这种变换是另一种形式的 *GBK* 操作，因此，必须使用一个新的 MapReduce 作业(Crunch Job 1)，此作业中的第二个操作是被命名为 *count values* 的 `mapValues()`

① 这种优化称为 `parallelDo` *fusion*，要想了解详细情况，请参阅本章开头引用的 FlumeJava 文献，网址为 *http://bit.ly/data-parallel_pipelines*，其中还包括一些 Crunch 使用的其他优化方法。需要注意的是，`parallelDo` fusion 允许在不损失任何效率的条件下，将管线操作分解为更小的、逻辑上独立的函数，因为 Crunch 要把它们融合为尽可能少的 MapReduce 阶段。

操作。mapValues()操作是一种简单的 parallelDo()操作，因此可以在 reduce 中运行。

请注意，图中省略了 Crunch Job 1 的 map 阶段，因为在这个阶段没有运行任何 Crunch 基本操作。

18.3.4 迭代算法

PObjects 的一种常见用法是检查迭代算法的收敛性。分布式迭代算法的一个典型的例子是 PageRank 算法，它为一组互相链接的网页(如 World Wide Web)排名打分，以体现每个网页的相对重要性。[①]在用 Crunch 实现的 PageRank 中，控制流如下所示：

```
PTable<String, PageRankData> scores = readUrls(pipeline, urlInput);
Float delta = 1.0f;
while (delta > 0.01) {
  scores = pageRank(scores, 0.5f);
  PObject<Float> pDelta = computeDelta(scores);
  delta = pDelta.getValue();
}
```

即使我们不了解 PageRank 算法本身的操作细节，也可以理解作为高级程序的 Crunch 是如何运行的。

该程序的输入是一个文本文件，其中每一行文本包含了两个 URL：一个是网页的 URL，另一个是该网页的出站超链接。例如，下面这个文件的含义是指网页 A 有超链接到达网页 B 和网页 C，并且网页 B 有超链接到达网页 D：

```
www.A.com www.B.com
www.A.com www.C.com
www.B.com www.D.com
```

重新回到代码，第一行代码读取输入文本并为每个唯一的网页创建一个初始 PageRankData 对象。PageRankData 是一个简单的 Java 类，其字段包括得分字段、上次得分字段(用于检查收敛性)以及出站链接列表：

```
public static class PageRankData {
  public float score;
  public float lastScore;
  public List<String> urls;
```

① 有关详细信息，请参见维基百科，网址为 *http://en.wikipedia.org/wiki/PageRank*。

```
  // ... methods elided
}
```

这个算法的目标是计算每个网页的得分，并通过得分来表示网页的相对重要性。所有网页一开始时的得分相同，每个网页的初始得分都是 1，且上次得分为 0。使用 Crunch 的分组操作，按第一个字段(网页)对输入进行分组，然后把值(即出站链接)聚合到一张列表中，就可以创建一个出站链接列表。[①]

迭代操作通过使用常规的 Java while 循环语句实现。每次迭代循环都要调用 PageRank()方法来更新得分。PageRank()方法使用了一系列的 Crunch 操作以实现 PageRank 算法。如果前一组得分与新得分之间的增量(delta)小于一个足够小的量(如 0.01)，那么说明得分已收敛，算法结束。这个增量值是通过 computeDelta()方法计算得到的，它是一个 Crunch 的聚合函数，用于为集合中的所有网页找出得分的最大绝对差。

那么，管线在什么时候运行？答案是在每次调用 pDelta.getValue()时。在第一次循环中，由于 PCollection 都没有物化，因此为了计算增量，必须运行 readUrls()、pageRank()和 computeDelta()作业。而后续的迭代循环则只需要运行用于计算新得分(pageRank())和增量(computeDelta())的作业。

对于此管线而言，如果在调用 pageRank()后立即调用 scores.materialize().iterator()，那么 Crunch planner 优化执行计划的工作做得不错。它确保 scores 表被显式地物化，因此在下一次迭代循环时它就可被用于下一个执行计划。如果不调用 materialize()，这段程序也能产生相同的结果，但效率比较低，因为 planner 可能会选择物化不同的中间结果，那么在下一个迭代循环中，有些计算必须重新执行才能获得 scores 并传递给 pageRank()。

18.3.5 给管线设置检查点

在上一节中我们看到，先前任意一次运行中被物化的 PCollection，只要是在同一管线中，都能被 Crunch 重用。不过，假如你创建了一个新的管线实例，那么它不会自动分享来自其他管线的任何物化的 PCollection，哪怕输入源相同。这就使得管线的开发变得相当耗时，因为即使已经接近管线底部，只要某个计算有细微的修改，Crunch 也要从头开始运行这个新的管线。

解决的办法是在持久存储器(通常是 HDFS)上设置一个 PCollection 检查点，以

① 在 Crunch 综合测试包的一个名为 PageRankIT 的类中可以找到完整的源代码。

便 Crunch 可以从这个检查点启动新的管线。

回顾范例 18-3 中用于计算词频统计直方图的 Crunch 计划。我们看到，Crunch planner 把该管线划分为两个 MapReduce 作业。如果再次运行程序，Crunch 将重新运行这两个 MapReduce 作业并覆盖原来的输出，因为 `WriteMode` 被设置为 `OVERWRITE`。

相反，如果我们设置了检查点 `inverseCounts`，那么第二次运行只会启动一个 MapReduce 作业(即用于计算 hist 的作业，因为它完全在 `inverseCounts` 后才执行)。设置检查点实际上就是把一个 `PCollection` 写入目标，并将其 `WriteMode` 设置为 `CHECKPOINT`：

```
PCollection<String> lines = pipeline.readTextFile(inputPath);
PTable<String, Long> counts = lines.count();
PTable<Long, String> inverseCounts = counts.parallelDo(
    new InversePairFn<String, Long>(), tableOf(longs(), strings()));
inverseCounts.write(To.sequenceFile(checkpointPath),
    Target.WriteMode.CHECKPOINT);
PTable<Long, Integer> hist = inverseCounts
    .groupByKey()
    .mapValues(new CountValuesFn<String>(), ints());
hist.write(To.textFile(outputPath), Target.WriteMode.OVERWRITE);
pipeline.done();
```

Crunch 会比较输入文件的时间戳与这些检查点文件的时间戳，如果有任何输入文件的时间戳晚于检查点文件的时间戳，那么自动重新计算相关检查点，因而不存在使用了管线中的过时数据的风险。

由于检查点在两次管线运行之间被持久保存，因此它们不会被 Crunch 清除，所以一旦代码如愿以偿地产生了你所预期结果后，你应当删除这些检查点。

18.4 Crunch 库

Crunch 自带了一组功能强大的库函数，在 `org.apache.crunch.lib` 包中，表 18-1 对这些函数进行了简单的概括。

Crunch 的一个强项是如果它没有提供你所需的函数，那么很简单，你可以自己编写，通常只需要几行 Java 语句。前面的范例 18-2 给出了一个通用函数的例子，用于查找 `PTable` 中的唯一值。

`Aggregate` 类有 `length()`、`min()`、`max()`和 `count()`方法，而在 `PCollection`

上也有相对应的方法。类似地，Aggregate 类的 top()方法(及其派生的 bottom()、collectValues()方法，还有 Join 的 join()方法以及 Cogroup 的 cogroup()方法，对于 PTable 来说都是完全一样的。

表 18-1. Crunch 库

类	方法	描述
Aggregate	length()	返回封装在 PObject 中的 PCollection 的元素数目
	min()	返回封装在 PObject 中的 PCollection 的最小值元素
	max()	返回封装在 PObject 中的 PCollection 的最大值元素
	count()	返回输入 PCollection 中的元素(唯一的)与其计数的映射表
	top()	返回由 PTable 的前 *N* 位或后 *N* 位键-值对构成的表，按值排序
	collectValues()	按表中的键分组，并把值聚合到 Java Collection 中，返回 PTable<K, Collection<V>>
Cartesian	cross()	计算两个 PCollection 或 PTable 的向量积
Channels	split()	把 PCollection<Pair<T, U>> 拆分为 Pair<PCollection<T>, Collection<U>>
Cogroup	cogroup()	按键分组，把元素聚合为两个或多个 PTable
Distinct	distinct()	创建一个删除了重复元素的新 PCollection 或 PTable
Join	join()	按键对两个 PTable 执行内连接。另外还有左连接、右连接和全连接方式
Mapred	map()	对 PTable<K1, V1>运行(旧的 API 的)mapper，以产生 PTable<K2, V2>
	reduce()	对 PGroupedTable<K1, V1> 运行(旧的 API 的)reducer，以产生 PTable<K2, V2>
Mapreduce	map(), reduce()	类似于 Mapred，只不过使用新的 MapReduce API
PTables	asPTable()	把 PCollection<Pair<K, V>>变换为 PTable<K, V>
	keys()	返回一个 PCollection，其中包含 PTable 的键
	values()	返回一个 PCollection，其中包含 PTable 的值
	mapKeys()	对 PTable 的所有键应用某个映射函数，并保持值不变
	mapValues()	对 PTable 或 PGroupedTable 的所有值应用某个映射函数，并保持键不变
Sample	sample()	以指定的概率，每个元素独立选择创建一个 PCollection 样本
	reservoirSample()	创建一个指定大小的 PCollection 样本，其中每个元素出现的概率相同

续表

类	方法	描述
Secondary Sort	sortAndApply()	按照先 K 后 V1 的顺序对 PTable<K, Pair<V1, V2>>进行排序，然后应用函数以给出一个输出 PCollection 或 PTable
Set	difference()	返回包含两个 PCollection 的差集的 PCollection
	intersection()	返回包含两个 PCollection 的交集的 PCollection
	comm()	返回一个三元组 PCollection，它对两个 PCollection 中的元素进行分类，以判断每个元素是仅出现在第一个集合中，还是仅出现在第二个集合中，或者是在两个集合中都有。(类似于 UNIX 的 comm 命令)
Shard	shard()	创建一个 PCollection，其中包含的元素与输入 PCollection 完全相同，只不过通过指定数量的文件进行分区(共享)
Sort	sort()	按照元素的自然顺序，对一个 PCollection 执行升序(默认)或降序的完全排序。另外还有一些方法可以按键对 PTable 排序，或者以指定的顺序按照列的子集来对 Pair 或元组的集合进行排序

范例 18-4 中的代码使用了一些聚合方法。

范例 18-4. PCollection 和 PTable 中的聚合方法的使用

```
PCollection<String> a = MemPipeline.typedCollectionOf(strings(),
    "cherry", "apple", "banana", "banana");

assertEquals((Long) 4L, a.length().getValue());
assertEquals("apple", a.min().getValue());
assertEquals("cherry", a.max().getValue());

PTable<String, Long> b = a.count();
assertEquals("{(apple,1),(banana,2),(cherry,1)}", dump(b));

PTable<String, Long> c = b.top(1);
assertEquals("{(banana,2)}", dump(c));

PTable<String, Long> d = b.bottom(2);
assertEquals("{(apple,1),(cherry,1)}", dump(d));
```

18.5 延伸阅读

本章简单介绍了 Crunch。要想了解更多信息，请参阅 Crunch 用户指南，网址为 *http://crunch.apache.org/user-guide.html*。

第 19 章

关于 Spark

Apache Spark (*https://spark.apache.org/*)是用于大数据处理的集群计算框架。与本书所讨论的其他大多数数据处理框架不同，Spark 并没有以 MapReduce 作为执行引擎，而是使用了它自己的分布式运行环境在集群上执行工作。不过，正如我们将在本章中看到的，Spark 与 MapReduce 在 API 和运行环境方面有许多相似之处。Spark 与 Hadoop 紧密集成，它可以在 YARN 上运行，并支持 Hadoop 文件格式及其存储后端(如 HDFS)。

Spark 最突出的表现在于它能将作业与作业之间产生的大规模的工作数据集存储在内存中。这种能力使得 Spark 在性能上超过了等效的 MapReduce 工作流，通常可高出 1 个数量级，在某些情况下则有可能高出更多[①]，原因是 MapReduce 的数据集始终需要从磁盘上加载。从 Spark 处理模型中获益最大的两种应用类型分别为迭代算法(即对一个数据集重复应用某个函数，直至满足退出条件)和交互式分析(用户向数据集发出一系列专用的探索性查询)。

即使你不需要在内存中进行缓存，Spark 还是会因其出色的 DAG 引擎和用户体验而极具吸引力。与 MapReduce 不同，Spark 的 DAG 引擎可以处理任意操作流水线，并为用户将其转换为单个作业。

Spark 的用户体验也是首屈一指的，它拥有丰富的 API 集，可用于执行多种常见的

① 详情参见 Matei Zaharia(Databricks 平台首席科学家，大数据处理框架 Apache Spark 创始人)等人的文章，标题为“Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”，原文发布于 2012 年 NSDI’12 Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation，网址为 *http://bit.ly/resilient_dist_datasets*。

数据处理任务，譬如连接。在写本章时，Spark 提供了三种语言的 API：Scala、Java 和 Python。本章大多数示例使用的都是 Scala API，不过，要想把它们转换为其他语言也很容易。Spark 还为 Scala 和 Python 提供了 REPL(read—eval—print loop)交互模式，可以快速且轻松地浏览数据集。

实践证明 Spark 还是用于构建分析工具的出色平台。为此，Apache Spark 项目包括用于处理机器学习(MLlib)、图算法(GraphX)、流式计算(Spark Streaming)和 SQL 查询(Spark SQL)等模块。这些模块超出了本章的讨论范围，感兴趣的读者可以参考 Apache Spark 网站，网址为 *http://spark.apache.org/*。

19.1 安装 Spark

从下载页面(*https://spark.apache.org/downloads.html*)下载一个稳定版本的 Spark 二进制发行包(应当选择与正在使用的 Hadoop 发行版匹配的版本)，然后在合适的位置解压缩该文件包：

```
% tar xzf spark-x.y.z-bin-distro.tgz
```

为了方便起见，可以把 Spark 的二进制文件路径添加到你的路径中，如下所示：

```
% export SPARK_HOME=~/sw/spark-x.y.z-bin-distro
% export PATH=$PATH:$SPARK_HOME/bin
```

接下来就可以运行一个 Spark 示例了。

19.2 示例

我们通过 *spark-shell* 程序运行一个交互式会话来演示 Spark。*Spark-shell* 是添加了一些 Spark 功能的 Scala REPL 交互式解释器。通过以下命令启动该 shell 环境：

```
% spark-shell
Spark context available as sc.

scala>
```

我们可以从控制台的输出看到已经创建了一个名为 `sc` 的 Scala 变量，它被用于保存 `SparkContext` 实例。这是 Spark 的主要入口点。接下来就可以加载一个文本文件：

```
scala> val lines = sc.textFile("input/ncdc/micro-tab/sample.txt")
```

```
lines: org.apache.spark.rdd.RDD[String] = MappedRDD[1] at textFile at
<console>:12
```

lines 变量引用的是一个弹性分布式数据集(Resilient Distributed Dataset，简称 RDD)。RDD 是 Spark 最核心的概念，它是在集群中跨多个机器分区存储的一个只读的对象集合。在典型的 Spark 程序中，首先要加载一个或多个 RDD，它们作为输入通过一系列转换得到一组目标 RDD，然后对这些目标 RDD 执行一个动作，例如计算出结果或者写入持久存储器。“弹性分布式数据集”中的术语“弹性”指的是 Spark 可以通过重新安排计算来自动重建丢失的分区。

加载 RDD 或执行转换并不会立即触发任何数据处理的操作，只不过是创建了一个计算的计划。只有当对 RDD 执行某个动作(比如 foreach())时，才会触发真正的计算。

继续讨论示例程序。我们希望执行的第一个转换是把文本行拆分为字段：

```
scala> val records = lines.map(_.split("\t"))
records: org.apache.spark.rdd.RDD[Array[String]] = MappedRDD[2] at map at
<console>:14
```

通过 RDD 的 map()方法可以对 RDD 中的每个元素应用某个函数。此处，我们将每一行文本(即一个 String)拆分成一个 String 类型的 Scala 数组。

然后，应用过滤器来滤除所有不良记录：

```
scala> val filtered = records.filter(rec => (rec(1) != "9999"
  && rec(2).matches("[01459]")))
filtered: org.apache.spark.rdd.RDD[Array[String]] = FilteredRDD[3]
at filter at <console>:16
```

RDD 的 filter()方法的输入是一个过滤谓词，也就是一个返回布尔值的函数。在本例中，此函数用于检查记录是否缺失了温度(以 9999 来表示)或者读数质量不达标。

为了找出每年的最高温度，我们需要按年份字段分组，这样才能对每年的所有温度值进行处理。Spark 的 reduceByKey()方法提供了分组功能，但是它需要一个用 Scala Tuple2 来表示的键-值对 RDD。因此，我们首先需要利用另一个 map 把 RDD 转换为适当的表示形式：

```
scala> val tuples = filtered.map(rec => (rec(0).toInt, rec(1).toInt))
tuples: org.apache.spark.rdd.RDD[(Int, Int)] = MappedRDD[4] at map at
<console>:18
```

然后，我们就可以执行聚合操作。reduceByKey()方法的参数是一个函数，它以一对值作为输入，并将它们组合形成一个值。在本例中，我们使用了 Java 的 Math.max 函数：

```
scala> val maxTemps = tuples.reduceByKey((a, b) => Math.max(a, b))
maxTemps: org.apache.spark.rdd.RDD[(Int, Int)] = MapPartitionsRDD[7] at
reduceByKey at <console>:21
```

通过调用 foreach()方法，并传递 println()参数，把其中的每个元素打印到控制台，我们就可以看到 maxTemps 的内容，：

```
scala> maxTemps.foreach(println(_))
(1950,22)
(1949,111)
```

foreach()方法与标准 Scala 集合(例如 List)的等效方法一样，它对 RDD 的每个元素应用一个函数(这个函数会产生一些附带效果)。正是这个操作触发了 Spark 运行一个作业来计算 RDD 中的值，从而使这些值能够传递给 println()方法。

或者，我们也可以将 RDD 保存到文件系统，如下所示：

```
scala> maxTemps.saveAsTextFile("output")
```

它创建了一个包含分区文件的名为 *output* 的目录：

```
% cat output/part-*
(1950,22)
(1949,111)
```

saveAsTextFile()方法也会触发 Spark 作业的运行。这两种方法的主要区别在于 saveAsTextFile()方法没有返回任何值，只是计算得到一个 RDD，并将其分区写入 *output* 目录下的文件中。

19.2.1 Spark 应用、作业、阶段和任务

正如我们在上面的例子中看到的，Spark 像 MapReduce 一样也有**作业**(job)的概念，只不过 Spark 的作业比 MapReduce 的作业更通用，因为 Spark 作业是由任意的**多阶段**(stages)有向无环图(DAG)构成，其中每个阶段大致相当于 MapReduce 中的 map 阶段或 reduce 阶段。

这些阶段又被 Spark 运行环境分解为多个**任务**(task)，任务并行运行在分布于集群中的 RDD 分区上，就像 MapReduce 中的任务一样。

Spark 作业始终运行在应用(application)上下文(用 SparkContext 实例来表示)中，它提供了 RDD 分组以及共享变量。一个应用可以串行或并行地运行多个作业，并为这些作业提供访问由同一应用的先前作业所缓存的 RDD 的机制(参见 19.3.3 节)。像 *spark-shell* 会话这样的交互式 Spark 会话只是应用的一个实际的例子。

19.2.2 Scala 独立应用

在通过 Spark shell 对程序进行完善之后，你可能希望将其打包形成一个自包含的应用，以便将来能够再次运行。通过范例 19-1 的 Scala 程序，可以了解它的具体做法。

范例 19-1. 使用 Spark 找出最高温度的 Scala 应用程序

```
import org.apache.spark.SparkContext._
import org.apache.spark.{SparkConf, SparkContext}

object MaxTemperature {
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("Max Temperature")
    val sc = new SparkContext(conf)

    sc.textFile(args(0))
      .map(_.split("\t"))
      .filter(rec => (rec(1) != "9999" && rec(2).matches("[01459]")))
      .map(rec => (rec(0).toInt, rec(1).toInt))
      .reduceByKey((a, b) => Math.max(a, b))
      .saveAsTextFile(args(1))
  }
}
```

在运行一个独立程序时，由于提供 SparkContext 的 shell 环境已经不存在了，因此我们需要自己创建 `SparkContext`。通过 `SparkConf` 来创建一个新的实例，有了这个实例就可以把各种 Spark 属性传递给应用。在前面的程序中，我们只是设置了应用的名称。

另外还要做一些其他方面的小改动。首先，我们要使用命令行参数来指定输入和输出路径。其次，还要利用方法链来避免为每个 RDD 创建中间变量。这样做可以使程序变得更加紧凑，同时在必要时仍然可以通过 Scala IDE 来查看每次转换的类型信息。

不是 Spark 定义的所有转换都可用于 RDD 类自身。在本例中，reducebyKey()(这个方法只能作用于键-值对 RDD)实际上是在 PairRDDFunctions 类中定义的，不过，我们可以使用以下的导入操作来让 Scala 将 RDD[(Int, Int)]隐式转换为 PairRDDFunctions：

```
import org.apache.spark.SparkContext._
```

这个导入操作让 Spark 可以使用更多样化的隐式转换，因此它当然值得被包含在程序中。

这一次，我们用 *spark-submit* 命令来运行程序，在作为参数传递的应用 JAR 中包含了经过编译的 Scala 程序，后面紧随着程序的命令行参数(输入和输出路径)：

```
% spark-submit --class MaxTemperature --master local \
  spark-examples.jar input/ncdc/micro-tab/sample.txt output
% cat output/part-*
(1950,22)
(1949,111)
```

我们还指定了两个选项：--class 告诉 Spark 应用的类名；--master 指定作业应该在哪里运行。值 local 告诉 Spark 所有作业都在本地机器上的一个 JVM 中运行。19.6 节将会介绍在集群上运行的选项。下面让我们来看看 Spark 如何使用其他语言，先从 Java 开始。

19.2.3 Java 示例

Spark 是用 Scala 实现的，而 Scala 作为基于 JVM 的语言，与 Java 有着良好集成关系。用 Java 语言来写前面的示例非常简单，只不过会有点冗长，参见范例 19-2。[①]

范例 19-2. 用 Spark 找出最高气温

```
public class MaxTemperatureSpark {

  public static void main(String[] args) throws Exception {
  if (args.length != 2) {
    System.err.println("Usage: MaxTemperatureSpark <input path> <output path>");
    System.exit(-1);
  }

  SparkConf conf = new SparkConf();
  JavaSparkContext sc = new JavaSparkContext("local", "MaxTemperatureSpark",conf);
  JavaRDD<String> lines = sc.textFile(args[0]);
```

① 如果用 Java 8 lambda 表达式来写，这段 Java 版的程序可以会更紧凑。

```
JavaRDD<String[]> records = lines.map(new Function<String, String[]>() {
  @Override public String[] call(String s) {
    return s.split("\t");
  }
});
JavaRDD<String[]> filtered = records.filter(new Function<String[], Boolean>() {
  @Override public Boolean call(String[] rec) {
    return rec[1] != "9999" && rec[2].matches("[01459]");
  }
});
JavaPairRDD<Integer, Integer> tuples = filtered.mapToPair(
  new PairFunction<String[], Integer, Integer>() {
    @Override public Tuple2<Integer, Integer> call(String[] rec) {
      return new Tuple2<Integer, Integer>(
        Integer.parseInt(rec[0]), Integer.parseInt(rec[1]));
    }
  }
);
JavaPairRDD<Integer, Integer> maxTemps = tuples.reduceByKey(
  new Function2<Integer, Integer, Integer>() {
    @Override public Integer call(Integer i1, Integer i2) {
      return Math.max(i1, i2);
    }
  }
);
maxTemps.saveAsTextFile(args[1]);
}
}
```

在 Spark 的 Java API 中，RDD 以 `JavaRDD` 实例来表示，或者在遇到键-值对 RDD 这种特殊情况时，使用 `JavaPairRDD` 来表示。这两个类都实现了 `JavaRDDLike` 接口，大多数与 RDD 相关的方法都可以在这个接口中找到，例如，通过查看类文档。

前面这段程序的运行与 Scala 版程序的运行类似，只不过它的类名变成了 `MaxTemperatureSpark`。

19.2.4 Python 示例

Spark 通过一个名为 *PySpark* 的 API 来为 Python 提供语言支持。利用 Python 的 lambda 表达式，我们可以重写前面的示例程序。范例 19-3 所示的这段程序看上去非常接近于等效的 Scala 程序。

范例 19-3. 使用 PySpark 找出最高温度的 Python 应用程序

```
from pyspark import SparkContext
import re, sys
```

```
sc = SparkContext("local", "Max Temperature")
sc.textFile(sys.argv[1]) \
  .map(lambda s: s.split("\t")) \
  .filter(lambda rec: (rec[1] != "9999" and re.match("[01459]", rec[2]))) \
  .map(lambda rec: (int(rec[0]), int(rec[1]))) \
  .reduceByKey(max) \
  .saveAsTextFile(sys.argv[2])
```

请注意，对于 reduceByKey()转换，我们使用的是 Python 内置的 max 函数。

前面这段程序是用常规的 CPython 语言来写的，这一点很重要。Spark 通过复刻(fork)Python 子进程来运行用户的 Python 代码(在启动程序以及集群上运行用户任务的 *executor* 中)，并且使用套接字连接两个进程，从而使父进程能够传递将要由 Python 代码处理的 RDD 分区数据。

为了运行这段代码，我们需要指定的是 Python 文件，而不是应用 JAR：

```
% spark-submit --master local \
  ch19-spark/src/main/python/MaxTemperature.py \
  input/ncdc/micro-tab/sample.txt output
```

Spark 还可以使用 pyspark 命令，以交互模式用 Python 来运行。

19.3 弹性分布式数据集

弹性分布式数据集(RDD)是所有 Spark 程序的核心，因此本节将详细讨论它们的使用情况。

19.3.1 创建

RDD 的创建有三种方法：来自一个内存中的对象集合(也称为并行化一个集合)；使用外部存储器(例如 HDFS)中的数据集；对现有的 RDD 进行转换。第一种方法适用于对少量的输入数据进行并行的 CPU 密集型计算。例如，下面这段代码对数字 1 到 10 运行独立计算[①]：

```
val params = sc.parallelize(1 to 10)
val result = params.map(performExpensiveComputation)
```

performExpensiveComputation 函数对输入的值并行运行，其并行度由

① 这就像使用 MapReduce 中的 NLineInputFormat 来执行参数扫描一样，参见 8.2.2 节对 NLineInputFormat 的描述。

spark.default.parallelism 属性确定，默认值取决于 Spark 作业的地点。在本地运行时，默认值就是该机器的内核(core)数，而在集群上运行时，它是集群中所有 executor 节点的内核的总数。

如果把并行度作为第二个参数传给 parallelize()，可以覆盖特定计算的并行度：

```
sc.parallelize(1 to 10, 10)
```

创建 RDD 的第二种方法是创建一个对外部数据集的引用。我们已经知道如何为文本文件创建一个 String 对象的 RDD：

```
val text: RDD[String] = sc.textFile(inputPath)
```

其中的路径可以是任何 Hadoop 文件系统路径，例如本地文件系统或者 HDFS 上的文件。Spark 内部使用了旧的 MapReduce API 的 TextInputFormat 来读取文件(参见 8.2.2 节对 TextInputFormat 的描述)，这意味着它的文件分割行为与 Hadoop 中的一致，因此在使用 HDFS 的情况下，每个 HDFS 块对应于一个 Spark 分区。这个默认值可以通过传递第二个参数以请求特定的分割数量来更改：

```
sc.textFile(inputPath, 10)
```

另一种变体是通过返回一个字符串对 RDD 来把文本文件作为一个完整的文件对待(类似于 8.2.1 节的“将整个文件作为记录处理”)，其中第一个字符串是文件路径，第二个字符串是文件内容。由于每个文件都要被加载到内存中，因此这种方式只适合小文件：

```
val files: RDD[(String, String)] = sc.wholeTextFiles(inputPath)
```

Spark 也可以处理文本文件以外的其他格式。例如，以下代码可用于读取顺序文件：

```
sc.sequenceFile[IntWritable, Text](inputPath)
```

其中值得注意的是顺序文件键和值的 Writable 类型是如何指定的。对于普通的 Writable 类型，Spark 可以将它们映射为等效的 Java 类型，因此我们也可以使用以下形式，效果都是一样的：

```
sc.sequenceFile[Int, String](inputPath)
```

从任意 Hadoop InputFormat 格式创建 RDD 的方法有两种：对于需要路径输入的那些基于文件的格式可以使用 hadoopFile()，而对于不需要路径输入的格式(例如 HBase 的 TableInputFormat)则可以使用 hadoopRDD()。这些方法适用于旧

的 MapReduce API，而对于新的 MapReduce API 来说，需要相应地使用 newAPIHadoopFile()和 newAPIHadoopRDD()。下面这段代码是读取 Avro 数据文件的一个例子，它利用了包含 WeatherRecord 类的 Specific API：

```
val job = new Job()
AvroJob.setInputKeySchema(job, WeatherRecord.getClassSchema)
val data = sc.newAPIHadoopFile(inputPath,
    classOf[AvroKeyInputFormat[WeatherRecord]],
    classOf[AvroKey[WeatherRecord]], classOf[NullWritable],
    job.getConfiguration)
```

除了路径之外，newAPIHadoopFile()方法还需要输入 InputFormat 类型、键类型、值类型以及 Hadoop 的配置。在配置中包含了 Avro 模式，我们在代码的第二行使用 AvroJob 帮助类对它进行设置。

第三种方式是通过对现有 RDD 的转换来创建 RDD。下面我们将详细讨论转换。

19.3.2 转换和动作

Spark 为 RDD 提供了两大类操作：**转换**(transformation)和**动作**(action)。转换是从现有 RDD 生成新的 RDD，而动作则触发对 RDD 的计算并对计算结果执行某种操作，要么返回给用户，要么保存到外部存储器中。

动作的效果立竿见影，但转换不是，转换是惰性的，因为在对 RDD 执行一个动作之前都不会为该 RDD 的任何转换操作采取实际行动。例如，下面这段代码用于小写字母化文本文件中的文本行：

```
val text = sc.textFile(inputPath)
val lower: RDD[String] = text.map(_.toLowerCase())
lower.foreach(println(_))
```

map()方法是一种转换操作，它在 Spark 内部被表示为可以在稍后某个时刻对输入 RDD(文本)的每个元素调用一个函数(即 toLowerCase())。在调用 foreach()方法(这是一个动作)之前，该函数实际上并没有被调用。事实上，Spark 在结果即将被写入控制台之前，才会运行一个作业来读取输入文件并对其中的每一行文本调用 toLowerCase()。

要想判断一个操作是转换还是动作，我们可以观察其返回类型：如果返回的类型是 RDD，那么它是一个转换，否则就是一个动作。在翻阅 RDD 文档(在 org.apache.spark.rdd 包中)时了解这一点会很有用。通过这些文档可以找到对 RDD 执行的大多数操作。另外，在 PairRDDFunctions 类中可以找到用于键-值

对 RDD 的各种转换和动作。

在 Spark 库中包含了一组丰富的操作，包括映射、分组、聚合、重新分区、采样、连接 RDD 以及把 RDD 作为集合来处理的各种转换，同时还包括将 RDD 物化为集合；对 RDD 进行统计数据的计算；从一个 RDD 中采样固定数量的元素；以及将 RDD 保存到外部存储器等各种动作，具体情况请参阅相应的类文档。

Spark 中的 MapReduce

Spark 的 map()和 reduce()操作与 Hadoop MapReduce 中的同名函数并没有直接对应关系，尽管从命名上看好像是有。Hadoop MapReduce 中的 map 和 reduce 的一般形式为(参见第 8 章):

```
map: (K1, V1) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)
```

注意，这两个函数都可以返回多个输出对，它们以 list 来表示，而在 Spark(以及常规的 Scala)中，这种情况需要通过 flatMap()操作来实现。flatMap()与 map()类似，只是少了一层嵌套:

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

scala> l.map(a => List(a))
res0: List[List[Int]] = List(List(1), List(2), List(3))

scala> l.flatMap(a => List(a))
res1: List[Int] = List(1, 2, 3)
```

想要在 Spark 中模拟 Hadoop MapReduce 的一种简单的方法是使用两个 flatMap()操作，并且两者之间使用 groupByKey()和 sortByKey()分隔，它们分别执行的是 MapReduce 的混洗(shuffle)和排序(sort)操作:

```
val input: RDD[(K1, V1)] = ...
val mapOutput: RDD[(K2, V2)] = input.flatMap(mapFn)
val shuffled: RDD[(K2, Iterable[V2])] = mapOutput.groupByKey().sortByKey()
val output: RDD[(K3, V3)] = shuffled.flatMap(reduceFn)
```

在这里，键的类型 K2 需要继承 Scala 的 Ordering 类型才能满足 sortByKey()的要求。这个例子可能有助于了解 MapReduce 和 Spark 之间的关系，但不应盲目应用。其中一个原因是 sortByKey()执行的是全排序，所以它的语义与 Hadoop MapReduce 中的略有不同。通过使用 repartitionAndSortWithinPartitions()来执行部分排序可以避免此问

题，但即使这样做，效率还是不高，因为 Spark 使用了两个 shuffle(一个用于 groupByKey()，一个用于排序)。

与其试图模仿 MapReduce，还不如使用你实际需要的操作。例如，如果不需要按键排序，那么可以省略 sortByKey()的调用(这在常规的 Hadoop MapReduce 中是不可能的)。

类似地，groupByKey()在大多数情况下会显得过于笼统。通常，我们只是想利用 shuffle 来聚合相应的值，因此使用 reduceByKey()、foldByKey()或者 aggregateByKey()(在下一节介绍)要比使用 groupByKey()效率更高，因为它们也可以作为 combiner 在 map 任务中运行。最后，flatMap()也不是必须的，如果总是只有一个返回值，那么首选使用 map()，而如果有零个或一个返回值，则应当使用 filter()。

聚合转换

按键来为键-值对 RDD 进行聚合操作的三个主要转换函数分别是 reduceByKey()、foldByKey()和 aggregateByKey()。它们的工作方式稍有不同，但都用于聚合给定键的值，并为每个键产生一个值。(对应的等效动作分别是 reduce()、fold()和 aggregate()，它们以类似的操作方式为整个 RDD 产生一个值。)

其中最简单的是 reduceByKey()，它为键-值对中的值重复应用一个二进制函数，直至产生一个结果值。例如：

```
val pairs: RDD[(String, Int)] =
    sc.parallelize(Array(("a", 3), ("a", 1), ("b", 7), ("a", 5)))
val sums: RDD[(String, Int)] = pairs.reduceByKey(_+_)
assert(sums.collect().toSet === Set(("a", 9), ("b", 7)))
```

键 a 的值使用加法函数(_+_)来聚合，即$(3 + 1)+ 5 = 9$，而键 b 只有一个值，因此不需要聚合。由于上述操作一般来说是分布式的，它们根据 RDD 的不同分区在不同的任务中执行，因此这些函数应当是可交换和可结合的。换句话说就是操作的先后次序及两两之间的结合关系并不重要。在这种情况下，上述聚合操作可以是 $5 +(3 + 1)$，也可以是 $3 +(1 + 5)$，两者返回的结果相同。

assert 语句中使用的三元等于运算符(===)来自于 ScalaTest，比起使用常规的==运算符，它可以提供更多有意义的失败消息。

下面这段代码给出了如何利用 foldByKey()来执行相同的操作：

```
val sums: RDD[(String, Int)] = pairs.foldByKey(0)(_+_)
assert(sums.collect().toSet === Set(("a", 9), ("b", 7)))
```

请注意，这一次我们必须要提供一个零值(zero value)，在做整数加法时它就是 0，但是对于其他的类型及操作来说，它有可能是不同的。此处，a 的值聚合为$((0 + 3) + 1) + 5) = 9$(也可以是其他顺序，但是，第一个操作必定是加 0)。对于 b 则是 $0 + 7 = 7$。

使用 foldByKey()与使用 reduceByKey()一样，没有谁比谁更强大，尤其是两者都不能改变聚合结果值的类型。要想达到这个目的，我们需要用 aggregateByKey()。例如，我们可以将一些整数值聚合形成一个集合：

```
val sets: RDD[(String, HashSet[Int])] =
    pairs.aggregateByKey(new HashSet[Int])(_+=_, _++=_)
assert(sets.collect().toSet === Set(("a", Set(1, 3, 5)), ("b", Set(7))))
```

对于集合加法运算来说，零值就是空集，因此我们使用 new HashSet[Int]来创建一个新的可变集合。必须为 aggregateByKey()提供两个输入的函数。第一个函数负责把 Int 合并到 HashSet[Int]中。此处，我们使用_+=_函数来把整数添加到集合中(如果使用_+_，则返回一个新集合并保留第一个集合不变)。

第二个函数负责合并两个 HashSet[Int]中的值(此操作发生在 map 任务的 combiner 运行之后，并且在 reduce 任务的两个分区进行聚合时)。此处我们用_++=_函数把第二个集合中的所有元素添加到第一个集合中。

对于键 a，操作顺序可能如下所示：

$$((\emptyset + 3) + 1) + 5) = (1, 3, 5)$$

或者，如果 Spark 使用了一个 combiner，则如下所示：

$$(\emptyset + 3) + 1) ++ (\emptyset + 5) = (1, 3) ++ (5) = (1, 3, 5)$$

转换后的 RDD 可以持久化存储在内存中，以提高其后的操作效率，关于这个问题我们将在下面讨论。

19.3.3 持久化

回顾 19.2 节介绍的示例，我们可以用下述命令把年份/温度对构成的中间数据集缓存到内存中：

```
scala> tuples.cache()
res1: tuples.type = MappedRDD[4] at map at <console>:18
```

调用 cache()并不会立即缓存 RDD，相反，它用一个标志来对该 RDD 进行标记，以指示该 RDD 应当在 Spark 作业运行时被缓存。因此，让我们先强制运行一个作业：

```
scala> tuples.reduceByKey((a, b) => Math.max(a, b)).foreach(println(_))
INFO BlockManagerInfo: Added rdd_4_0 in memory on 192.168.1.90:64640
INFO BlockManagerInfo: Added rdd_4_1 in memory on 192.168.1.90:64640
(1950,22)
(1949,111)
```

来自 BlockManagerInfo 的日志行表明 RDD 分区已作为作业运行的一部分保存在内存中。日志显示 RDD 的编号为 4(这个编号在调用 cache()方法后显示在控制台上)，它有两个分区，分别被标记为 0 和 1。如果我们对缓存的数据集运行另一个作业，将会看到从内存中加载该 RDD。这次我们要计算的是最低温度：

```
scala> tuples.reduceByKey((a, b) => Math.min(a, b)).foreach(println(_))
INFO BlockManager: Found block rdd_4_0 locally
INFO BlockManager: Found block rdd_4_1 locally
(1949,78)
(1950,-11)
```

这只是一个简单的小型数据集示例，对于大规模作业来说，这样做能够节省可观的时间。相比较而言，MapReduce 在执行另一个计算时必须从磁盘中重新加载输入数据集，即使它可以使用中间数据集作为输入(例如，经过了对无效行和不必要字段清理的数据集)，也始终无法摆脱必须从磁盘加载的事实，这必然会影响其执行速度。Spark 可以在跨集群的内存中缓存数据集，这也就意味着对数据集所做的任何计算都会非常快。

事实证明，这对于数据的交互式探索查询(interactive exploration)非常有用。它天生适用于某些特定风格的算法，例如迭代算法。在迭代算法中，上一次迭代计算的结果可以被缓存在内存中，以用作下一次迭代的输入。这些算法也可以用 MapReduce 来表示，但是每次迭代都要作为单个 MapReduce 作业来运行，因此每次迭代的结果必须写入磁盘，然后在下一次迭代时从磁盘中读回。

被缓存的 RDD 只能由同一应用的作业来读取。如果要在应用之间共享数据集，则必须在第一个应用中使用 saveAs*()方法(譬如 saveAsTextFile()、saveAsHadoopFile()等等)将其写入外部存储器，然后在第二个应用中使用 SparkContext 的相应方法(如 textFile()、hadoopFile()等等)进行加载。同理，当应用终止时，它缓存的所有 RDD 都将被销毁，除非这些 RDD 已被显式保存，否则无法再次访问。

持久化级别

调用 cache()将会在 executor 的内存中持久化保存 RDD 的每个分区。如果 executor 没有足够的内存来存储 RDD 分区，计算并不会失败，只不过是根据需要重新计算分区。对于包含大量转换操作的复杂程序来说，重新计算的代价可能太高，因此 Spark 提供了不同级别的持久化行为，我们可以通过调用 persist()并指定 StorageLevel 参数来做出选择。

默认的持久化级别是 MEMORY_ONLY，它使用对象在内存中的常规表示方法。另一种更紧凑的表示方法是通过把分区中的元素序列化为字节数组来实现的。这一级别称为 MEMORY_ONLY_SER。与 MEMORY_ONLY 相比，MEMORY_ONLY_SER 多了一份 CPU 开销，但是，如果它生成的序列化 RDD 分区的大小适合被保存到内存中，而常规的表示方法却无法做到这一点时，这份额外开销就是值得的。MEMORY_ONLY_SER 还能减少垃圾回收的压力，因为每个 RDD 被存储为一个字节数组，而不是大量的对象。

通过检查 driver 日志文件中的 BlockManager 消息，可以了解 RDD 分区的大小是否适合被保存到内存中。此外，每个 driver 的 SparkContext 都运行了一个 HTTP 服务器(端口 4040)，可提供运行环境以及正在运行的作业的相关信息，包括缓存的 RDD 分区的信息。

默认情况下，RDD 分区的序列化使用的是常规的 Java 序列化方法，但是，无论从大小还是速度来看，使用 Kryo 序列化方法(在下一小节介绍)通常都是更好的选择。通过压缩序列化分区可以进一步节省空间(再次以牺牲 CPU 为代价)，其做法是把 spark.rdd.compress 属性设置为 true，并且可选地设置 spark.io.compression.codec 属性。

如果重新计算数据集的代价太过高昂，那么可以使用 MEMORY_AND_DISK(如果数据集的大小不适合保存到内存中，就将其溢出到磁盘)，或 MEMORY_AND_DISK_SER (如果序列化数据集的大小不适合保存到内存中，就将其溢出到磁盘)。

另外还有一些更高级的以及实验性的持久化级别，它们可用于在集群中的多个节点上复制分区，或者也可以使用堆外内存，详情请参阅 Spark 文档。

19.3.4 序列化

在使用 Spark 时，要从两个方面来考虑序列化：数据序列化和函数序列化(或称为闭包函数)。

1. 数据

让我们先来了解一下数据序列化。默认情况下，Spark 在通过网络将数据从一个 executor 发送到另一个 executor 时，或者以序列化的形式缓存(持久化)数据时(参见 19.3.3 节)，所使用的都是 Java 序列化机制。Java 序列化机制为程序员所熟知(你必须确保所使用的类实现了 `java.io.Serializable` 或 `java.io.Externalizable` 接口)，但从性能或大小来看，这种做法效率并不高。

使用 Kryo 序列化机制(*https://github.com/EsotericSoftware/kryo*)对于大多数 Spark 程序都是一个更好的选择。Kryo 是一个高效的通用 Java 序列化库。要想使用 Kryo 序列化机制，需要在你的驱动器程序的 `SparkConf` 中设置 `spark.serializer` 属性，如下所示：

```
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
```

Kryo 不要求被序列化的类实现某个特定的接口(例如 `java.io.Serializable`)。因此，旧的纯 Java 对象也可以在 RDD 中使用，除了需要启用 Kryo 序列化之外，没有什么更多的工作需要做。话虽如此，但是如果使用之前先在 Kryo 中对这些类进行注册，那么可以提高其性能。这是因为 Kryo 需要写被序列化对象的类的引用(每个被写的对象都需要写一个引用)，如果已经注册了类，那么该引用就只是一个整数标识符，否则就是完整的类名。上述原则仅适用于你自己的类，Spark 已经代你注册了 Scala 类和其他一些框架类(比如 Avro `Generic` 类或 `Thrift` 类)。

在 Kryo 中注册类很简单，先创建一个 `KryoRegistrator` 子类，然后重写 `registerClasses()`方法：

```
class CustomKryoRegistrator extends KryoRegistrator {
  override def registerClasses(kryo: Kryo) {
    kryo.register(classOf[WeatherRecord])
  }
}
```

最后，在 driver 程序中将 spark.kryo.registrator 属性设置为你的 KryoRegistrator 实现的完全限定类名：

```
conf.set("spark.kryo.registrator", "CustomKryoRegistrator")
```

2. 函数

通常函数的序列化会“谨守本份”：Scala 中的函数都可以通过标准的 Java 序列化机制来序列化，这也是 Spark 用于向远程 executor 节点发送函数的手段。但是对 Spark 来说，即使在本地模式下运行，也需要序列化函数，因此假若你无意中引入了一个不可序列化的函数(例如，从不可序列化的类的方法转换得到的函数)，那么你应该在开发初期就会发现它。

19.4 共享变量

Spark 程序经常需要访问一些不属于 RDD 的数据。例如，下面这段程序在 map() 操作中用到了一张查找表(lookup table)：

```
val lookup = Map(1 -> "a", 2 -> "e", 3 -> "i", 4 -> "o", 5 -> "u")
val result = sc.parallelize(Array(2, 1, 3)).map(lookup(_))
assert(result.collect().toSet === Set("a", "e", "i"))
```

虽然这段程序可以正常工作(变量 lookup 作为闭包函数的一部分被序列化后传递给 map()，但是使用广播变量可以更高效地完成相同的工作。

19.4.1 广播变量

广播变量(broadcast variable)在经过序列化后被发送给各个 executor，然后缓存在那里，以便后期任务可以在需要时访问它。它与常规变量不同，常规变量是作为闭包函数的一部分被序列化的，因此它们在每个任务中都要通过网络被传输一次。广播变量的作用类似于 MapReduce 中的分布式缓存(参见 9.4.2 节)，两者的不同之处在于 Spark 将数据保存在内存中，只有在内存耗尽时才会溢出到磁盘上。

我们通过向 SparkContext 的 broadcast()方法传递即将被广播的变量来创建一个广播变量。它返回 Broadcast[T]，即对类型为 T 的变量的一个封装：

```
val lookup: Broadcast[Map[Int, String]] =
    sc.broadcast(Map(1 -> "a", 2 -> "e", 3 -> "i", 4 -> "o", 5 -> "u"))
val result = sc.parallelize(Array(2, 1, 3)).map(lookup.value(_))
assert(result.collect().toSet === Set("a", "e", "i"))
```

请注意，要想在 RDD 的 map()操作中访问这些变量，需要对它们调用 value。

顾名思义，广播变量是单向传播的，即从 driver 到任务，因此一个广播变量是没有办法更新的，也不可能将更新传回 driver。要想做到这一点，我们需要累加器。

19.4.2 累加器

累加器(accumulator)是在任务中只能对它做加法的共享变量，类似于 MapReduce 中的计数器(参见 9.1 节)。当作业完成后，driver 程序可以检索累加器的最终值。下面这个例子使用累加器来对一个整数 RDD 中的元素个数进行计数，同时使用 reduce()动作对 RDD 中的值求和：

```
val count: Accumulator[Int] = sc.accumulator(0)
val result = sc.parallelize(Array(1, 2, 3))
  .map(i => { count += 1; i })
  .reduce((x, y) => x + y)
assert(count.value === 3)
assert(result === 6)
```

代码的第一行使用了 SparkContext 的 accumulator()方法来创建一个累加器变量 count。map()操作是一个恒等函数，其附带效果是使 count 递增。当 Spark 作业的结果被计算出来后，可以通过对累加器调用 value 来访问它的值。

在这个例子中，累加器的类型为 Int，事实上它可以是任意的数值类型。Spark 还可以让累加器的结果类型与加数类型不同(参见 SparkContext 的 accumulable()方法)，也能够累积可变集合中的值(通过 accumulableCollection()方法)。

19.5 剖析 Spark 作业运行机制

下面来看看当我们运行 Spark 作业时会发生些什么。在最高层，它有两个独立的实体：*driver* 和 *executor*。driver 负责托管应用(SparkContext)并为作业调度任务。executor 专属于应用，它在应用运行期间运行，并执行该应用的任务。通常，driver 作为一个不由集群管理器(cluster manager)管理的客户端来运行，而 executor 则运行在集群的计算机上。不过，也并不总是这样(参见 19.6 节)。在下面的讨论中，我们假设应用的 executor 已经运行。

19.5.1 作业提交

图 19-1 描绘了 Spark 运行作业的过程。当对 RDD 执行一个动作(比如 `count()`)时，会自动提交一个 Spark 作业。从内部看，它导致对 `SparkContext` 调用 `runJob()`(图 19-1 中的步骤 1)，然后将调用传递给作为 driver 的一部分运行的调度程序(步骤 2)。调度程序由两部分组成：DAG 调度程序和任务调度程序。DAG 调度程序把作业分解为若干阶段，并由这些阶段构成一个 DAG。任务调度程序则负责把每个阶段中的任务提交到集群。

图 19-1. Spark 如何运行作业

接下来，让我们看看 DAG 调度程序如何构建一个 DAG。

19.5.2 DAG 的构建

要想了解一个作业如何被划分为阶段，首先需要了解在阶段中运行的任务的类

型。有两种类型的任务：shuffle map 任务和 result 任务。从任务类型的名称可以看出 Spark 会怎样处理任务的输出。

shuffle map 任务　顾名思义，shuffle map 任务就像是 MapReduce 中 shuffle 的 map 端部分。每个 shuffle map 任务在一个 RDD 分区上运行计算，并根据分区函数把输出写入一组新的分区中，以允许在后面的阶段中取用(后面的阶段可能由 shuffle map 任务组成，也可能由 result 任务组成)。，shuffle map 任务运行在除最终阶段之外的其他所有阶段中。

result 任务　result 任务运行在最终阶段，并将结果返回给用户程序(例如 `count()`的计算结果)。每个 result 任务在它自己的 RDD 分区上运行计算，然后把结果发送回 driver，再由 driver 将每个分区的计算结果汇集成最终结果(比如，在 `saveAsTextFile()`操作的情况下，结果有可能是一个 `Unit`)。

最简单的 Spark 作业不需要使用 shuffle，因此它只有一个由 result 任务构成阶段，这就像是 MapReduce 中的仅有 map 的作业一样。

比较复杂的作业要涉及到分组操作，并且要求一个或多个 shuffle 阶段。例如，下面这个作业用于为存储在 *inputPath* 目录下的文本文件计算词频统计分布图(每行文本只有一个单词)：

```
val hist: Map[Int, Long] = sc.textFile(inputPath)
  .map(word => (word.toLowerCase(), 1))
  .reduceByKey((a, b) => a + b)
  .map(_.swap)
  .countByKey()
```

前两个转换是 `map()`和 `reduceByKey()`，它们用于计算每个单词出现的频率。第三个转换是 `map()`，它交换每个键-值对中的键和值，从而得到(*count*, *word*)对。最后是 `countByKey()`动作，它返回的是每个计数对应的单词量(即词频分布)。

由于 `reduceByKey()`必须要有 shuffle 阶段，因此 Spark 的 DAG 调度程序将此作业分为两个阶段。[①]结果得到的 DAG 如图 19-2 所示。

通常，每个阶段的 RDD 都要在 DAG 中显示，并且在 DAG 图中给出了这些 RDD 的类型以及创建它的操作。例如，`RDD[String]`是由 `textFile()`创建的。为了简化，图中省略了 Spark 内部产生的一些中间 RDD。例如，由 `textFile()`返回的

① 注意，`countByKey()`在本地 driver 上执行最后的聚合操作，而不是使用第二个 shuffle 阶段。这与示例 18-3 中等效的 Crunch 程序不同，后者使用了第二个 MapReduce 作业用于计数。

RDD 实际上是一个 MappedRDD[String]，而其父对象是 HadoopRDD [LongWritable, Text]。

图 19-2. 用于计算词频统计分布图的 Spark 作业中的各个阶段以及 RDD

注意，reduceByKey()转换跨越了两个阶段，这是因为它是使用 shuffle 实现的，并且就像 MapReduce 一样，reduce 函数一边在 map 端作为 combiner 运行(阶段 1)，一边又在 reduce 端又作为 reducer 运行(阶段 2)。它与 MapReduce 相似的另一个地方是，Spark 的 shuffle 实现将其输出写入本地磁盘上的分区文件中(即使对内存中的 RDD 也一样)，并且这些文件将由下一个阶段的 RDD 读取。[①]

如果 RDD 已经被同一应用(SparkContext)中先前的作业持久化保存，那么 DAG 调度程序将会省掉一些工作，不会再创建一些阶段来重新计算它(或者它的父

① 可以通过配置(*http://bit.ly/shuffle_behavior*)来调整 shuffle 的性能。同时还要注意，Spark 使用了自己定义的 shuffle 实现，并没有与 MapReduce 的 shuffle 实现共享任何代码。

RDD)。

DAG 调度程序负责将一个阶段分解为若干任务以提交给任务调度程序。在本例的第一个阶段中，输入文件的每个分区运行一个 shuffle map 任务。`reduceByKey()`操作的并行度可以通过传递第二个参数来显式设置。如果没有设置并行度，则根据父 RDD 来确定，在这种情况下就是输入数据的分区数。

DAG 调度程序会为每个任务赋予一个位置偏好(placement preference)，以允许任务调度程序充分利用数据本地化(data locality)。例如，对于存储在 HDFS 上的输入 RDD 分区来说，它的任务的位置偏好就是托管了这些分区的数据块的 datanode(称为 *node local*)，而对于在内存中缓存的 RDD 分区，其任务的位置偏好则是那些保存 RDD 分区的 executor(称为 *process local*)。

回到图 19-1，一旦 DAG 调度程序已构建一个完整的多阶段 DAG，它就将每个阶段的任务集合提交给任务调度程序(步骤 3)。子阶段只有在其父阶段成功完成后才能提交。

19.5.3 任务调度

当任务集合被发送到任务调度程序后，任务调度程序用为该应用运行的 executor 的列表，在斟酌位置偏好的同时构建任务到 executor 的映射。接着，任务调度程序将任务分配给具有可用内核的 executor(如果同一应用中的另一个作业正在运行，则有可能分配不完整)，并且在 executor 完成运行任务时继续分配更多的任务，直到任务集合全部完成。默认情况下，每个任务到分派一个内核，不过也可以通过设置 `spark.task.cpus` 来更改。

请注意，任务调度程序在为某个 executor 分配任务时，首先分配的是进程本地(process-local)任务，再分配节点本地(node-local)任务，然后分配机架本地(rack-local)任务，最后分配任意(非本地)任务或者推测任务(speculative task)，如果没有其他任务候选者的话。[①]

这些被分配的任务通过调度程序后端启动(图 19-1 中的步骤 4)。调度程序后端向 executor 后端发送远程启动任务的消息(步骤 5)，以告知 executor 开始运行任务(步骤 6)。

① 推测任务是现有任务的复本，如果任务运行得比预期的缓慢，则调度程序可以将其作为备份来运行，详情可以参见 7.4.2 节。

Spark 利用 Akka (一个基于 Actor 的平台，*http://akka.io/*)来构建高度可扩展的事件驱动分布式应用，而不是使用 Hadoop RPC 进行远程调用。

当任务成功完成或者失败时，executor 都会向 driver 发送状态更新消息。如果失败了，任务调度程序将在另一个 executor 上重新提交任务。若是启用了推测任务(默认情况下不启用)，它还会为运行缓慢的任务启动推测任务。

19.5.4 任务执行

Executor 以如下方式运行任务(步骤 7)。首先，它确保任务的 JAR 包和文件依赖关系都是最新的。executor 在本地高速缓存中保留了先前任务已使用的所有依赖，因此只有在它们更新的情况下才会重新下载。第二步，由于任务代码是以启动任务消息的一部分而发送的序列化字节，因此需要反序列化任务代码(包括用户自己的函数)。第三步，执行任务代码。请注意，因为任务运行在与 executor 相同的 JVM 中，因此任务的启动没有进程开销。[①]

任务可以向 driver 返回执行结果。这些执行结果被序列化并发送到 executor 后端，然后以状态更新消息的形式返回 driver。shuffle map 任务返回的是一些可以让下一个阶段检索其输出分区的信息，而 result 任务则返回其运行的分区的结果值，driver 将这些结果值收集起来，并把最终结果返回给用户的程序。

19.6 执行器和集群管理器

我们已经看到 Spark 如何依靠 executor(执行器)来运行构成 Spark 作业的任务，但是对 executor 实际上是如何开始工作的却只粗略带过。负责管理 executor 生命周期的是集群管理器(cluster manager)，Spark 提供了好多种具有不同特性的集群管理器。

本地模式 在使用本地模式时，有一个 executor 与 driver 运行在同一个 JVM 中。此模式对于测试或运行小规模作业非常有用。这种模式的主 URL 为 `local`(使用一个线程)、`local[n]`(n 个线程)或 `local(*)`(机器的每个内核一个线程)。

独立模式 独立模式的集群管理器是一个简单的分布式实现，它运行了一个

① 在 Mesos 细粒度模式下，情况并非如此，它的每个任务作为单独的进程运行。有关详情，请参阅下一节。

master 以及一个或多个 worker。当 Spark 应用启动时，master 要求 worker 代表应用生成多个 executor 进程。这种模式的主 URL 为 spark://*host*:*port*。

Mesos 模式 Apache Mesos 是一个通用的集群资源管理器，它允许根据组织策略在不同的应用之间细化资源共享。默认情况下(细粒度模式)，每个 Spark 任务被当作是一个 Mesos 任务运行。这样做可以更有效地使用集群资源，但是以额外的进程启动开销为代价。在粗粒度模式下，executor 在进程中运行任务，因此在 Spark 应用运行期间的集群资源由 executor 进程来掌管。这种模式的主 URL 为 mesos://*host*:*port*。

YARN 模式 YARN 是 Hadoop 中使用的资源管理器(参见第 4 章)。每个运行的 Spark 应用对应于一个 YARN 应用实例，每个 executor 在自己的 YARN 容器中运行。这种模式的主 URL 为 `yarn-client` 或 `yarn-cluster`。

Mesos 和 YARN 集群管理器优于独立模式的集群管理器，因为它们考虑了在集群上运行的其他应用(例如 MapReduce 作业)的资源需求，并统筹实施调度策略。独立模式的集群管理器对集群的资源采用静态分配方法，因此不能随时适应其他应用的变化的需求。此外，YARN 是唯一一个能够与 Hadoop 的 Kerberos 安全机制集成的集群管理器(参见 10.4 节)。

运行在 YARN 上的 Spark

在 YARN 上运行 Spark 提供了与其他 Hadoop 组件最紧密的集成，也是在已有 Hadoop 集群的情况下使用 Spark 的最简便的方法。为了在 YARN 上运行，Spark 提供了两种部署模式：YARN 客户端模式和 YARN 集群模式。YARN 客户端模式的 driver 在客户端运行，而 YARN 集群模式的 driver 在 YARN 的 application master 集群上运行。

对于具有任何交互式组件的程序(例如 *spark-shell* 或 *pyspark*)都必须使用 YARN 客户端模式。客户端模式在构建 Spark 程序时也很有用，因为任何调试输出都是立即可见的。

另一方面，YARN 集群模式适用于生成作业(production job)，因为整个应用在集群上运行，这样做更易于保留日志文件(包括来自 driver 的日志文件)以供稍后检查。如果 application master 出现故障，YARN 还可以尝试重新运行该应用，详情参见 7.2.2 节)。

1. YARN 客户端模式

在 YARN 客户端模式下，当 driver 构建新的 SparkContext 实例时就启动了与 YARN 之间的交互(图 19-3 中的步骤 1)。该 Context 向 YARN 资源管理器提交一个 YARN 应用(步骤 2)，YARN 资源管理器则启动集群节点管理器上的 YARN 容器，并在其中运行一个名为 SparkExecutorLauncher 的 application master(步骤 3)。ExecutorLauncher 的工作是启动 YARN 容器中的 executor，为了做到这一点，ExecutorLauncher 要向资源管理器请求资源(步骤 4)，然后启动 ExecutorBackend 进程作为分配给它的容器(步骤 5)。

图 19-3. 在 YARN 客户端模式下 Spark executor 的启动流程

每个 executor 在启动时都会连接回 SparkContext，并注册自身。这就向 SparkContext 提供了关于可用于运行任务的 executor 的数量及其位置的信息，这些信息被用在任务的位置偏好策略中(参见 19.5.3 节)。启动的 executor 的数量在 *spark-shell*、*spark-submit* 或 *py-spark* 中设置(如果未设置，则默认为两个)，同时还要设置每个 executor 使用的内核数(默认值为 1)以及内存量(默认值为 1,024 MB)。下面这个例子显示了如何在 YARN 上运行具有 4 个 executor 且每个 executor 使用 1 个内核和 2 GB 内存的 *spark-shell*：

```
% spark-shell --master yarn-client \
  --num-executors 4 \
  --executor-cores 1 \
  --executor-memory 2g
```

YARN 资源管理器的地址并没有在主 URL 中指定(这与使用独立模式或 Mesos 模式的集群管理器不同)，而是从 `HADOOP_CONF_DIR` 环境变量指定的目录中的 Hadoop 配置中选取。

2. YARN 集群模式

在 YARN 集群模式下，用户的 driver 程序在 YARN 的 application master 进程中运行。使用 `spark-submit` 命令时需要输入 `yarn-cluster` 的主 URL：

```
% spark-submit --master yarn-cluster ...
```

所有其他参数，比如`--num-executors` 和应用 JAR(或 Python 文件)，都与 YARN 客户端模式相同(具体用法可通过 `spark-submit --help` 获得)。

spark-submit 客户端将会启动 YARN 应用(图 19-4 中的步骤 1)，但是它不会运行任何用户代码。剩余过程与客户端模式相同，除了 application master 在为 executor 分配资源(步骤 4)之前先启动 driver 程序(步骤 3b)外。

图 19-4. 在 YARN 集群模式下 Spark executor 的启动流程

在这两种 YARN 模式下，executor 都是在还没有任何本地数据位置信息之前先启动的，因此最终有可能会导致 executor 与存有作业所希望访问文件的 datanode 并不在一起。对于交互式会话，这是可以接受的，特别是因为会话开始之前可能并不知道需要访问哪些数据集。但是对于生成作业来说，情况并非如此，所以 Spark 提供了一种方法，可以在 YARN 群集模式下运行时提供一些有关位置的提示，以提高数据本地性。

SparkContext 构造函数可以使用第二个参数来传递一个优选位置。这个优选位置是利用 InputFormatInfo 辅助类根据输入格式和路径计算得到的。例如，对于文本文件，我们使用的是 TextInputFormat：

```
val preferredLocations = InputFormatInfo.computePreferredLocations(
    Seq(new InputFormatInfo(new Configuration(), classOf[TextInputFormat],
    inputPath)))
val sc = new SparkContext(conf, preferredLocations)
```

当向资源管理器请求分配时，application master 需要用到这个优选位置(步骤 4)。[1]

19.7 延伸阅读

本章只讨论了 Spark 的一些基础知识。更多细节请参阅 O'Reilly 在 2014 出版的 *Learning Spark*，网址为 *http://shop.oreilly.com/product/ 0636920028512.do*，作者 Holden Karau、Andy Konwinski、Patrick Wendell 和 Matei Zaharia。Apache Spark 网站(*http://spark.apache.org/*)上还有关于最新 Spark 发行版的更新文档。

① 在写这一章内容时，最新版本 Spark 1.2.0 中优选位置 API 还不太稳定，可能在以后的版本中会有所改善。

第 20 章

关于 HBase

(作者：Jonathan Gray 和 Michael Stack)

20.1 HBase 基础

HBase 是一个在 HDFS 上开发的面向列的分布式数据库。如果需要实时地随机访问超大规模数据集，就可以使用 HBase 这一 Hadoop 应用。

虽然数据库存储和检索的实现可以选择很多不同的策略，但是绝大多数解决办法，特别是关系型数据库技术的变种，不是为大规模可伸缩的分布式处理设计的。很多厂商提供了复制(replication)和分区(partitioning)解决方案，让数据库能够从单个节点上扩展出去，但是这些附加的技术大都属于“事后”的解决办法，而且非常难以安装和维护。并且这些解决办法常常要牺牲一些重要的 RDBMS 特性。在一个“扩展的”RDBMS 上，连接、复杂查询、触发器、视图以及外键约束这些功能要么运行开销大，要么根本无法用。

HBase 从另一个方向来解决可伸缩性的问题。它自底向上地进行构建，能够简单地通过增加节点来达到线性扩展。HBase 并不是关系型数据库，它不支持 SQL。[①] 但在特定的问题空间里，它能够做 RDBMS 不能做的事：在廉价硬件构成的集群上管理超大规模的稀疏表。

HBase 的一个典型应用是 *webtable*，一个以网页 URL 为主键的表，其中包含爬取

① 不过，可以了解一下 17.4.3 节中提到的 Apache Phoenix 项目以及名为 Trafodion 的基于 HBase 的事务 SQL 数据库(*https://wiki.trafodion.org/*)。

的页面和页面的属性(例如语言和 MIME 类型)。webtable 非常大，行数可以达十亿级(billion)之级。在 webtable 上连续运行用于批处理分析和解析的 MapReduce 作业，能够获取相关的统计信息，增加验证的 MIME 类型列以及供搜索引擎进行索引的解析后的文本内容。同时，表格还会被以不同运行速度的“爬取器”(crawler)随机访问并随机更新其中的行。在用户点击访问网站的缓存页面时，需要实时地将这些被随机访问的页面提供给他们。

背景

HBase 项目是由 Powerset 公司的 Chad Walters 和 Jim Kelleman 在 2006 年末发起的。当时，它起源于在此之前 Google 刚刚发布的 Bigtable。[①] 2007 年 2 月，Mike Cafarella 提供代码，形成了一个基本可以用的系统，然后 Jim Kellerman 接手继续推进该项目。

HBase 的第一个发布版本是在 2007 年 10 月和 Hadoop 0.15.0 捆绑在一起发布的。2010 年 5 月，HBase 从 Hadoop 子项目升级成 Apache 顶层项目。今天，HBase 已然成为一种广泛应用于各种行业生产中的成熟技术。。

20.2 概念

在本节中，我们只对 HBase 的核心概念进行快速、简单的介绍。掌握这些概念至少有助于消化后续内容。

20.2.1 数据模型的“旋风之旅”

应用把数据存放在带标签的表中。表由行和列组成。表格的“单元格”(cell)由行和列的坐标交叉决定，是有版本的。默认情况下，版本号是自动分配的，为 HBase 插入单元格时的时间戳。单元格的内容是未解释的字节数组。例如，图 20-1 所示为用于存储照片的 HBase 表。

表中行的键也是字节数组。所以理论上，任何东西都可以通过表示成字符串或将二进制形式转化为长整型或直接对数据结构进行序列化，来作为键值。表中的行

① 引自 Fay Chang 等人的文章“Bigtable: A Distributed Storage System for Structured Data”(Bigtable：一个结构化数据的分布式存储系统)，发表于 2006 年 11 月，网址为 *http://research.google.com/archive/bigtable.html*。

根据行的键值(也就是表的主键)进行排序。排序根据字节序进行。所有对表的访问都要通过表的主键。[1]

图 20-1. 用于描述存储照片的表的 HBase 数据模型

行中的列被分成“列族”(column family)。同一个列族的所有成员具有相同的前缀。因此，像列 *info:format* 和 *info:geo* 都是列族 *info* 的成员，而 *contents:image* 则属于 *contents* 族。列族的前缀必须由“可打印的”(printable)字符组成。而修饰性的结尾字符，即列族修饰符，可以为任意字节。列族和修饰符之间始终以冒号(:)分隔。

一个表的列族必须作为表模式定义的一部分预先给出，但是新的列族成员可以随后按需要加入。例如，只要目标表中已经有了列族 *info*，那么客户端就可在更新时提供新的列 *info:camera*，并存储它的值。

物理上，所有的列族成员都一起存放在文件系统中。所以，虽然我们前面把 HBase 描述为一个面向列的存储器，但实际上更准确的说法是：它是一个面向列族的存储器。由于调优和存储都是在列族这个层次上进行的，所以最好使所有列族成员都有相同的**访问模式**(access pattern)和大小特征。对于存储照片的表，由

① HBase 不支持表中的其他列建立索引(也称为辅助索引)。不过，有几种策略可用于支持辅助索引提供的查询类型，每种策略在存储空间、处理负载和查询执行时间之间存在不同的利弊权衡，关于这个问题，请参阅 HBase 参考指南，网址为 *http://hbase.apache.org/book.html*。

于图像数据比较大(兆字节)，因而跟较小的元数据(千字节)分别存储在不同的列族中。

简而言之，HBase 表和 RDBMS 中的表类似，只不过它的单元格有版本，行是排序的，而只要列族预先存在，客户端随时可以把列添加到列族中去。

1. 区域

HBase 自动把表水平划分成**区域**(region)。每个区域由表中行的子集构成。每个区域由它所属于的表、它所包含的第一行及其最后一行(不包括这行)来表示。一开始，一个表只有一个区域。但是随着区域开始变大，等到它超出设定的大小阈值，便会在某行的边界上把表分成两个大小基本相同的新分区。在第一次划分之前，所有加载的数据都放在原始区域所在的那台服务器上。随着表变大，区域的个数也会增加。区域是在 HBase 集群上分布数据的最小单位。用这种方式，一个因为太大而无法放在单台服务器上的表会被放到服务器集群上，其中每个节点都负责管理表所有区域的一个子集。表的加载也是使用这种方法把数据分布到各个节点。在线的所有区域按次序排列就构成了表的所有内容。

2. 加锁

无论对行进行访问的事务牵涉多少列，对行的更新都是“原子的”(atomic)。这使得“加锁模型”(locking model)能够保持简单。

20.2.2 实现

正如 HDFS 和 YARN 是由客户端、从属机(slave)和协调主控机(master)——(即 HDFS 的 *namenode* 和 *datanode*，以及 YARN 的资源管理器和节点管理器)——组成，HBase 也采用相同的模型，它用一个 master 节点协调管理一个或多个 regionserver 从属机(参见图 20-2)。HBase 主控机(master)负责启动(bootstrap)一个全新的安装，把区域分配给注册的 regionserver，恢复 regionserver 的故障。master 的负载很轻。regionserver 负责零个或多个区域的管理以及响应客户端的读/写请求。regionserver 还负责区域的划分并通知 HBase master 有了新的子区域(daughter region)，这样一来，主控机就可以把父区域设为离线，并用子区域替换父区域。

图 20-2. HBase 集群的成员

HBase 依赖于 ZooKeeper(参见第 21 章)。默认情况下，它管理一个 ZooKeeper 实例，作为集群的“权威机构”(authority)，尽管也可以通过配置来使用已有的 ZooKeeper 集群。ZooKeeper 集合体(ensemble)负责管理诸如 `hbase:meta` 目录表的位置以及当前集群主控机地址等重要信息。如果在区域的分配过程中有服务器崩溃，就可以通过 ZooKeeper 来进行分配的协调。在 ZooKeeper 上管理分配事务的状态有助于在恢复时能够从崩溃服务器遗留的状态开始继续分配。在启动一个客户端到 HBase 集群的连接时，客户端必须至少拿到到集群所传递的 ZooKeeper 集合体的位置。这样，客户端才能访问 ZooKeeper 的层次结构，从而了解集群的属性，例如服务器的位置。

类似于在 Hadoop 中可以通过 *etc/hadoop/slaves* 文件查看 datanode 和节点管理器一样，regionserver 从属机节点列在 HBase 的 *conf/regionservers* 文件中。启动和结束服务的脚本和 Hadoop 类似，使用相同的基于 SSH 的机制来运行远程命令。集群的站点配置(site-specific configuration)在 HBase 的 *conf/hbase-site.xml* 和 *conf/hbase-env.sh* 文件中。它们的格式和 Hadoop 父项目中对应的格式相同(参见第 10 章)。

对于 HBase 和 Hadoop 上相同的服务或类型，HBase 实际上直接使用或继承 Hadoop 的实现。在无法直接使用或继承时，HBase 会尽量遵循 Hadoop 的模型。例如，HBase 使用 Hadoop Configuration 系统，所以它们的配置文件格式相同。对于作为用户的你来说，这意味着你使用 HBase 时感觉就像使用 Hadoop 一样“亲切”。HBase 只是在增加特殊功能时才不遵循这一规则。

HBase 通过 Hadoop 文件系统 API 来持久化存储数据。多数人使用 HDFS 作为存储来运行 HBase。但是，在默认情况下，除非另行指明，HBase 会将存储写入本地文件系统。如果是体验一下新装 HBase，这是没有问题的，但如果稍后要使用 HBase 集群，首要任务通常是把 HBase 的存储配置为指向要使用的 HDFS 集群。

运行中的 HBase

HBase 内部保留名为 *hbase:meta* 的特殊目录表(catalog table)。它们维护着当前集群上所有区域的列表、状态和位置。*hbase:meta* 表中的项使用区域名作为键。区域名由所属的表名、区域的起始行、区域的创建时间以及对其整体进行的 MD5 哈希值(即对表名、起始行、创建的时间戳进行哈希后的结果)组成。

例如，表 *TestTable* 中起始行为 *xyz* 的区域的名称如下：

```
TestTable,xyz,1279729913622.1b6e176fb8d8aa88fd4ab6bc80247ece.
```

在表名、起始行、时间戳中间用逗号分隔。MD5 哈希值则使用前后两个句号包围。

如前所述，行的键是排序的。因此，要查找一个特定行所在的区域只要在目录表中找到第一个键大于或等于给定行键的项即可。区域变化时，即分裂、禁用/启用(disable/enable)、删除、为负载均衡重新部署区域或由于 regionserver 崩溃而重新部署区域时，目录表会进行相应的更新。这样，集群上所有区域的状态信息就能保持是最新的。

新连接到 ZooKeeper 集群上的客户端首先查找 `hbase:meta` 的位置。然后客户端通过查找合适的 `hbase:meta` 区域来获取用户空间区域所在节点及其位置。接着，客户端就可以直接和管理那个区域的 regionserver 进行交互。

每个行操作可能要访问三次远程节点。为了节省这些代价，客户端会缓存它们遍历 `hbase:meta` 时获取的信息。需要缓存的不仅有位置信息，还有用户空间区域的开始行和结束行。这样，它们以后不需要访问 `hbase:meta` 表也能得知区域存放的位置。客户端在碰到错误之前会一直使用所缓存的项。当发生错误时，即区域被移动了，客户端会再去查看 `hbase:meta` 获取区域的新位置。如果

hbase:meta 区域也被移动了，客户端会重新查找。

到达 Regionserver 的写操作首先追加到“提交日志”(commit log)中，然后加入内存中的 memstore。如果 memstore 满，它的内容会被“刷入”(flush)文件系统。

提交日志存放在 HDFS 中，因此即使一个 regionserver 崩溃，提交日志仍然可用。如果发现一个 regionserver 不能访问(通常因为服务器的 znode 在 ZooKeeper 中过期)，主控机会根据区域对死掉的 regionserver 的提交日志进行分割。重新分配后，在打开并使用死掉的 regionserver 上的区域之前，这些区域会找到属于它们的从被分割提交日志中得到的文件，其中包含还没有被持久化存储的更新。这些更新会被“重做”(replay)以使区域恢复到服务器失败前的状态。

在读的时候首先查看区域的 memstore。如果在 memstore 中找到了需要的版本，查询就结束了。否则，需要按照次序从新到旧检查“刷新文件”(flush file)，直到找到满足查询的版本，或所有刷新文件都处理完为止。

有一个后台进程负责在刷新文件个数到达一个阈值时压缩它们。它把多个文件重新写入一个文件。这是因为读操作检查的文件越少，它的执行效率越高。在压缩(compaction)时，进程会清理掉超出模式所设最大值的版本以及删除单元格或标识单元格为过期。在 regionserver 上，另外有一个独立的进程监控着刷新文件的大小，一旦文件大小超出预先设定的最大值，便对区域进行分割。

20.3 安装

从一个 Apache Download Mirror (*http://www.apache.org/dyn/closer.cgi/hbase/*)中挑选并下载一个 HBase 的稳定发布版本，然后在本地文件系统解压。示例如下：

```
% tar xzf hbase-x.y.z.tar.gz
```

和用 Hadoop 一样，首先需要告诉 HBase 系统中的 Java 在哪里。如果设置了 JAVA_HOME 环境变量，把它指向了正确的 Java 安装，HBase 就会使用那个 Java 安装。这样便不需要进行其他配置。否则，可以通过编辑 HBase 的 *conf/hbase-env.sh*，并指明 JAVA_HOME 变量的值(参见附录 A 的示例)，从而设置 HBase 所使用的 Java 安装。

为了方便，把 HBase 的二进制文件目录加入命令行路径中。示例如下：

```
% export HBASE_HOME=~/sw/hbase-x.y.z
% export PATH=$PATH:$HBASE_HOME/bin
```

要想获取 HBase 的选项列表，输入以下命令即可：

```
% hbase
Options:
  --config DIR Configuration direction to use. Default: ./conf
  --hosts HOSTS Override the list in 'regionservers' file
Commands:
Some commands take arguments. Pass no args or -h for usage.
  shell Run the HBase shell
  hbck Run the hbase 'fsck' tool
  hlog Write-ahead-log analyzer
  hfile Store file analyzer
  zkcli Run the ZooKeeper shell
  upgrade Upgrade hbase
  master Run an HBase HMaster node
  regionserver Run an HBase HRegionServer node
  zookeeper Run a Zookeeper server
  rest Run an HBase REST server
  thrift Run the HBase Thrift server
  thrift2 Run the HBase Thrift2 server
  clean Run the HBase clean up script
  classpath Dump hbase CLASSPATH
  mapredcp Dump CLASSPATH entries required by mapreduce
  pe Run PerformanceEvaluation
  ltt Run LoadTestTool
  version Print the version
  CLASSNAME Run the class named CLASSNAME
```

测试驱动

要启动一个使用本地文件系统*/tmp* 目录作为持久化存储的 HBase 临时实例，键入以下命令：

```
% start-hbase.sh
```

在默认情况下，HBase 会被写入到*/${java.io.tmpdir}/hbase-${user.name}*中。`${java.io.tmpdir}`通常被映射为*/tmp*，不过，还是应该通过设置 *hbase-site.xml* 中的 `hbase.tmp.dir` 来对 HBase 进行配置，以便使用更长久的存储位置。在独立模式下，HBase 主控机、regionserver 和 ZooKeeper 实例都是在同一个 JVM 中运行的。

要管理 HBase 实例，键入以下命令启动 HBase 的 shell 环境即可：

```
% hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.98.7-hadoop2, r800c23e2207aa3f9bddb7e9514d8340bcfb89277, Wed Oct 8
15:58:11 PDT 2014

hbase(main):001:0>
```

这将启动一个加入了一些 HBase 特有命令的 JRuby IRB 解释器。输入 help 然后按 RETURN 键可以查看已分组的 shell 环境的命令列表。输入 help COMMAND_GROUP 可以查看某一类命令的帮助，而输入 help COMMAND 则能获得某个特定命令的帮助信息和用法示例。命令使用 Ruby 的格式来指定列表和目录。主帮助屏幕的最后包含一个快速教程。

现在，让我们创建一个简单的表，添加一些数据，再把表清空。

要新建一个表，首先必须为你的表起一个名字，并为其定义模式。一个表的模式包含表的属性和列族的列表。列族本身也有属性，可以在定义模式时依次定义它们。例如，列族的属性包括列族是否应该在文件系统中被压缩存储以及一个单元格要保存多少个版本等。模式可以被修改，需要修改时把表设为“离线”(offline)即可。在 shell 环境中使用 disable 命令可以把表设为离线，使用 alter 命令可以进行必要的修改，而 enable 命令则可以把表重新设为“在线”(online)。

要想新建一个名为 test 的表，使其只包含一个名为 data 的列，表和列族属性都为默认值，则键入以下命令：

```
hbase(main):001:0> create 'test', 'data'
0 row(s) in 0.9810 seconds
```

如果前面有命令没有成功完成，那么 shell 环境会提示错误并显示堆栈跟踪(stack trace) 信息。这时你的安装肯定没有成功。请检查 HBase 日志目录中的主控机日志，查看哪里出了问题。默认的日志目录是*${HBASEHOME}/logs*。

关于如何在定义模式时添加表和列族属性的示例，可参见 help 命令的输出。

为了验证新表是否创建成功，运行 list 命令。这会输出用户空间中的所有表：

```
hbase(main):002:0> list
TABLE
test
1 row(s) in 0.0260 seconds
```

要在列族 data 中三个不同的行和列上插入数据，获取第一行，然后列出表的内

容，输入如下：

```
hbase(main):003:0> put 'test', 'row1', 'data:1', 'value1'
hbase(main):004:0> put 'test', 'row2', 'data:2', 'value2'
hbase(main):005:0> put 'test', 'row3', 'data:3', 'value3'
hbase(main):006:0> get 'test', 'row1'
COLUMN                CELL
  data:1              timestamp=1414927084811, value=value1
1 row(s) in 0.0240 seconds
hbase(main):007:0> scan 'test'
ROW                   COLUMN+CELL
  row1                column=data:1, timestamp=1414927084811, value=value1
  row2                column=data:2, timestamp=1414927125174, value=value2
  row3                column=data:3, timestamp=1414927131931, value=value3
3 row(s) in 0.0240 seconds
```

请注意我们是如何在添加三个新列的时候不修改模式的。

为了移除这个表，首先要把它设为禁用，然后删除：

```
hbase(main):009:0> disable 'test'
0 row(s) in 5.8420 seconds
hbase(main):010:0> drop 'test'
0 row(s) in 5.2560 seconds
hbase(main):011:0> list
TABLE
0 row(s) in 0.0200 seconds
```

通过运行以下命令来关闭 HBase 实例：

```
% stop-hbase.sh
```

要想了解如何设置分布式的 HBase，并把它指向正运行的 HDFS，请参见 HBase 文档中有关 configuration section 的小节，网址为 *http://hbase.apache.org/book/configuration.html*。

20.4 客户端

和 HBase 集群进行交互，有很多种不同的客户端可供选择。

20.4.1 Java

HBase 和 Hadoop 一样，都是用 Java 开发的。范例 20-1 展示了 20.3.1 节中在 shell 环境下运行的 Java 实现版本。

范例 20-1. 基本的表管理与访问

```
public class ExampleClient {

    public static void main(String[] args) throws IOException {
        Configuration config = HBaseConfiguration.create();
        // Create table
        HBaseAdmin admin = new HBaseAdmin(config);
        try {
            TableName tableName = TableName.valueOf("test");
            HTableDescriptor htd = new HTableDescriptor(tableName);
            HColumnDescriptor hcd = new HColumnDescriptor("data");
            htd.addFamily(hcd);
            admin.createTable(htd);
            HTableDescriptor[] tables = admin.listTables();
            if (tables.length != 1 &&
            Bytes.equals(tableName.getName(), tables[0].getTableName().getName())) {
            throw new IOException("Failed create of table");
        }
    // Run some operations -- three puts, a get, and a scan -- against the table.
    HTable table = new HTable(config, tableName);
    try {
    for (int i = 1; i <= 3; i++) {
    byte[] row = Bytes.toBytes("row" + i);
    Put put = new Put(row);
    byte[] columnFamily = Bytes.toBytes("data");
    byte[] qualifier = Bytes.toBytes(String.valueOf(i));
    byte[] value = Bytes.toBytes("value" + i);
    put.add(columnFamily, qualifier, value);
    table.put(put);
    }
    Get get = new Get(Bytes.toBytes("row1"));
    Result result = table.get(get);
    System.out.println("Get: " + result);
    Scan scan = new Scan();
    ResultScanner scanner = table.getScanner(scan);
    try {
    for (Result scannerResult : scanner) {
    System.out.println("Scan: " + scannerResult);
    }
    } finally {
    scanner.close();
    }
    // Disable then drop the table
    admin.disableTable(tableName);
    admin.deleteTable(tableName);
    } finally {
    table.close();
    }
    } finally {
admin.close();
}
}
}
```

这个类只有一个主方法。为了简洁，我们没有给出导入包的信息。大多数 HBase 类都位于 org.apache.hadoop.hbase 和 org.apache.hadoop. hbase.client 包中。

在这个类中，我们首先让 HBaseConfiguration 类来创建 Configuration 对象。这个类会返回一个 Configuration，其中已经读入了程序 classpath 下 *hbase-site.xml* 和 *hbase-default.xml* 文件中的 HBase 配置信息。这个 Configuration 接下来会被用于创建 HBaseAdmin 和 HTable 实例。HBaseAdmin 用于管理 HBase 集群，添加和丢弃表。HTable 则用于访问指定的表。Configuration 实例将这些类指向了执行这些代码的集群。

从 HBase 1.0 开始，新的客户端 API 更加干净且直观。HBaseAdmin 和 HTable 的构造函数已被弃用，不建议客户端显式引用这些旧类。取而代之的是，客户端应该使用新的 ConnectionFactory 类创建一个 Connection 对象，然后根据需要调用 getAdmin()或 getTable()来检索 Admin 或 Table 实例。连接管理以前是在幕后为用户完成的，而现在则是客户端的职责。在本书的附带网站上可以找到使用了新的 API 的本章示例的更新版本。

要创建一个表，我们需要首先创建一个 HBaseAdmin 的实例，然后让它来创建名为 test 且只有一个列族 data 的表。在我们的示例中，表的模式是默认的。可以使用 HTableDescripttor 和 HColumnDescriptor 中的方法来修改表的模式。接下来的代码测试了表是否真的被创建了，如果没有，则抛出异常。

要对一个表进行操作，我们需要新建一个 HTable 的实例，并把我们的 Configuration 实例和我们要操作的表的名称传递给它。然后，我们在循环中创建 Put 对象，以便将数据插入到表中。Put 把单个的单元格值 valuen 放入名为 rown 的行的名为 data:n 的列上，其中 n 为 1 到 3。列名通过两个部分指定：列族名和列族修饰词。上面这段代码使用了 HBase 的 Bytes 实用类来把标识符和值转换为 HBase 所需的字节数组。

接着，我们新建一个 Get 对象来获取和打印刚添加的第一行。然后，再使用 Scan 对象来扫描新建的表，并打印扫描结果。

在程序的最后，我们首先禁用表，接着删除它，把这张表清除掉。我们曾提到过在丢弃表前必须先禁用它。

扫描器

HBase 扫描器(scanner)和传统数据库中的游标(cursor)或 Java 中的迭代器(iterator)类似。它和后者的不同之处在于使用后需要关闭。扫描器按次序返回数据行。用户使用已设置的 Scan 对象实例作为 scan 参数，调用 getScanner()，以此来获取 HBase 中一个表上的扫描器。通过 Scan 实例，你可以传递扫描开始位置和结束位置的数据行、返回结果中要包含的列以及运行在服务器端的过滤器。ResultScanner 接口是调用 getScanner()时返回的接口，它的定义如下：

```
public interface ResultScanner extends Closeable, Iterable<Result> {
  public Result next() throws IOException;
  public Result [] next(int nbRows) throws IOException;
  public void close();
}
```

可以查看接下来的一个或多个数据行。扫描器会在幕后每次获取 100 行数据，把这些结果放在客户端，并只有在当前这批结果都被取光后才再去服务器端获取下一批结果。以这种方式获取和缓存的行数是由 hbase.client.scanner.caching 配置选项所决定的，或者也可以通过 setCaching()方法设置 Scan 实例自己缓存(cache)/预取(prefetch)的行数。

较大的缓存值可以加速扫描器的运行，但也会在客户端使用较多的内存。而且，还要避免把缓存值设得太高，因为它会导致客户端用于处理一批数据的时间超出扫描器的超时时间。如果在扫描器超时之前，客户端没有能再次访问服务器，那么扫描器在服务器端所用的资源会被服务器端的垃圾收集器自动回收。默认的扫描器超时时间为 60 秒，它在 hbase.client.scanner.timeout.period 中设置。如果扫描器超时，客户端会收到一个 UnknownScannerException 异常。

编译这段程序的最简单的方法是使用本书示例代码附带的 Maven POM。然后，我们可以用后面跟着类名的 hbase 命令来运行程序，如下所示：

```
% mvn package
% export HBASE_CLASSPATH=hbase-examples.jar
% hbase ExampleClient
Get: keyvalues={row1/data:1/1414932826551/Put/vlen=6/mvcc=0}
Scan: keyvalues={row1/data:1/1414932826551/Put/vlen=6/mvcc=0}
Scan: keyvalues={row2/data:2/1414932826564/Put/vlen=6/mvcc=0}
Scan: keyvalues={row3/data:3/1414932826566/Put/vlen=6/mvcc=0}
```

通过 Result 的 toString()方法可以让每行输出显示一个 HBase 数据行。字段由斜杠字符分隔且顺序如下：行名称，列名称，单元格时间戳，单元格类型，值的字节数组(vlen)长度和一个内部 HBase 字段(mvcc)。稍后我们将看到如何使用 getValue()方法从 Result 对象中获取值。

20.4.2 MapReduce

org.apache.hadoop.hbase.mapreduce 包中的类和工具有利于将 HBase 作为 MapReduce 作业的源/输出。TableInputFormat 类可以在区域的边界进行分割，使 map 能够拿到单个的区域进行处理。TableOutputFormat 将把 reduce 的结果写入 HBase。

范例 20-2 中的 SimpleRowCounter 类(它是 HBase mapreduce 包中 RowCounter 类的简化版本)使用 TableInputFormat 来运行一个 map 任务，以计算行数。

范例 20-2. 一个计算 HBase 表中行数的 MapReduce 应用程序

```
public class SimpleRowCounter extends Configured implements Tool {

  static class RowCounterMapper extends TableMapper<ImmutableBytesWritable,
      Result> {
    public static enum Counters { ROWS }

    @Override
    public void map(ImmutableBytesWritable row, Result value, Context context) {
      context.getCounter(Counters.ROWS).increment(1);
    }
  }

  @Override
  public int run(String[] args) throws Exception {
    if (args.length != 1) {
      System.err.println("Usage: SimpleRowCounter <tablename>");
      return -1;
    }
    String tableName = args[0];
    Scan scan = new Scan();
    scan.setFilter(new FirstKeyOnlyFilter());
    Job job = new Job(getConf(), getClass().getSimpleName());
    job.setJarByClass(getClass());
    TableMapReduceUtil.initTableMapperJob(tableName, scan,
        RowCounterMapper.class, ImmutableBytesWritable.class, Result.class, job);
    job.setNumReduceTasks(0);
    job.setOutputFormatClass(NullOutputFormat.class);
    return job.waitForCompletion(true) ? 0 : 1;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(HBaseConfiguration.create(),
        new SimpleRowCounter(), args);
    System.exit(exitCode);
```

```
  }
}
```

嵌套类 RowCounterMapper 是 HBase 的 TableMapper 抽象类的一个子类。后者是 org.apache.hadoop.mapreduce.Mapper 的“特化”(specialization)。它设定 map 的输入类型由 TableInputFormat 来传递。输入的键为 ImmutableBytesWritable 对象(行键)，值为 Result 对象(扫描的行结果)。由于这个作业只对行进行计数，而且没有从 map 中发送任何输出，因此我们仅为看到的每一行把 Counters.ROWS 的值递增 1。

在 run() 方法中，我们创建了一个扫描对象。这个扫描对象通过调用 TableMapReduceUtil.initTableMapJob()实用方法来对作业进行配置，它除了设置输入格式到 TableInputFormat 之外，还可以做其他一些事情(例如设置要使用的 map 类)。

注意这里是如何设置扫描过滤器(即 FirstKeyOnlyFilter 实例)的。这个过滤器告诉服务器，让它在运行服务器端任务时只用每行的第一个单元格来填充 mapper 中的 Result 对象。由于 mapper 忽略了单元格值，因此这个优化是有意义的。

也可以通过在 HBase 的 shell 环境中输入 count 'tablename'命令来查询某个表的行数。不过，它不是分布式命令，因此适用于 MapReduce 程序中的大型表。

20.4.3 REST 和 Thrift

HBase 提供了 REST 和 Thrift 接口。在使用 Java 以外的编程语言和 HBase 交互时，会用到这些接口。在所有情况下，Java 服务器上都运行着一个 HBase 客户端实例，它负责协调 REST 和 Thrift 应用请求和 HBase 集群间的双向交互。有关服务的运行以及客户端接口的详细情况，请参阅参考指南，网址为 *http://hbase.apache.org/book.html*。

20.5 创建在线查询应用

虽然 HDFS 和 MapReduce 是用于对大数据集进行批处理的强大工具，但对于读或写单独的记录，效率却很低。在这个示例中，我们将看到如何用 HBase 来填补它们之间的鸿沟。

前面几章描述的气象数据集包含过去 100 多年上万个气象站的观测数据。这个数据集还在继续增长，它的大小几乎是无限的。在这个示例中，我们将构建一个简

单的在线查询(而不是批处理)界面，允许用户按时间顺序导航不同的观测站并浏览历史气象观测值。我们将为此而构建一个简单的命令行 Java 应用程序，不过也不难从中看出应当如何用相同的技术来构建一个具有同样效果的 Web 应用程序。

为此，让我们假设数据集非常大，观测数据达到上亿条记录，且气温更新数据到达的速度很快——比如从全球观测站收到超过每秒几百到几千次更新。不仅如此，我们还假设这个在线应用必须能够及时(most up-to-date)显示观测数据，即在收到数据后大约 1 秒就能进行显示。

对数据集的第一个要求使我们排除了使用 RDBMS。HBase 是一个可能的选择。对于查询延时的第二个要求排除了直接使用 HDFS。MapReduce 作业可以用于建立索引以支持对观测数据进行随机访问，但 HDFS 和 MapReduce 并不擅长在有更新到达时维护索引。

20.5.1 模式设计

在我们的示例中，有两个表。

1. stations 表

这个表包含观测站数据。行的键是 `stationid`。这个表还有一个列族 `info`，它能作为键/值字典来支持对观测站信息的查找。字典的键就是列名 `info:name`、`info:location` 以及 `info:description`。这个表是静态的，在这里，列族 `info` 的设计类似于 RDBMS 中表的设计。

2. observations 表

这个表存放气温观测数据。行的键是 `stationid` 和逆序时间戳构成的组合键。这个表有一个列族 `data`，它包含一列 `airtemp`，其值为观测到的气温值。

对模式的选择取决于我们知道最高效的读取 HBase 的方式。行和列以字典序升序保存。虽然有二级索引和正则表达式匹配工具，但它们会损失其他性能。清楚地理解查询数据最高效的方式对于选择最有效的存储和访问数据的设置非常关键。

在 `stations` 表中，显然选择 `stationid` 作为键，因为我们总是根据特定站点的 ID 来访问观测站的信息。但 `observations` 表使用的是一个组合键(把观测的时间戳加在键之后)。这样，同一个观测站的观测数据就会被分组放到一起，使用逆序

时间戳(Long.MAX_VALUE - epoch)的二进制存储，系统把每个观测站观测数据中最新的数据存储在最前面。

站点 ID 是定长的这件事非常重要。在某些情况下我们需要对数字组件填零以便行键能够正确排序，否则有可能会遇到一些问题，比如说在仅考虑按字节排序时，10 将会排在 2 之前，而 02 则排在 10 之前。

此外，如果键是整数，那么它会采用二进制来表示，而不是以数字的字符串版本形式存储，因为前者消耗的空间更少。

在 shell 环境中，可以用以下方法来定义表：

```
hbase(main):001:0> create 'stations', {NAME => 'info'}
0 row(s) in 0.9600 seconds
hbase(main):002:0> create 'observations', {NAME => 'data'}
0 row(s) in 0.1770 seconds
```

在两个表中，我们都只对表单元格的最新版本感兴趣，所以把 VERSIONS 设为 1 (默认值是 3)。

宽表(Wide Table)

在 HBase 中，所有访问都是通过主键的，因此在键的设计上应当尽量照顾到如何查询这些数据。在对 HBase 这样的面向列(族)的存储(*http://en.wikipedia.org/ wiki/Column-oriented_DBMS*)设计模式时，另一件需要记住的是它可以以极小的开销管理较宽的稀疏表。[①]

HBase 并没有内置对数据库连接的支持。但是宽表使我们并不需要让第一个表和第二个表或第三个表进行数据库连接。一个宽行有时可以容下一个主键相关的所有数据。

20.5.2 加载数据

观测站的数量相对较少，所以我们可以使用任何一种接口来插入这些观测站的静态数据。在示例代码中包括一个用于执行此操作的 Java 应用程序，运行方式如下：

```
% hbase HBaseStationImporter input/ncdc/metadata/stations-fixed-idth.txt
```

① 引自 Daniel J. Abadi 的文章，标题为“Column-Stores for Wide and Sparse Data”，发表于 2007 年 1 月，网址为 *http://bit.ly/column-stores*。

但是，假设我们要加载数十亿条观测数据。这种数据导入是一个极为复杂的过程，是一个需要长时间运行的数据库操作。MapReduce 和 HBase 的分布式模型让我们可以充分利用集群。通过把原始输入数据复制到 HDFS，接着运行 MapReduce 作业，我们就能读到输入数据并将其写入 HBase。

范例 20-3 展示了一个 MapReduce 作业，它将观测数据从前几章所用的输入文件导入 HBase。

范例 20-3. 从 HDFS 向 HBase 表导入气温数据的 MapReduce 应用

```
public class HBaseTemperatureImporter extends Configured implements Tool {

  static class HBaseTemperatureMapper<K> extends Mapper<LongWritable,
      Text, K, Put> {
    private NcdcRecordParser parser = new NcdcRecordParser();

    @Override
    public void map(LongWritable key, Text value, Context context) throws
        IOException, InterruptedException {
      parser.parse(value.toString());
      if (parser.isValidTemperature()) {
        byte[] rowKey =
            RowKeyConverter.makeObservationRowKey(parser.getStationId(),
            parser.getObservationDate().getTime());
        Put p = new Put(rowKey);
        p.add(HBaseTemperatureQuery.DATA_COLUMNFAMILY,
            HBaseTemperatureQuery.AIRTEMP_QUALIFIER,
            Bytes.toBytes(parser.getAirTemperature()));
        context.write(null, p);
      }
    }
  }

  @Override
  public int run(String[] args) throws Exception {
    if (args.length != 1) {
      System.err.println("Usage: HBaseTemperatureImporter <input>");
      return -1;
    }
    Job job = new Job(getConf(), getClass().getSimpleName());
    job.setJarByClass(getClass());
    FileInputFormat.addInputPath(job, new Path(args[0]));
    job.getConfiguration().set(TableOutputFormat.OUTPUT_TABLE, "observations");
    job.setMapperClass(HBaseTemperatureMapper.class);
    job.setNumReduceTasks(0);
    job.setOutputFormatClass(TableOutputFormat.class);
    return job.waitForCompletion(true) ? 0 : 1;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(HBaseConfiguration.create(),
        new HBaseTemperatureImporter(), args);
    System.exit(exitCode);
```

```
  }
}
```

HBaseTemperatureImporter 有一个名为 HbaseTemperatureMapper 的嵌套类，它类似于第 6 章的 MaxTemperatureMaper 类。外部类实现了 Tool，并对调用这个仅含有 map 的作业进行设置。HBaseTemperatureMapper 和 MaxTemperatureMapper 的输入相同，所进行的解析方法——都使用第 6 章所介绍的 NcdcRecordParser 来进行——也相同。解析时会检查输入是否为有效的气温。但是不同于在 MaxTemperatureMapper 中仅把有效气温加到输出集合中，这个类创建一个 Put 对象以便把有效的气温值添加到 HBase 的 observations 表的 data:airtemp 列。我们使用了从 HBaseTemperatureQuery 类中导出的 data 和 airtemp 静态常量。后面对此会有介绍。

我们所用的行的键在 RowKeyConverter 上的 makeObservationRowKey()方法中，用观测站 ID 和观测时间创建：

```
public class RowKeyConverter {

  private static final int STATION_ID_LENGTH = 12;

  /**
   * @return A row key whose format is: <station_id> <reverse_order_timestamp>
   */
  public static byte[] makeObservationRowKey(String stationId,
      long observationTime) {
    byte[] row = new byte[STATION_ID_LENGTH + Bytes.SIZEOF_LONG];
    Bytes.putBytes(row, 0, Bytes.toBytes(stationId), 0, STATION_ID_LENGTH);
    long reverseOrderTimestamp = Long.MAX_VALUE - observationTime;
    Bytes.putLong(row, STATION_ID_LENGTH, reverseOrderTimestamp);
    return row;
  }
}
```

观测站 ID 其实是一个定长字符串。在转换时利用了这一点。与前面的例子一样，我们使用 HBase 的 Bytes 类在字节数组和常用的 Java 类型之间进行转换。Bytes.SIZEOF_LONG 常量用于计算行键字节数组的时间戳部分的大小。putBytes()和 putLong()方法用于填充行键字节数组中的观测站 ID 和时间戳部分，使它们处于相应的偏移位置。

该作业在 run()方法中被配置为使用 HBase 的 TableOutputFormat。将要写入的表必须在作业配置中通过设置 TableOutputFormat.OUTPUTTABLE 属性来指定。

TableOutputFormat 的使用为我们提供了方便，因为它负责管理 HTable 实例的

创建，否则我们需要在 mapper 的 setup()方法中来做这件事(另外还需要在 cleanup()方法中调用 close())。TableOutputFormat 禁用了 HTable 的自动刷新功能，因此可以缓存对 put()的调用以提高效率。这段示例代码包括一个名为 HBaseTemperatureDirectImporter 的类，以演示如何在 MapReduce 程序中直接使用 HTable。我们可以用下面的命令来运行程序：

```
% hbase HBaseTemperatureImporter input/ncdc/all
```

1. 加载的分布

要特别当心数据导入所引发的“步调一致”的情况。这时所有的客户端都对同一个表的区域(在单个节点上)进行操作，然后再对下一个区域进行操作，依次进行。这时加载操作并没有均匀地分布在所有区域上。这通常是由“排序后输入”(sorted input)和切分的原理共同造成的。如果在插入数据前，针对行键按数据排列的次序进行随机处理，可能有助于减少这种情况。在我们的示例中，基于当前 stationid 值的分布情况和 TextInputFormat 分割数据的方式，上传操作应该能够保证足够的分布式特性。

如果一个表是新的，一开始它只有一个区域。此时所有的更新都会加载到这个区域，直到区域分裂为止。即使数据行的键是随机分布的，我们也无法避免这种情况。这种启动现象意味着上传数据在开始时比较慢，直到有足够多的区域被分布到各个节点，集群的成员都能够参与到上传为止。不要把这种情况与前面一段所描述的情况混为一谈，它们是不同的。

这两个问题都可以通过使用批量加载来避免，下面我们将讨论批量加载。

2. 批量加载

HBase 有一个高效的“批量加载”(bulk loading)工具。它从 MapReduce 把以内部格式表示的数据直接写入文件系统，从而实现批量加载。顺着这条路，我们加载 HBase 实例的速度比用 HBase 客户端 API 写入数据的方式至少快一个数量级。

批量加载的处理过程分为两步。第一步使用 HFileOutputFormat2 通过一个 MapReduce 作业将 HFiles 写入 HDFS 目录。由于数据行必须按顺序写入，因此这个作业需要执行行键的完全排序(参见 9.2.3 节)。HFileOutputFormat2 的 configureIncrementalLoad()方法可以完成所有必要的配置。

批量加载的第二步涉及将 HFiles 从 HDFS 移入现有的 HBase 表中。这张表在此

过程中可以是活跃的。在示例代码中包括了一个名为 HBaseTemperatureBulkImporter 的类用于以批量加载方法来加载气象观测数据。

20.5.3 在线查询

为了实现在线查询应用，我们将直接使用 HBase 的 Java API。在这里，我们将深刻体会到选择模式和存储格式的重要性。

1. 观测站信息查询

最简单的查询就是获取静态的观测站信息。这是一个单行的查找，通过使用 get()操作来执行。这一类查询在传统数据库中也很简单，但 HBase 提供了额外的控制功能和灵活性。我们把 info 列族作为键/值字典(列名作为键，列值作为值)，HBaseStationQuery 中的这段代码如下所示：

```
static final byte[] INFO_COLUMNFAMILY = Bytes.toBytes("info");
static final byte[] NAME_QUALIFIER = Bytes.toBytes("name");
static final byte[] LOCATION_QUALIFIER = Bytes.toBytes("location");
static final byte[] DESCRIPTION_QUALIFIER = Bytes.toBytes("description");

public Map<String, String> getStationInfo(HTable table, String stationId)
    throws IOException {
  Get get = new Get(Bytes.toBytes(stationId));
  get.addColumn(INFO_COLUMNFAMILY);
  Result res = table.get(get);
  if (res == null) {
    return null;
  }
  Map<String, String> resultMap = new HashMap<String, String>();
  resultMap.put(" name" , getValue(res, INFO_COLUMNFAMILY, NAME_QUALIFIER));
  resultMap.put("location", getValue(res, INFO_COLUMNFAMILY, LOCATION_QUALIFIER));
  resultMap.put("description", getValue(res, INFO_COLUMNFAMILY,
    DESCRIPTION_QUALIFIER));
  return resultMap;
  }

  private static String getValue(Result res, byte [] cf, byte [] qualifier) {
    byte [] value = res.getValue(cf, qualifier);
    return value == null? ": Bytes.toString(value);
  }
```

在这个示例中，getStationInfo()接收一个 HTable 实例和一个观测站 ID。为了获取观测站的信息，我们使用 HTable.get()来传递一个 Get 实例。它被设置为用于获取已定义列族 INFO_COLUMNFAMILY 中由观测站 ID 所指明的列的值。

get()的结果在 Result 中返回。它包含数据行，你可以通过操作需要的列单元格

来取得单元格的值。getStationInfo()方法把 Result 转换为更便于使用的由 String 类型的键和值构成的 Map。

我们已经看出在使用 HBase 时为什么需要工具函数了。在 HBase 上，为了处理底层的交互，我们已经开发出越来越多的抽象。但是，理解它们的工作机理以及各个存储选项之间的差异，非常重要。

和关系型数据库相比，HBase 的优势之一是不需要我们预先设定列。所以在将来，如果每个观测站在这三个必有的属性以外还有几百个可选的属性，我们便可以直接插入这些属性而不需要修改模式。当然，你的应用中读和写的代码是需要修改的。在示例中，我们可以循环遍历 Result 来获取每个值，而不用显式获取各个值。

以下是观测站查询的示例：

```
% hbase HBaseStationQuery 011990-99999
name SIHCCAJAVRI
location (unknown)
description (unknown)
```

2. 观测数据查询

对 observations 表的查询需要输入的参数包括站点 ID、开始时间以及要返回的最大行数。由于数据行是按观测站以观测时间逆序存储的，因此查询将返回发生在开始时间之后的观察值。范例 20-4 中的 getStationObservations()方法使用 HBase 扫描器对表行进行遍历。它返回一个 NavigableMap<Long, Integer>，其中键是时间戳，值是温度。由于这个 map 按键的升序来排序，因此其中的数据项是按时间顺序来排列的。

范例 20-4. 检索 HBase 表中某范围内气象站观测数据行的程序

```
public class HBaseTemperatureQuery extends Configured implements Tool {
  static final byte[] DATA_COLUMNFAMILY = Bytes.toBytes("data");
  static final byte[] AIRTEMP_QUALIFIER = Bytes.toBytes("airtemp");

  public NavigableMap<Long, Integer> getStationObservations(HTable table,
      String stationId, long maxStamp, int maxCount) throws IOException {
    byte[] startRow = RowKeyConverter.makeObservationRowKey(stationId, maxStamp);
    NavigableMap<Long, Integer> resultMap = new TreeMap<Long, Integer>();
    Scan scan = new Scan(startRow);
    scan.addColumn(DATA_COLUMNFAMILY, AIRTEMP_QUALIFIER);
    ResultScanner scanner = table.getScanner(scan);
    try {
      Result res;
```

```
    int count = 0;
    while ((res = scanner.next()) != null && count++ < maxCount) {
      byte[] row = res.getRow();
      byte[] value = res.getValue(DATA_COLUMNFAMILY, AIRTEMP_QUALIFIER);
      Long stamp = Long.MAX_VALUE -
          Bytes.toLong(row, row.length - Bytes.SIZEOF_LONG, Bytes.SIZEOF_LONG);
      Integer temp = Bytes.toInt(value);
      resultMap.put(stamp, temp);
    }
  } finally {
    scanner.close();
  }
  return resultMap;
}

public int run(String[] args) throws IOException {
  if (args.length != 1) {
    System.err.println("Usage: HBaseTemperatureQuery <station_id>");
    return -1;
  }

  HTable table = new HTable(HBaseConfiguration.create(getConf()),
      "observations");
  try {
    NavigableMap<Long, Integer> observations =
      getStationObservations(table, args[0],
          Long.MAX_VALUE, 10).descendingMap();
    for (Map.Entry<Long, Integer> observation : observations.entrySet()) {
      // Print the date, time, and temperature
      System.out.printf("%1$tF %1$tR\t%2$s\n", observation.getKey(),
          observation.getValue());
    }
    return 0;
  } finally {
    table.close();
  }
}

public static void main(String[] args) throws Exception {
  int exitCode = ToolRunner.run(HBaseConfiguration.create(),
  new HBaseTemperatureQuery(), args);
  System.exit(exitCode);
}
}
```

run()方法调用 geStationObservations()以请求最近的 10 个观测值，并通过调用 descendingMap()使返回值仍然回归到降序。观测值被格式化并打印到控制台(记住，温度的单位是十分之一度)。例如：

```
% hbase HBaseTemperatureQuery 011990-99999
1902-12-31 20:00 -106
1902-12-31 13:00 -83
1902-12-30 20:00 -78
```

```
1902-12-30 13:00 -100
1902-12-29 20:00 -128
1902-12-29 13:00 -111
1902-12-29 06:00 -111
1902-12-28 20:00 -117
1902-12-28 13:00 -61
1902-12-27 20:00 -22
```

按时间逆序存储时间戳的优点是，通常在线应用程序需要的是最新的观测值，而这样做能更好地满足这一点。如果观测数据直接用实际的时间戳来存放，我们就只能根据偏移量和限制范围高效地获取最老的观测数据。要获取最新的数据则意味着要拿到所有的数据，直到最后才能获得结果。与此相比，获取前 *n* 行然后退出扫描程序(这种场景有时被称为“提早过滤”)的效率显然更高。

HBase 0.98 新增了反向扫描的能力，也就是说现在可以按时间顺序存储观测值，并从给定的起始行开始反向扫描。反向扫描比正向扫描要慢几个百分点。若要使用反向扫描，请在开始扫描之前调用 `Scan` 对象的 `setReversed(true)` 方法。

20.6 HBase 和 RDBMS 的比较

HBase 和其他面向列的数据库常常被拿来和更流行的传统关系型数据库(或简写为 RDBMS)进行比较。虽然它们在实现和设计上的出发点有着较大的区别，但它们都力图解决相同的问题。所以，虽然它们有很多不同点，但我们仍然能够对它们进行客观的比较。

如前所述，HBase 是一个分布式的、面向列的数据存储系统。它通过在 HDFS 上提供随机读/写来解决 Hadoop 不能处理的问题。HBase 自底层设计开始即聚焦于各种可伸缩性问题：表可以很“高”(数十亿个数据行)；表可以很“宽”(数百万个列)；水平分区并在上千个普通商用机节点(commodity node)上自动复制。表的模式是物理存储的直接反映，使系统有可能提供高效的数据结构的序列化、存储和检索。但是，应用程序的开发者必须承担重任，选择以正确的方式使用这种存储和检索方式。

严格来说，RDBMS 是一个遵循“Codd 的 12 条规则”(*http://en.wikipedia.org/wiki/Codd%27s_12_rules*)的数据库。标准的 RDBMS 是模式固定、面向行的数据库且具有 ACID 性质和复杂的 SQL 查询处理引擎。RDBMS 强调事务的“强一致性”(strong consistency)、参照完整性(referential integrity)、数据抽象与物理存储层

相对独立，以及基于 SQL 语言的复杂查询支持。在 RDBMS 中，可以非常容易地建立“二级索引”(secondary index)，执行复杂的内连接和外连接，执行计数、求和、排序、分组等操作，或对表、行和列中的数据进行分页存放。

对于大多数中小规模的应用，如 MySQL 和 PostgreSQL 之类现有开源 RDBMS 解决方案所提供的易用性、灵活性、产品成熟度以及强大、完整的功能特性几乎是无可替代的。但是，如果要在数据规模和并发读/写这两方面中的任何一个(或全部)上进行大规模向上扩展(scale up)，就会很快发现 RDBMS 的易用性会让你损失不少性能，而如果要进行分布式处理，更是非常困难。RDBMS 的扩展通常要求打破 Codd 的规则，如放松 ACID 的限制，使 DBA 的管理变得复杂，并同时放弃大多数关系型数据库引以为荣的易用性。

20.6.1 成功的服务

这里将简单介绍一个典型的 RDBMS 如何进行扩展。下面给出一个成功服务从小到大的生长过程。

(1) **服务首次提供公开访问。**
将服务从本地工作站迁移到拥有良好模式定义的、共享的远程 MySQL 实例上。

(2) **服务越来越受欢迎：数据库收到太多的读请求。**
用 memcached 来缓存常用查询结果。这时读不再是严格意义上的 ACID；缓存数据必须在某个时间到期。

(3) **对服务的使用继续增多：数据库收到太多的写请求。**
通过购买一个 16 核、128GB RAM、配备一组 15k RPM 硬盘驱动器的增强型服务器来垂直升级 MySQL。非常昂贵。

(4) **新的特性增加了查询的复杂度；包含很多连接操作。**
对数据进行反规范化以减少连接的次数。(这和 DBA 培训时所教的不一样！)

(5) **服务被广泛使用：所有的服务都变得非常慢。**
停止使用任何服务器端计算(server-side computation)。

(6) **有些查询仍然太慢。**
定期对最复杂的查询进行“预物化”(prematerialize)，并尝试在大多数情

况下停止使用连接。

(7) **读性能尚可，但写仍然越来越慢。**
放弃使用二级索引和触发器。(没有索引了？)

迄今为止，如何解决以上扩展问题并没有一个清晰的解决办法。无论怎样，都需要开始横向进行扩展。可以尝试在大表上进行某种分区或查看一些能提供多主控机的商业解决方案。

无数应用、行业以及网站都成功实现了 RDBMS 的可伸缩性、容错和分布式数据系统。它们都使用了前面提到的很多策略。但最终，你所拥有的已经不再是一个真正的 RDBMS。由于妥协和复杂性问题，系统放弃了很多易用性特性。任何种类的从属复本或外部缓存都会对反规范化的数据引入弱一致性(weak consistency)。连接和二级索引的低效意味着绝大多数查询成为主键查找。而对于多写入机制(multiwriter)的设置很可能意味着根本没有实际的连接，而分布式事务会成为一个噩梦。这时，要管理一个单独用于缓存的集群，网络拓扑会变得异常复杂。即使有一个做了那么多妥协的系统，你仍然会情不自禁地担心主控机崩溃或担心在几个月后，数据或负载可能会增长到当前的 10 倍。

20.6.2 HBase

让我们考虑 HBase，它具有以下特性。

- **没有真正的索引** 行是顺序存储的，每行中的列也是，所以不存在索引膨胀的问题，而且插入性能和表的大小无关。
- **自动分区** 在表增长的时候，表会自动分裂成区域，并分布到可用的节点上。
- **线性扩展和对于新节点的自动处理** 增加一个节点，把它指向现有集群并运行 regionserver。区域自动重新进行平衡，负载均匀分布。
- **普通商用硬件支持** 集群可以用 1000～5000 美金的单个节点搭建，而不需要使用单个得花 5 万美金的节点。RDBMS 需要支持大量 I/O，因此要求更昂贵的硬件。
- **容错** 大量节点意味着每个节点的重要性并不突出。不用担心单个节点失效。

- **批处理** MapReduce 集成功能使我们可以用全并行的分布式作业根据“数据的位置”(location awareness)来处理它们。

如果没日没夜地担心数据库(正常运行时间、扩展性问题、速度)，应该好好考虑从 RDBMS 转向使用 HBase。应该使用一个针对扩展性问题的解决方案，而不是性能越来越差却需要大量投入的曾经可用的方案。有了 HBase，软件是免费的，硬件是廉价的，而分布式处理则是与生俱来的。

20.7 Praxis

在这一小节，我们将讨论在应用中运行 HBase 集群时用户常常遇到的一些问题。

20.7.1 HDFS

HBase 使用 HDFS 的方式与 MapReduce 使用 HDFS 的方式截然不同。在 MapReduce 中，首先打开 HDFS 文件，然后 map 任务流式处理文件的内容，最后关闭文件。在 HBase 中，数据文件在启动时就被打开，并在处理过程中始终保持打开状态，这是为了节省每次访问操作打开文件所需的代价。所以，HBase 更容易碰到 MapReduce 客户端不常碰到的问题：

1. 文件描述符用完

由于我们在连接的集群上保持文件的打开状态，所以用不了太长时间就可能达到系统和 Hadoop 设定的限制。例如，我们有一个由三个节点构成的集群，每个节点上运行一个 datanode 实例和一个 regionserver。如果我们正在运行一个加载任务，表中有 100 个区域和 10 个列族。我们允许每个列族平均有两个“刷入文件”(flush file)。通过计算，我们知道同时打开了 $100 \times 10 \times 2$，即 2000 个文件。此外，还有各种外部扫描器和 Java 库文件占用了其他文件描述符。每个打开的文件在远程 datanode 上至少占用一个文件描述符。

一个进程默认的文件描述符限制是 1024。当我们使用的描述符个数超过文件系统的 *ulimit* 值，我们会在日志中看到“Too many open files”(打开了太多文件)的错误信息。但在这之前，往往就已经能看出 HBase 的行为不正常。要修正这个问题需要增加文件描述符的 *ulimit* 参数值。有关如何增加集群的 ulimit 值，请参阅 HBase 参考指南，网址为 *http://hbase.apache.org/book.html*。

2. datanode 上的线程用完

和前面的情况类似，Hadoop 1 的 datanode 上同时运行的线程数不能超过 256 这一限制值(`dfs.datanode.max.xcievers`)，这个限制值会导致 HBase 异常运行。Hadoop 2 将默认值提高到 4,096，因此最近几个版本的 HBase(仅在 Hadoop 2 及更高版本上运行)出现问题的可能性较小。可以通过在 *hdfs-site.xml* 中配置 `dfs.datanode.max.transfer.threads`(此属性的新名称)来更改设置。

20.7.2 用户界面

HBase 在主控机上运行了一个 Web 服务器，它能提供运行中集群的状态视图。在默认情况下，它监听 60010 端口。主界面显示了基本的属性(包括软件版本、集群负载、请求频率、集群表的列表)和加入的 regionserver 等。在主界面上单击选中 regionserver 会把你带到那个 regionserver 上运行的 Web 服务器，它列出了这个服务器上所有区域的列表及其他基本的属性值，比如如使用的资源和请求频率。

20.7.3 度量

Hadoop 有一个度量(metric)系统。可以用它每过一段时间获取系统重要组件的信息，并输出到上下文(context)，详情参见 11.2.2 节。启用 Hadoop 度量系统，并把它捆绑入 Ganglia 或导出到 JMX，就能得到集群上正在做和刚才做的事情的视图。HBase 也有它自己的度量(比如请求频率、组件计数、资源使用情况等)。相关信息可以参见 HBase *conf* 目录下的 *hadoop-metrics2-properties* 文件。

20.7.4 计数器

在 StumbleUpon 公司(*https://www.stumbleupon.com/*)，第一个在 HBase 上部署的产品特性是为 `stumbleupon.com` 前端维护计数器。计数器以前存储在 MySQL 中，但计数器的更新太频繁，计数器所导致的写操作太多，所以 Web 设计者必须对计数值进行限定。使用 `HTable` 的 `incrementColumnValue()`方法以后，计数器每秒可以实现数千次更新。

20.8 延伸阅读

本章对 HBase 做了简单介绍，并没有深入探讨 HBase 所具有的潜力。有关 HBase

更详细的描述，请参阅 O'Reilly 在 2011 年出版的 *HBase: The DefinitiveGuide*，网址为 *http://hbase.apache.org/book.html*，作者 Lars George，且其新版即将发行，或者 Manning 出版社 2012 年在 *HBase in Action*，网址为 *http://www.manning.com/dimidukkhurana/*，作者 Nick Dimiduk 和 Amandeep Khurana。

第 21 章

关于 ZooKeeper

迄今为止，整本书都是在教我们大规模数据处理技术。但本章的内容则有所不同，将介绍如何使用 ZooKeeper 来构建一般的分布式应用。ZooKeeper 是 Hadoop 的分布式协调服务。

写分布式应用的主要困难在于会出现“部分失败”(partial failure)。当一条消息在网络中两个节点之间传送时，如果出现网络错误，发送者无法知道接收者是否已经收到这条消息。接收者可能在出现网络错误之前就已经收到这条消息，也有可能没有收到，又或者接收者的进程已经死掉。发送者能够获得真实情况的唯一途径就是重新连接接收者，并向它发出询问。这种情况就是部分失败，即我们甚至不知道一个操作是否已经失败。

由于部分失败是分布式系统固有的特征，因此，使用 ZooKeeper 并不能避免出现部分失败，当然它也不会隐藏部分失败。[①] ZooKeeper 可以提供一组工具，使你在构建分布式应用时能够对部分失败进行正确处理。

ZooKeeper 具有以下特点。

- ZooKeeper 是简单的　ZooKeeper 的核心是一个精简的文件系统，它提供一些简单的操作和一些额外的抽象操作，例如，排序和通知。
- ZooKeeper 是富有表现力的　ZooKeeper 的基本操作是一组丰富的“构

① 详情参见 J. Waldo 等人在 1994 年发表的文章，标题为“A Note on Distributed Computing”，网址为 *http://research.sun.com/techrep/1994/smli_tr-94-29.pdf*。分布式编程与本地编程有着根本的不同，不可忽视。

件”(building block)，可用于实现多种协调数据结构和协议。相关的例子包括：分布式队列、分布式锁和一组节点中的“领导者选举”(leader election)。

- ZooKeeper 具有高可用性　ZooKeeper 运行于一组机器之上，并且在设计上具有高可用性，因此应用程序完全可以依赖于它。ZooKeeper 可以帮助系统避免出现单点故障，因此可以用于构建一个可靠的应用程序。
- ZooKeeper 采用松耦合交互方式　在 ZooKeeper 支持的交互过程中，参与者不需要彼此了解。例如，ZooKeeper 可以被用于实现“数据汇集”(rendezvous)机制，让进程在不了解其他进程(或网络状况)的情况下能够彼此发现并进行信息交互。参与的各方甚至可以不必同时存在，因为一个进程可以在 ZooKeeper 中留下一条消息，在该进程结束后，另外一个进程还可以读取这条消息。
- ZooKeeper 是一个资源库　ZooKeeper 提供了一个通用协调模式实现方法的开源共享库，使程序员免于写这类通用的协议(这通常是很难写正确的)。所有人都能够对这个资源库进行添加和改进，久而久之，会使每个人都从中受益。

同时，ZooKeeper 也是高性能的。在它的诞生地 Yahoo!公司，对于以写操作为主的工作负载来说，ZooKeeper 的基准吞吐量已经超过每秒 10000 个操作。对于常规的以读操作为主的工作负载来说，吞吐量更是高出好几倍。[①]

21.1　安装和运行 ZooKeeper

首次尝试使用 ZooKeeper 时，最简单的方式是在一台 ZooKeeper 服务器上以独立模式(standalone mode)运行。例如，可以在一台用于开发的机器上尝试运行。运行 ZooKeeper 需要 Java，因此首先要确认已经安装了 Java。

可以从 Apache 的 ZooKeeper 发布页面(*http://hadoop.apache.org/zookeeper/releases.html*)下载 ZooKeeper 的一个稳定版本，然后在合适的位置将下载的压缩包

① 详细的基准数据可以参见 Patrick Hunt、Mahadev Konar、Flavio P. Junqueira 和 Benjamin Reed 发表的优秀论文，标题为“ZooKeeper: Wait-free Coordination for Internet-Scale Systems”，网址为 *http://bit.ly/wait-free_coordination*，发表于 2010 年 USENIX 年度技术大会。

解压：

```
% tar xzf zookeeper-x.y.z.tar.gz
```

ZooKeeper 提供了几个能够运行服务并与之交互的二进制可执行文件，可以很方便地将包含这些二进制文件的目录加入命令行路径：

```
% export ZOOKEEPER_HOME=~/sw/zookeeper-x.y.z
% export PATH=$PATH:$ZOOKEEPER_HOME/bin
```

在运行 ZooKeeper 服务之前，我们需要创建一个配置文件。这个配置文件习惯上被命名为 *zoo.cfg*，并被保存在 *conf* 子目录中(也可以把它保存在*/etc/zookeeper* 子目录中；如果设置了环境变量 `ZOOCFGDIR`，也可以保存在该环境变量所指定的目录中)。配置文件的内容示例如下：

```
tickTime=2000
dataDir=/Users/tom/zookeeper
clientPort=2181
```

这是一个标准的 Java 属性文件，本例中定义的三个属性是以独立模式运行 ZooKeeper 所需的最低要求。简单地说，`tickTime` 属性指定了 ZooKeeper 中的基本时间单元(以毫秒为单位)；`dataDir` 属性指定了 ZooKeeper 存储持久数据的本地文件系统位置；`clientPort` 属性指定了 ZooKeeper 用于监听客户端连接的端口(通常使用 2181 端口)。用户应该将 `dataDir` 属性的值修改为自己系统所要求的合适位置。

定义好合适的配置文件之后，我们现在可以启动一个本地 ZooKeeper 服务器：

```
% zkServer.sh start
```

使用 `nc`(也可以使用 `telnet`)发送 `ruok` 命令(Are you OK？)到监听端口，检查 ZooKeeper 是否正在运行：

```
% echo ruok | nc localhost 2181
imok
```

`imok` 是 ZooKeeper 在说“I'm OK”。还有其他一些用于管理 ZooKeeper 的命令，都采用类似的四字母组合，如表 21-1 所示。

除了 `mntr` 命令以外，ZooKeeper 还通过 JMX 来披露统计信息。请访问 *http://zookeeper.apache.org/*，找到 ZooKeeper 文档，获取详细的相关信息。在安装目录的 *src/contrib* 子目录中包含有相关的监控工具及方法。

表 21-1. ZooKeeper 命令：四字母组合

类别	命令	描述
服务器状态	ruok	如果服务器正在运行并且未处于出错状态，则输出 imok
	conf	输出服务器的配置信息(根据配置文件 zoo.cfg)
	envi	输出服务器的环境信息，包括 ZooKeeper 版本、Java 版本和其他系统属性
	srvr	输出服务器的统计信息，包括延迟统计、znode 的数量和服务器运行模式(standalone、leader 或 follower)
	stat	输出服务器的统计信息和已连接的客户端
	srst	重置服务器的统计信息
	isro	显示服务器是否处于只读(ro)模式(由于网络分区)，或者读/写(rw)模式
客户端连接	dump	列出集合体中的所有会话和短暂 znode。必须连接到 leader 才能够使用此命令(参考 srvr 命令)
	cons	列出所有服务器客户端的连接统计信息
	crst	重置连接统计信息
观察	wchs	列出服务器上所有观察的摘要信息
	wchc	按连接列出服务器上所有的观察。注意：如果观察的数量较多，此命令会影响服务器的性能
	wchp	按 znode 路径列出服务器上所有的观察。注意：如果观察的数量较多，此命令会影响服务器的性能
监控	mntr	按 Java 属性格式列出服务器统计信息。适合于用作 Ganglia 和 Nagios 等监控系统的信息源

ZeeKeeper 从 3.5.0 版本开始内建了用于提供与“四字母组合”相同信息的 web server。可以访问 *http://localhost:8080/commands*，获取命令列表。

21.2 示例

假设有一组服务器用于为客户端提供某种服务。我们希望每个客户端都能找到其中一台服务器，这样一来，它们就可以使用这项服务。在这个例子中，一个挑战是如何维护这组服务器的成员列表。

这组服务器的成员列表显然不能存储在网络中的单个节点上，否则该节点的故障将意味着整个系统的故障(我们希望这个成员列表是高度可用的)。我们先假设已经有了一种可靠的方法来解决成员列表的存储问题。接下来，如果其中一台服务器出现故障，我们需要解决如何从服务器成员列表中将它删除的问题。某个进程需

要去负责删除故障服务器，但注意不能由故障服务器自己来完成，因为故障服务器已经不再运行！

我们所描述的不是一个被动的分布式数据结构，而是一个主动的、能够在某个外部事件发生时修改数据项状态的数据结构。ZooKeeper 提供了这种服务，接下来让我们看看如何使用它来实现这种众所周知的组成员管理应用的。

21.2.1 ZooKeeper 中的组成员关系

理解 ZooKeeper 的一种方法是将其看作一个具有高可用性特征的文件系统。这个文件系统中没有文件和目录，而是统一使用“节点”(node)的概念，称为 znode。znode 既可以作为保存数据的容器(如同文件)，也可以作为保存其他 znode 的容器(如同目录)。所有的 znode 构成了一个层次化的命名空间，一种自然的建立组成员列表的方式就是利用这种层次结构，创建一个以组名为节点名的 znode 作为父节点，然后以组成员名(服务器名)为节点名来创建作为子节点的 znode。图 21-1 给出了一组具有层次结构的 znode。

图 21-1. ZooKeeper 中的 znode

在这个示例中，我们没有在任何 znode 中存储数据，但在一个真实的应用中，你可以想象将成员相关的数据存储在它们的 znode 中，例如主机名。

21.2.2 创建组

让我们通过写一段程序的方式来介绍 ZooKeeper 的 Java API，这段示例程序用于

创建组名为/*zoo* 的 znode，参见范例 21-1。

范例 21-1. 该程序在 Zookeeper 中新建表示组的 znode

```
public class CreateGroup implements Watcher {

  private static final int SESSION_TIMEOUT = 5000;

  private ZooKeeper zk;
  private CountDownLatch connectedSignal = new CountDownLatch(1);

  public void connect(String hosts) throws IOException, InterruptedException {
    zk = new ZooKeeper(hosts, SESSION_TIMEOUT, this);
    connectedSignal.await();
  }

  @Override
  public void process(WatchedEvent event) { // Watcher interface
    if (event.getState() == KeeperState.SyncConnected) {
      connectedSignal.countDown();
    }
  }

  public void create(String groupName) throws KeeperException,
      InterruptedException {
    String path = "/" + groupName;
    String createdPath = zk.create(path, null/*data*/, Ids.OPEN_ACL_UNSAFE,
        CreateMode.PERSISTENT);
    System.out.println("Created " + createdPath);
  }

  public void close() throws InterruptedException {
    zk.close();
  }

  public static void main(String[] args) throws Exception {
    CreateGroup createGroup = new CreateGroup();
    createGroup.connect(args[0]);
    createGroup.create(args[1]);
    createGroup.close();
  }
}
```

在 `main()`方法执行时，创建一个 `CreateGroup` 的实例然后调用这个实例的 `connect()`方法。`connect` 方法实例化了一个新的 `ZooKeeper` 类的对象，这个类是客户端 API 中的主要类，用于维护客户端和 ZooKeeper 服务之间的连接。ZooKeeper 类的构造函数有三个参数：第一个参数是 ZooKeeper 服务的主机地址

(可指定端口，默认端口是 2181)；[1]第二个参数是以毫秒为单位的会话超时参数(这里我们设成 5 秒)，后文中将给出该参数的详细解释；第三个参数是一个 Watcher 对象的实例。Watcher 对象接收来自于 ZooKeeper 的回调，以获得各种事件的通知。在这个例子中，CreateGroup 是一个 Watcher 对象，因此我们将它传递给 ZooKeeper 构造函数。

当一个 ZooKeeper 的实例被创建时，会启动一个线程连接到 ZooKeeper 服务。由于对构造函数的调用是立即返回的，因此在使用新建的 ZooKeeper 对象前一定要等待其与 ZooKeeper 服务之间成功建立连接。我们使用 Java 的 CountDownLatch 类(位于 java.util.concurrent 包中)来阻止使用新建的 ZooKeeper 对象，直到这个 ZooKeeper 对象已经准备就绪。Watcher 类用于获取 ZooKeeper 对象是否准备就绪的信息，在它的接口中只有一个方法：

```
public void process(WatchedEvent event);
```

当客户端已经与 ZooKeeper 服务建立连接后，Watcher 的 process()方法会被调用，参数是一个用于表示该连接的事件。在接收到一个连接事件(以 Watcher.Event.KeeperState 的枚举型值 SyncConnected 来表示)时，我们通过调用 CountDownLatch 的 countDown()方法来递减它的计数器。锁存器(latch)创建时带有一个值为 1 的计数器，用于表示在它释放所有等待线程之前需要发生的事件数。在调用一次 countDown()方法之后，计数器的值变为 0，则 await()方法返回。

现在 connect()方法已经返回，下一个执行的是 CreateGroup 的 create()方法。在这个方法中，我们使用 ZooKeeper 实例中的 create()方法来创建一个新的 ZooKeeper 的 znode。所需的参数包括：路径(用字符串表示)、znode 的内容(字节数组，本例中使用空值)、访问控制列表(简称 ACL，本例中使用了完全开放的 ACL，允许任何客户端对 znode 进行读/写)和创建 znode 的类型。

有两种类型的 znode：**短暂的**(ephemeral)和**持久的**(persistent)。创建 znode 的客户端断开连接时，无论客户端是明确断开还是因为任何原因而终止，短暂 znode 都会被 ZooKeeper 服务删除。与之相反，当客户端断开连接时，持久 znode 不会被删除。我们希望代表一个组的 znode 存活的时间应当比创建程序的生命周期要长，因此在本例中我们创建了一个持久的 znode。

① 对于复制模式下的 ZooKeeper 服务来说，这个参数是一个以逗号分隔的服务器(主机和可选端口)列表。

create()方法的返回值是 ZooKeeper 所创建的节点路径，我们用这个返回值来打印一条表示节点路径被成功创建的消息。当我们查看“顺序 znode”(sequential znode)时，会发现 create()方法返回的路径与传递给该方法的路径不同。

为了观察程序的执行，我们需要在本地机器上运行 ZooKeeper，然后可以输入以下命令：

```
% export CLASSPATH=ch21-zk/target/classes/:$ZOOKEEPER_HOME/*:\
  $ZOOKEEPER_HOME/lib/*:$ZOOKEEPER_HOME/conf
% java CreateGroup localhost zoo
Created /zoo
```

21.2.3 加入组

这个应用的下一部分是一段用于注册组成员的程序。每个组成员将作为一个程序运行，并且加入到组中。当程序退出时，这个组成员应当从组中被删除。为了实现这一点，我们在 ZooKeeper 的命名空间中使用短暂 znode 来代表一个组成员。

范例 21-2 中的程序 JoinGroup 实现了这个想法。在基类 ConnectionWatcher 中，对创建和连接 ZooKeeper 实例的程序逻辑进行了重构，如范例 21-3 所示。

范例 21-2. 该程序将成员加入组

```
public class JoinGroup extends ConnectionWatcher {

  public void join(String groupName, String memberName) throws KeeperException,
      InterruptedException {
    String path = "/" + groupName + "/" + memberName;
    String createdPath = zk.create(path, null/*data*/, Ids.OPEN_ACL_UNSAFE,
      CreateMode.EPHEMERAL);
    System.out.println("Created " + createdPath);
  }

  public static void main(String[] args) throws Exception {
    JoinGroup joinGroup = new JoinGroup();
    joinGroup.connect(args[0]);

    joinGroup.join(args[1], args[2]);

    // stay alive until process is killed or thread is interrupted
    Thread.sleep(Long.MAX_VALUE);
  }
}
```

范例 21-3. 该辅助类等待与 ZooKeeper 建立连接

```
public class ConnectionWatcher implements Watcher {

  private static final int SESSION_TIMEOUT = 5000;

  protected ZooKeeper zk;
  private CountDownLatch connectedSignal = new CountDownLatch(1);

  public void connect(String hosts) throws IOException, InterruptedException {
    zk = new ZooKeeper(hosts, SESSION_TIMEOUT, this);
    connectedSignal.await();
  }

  @Override
  public void process(WatchedEvent event) {
    if (event.getState() == KeeperState.SyncConnected) {
    connectedSignal.countDown();
  }
}

public void close() throws InterruptedException {
    zk.close();
  }
}
```

JoinGroup 的代码与 CreateGroup 的非常相似。在它的 join()方法中，创建短暂 znode 作为组 znode 的子节点，然后通过休眠来模拟正在做某种工作，直到该进程被强行终止。接着，你会看到随着进程终止，这个短暂 znode 被 ZooKeeper 删除。

21.2.4 列出组成员

现在，我们需要一段程序来查看组成员(参见范例 21-4)。

范例 21-4. 用于列出组成员的程序

```
public class ListGroup extends ConnectionWatcher {

  public void list(String groupName) throws KeeperException, InterruptedException {
    String path = "/" + groupName;

  try {
    List<String> children = zk.getChildren(path, false);
    if (children.isEmpty()) {
      System.out.printf("No members in group %s\n", groupName);
      System.exit(1);
    }
    for (String child : children) {
      System.out.println(child);
```

```
    }
  } catch (KeeperException.NoNodeException e) {
    System.out.printf("Group %s does not exist\n", groupName);
    System.exit(1);
  }
}

public static void main(String[] args) throws Exception {
  ListGroup listGroup = new ListGroup();
  listGroup.connect(args[0]);
  listGroup.list(args[1]);
  listGroup.close();
}
}
```

在 list()方法中，我们调用了 getChildren()方法来检索并打印输出一个 znode 的子节点列表，调用参数为该 znode 的路径和观察标志。如果在一个 znode 上设置了观察标志，那么一旦该 znode 的状态改变，关联的观察(Watcher)会被触发。虽然在这里我们没有使用观察，但在查看一个 znode 的子节点时，通过设置观察可以让应用程序接收到组成员加入、退出和组被删除的有关通知。

在这段程序中，我们捕捉了 KeeperException.NoNodeException 异常，代表组的 znode 不存在时，这个异常就会被抛出。

让我们看看 ListGroup 程序是如何工作的。起初，由于我们还没有在组中添加任何成员，因此 zoo 组是空的：

```
% java ListGroup localhost zoo
No members in group zoo
```

我们可以使用 JoinGroup 来向组中添加成员。由于这些作为组成员的 znode 不会自己终止(因为 sleep 语句)，所以我们以后台进程的方式来启动它们：

```
% java JoinGroup localhost zoo duck &
% java JoinGroup localhost zoo cow &
% java JoinGroup localhost zoo goat &
% goat_pid=$!
```

最后一行命令保存了将 goat 添加到组中的 Java 进程的 ID。我们需要保存这个进程 ID，以便能够在查看组成员之后杀死该进程：

```
% java ListGroup localhost zoo
goat
duck
cow
```

为了从组中删除一个成员，我们杀死了 goat 所对应的进程：

```
% kill $goat_pid
```

几秒钟之后，由于该进程的 ZooKeeper 会话已经结束(超时设置为 5 秒)，并且所对应的短暂 znode 也已经被删除，所以 goat 会从组成员列表中消失。

```
% java ListGroup localhost zoo
duck
cow
```

让我们回顾一下，看看已经实现了哪些功能。对于参与到一个分布式系统中的节点，我们已经有了一个建立节点列表的方法。这些节点相互之间并不了解。例如，一个想使用列表中节点来完成某些工作的客户端，能够在这些节点不知情的情况下发现它们。

最后要注意的是组成员关系管理并不能解决与节点通信过程中出现的网络问题。在与一个组中的成员节点进行通信的过程中可能会出现故障，这些故障必须以一种合适的方式来解决(重试、使用组中另外一个成员等)。

ZooKeeper 命令行工具

ZooKeeper 提供了一个用于与其命名空间进行交互的命令行工具。我们可以使用这个工具列出*/zoo znode* 之下的 znode 列表，如下所示：

```
% zkCli.sh -server localhost ls /zoo
[cow, duck]
```

不使用任何参数直接运行这个命令行工具，可以显示该工具的使用帮助。

21.2.5 删除组

为了使这个例子比较完整，让我们来看看如何删除一个组。ZooKeeper 类提供了一个 delete()方法，该方法有两个参数：节点路径和版本号。如果所提供的版本号与 znode 的版本号一致，ZooKeeper 会删除这个 znode。这是一种乐观的加锁机制，使客户端能够检测出对 znode 的修改冲突。通过将版本号设置为-1，可以绕过这个版本检测机制，不管 znode 的版本号是什么而直接将其删除。

ZooKeeper 不支持递归的删除操作，因此在删除父节点之前必须先删除子节点。在范例 21-5 中，DeleteGroup 类用于删除一个组及其所有成员。

范例 21-5. 用于删除一个组及其所有成员的程序

```
public class DeleteGroup extends ConnectionWatcher {
```

```
public void delete(String groupName) throws KeeperException,
    InterruptedException {
  String path = "/" + groupName;

  try {
    List<String> children = zk.getChildren(path, false);
    for (String child : children) {
      zk.delete(path + "/" + child, -1);
    }
    zk.delete(path, -1);
  } catch (KeeperException.NoNodeException e) {
    System.out.printf("Group %s does not exist\n", groupName);
    System.exit(1);
  }
}

public static void main(String[] args) throws Exception {
  DeleteGroup deleteGroup = new DeleteGroup();
  deleteGroup.connect(args[0]);
  deleteGroup.delete(args[1]);
  deleteGroup.close();
}
}
```

最后，我们可以删除之前所创建的 zoo 组：

```
% java DeleteGroup localhost zoo
% java ListGroup localhost zoo
Group zoo does not exist
```

21.3 ZooKeeper 服务

ZooKeeper 是一个具有高可用性的高性能协调服务。在本小节中，我们将从三个方面来了解这个服务：模型、操作和实现。

21.3.1 数据模型

ZooKeeper 维护着一个树形层次结构，树中的节点被称为 znode。znode 可以用于存储数据，并且有一个与之相关联的 ACL。ZooKeeper 被设计用来实现协调服务(这类服务通常使用小数据文件)，而不是用于大容量数据存储，因此一个 znode 能存储的数据被限制在 1 MB 以内。

ZooKeeper 的数据访问具有原子性。客户端在读取一个 znode 的数据时，要么读到所有的数据，要么读操作失败，不会只读到部分数据。同样，一个写操作将替换

znode 存储的所有数据。ZooKeeper 会保证写操作不成功就失败，不会出现部分写之类的情况，也就是不会出现只保存客户端所写部分数据的情况。ZooKeeper 不支持添加操作。这些特征都是与 HDFS 所不同的。HDFS 被设计用于大容量数据存储，支持流式数据访问和添加操作。

znode 通过路径被引用。像 Unix 中的文件系统路径一样，在 ZooKeeper 中路径被表示成用斜杠分隔的 Unicode 字符串。与 Unix 中的文件系统路径不同的是，ZooKeeper 中的路径必须是绝对路径，也就是说每条路径必须从一个斜杠字符开始。此外，所有的路径表示必须是规范的，即每条路径只有唯一的一种表示方式，不支持路径解析。例如，在 Unix 中，一个具有路径*/a/b* 的文件也可以通过路径*/a/./b* 来表示，原因在于“.”在 Unix 的路径中表示当前目录(“..”表示当前目录的上一级目录)。在 ZooKeeper 中，“.”不具有这种特殊含义，这样表示的路径名是不合法的。

在 ZooKeeper 中，路径由 Unicode 字符串构成，并且有一些限制(参见 ZooKeeper 的参考文档)。字符串“zookeeper”是一个保留词，不能将它作为路径表示中的一部分。需要特别指出的是，ZooKeeper 使用*/zookeeper* 子树来保存管理信息，例如关于配额的信息。

注意，ZooKeeper 的路径与 URI 不同，前者在 Java API 中通过 `java.lang.String` 来使用，而后者是通过 Hadoop Path 类(或 `java.net.URI` 类)来使用。

znode 有一些性质非常适合用于构建分布式应用，我们将在接下来的几个小节中进行讨论。

1. 短暂 znode

znode 有两种类型：短暂的和持久的。znode 的类型在创建时被确定并且之后不能再修改。在创建短暂 znode 的客户端会话结束时，ZooKeeper 会将该短暂 znode 删除。相比之下，持久 znode 不依赖于客户端会话，只有当客户端(不一定是创建它的那个客户端)明确要删除该持久 znode 时才会被删除。短暂 znode 不可以有子节点，即使是短暂子节点。

虽然每个短暂 znode 都会被绑定到一个客户端会话，但它们对所有的客户端还是可见的(当然，还是要符合其 ACL 的定义)。

对于那些需要知道特定时刻有哪些分布式资源可用的应用来说，使用短暂 znode

是一种理想的选择。本章前面的例子就使用了短暂 znode 来实现一个组成员管理服务，让任何进程都知道在特定的时刻有哪些组成员可用。

2. 顺序号

顺序(sequential)znode 是指名称中包含 ZooKeeper 指定顺序号的 znode。如果在创建 znode 时设置了顺序标识，那么该 znode 名称之后便会附加一个值，这个值是由一个单调递增的计数器(由父节点维护)所添加的。

例如，如果一个客户端请求创建一个名为*/a/b-*的顺序 znode，则所创建 znode 的名字可能是*/a/b-3*。[①]如果稍后，另外一个名为*/a/b-*的顺序 znode 被创建，计数器会给出一个更大的值来保证 znode 名称的唯一性，例如，*/a/b-5*。在 Java 的 API 中，顺序 znode 的实际路径会作为 `create()`调用的返回值被传回客户端。

在一个分布式系统中，顺序号可以被用于为所有的事件进行全局排序，这样客户端就可以通过顺序号来推断事件的顺序。21.4.3 节介绍了如何使用顺序 znode 来实现共享锁。

3. 观察

znode 以某种方式发生变化时，“观察”(watch)机制可以让客户端得到通知。可以针对 ZooKeeper 服务的操作来设置观察，该服务的其他操作可以触发观察。例如，客户端可以对一个 znode 调用 `exists` 操作，同时设定一个观察。如果这个 znode 不存在，则客户端所调用的 `exists` 操作将返回 `false`。如果一段时间之后，另外一个客户端创建了这个 znode，则这个观察会被触发，通知前一个客户端这个 znode 被创建。在下一小节中，将完整介绍哪些操作会触发其他操作。

观察只能够触发一次[②]。为了能够多次收到通知，客户端需要重新注册所需的观察。在前面的例子中，如果客户端希望收到更多 znode 是否存在的通知(例如在这个 znode 被删除时也能收到通知)，则需要再次调用 `exists` 操作，设定一个新的观察。

① 习惯上(并非必需)在顺序 znode 的路径名后会跟一个连字符，使其顺序号更易读并且易于(被应用程序)解析。
② 对连接事件的回调除外，这种观察不需要重新注册。

在 21.4.1 节中，将有一个例子来演示如何使用观察来更新集群的配置。

21.3.2 操作

如表 21-2 所示，ZooKeeper 中有 9 种基本操作。

表 21-2. ZooKeeper 服务的操作

操作	描述
create	创建一个 znode(必须要有父节点)
delete	删除一个 znode(该 znode 不能有任何子节点)
exists	测试一个 znode 是否存在并且查询它的元数据
getACL, setACL	获取/设置一个 znode 的 ACL
getChildren	获取一个 znode 的子节点列表
getData, setData	获取/设置一个 znode 所保存的数据
sync	将客户端的 znode 视图与 ZooKeeper 同步

ZooKeeper 中的更新操作是有条件的。在使用 `delete` 或 `setData` 操作时必须提供被更新 znode 的版本号(可以通过 `exists` 操作获得)。如果版本号不匹配，则更新操作会失败。更新操作是非阻塞操作，因此一个更新失败的客户端(由于其他进程同时在更新同一个 znode)可以决定是否重试，或执行其他操作，并不会因此而阻塞其他进程的执行。

虽然 ZooKeeper 可以被看作是一个文件系统，但出于简单性的需要，有一些文件系统的基本操作被它摒弃了。由于 ZooKeeper 中的文件较小并且总是被整体读/写，因此没有必要提供打开、关闭或查找操作。

`sync` 操作与 POSIX 文件系统中的 `fsync()`操作是不同的。如前所述，ZooKeeper 中的写操作具有原子性，一个成功的写操作会保证将数据写到 ZooKeeper 服务器的持久存储介质中。然而，ZooKeeper 允许客户端读到的数据滞后于 ZooKeeper 服务的最新状态，因此客户端可以使用 `sync` 操作来获取数据的最新状态。相关详情请参见 21.3.4 节。

1. 集合更新

ZooKeeper 中有一个被称为 `multi` 的操作(Multiupdate)，用于将多个基本操作集合成一个操作单元，并确保这些基本操作同时被成功执行，或者同时失败，不会发生其中部分基本操作被成功执行而其他基本操作失败的情况。

集合更新可以被用于在 ZooKeeper 中构建需要保持全局一致性的数据结构，例如构建一个无向图。在 ZooKeeper 中用一个 znode 来表示无向图中的一个顶点，为了在两个顶点之间添加或删除一条边，我们需要同时更新两个顶点所分别对应的两个 znode，因为每个 znode 中都有指向对方的引用。如果我们只用 ZooKeeper 的基本操作来实现边的更新，可能会让其他客户端发现无向图处于不一致的状态，即一个顶点具有指向另一个顶点的引用而对方却没有对应的引用。将针对两个 znode 的更新操作集合到一个 multi 操作中可以保证这组更新操作的原子性，也就保证了一对顶点之间不会出现不完整的连接。

2. 关于 API

对于 ZooKeeper 客户端来说，主要有两种语言绑定(binding)可以使用：Java 和 C；当然也可以使用 Perl、Python 和 REST 的 contrib 绑定。对于每一种绑定语言来说，在执行操作时都可以选择同步执行或异步执行。我们已经看过同步执行的 Java API。下面是 exists 操作的签名，它返回一个封装有 znode 元数据的 Stat 对象(如果 znode 不存在，则返回 null)：

```
public Stat exists(String path, Watcher watcher) throws KeeperException,
        InterruptedException
```

在 ZooKeeper 类中同样可以找到异步执行的签名，如下所示：

```
public void exists(String path, Watcher watcher, StatCallback cb, Object ctx)
```

因为所有异步操作的结果都是通过回调来传送的，因此在 Java API 中异步方法的返回类型都是 void。调用者传递一个回调的实现，当 ZooKeeper 响应时，该回调方法被调用。在这种情况下，回调采用 StatCallback 接口，它有以下方法：

```
public void processResult(int rc, String path, Object ctx, Stat stat);
```

其中 rc 参数是返回代码，对应于 KeeperException 的代码。每个非零代码都代表一个异常，在这种情况下，stat 参数是 null。path 和 ctx 参数对应于客户端传递给 exists()方法的参数，用于识别这个回调所响应的请求。ctx 参数可以是任意对象，当 path 参数不能提供足够的信息时，客户端可以通过 ctx 参数来区分不同请求。如果 path 参数提供了足够的信息，可以将 ctx 参数设成 null。

实际上，有两个 C 语言的共享库。单线程库 zookeeper_st 只支持异步 API，并且主要在没有 pthread 库或 pthread 库不稳定的平台上使用。大部分开发人员都使用多线程库 zookeeper_mt，它既支持同步 API 也支持异步 API。要想进一步

了解如何构建和使用 C 语言 API，请参考 ZooKeeper 安装目录下 *src/c* 子目录中的 *README* 文件。

我该使用同步 API 还是异步 API?

两种类型的 API 提供相同的功能，因此选择哪一种只是风格问题。例如，如果你习惯于事件驱动的编程模型，则异步 API 更合适一些。

异步 API 允许你以流水线方式处理请求，这在某些情况下可以提供更好的吞吐量。想象一下，你打算读取一大批 znode 并且分别对它们进行处理。如果使用同步 API，每一个读操作都会阻塞进程，直到该读操作返回；但如果使用异步 API，你可以非常快速地启动所有的异步读操作并在另外一个单独的线程中来处理读操作的返回。

3. 观察触发器

在 `exists`、`getChildren` 和 `getData` 这些读操作上可以设置观察，这些观察可以被写操作 `create`、`delete` 和 `setData` 触发。ACL 相关的操作不参与触发任何观察。当一个观察被触发时会产生一个观察事件，这个观察和触发它的操作共同决定着观察事件的类型。

- 当所观察的 znode 被创建、删除或其数据被更新时，设置在 `exists` 操作上的观察将被触发。
- 当所观察的 znode 被删除或其数据被更新时，设置在 `getData` 操作上的观察将被触发。创建 znode 不会触发 `getData` 操作上的观察，因为 `getData` 操作成功执行的前提是 znode 必须已经存在。
- 所观察的 znode 的一个子节点被创建或删除时，或所观察的 znode 自己被删除时，设置在 `getChildren` 操作上的观察将会被触发。可以通过观察事件的类型来判断被删除的是 znode 还是其子节点：`NodeDelete` 类型代表 znode 被删除；`NodeChildrenChanged` 类型代表一个子节点被删除。

表 21-3 列出了观察及其触发操作所对应的事件类型。

表 21-3. 观察及其触发操作所对应的事件类型

设置观察的操作	观察触发器				
	创建 znode	创建子节点	删除 znode	删除子节点	setData
exists	NodeCreated		NodeDeleted		NodeData Changed
getData			NodeDeleted		NodeData Changed
getChildren		NodeChildren Changed	NodeDeleted	NodeChildren Changed	

一个观察事件中包含涉及该事件的 znode 的路径，因此对于 NodeCreated 和 NodeDeleted 事件来说，可通过路径来判断哪一个节点被创建或删除。为了能够在 NodeChildrenChanged 事件发生之后判断是哪些子节点被修改，需要重新调用 getChildren 来获取新的子节点列表。与之类似，为了能够在 NodeDataChanged 事件之后获取新的数据，需要调用 getData。在这两种情况下，从收到观察事件到执行读操作(getChildren 或 getData)期间，znode 的状态可能会发生改变，在写程序的时候必须牢记这一点。

4. ACL 列表

每个 znode 创建时都会带有一个 ACL 列表，用于决定谁可以对它执行何种操作。

ACL 依赖于 ZooKeeper 的客户端身份验证机制。ZooKeeper 提供了以下几种身份验证方式：

- digest　通过用户名和密码来识别客户端
- sasl　通过 Kerberos 来识别客户端
- ip　通过客户端的 IP 地址来识别客户端

在建立一个 ZooKeeper 会话之后，客户端可以对自己进行身份验证。虽然 znode 的 ACL 列表会要求所有的客户端是经过验证的，但 ZooKeeper 的身份验证过程却是可选的，客户端必须自己进行身份验证来支持对 znode 的访问。这里有一个使用 digest 方式(用户名和密码)进行身份验证的例子：

```
zk.addAuthInfo("digest", "tom:secret".getBytes());
```

每个 ACL 都是身份验证方式、符合该方式的一个身份和一组权限的组合。例如，如果我们打算给 IP 地址为 10.0.0.1 的客户端对某个 znode 的读权限，可以使用

ip 验证方式、10.0.0.1 和 READ 权限在该 znode 上设置一个 ACL。在 Java 语言中，我们可以如下所示来创建这个 ACL 对象：

```
new ACL(Perms.READ,
new Id("ip", "10.0.0.1"));
```

表 21-4 列出了一个完整的权限集合。注意，exists 操作并不受 ACL 权限的限制，因此任何客户端都可以调用 exists 来检索一个 znode 的状态或查询一个 znode 是否存在。

表 21-4. ACL 权限

ACL 权限	允许的操作
CREATE	create(子节点)
READ	getChildren
	getData
WRITE	setData
DELETE	delete(子节点)
ADMIN	setACL

在类 ZooDefs.Ids 中有一些预定义的 ACL，OPEN_ACL_UNSAFE 是其中之一，它将所有的权限(不包括 ADMIN 权限)授予每个人。

此外，ZooKeeper 还支持插入式身份验证机制，如果需要的话，它可以集成第三方的身份验证系统。

21.3.3 实现

ZooKeeper 服务有两种不同的运行模式。一种是**独立模式**(standalone mode)，即只有一个 ZooKeeper 服务器。这种模式较为简单，比较适合于测试环境(甚至可以在单元测试中采用)，但是不能保证高可用性和可恢复性。在生产环境中的 ZooKeeper 通常以**复制模式**(replicated mode)运行于一个计算机集群上，这个计算机集群被称为一个**集合体**(ensemble)。ZooKeeper 通过复制来实现高可用性，只要集合体中半数以上的机器处于可用状态，它就能够提供服务。例如，在一个有 5 个节点的集合体中，任意 2 台机器出现故障，都可以保证服务继续，因为剩下的 3 台机器超过了半数。注意，6 个节点的集合体也只能够容忍 2 台机器出现故障，因为如果 3 台机器出现故障，剩下的 3 台机器没有超过集合体的半数。出于这个原因，一个集合体通常包含奇数台机器。

从概念上来说，ZooKeeper 是非常简单的：它所做的就是确保对 znode 树的每一个修改都会被复制到集合体中超过半数的机器上。如果少于半数的机器出现故障，则最少有一台机器会保存最新的状态，其余的副本最终也会更新到这个状态。

然而，这个简单想法的实现却不简单。ZooKeeper 使用了 Zab 协议，该协议包括两个可以无限重复的阶段。

1. 阶段 1：领导者选举

集合体中的所有机器通过一个选择过程来选出一台被称为**领导者**(leader)的机器，其他的机器被称为**跟随者**(follower)。一旦半数以上(或指定数量)的跟随者已经将其状态与领导者同步，则表明这个阶段已经完成。

2. 阶段 2：原子广播

所有的写请求都会被转发给领导者，再由领导者将更新广播给跟随者。当半数以上的跟随者已经将修改持久化之后，领导者才会提交这个更新，然后客户端才会收到一个更新成功的响应。这个用来达成共识的协议被设计成具有原子性，因此每个修改要么成功要么失败。这类似于数据库中的两阶段提交协议。

ZooKeeper 是否使用 Paxos？

否。ZooKeeper 的 Zab 协议不同于众所周知的 Paxos 算法[①]。虽然有些类似，但是 Zab 在操作方面是不同的，例如它依靠 TCP 来保证其消息的顺序。[②]

Google 的 Chubby 锁服务[③]是基于 Paxos 的，其功能与 ZooKeeper 的功能类似。

如果领导者出现故障，其余的机器会选出另外一个领导者，并和新的领导者一起继续提供服务。随后，如果之前的领导者恢复正常，会成为一个跟随者。领导者选举的过程是非常快的，根据一个已公布的结果(*http://bit.ly/dist_coordination*)来

① 参见图灵奖得主 Leslie Lamport 的文章，标题为“Paxos Made Simple”，发表于 2001 年 12 月的 ACM SIGACT News，网址为 *http://bit.ly/simple-paxos*。

② 有关 Zab 的描述可以参见由 Benjamin Reed 和 Flavio Junqueira 的文章，标题为“A simple totally ordered broadcast protocol”，发布于 2008 年大规模分布式系统和中间件工作坊，LADIS '08 Proceedings of the 2nd Workshop，*http://bit.ly/ordered_protocol*。

③ 参见 Mike Burrows 的文章，标题为“The Chubby Lock Service for Loosely-Coupled Distributed Systems”，发布于 2000 年 11 月，网址为 *http://research.google.com/archive/chubby.html*。

看，只需要大约 200 毫秒，因此在领导者选举的过程中不会出现系统性能的明显降低。

在更新内存中的 znode 树之前，集合体中的所有机器都会先将更新写入磁盘。任何一台机器都可以为读请求提供服务，并且由于读请求只涉及内存检索，因此非常快。

21.3.4 一致性

理解 ZooKeeper 的实现有助于理解其服务所提供的一致性保证。在集合体中所使用的术语“领导者”和“跟随者”是恰当的，它们表明一个跟随者可能滞后于领导者几个更新。这也表明在一个修改被提交之前，只需要集合体中半数以上机器已经将该修改持久化即可。对 ZooKeeper 来说，理想的情况就是将客户端都连接到与领导者状态一致的服务器上。每个客户端都有可能被连接到领导者，但客户端对此无法控制，甚至它自己都无法知道是否连接到领导者，[①]参见图 21-2。

图 21-2. 跟随者负责响应读请求，领导者负责提交写请求

① 可以对 ZooKeeper 进行配置，使领导者不接受任何客户端连接。在这种情况下，领导者的唯一任务就是协调更新。可以通过将 leaderServes 属性设置为 no 来实现这一点。推荐在超过 3 台服务器的集群中使用该设置。

每一个对 znode 树的更新都被赋予一个全局唯一的 ID，称为 *zxid*(代表“ZooKeeper Transaction ID”)。ZooKeeper 要求对所有的更新进行编号并排序，它决定了分布式系统的执行顺序，如果 *zxid* z_1 小于 z_2，则 z_1 一定发生在 z_2 之前。

在 ZooKeeper 的设计中，以下几点考虑保证了数据的一致性。

1. 顺序一致性

来自任意特定客户端的更新都会按其发送顺序被提交。也就是说，如果一个客户端将 znode z 的值更新为 a，在之后的操作中，它又将 z 的值更新为 b，则没有客户端能够在看到 z 的值是 b 之后再看到值 a(如果没有其他对 z 的更新)。

2. 原子性

每个更新要么成功，要么失败。这意味着如果一个更新失败，则不会有客户端看到这个更新的结果。

3. 单一系统映像

一个客户端无论连接到哪一台服务器，它看到的都是同样的系统视图。这意味着，如果一个客户端在同一个会话中连接到一台新的服务器，它所看到的系统状态不会比在之前服务器上所看到的更老。当一台服务器出现故障，导致它的一个客户端需要尝试连接集合体中其他的服务器时，所有状态滞后于故障服务器的服务器都不会接受该连接请求，除非这些服务器将状态更新至故障服务器的水平。

4. 持久性

一个更新一旦成功，其结果就会持久存在并且不会被撤销。这表明更新不会受到服务器故障的影响。

5. 及时性

任何客户端所看到的滞后系统视图都是有限的，不会超过几十秒。这意味着与其允许一个客户端看到非常陈旧的数据，还不如将服务器关闭，强迫该客户端连接到一个状态较新的服务器。

出于性能的原因，所有的读操作都是从 ZooKeeper 服务器的内存获得数据，它们不参与写操作的全局排序。如果客户端之间通过 ZooKeeper 之外的机制进行通信，则客户端可能会发现它们所看到的 ZooKeeper 状态是不一致的。例如，客户

端 A 将 znode *z* 的值从 *a* 更新为 *a*'，接着 A 告诉 B 去读 *z* 的值，而 B 读到的值是 *a* 而不是 *a*'。这与 ZooKeeper 的一致性保证是完全兼容的(这种情况称为“跨客户端视图的同时一致性”)。为了避免这种情况发生，B 应该在读 *z* 的值之前对 *z* 调用 sync 操作。sync 操作会强制 B 所连接的 ZooKeeper 服务器“赶上”领导者，这样当 B 读 *z* 的值时，所读到的将会是 A 所更新的(或后来更新的)。

容易让人疑惑的是，sync 操作只能以异步的方式被调用。你不需要等待 sync 调用的返回，ZooKeeper 会保证任何后续的操作都在服务器的 sync 操作完成后才执行，哪怕这些操作是在 sync 操作完成之前发出的。

21.3.5 会话

每个 ZooKeeper 客户端的配置中都包括集合体中服务器的列表。在启动时，客户端会尝试连接到列表中的一台服务器。如果连接失败，它会尝试连接另一台服务器，以此类推，直到成功与一台服务器建立连接或因为所有 ZooKeeper 服务器都不可用而失败。

一旦客户端与一台 ZooKeeper 服务器建立连接，这台服务器就会为该客户端创建一个新的会话。每个会话都会有一个超时的时间设置，这个设置由创建会话的应用来设定。如果服务器在超时时间段内没有收到任何请求，则相应的会话会过期。一旦一个会话已经过期，就无法重新被打开，并且任何与该会话相关联的短暂 znode 都会丢失。会话通常都会长期存在，而会话过期则是一种比较罕见的事件，但对于应用来说，如何处理会话过期仍是非常重要的，详情可以参见 21.4.2 节。

只要一个会话空闲超过一定时间，都可以通过客户端发送 ping 请求(也称为心跳)来保持会话不过期。(ping 请求是由 ZooKeeper 的客户端库自动发送，因此在你的代码中不需要考虑如何维护会话。)这个时间长度的设置应当足够低，以便能够检测出服务器故障(由读超时体现)，并且能够在会话超时的时间段内重新连接到另外一台服务器。

ZooKeeper 客户端可以自动地进行故障切换，切换至另一台 ZooKeeper 服务器，并且关键的是，在另一台服务器接替故障服务器之后，所有的会话(和相关的短暂 znode)仍然是有效的。

在故障切换过程中，应用程序将收到断开连接和连接至服务的通知。当客户端断开连接时，观察通知将无法发送；但是当客户端成功恢复连接后，这些延迟的通知还会被发送。当然，在客户端重新连接至另一台服务器的过程中，如果应用程

序试图执行一个操作，这个操作将会失败。这充分说明在真实的 ZooKeeper 应用中处理连接丢失异常的重要性，详情可以参见 21.4.2 节。

时间

在 ZooKeeper 中有几个时间参数。“滴答”(tick time)参数定义了 ZooKeeper 中的基本时间周期，并被集合体中的服务器用来定义相互交互的时间表。其他设置都是根据滴答参数来定义的，或至少受它限制。例如，会话超时(session timeout)参数的值不可以小于 2 个滴答并且不可以大于 20 个滴答。如果你试图将会话超时参数设置在这个范围之外，它将会被自动修改到这个范围之内。

通常将滴答参数设置为 2 秒(2000 毫秒)，对应于允许的会话超时范围是 4 到 40 秒。在选择会话超时设置时有几点需要考虑。

较短的会话超时设置会较快地检测到机器故障。在组成员管理的例子中，会话超时的时间就是用来将故障机器从组中删除的时间。但要避免将会话超时时间设得太低，因为繁忙的网络会导致数据包传输延迟，从而可能会无意中导致会话过期。在这种情况下，机器可能会出现“振动”(flap)现象：在很短的时间内反复出现离开后又重新加入组的情况。

对于那些创建较复杂暂时状态的应用程序来说，由于重建的代价较大，因此比较适合设置较长的会话超时。在某些情况下，可以对应用程序进行设计，使它能够在会话超时之前重启，从而避免出现会话过期的情况(这适合于对应用进行维护或升级)。服务器会为每个会话分配一个唯一的 ID 和密码，如果在建立连接的过程中将它们传递给 ZooKeeper，可以用于恢复一个会话(只要该会话没有过期)。将会话 ID 和密码保存在稳定存储器中之后，可以将一个应用程序正常关闭，然后在重启应用之前凭借所保存的会话 ID 和密码来恢复会话环境。

你可以将这个特征看成是一种用来帮助避免会话过期的优化技术，但不能因此忽略对会话过期异常的处理，因为机器的意外故障也会导致会话过期，或者，即使应用程序是正常关闭，也有可能因任何原因而导致它没有在会话未过期之前完成重启。

一般的规则是，ZooKeeper 集合体中的服务器越多，会话超时的设置应越大。连接超时、读超时和 ping 周期都被定义为集合体中服务器数量的函数，因此集合体中服务器数量越多，这些参数的值反而越小。如果频繁遇到连接丢失的情况，应考虑增大超时的设置。可以使用 JMX 来监控 ZooKeeper 的度量指标，例如请求延迟

的统计信息。

21.3.6 状态

ZooKeeper 对象在其生命周期中会经历几种不同的状态(参见图 21-3)。你可以在任何时刻通过 getState()方法来查询对象的状态：

```
public States getState()
```

States 被定义成代表 ZooKeeper 对象不同状态的枚举类型值(不管是什么枚举值，一个 ZooKeeper 的实例在一个时刻只能处于一种状态)。在试图与 ZooKeeper 服务建立连接的过程中，一个新建的 ZooKeeper 实例处于 CONNECTING 状态。一旦建立连接，它就会进入 CONNECTED 状态。

图 21-3. ZooKeeper 状态转换图

通过注册观察对象，使用了 ZooKeeper 对象的客户端就可以收到状态转换通知。一旦进入 CONNECTED 状态，观察对象就会收到一个 WatchedEvent 通知，其中 KeeperState 的值是 SyncConnected。

ZooKeeper 的 watcher 对象肩负着双重责任：一方面它可以被用于获得 ZooKeeper 状态变化的相关通知(如本节所述)；另一方面还可以被用于获得 znode 变化的相关通知(参见 21.3.2 节对观察触发器的讨论)。传递给 ZooKeeper 对象构造函数的(默认的)观察被用于监视其状态的变化。监视 znode 的变化可以使用一个专用的观察对象(将其传递给适当的读操作)，也可以通过读操作中的布尔标识来设定是否共享使用默认的观察。

ZooKeeper 实例可以断开然后重新连接到 ZooKeeper 服务，此时它的状态就在 CONNECTED 和 CONNECTING 之间转换。如果它断开连接，观察会收到一个 Disconnected 事件。注意，这些状态转换都是由 ZooKeeper 实例自己发起的，如果连接丢失，它会自动尝试重新连接。

如果 close()方法被调用或出现会话超时(观察事件的 KeeperState 值为 Expired)时，ZooKeeper 实例会转换到第三个状态 CLOSED。一旦处于 CLOSED 状态，ZooKeeper 对象不再被认为是活跃的(可以对 States 使用 isAlive()方法来测试)，并且不能再用。为了重新连接到 ZooKeeper 服务，客户端必须创建一个新的 ZooKeeper 实例。

21.4 使用 ZooKeeper 来构建应用

在一定程度上了解 ZooKeeper 之后，我们接下来用 ZooKeeper 写一些有用的应用程序。

21.4.1 配置服务

配置服务是分布式应用所需要的基本服务之一，它使集群中的机器可以共享配置信息中那些公共的部分。简单地说，ZooKeeper 可以作为一个具有高可用性的配置存储器，允许分布式应用的参与者检索和更新配置文件。使用 ZooKeeper 中的观察机制，可以建立一个活跃的配置服务，使那些感兴趣的客户端能够获得配置信息修改的通知。

让我们来写一个这样的服务。我们通过两个假设来简化所需实现的服务(稍加修改就可以取消这两个假设)。第一，我们唯一需要存储的配置数据是字符串，关键字是 znode 的路径，因此我们在每个 znode 上存储了一个键-值对。第二，在任何时候只有一个客户端会执行更新操作。除此之外，这个模型看起来就像是有一个主节点(类似于 HDFS 中的 namenode)在更新信息，而它的工作节点则需要遵循这些

信息。

我们在 ActiveKeyValueStore 类中写了如下代码：

```
public class ActiveKeyValueStore extends ConnectionWatcher {

  private static final Charset CHARSET = Charset.forName("UTF-8");

  public void write(String path, String value) throws InterruptedException,
      KeeperException {
    Stat stat = zk.exists(path, false);
    if (stat == null) {
      zk.create(path, value.getBytes(CHARSET), Ids.OPEN_ACL_UNSAFE,
        CreateMode.PERSISTENT);
    } else {
      zk.setData(path, value.getBytes(CHARSET), -1);
    }
  }
}
```

write()方法的任务是将一个关键字及其值写入 ZooKeeper。它隐藏了创建一个新的 znode 和用一个新值更新现有 znode 之间的区别，而是使用 exists 操作来检测 znode 是否存在，然后再执行相应的操作。其他值得一提的细节是需要将字符串值转换为字节数组，因为我们只用了 UTF-8 编码的 getBytes()方法。

为了说明 ActiveKeyValueStore 的用法，我们写了一个用来更新配置属性值的类 ConfigUpdater，如范例 21-6 所示。

范例 21-6. 该程序随机更新 ZooKeeper 中配置属性值的程序

```
public class ConfigUpdater {

    public static final String PATH = "/config";

    private ActiveKeyValueStore store;
    private Random random = new Random();

    public ConfigUpdater(String hosts) throws IOException, InterruptedException {
        store = new ActiveKeyValueStore();
        store.connect(hosts);
    }

    public void run() throws InterruptedException, KeeperException {
      while (true) {
          String value = random.nextInt(100) + "";
          store.write(PATH, value);
          System.out.printf("Set %s to %s\n", PATH, value);
          TimeUnit.SECONDS.sleep(random.nextInt(10));
      }
    }
```

```
    public static void main(String[] args) throws Exception {
        ConfigUpdater configUpdater = new ConfigUpdater(args[0]);
        configUpdater.run();
    }
}
```

这个程序很简单，ConfigUpdater 中定义了一个 ActiveKeyValueStore，它在 ConfigUpdater 的构造函数中连接到 ZooKeeper。run()方法永远在循环，在随机时间以随机值更新/*config* znode。

接下来，让我们看看如何读取/*config* 配置属性的值。首先，我们在 ActiveKeyValueStore 中添加一个读方法：

```
public String read(String path, Watcher watcher) throws InterruptedException,
    KeeperException {
  byte[] data = zk.getData(path, watcher, null/*stat*/);
  return new String(data, CHARSET);
}
```

ZooKeeper 的 getData()方法有三个参数：路径、一个观察对象和一个 Stat 对象。Stat 对象由 getData()方法返回的值填充，用来将信息回传给调用者。通过这个方法，调用者可以获得一个 znode 的数据和元数据，但在这个例子中，由于我们对元数据不感兴趣，因此将 Stat 参数设为 null。

作为配置服务的用户，ConfigWatcher(参见范例 21-7)创建了一个 ActiveKeyValueStore 对象 store，并且在启动之后调用了 store 的 read()方法(在 displayConfig()方法中)，将自身作为观察传递给 store。displayConfig()方法用于显示它所读到的配置信息的初始值。

范例 21-7. 观察 ZooKeeper 中配置属性的更新情况并将其打印到控制台的应用

```
public class ConfigWatcher implements Watcher {

  private ActiveKeyValueStore store;

  public ConfigWatcher(String hosts) throws IOException, InterruptedException {
    store = new ActiveKeyValueStore();
    store.connect(hosts);
  }

  public void displayConfig() throws InterruptedException, KeeperException {
    String value = store.read(ConfigUpdater.PATH, this);
    System.out.printf("Read %s as %s\n", ConfigUpdater.PATH, value);
  }

  @Override
```

```
  public void process(WatchedEvent event) {
    if (event.getType() == EventType.NodeDataChanged) {
      try {
        displayConfig();
      } catch (InterruptedException e) {
        System.err.println("Interrupted. Exiting.");
        Thread.currentThread().interrupt();
      } catch (KeeperException e) {
        System.err.printf("KeeperException: %s. Exiting.\n", e);
      }
    }
  }

  public static void main(String[] args) throws Exception {
    ConfigWatcher configWatcher = new ConfigWatcher(args[0]);
    configWatcher.displayConfig();

    // stay alive until process is killed or thread is interrupted
    Thread.sleep(Long.MAX_VALUE);
  }
}
```

当 ConfigUpdater 更新 znode 时，ZooKeeper 产生一个类型为 EventType.NodeDataChanged 的事件，从而触发观察。ConfigWatcher 在它的 process()方法中对这个事件做出反应，读取并显示配置的最新版本。

由于观察仅发送单次信号，因此每次我们调用 ActiveKeyValueStore 的 read()方法时，都将一个新的观察告知 ZooKeeper，以确保我们可以看到将来的更新。尽管如此，我们还是不能保证接收到每一个更新，因为在收到观察事件通知与下一次读之间，znode 可能已经被更新过，而且可能是很多次更新，由于客户端在这段时间没有注册任何观察，因此不会收到通知。对于示例中的配置服务，这不是问题，因为客户端只关心属性的最新值，最新值优先于之前的值。但在一般情况下，这个潜在的问题是不容忽视的。

让我们看看如何使用这个程序。在一个终端窗口中运行 ConfigUpdater：

```
% java ConfigUpdater localhost
Set /config to 79
Set /config to 14
Set /config to 78
```

然后紧接着在另一个终端窗口启动 ConfigWatcher：

```
% java ConfigWatcher localhost
Read /config as 79
Read /config as 14
Read /config as 78
```

21.4.2 可复原的 ZooKeeper 应用

关于分布式计算(*http://bit.ly/dist_computing*)的第一个误区是“网络是可靠的”。按照他们的观点，程序总是有一个可靠的网络，因此当程序运行在真正的网络中时，往往会出现各种各样的故障。让我们看看各种可能的故障模式以及能够解决故障的措施，使我们的程序在面对故障时能够及时复原。

在 Java API 中的每一个 ZooKeeper 操作都在其 `throws` 子句中声明了两种类型的异常，分别是 `InterruptedException` 和 `KeeperException`。

1. InterruptedException 异常

如果操作被中断，则会有一个 `InterruptedException` 异常被抛出。在 Java 语言中有一个取消阻塞方法的标准机制，即针对存在阻塞方法的线程调用 `interrupt()`。一个成功的取消操作将产生一个 `InterruptedException` 异常。ZooKeeper 也遵循这一机制，因此你可以使用这种方法来取消一个 ZooKeeper 操作。使用了 ZooKeeper 的类或库通常会传播 `InterruptedException` 异常，使客户端能够取消它们的操作。[①]

`InterruptedException` 异常并不意味着有故障，而是表明相应的操作已经被取消，所以在配置服务的示例中，可以通过传播异常来中止应用程序的运行。

2. KeeperException 异常

如果 ZooKeeper 服务器发出一个错误信号或与服务器存在通信问题，抛出的则是 `KeeperException` 异常。针对不同的错误情况，`KeeperException` 异常存在不同的子类。例如，`KeeperException.NoNodeException` 是 `KeeperException` 的一个子类，如果你试图针对一个不存在的 znode 执行操作，就会抛出这个异常。

每一个 `KeeperException` 异常的子类都对应一个关于错误类型信息的代码。例如，`KeeperException.NoNodeException` 异常的代码是 `KeeperException.Code.NONODE`(一个枚举值)。

① 详情请参阅 Brian Goetz 的优秀文章，标题为“Java theory and practice: Dealing with InterruptedException”，2006 年 5 月发表于 IBM 开发者网络，网址为 *http://www.ibm.com/developerworks/java/library/j-jtp05236.html*。

有两种方法被用来处理KeeperException 异常：一种是捕捉KeeperException 异常并且通过检测它的代码来决定采取何种补救措施；另一种是捕捉等价的KeeperException 子类并且在每段捕捉代码中执行相应的操作。

KeeperException 异常分为三大类。

- **状态异常** 当一个操作因不能被应用于 znode 树而导致失败时，就会出现状态异常。状态异常产生的原因通常是在同一时间有另外一个进程正在修改 znode。例如，如果一个 znode 先被另外一个进程更新了，根据版本号执行setData 操作的进程就会失败，并收到一个KeeperException.BadVersionException 异常，这是因为版本号不匹配。程序员通常都知道这种冲突总是存在的，所以也都会写代码来进行处理。

一些状态异常会指出程序中的错误，例如 KeeperException. NoChildrenForEphemeralsException 异常，试图在短暂 znode 下创建子节点时就会抛出该异常。

- **可恢复的异常** 可恢复的异常是指那些应用程序能够在同一个 ZooKeeper 会话中恢复的异常。一个可恢复的异常是通过 KeeperException.ConnectionLossException 来表示的，它意味着已经丢失了与 ZooKeeper 的连接。ZooKeeper 会尝试重新连接，并且在大多数情况下重新连接会成功，并确保会话是完整的。

但是 ZooKeeper 不能判断与 KeeperException.Connection LossException 异常相关的操作是否成功执行。这种情况就是部分失败的一个例子(在本章开始时提到的)。这时程序员有责任来解决这种不确定性，并且根据应用的情况来采取适当的操作。

在这一点上，就需要对**幂等**(idempotent)操作和**非幂等**(Nonidempotent)操作进行区分。幂等操作是指那些一次或多次执行都会产生相同结果的操作，例如读请求或无条件执行的 setData 操作。对于幂等操作，只需要简单地进行重试即可。

对于非幂等操作，就不能盲目地进行重试，因为它们多次执行的结果与一次执行是完全不同的。程序可以通过在 znode 的路径和它的数据中编码信息来检测是否非幂等操作的更新已经完成。在 21.4.3 节对可恢复异常的讨论中，我们将通过实现一个锁服务来讨论如何处理失败的非幂等操作。

- **不可恢复的异常** 在某些情况下，ZooKeeper 会话会失效，也许因为超时或因为会话被关闭(两种情况下都会收到 KeeperException.

SessionExpiredException 异常)，或因为身份验证失败(KeeperException.AuthFailedException 异常)。无论上述哪种情况，所有与会话相关联的短暂 znode 都将丢失，因此应用程序需要在重新连接到 ZooKeeper 之前重建它的状态。

3. 可靠的配置服务

让我们回到 ActiveKeyValueStore 的 write()方法，它由一个 exists 操作紧跟着一个 create 操作或 setData 操作组成：

```
public void write(String path, String value) throws InterruptedException,
    KeeperException {
  Stat stat = zk.exists(path, false);
  if (stat == null) {
    zk.create(path, value.getBytes(CHARSET), Ids.OPEN_ACL_UNSAFE,
      CreateMode.PERSISTENT);
  } else {
    zk.setData(path, value.getBytes(CHARSET), -1);
  }
}
```

作为一个整体，write()方法是一个幂等操作，所以我们可以对它进行无条件重试。这里有一个 write()方法修改后的版本，能够循环执行重试。

其中设置了重试的最大次数 MAX_RETRIES 和两次重试之间的时间间隔 RETRY_PERIOD_SECONDS：

```
public void write(String path, String value) throws InterruptedException,
      KeeperException {
  int retries = 0;
  while (true) {
     try {
       Stat stat = zk.exists(path, false);
       if (stat == null) {
         zk.create(path, value.getBytes(CHARSET), Ids.OPEN_ACL_UNSAFE,
               CreateMode.PERSISTENT);
       } else {
          zk.setData(path, value.getBytes(CHARSET), stat.getVersion());
       }
       return;
     } catch (KeeperException.SessionExpiredException e) {
       throw e;
     } catch (KeeperException e) {
       if (retries++ == MAX_RETRIES) {
       throw e;
       }
       // sleep then retry
```

```
            TimeUnit.SECONDS.sleep(RETRY_PERIOD_SECONDS);
        }
    }
  }
```

这段代码没有在 KeeperException.SessionExpiredException 异常处进行重试，因为当一个会话过期时，ZooKeeper 对象会进入 CLOSED 状态，此状态下，它不能再进行重新连接(参见图 21-3)。我们只是简单地将这个异常重新抛出[①]并且让调用者创建一个新的 ZooKeeper 实例，以重试整个 write()方法。一个简单的创建新实例的方法是创建一个新的 ConfigUpdater(实际上我们已将其改名为 ResilientConfigUpdater)用于恢复过期会话：

```
public static void main(String[] args) throws Exception {
  while (true) {
    try {
      ResilientConfigUpdater configUpdater =
        new ResilientConfigUpdater(args[0]);
      configUpdater.run();
    } catch (KeeperException.SessionExpiredException e) {
      // start a new session
    } catch (KeeperException e) {
      // already retried, so exit
      e.printStackTrace();
      break;
    }
  }
}
```

处理会话过期的另一种方式是在观察中(本例子中应该是 ConnectionWatcher)监测类型为 Expired 的 KeeperState，然后在监测到的时候创新一个连接。即使收到 KeeperException.SessionExpiredException 异常，但由于连接最终是能够重新建立的，我们就可以使用这种方式在 write()方法内不断进行重试。不管我们采用何种机制从过期会话中恢复，重要的是需要对这种不同于连接丢失的故障类型进行不同的处理。

实际上，这里忽略了另一种故障模式。当 ZooKeeper 对象被创建时，它会尝试连接一个 ZooKeeper 服务器。如果连接失败或超时，那么它会尝试连接集合体

① 另外一种写代码的方式是只使用一段用于捕捉 KeeperException 异常的代码，然后检测所捕获异常的编码值是否为 KeeperException.Code.SESSIONEXPIRED。选择使用哪种方式取决于编程风格，因为两种方式的效果相同。

中的另一台服务器。如果在尝试集合体中所有服务器之后仍然无法建立连接，它会抛出一个 `IOException` 异常。由于所有 ZooKeeper 服务器都不可用的可能性很小，所以一些应用程序选择循环重试操作，直到 ZooKeeper 服务可用为止。

这仅仅是一种重试处理策略，还有许多其他策略，例如使用“指数退回”(exponential backoff)，每次将重试的间隔乘以一个常数。

21.4.3 锁服务

分布式锁能够在一组进程之间提供互斥机制，使得在任何时刻只有一个进程可以持有锁。分布式锁可以用于在大型分布式系统中实现领导者选举，在任何时间点，持有锁的那个进程就是系统的领导者。

不要将 ZooKeeper 自己的领导者选举和使用 ZooKeeper 基本操作实现的一般的领导者选举服务混为一谈。事实上，ZooKeeper 中包含有一个领导者选举服务的实现。ZooKeeper 自己的领导者选举机制是不对外公开的，我们这里所描述的一般领导者选举服务则不同，它是为那些需要所有进程与主进程保持一致的分布式系统所设计的。

为了使用 ZooKeeper 来实现分布式锁服务，我们使用顺序 znode 来为那些竞争锁的进程强制排序。思路很简单：首先指定一个作为锁的 znode，通常用它来描述被锁定的实体，称为*/leader*；然后希望获得锁的客户端创建一些短暂顺序 znode，作为锁 znode 的子节点。在任何时间点，顺序号最小的客户端将持有锁。例如，有两个客户端差不多同时创建 znode，分别为*/leader/lock-1* 和*/leader/lock-2*，那么创建*/leader/lock-1* 的客户端将会持有锁，因为它的 znode 顺序号最小。ZooKeeper 服务是顺序的仲裁者，因为它负责分配顺序号。

通过删除 znode */leader/lock-1* 即可简单地将锁释放；另外，如果客户端进程死亡，对应的短暂 znode 也会被删除。接下来，创建*/leader/lock-2* 的客户端将持有锁，因为它的顺序号紧跟前一个。通过创建一个关于 znode 删除的观察，可以使客户端在获得锁时得到通知。

如下是申请获取锁的伪代码。

(1) 在锁 znode 下创建一个名为 *lock-*的短暂顺序 znode，并且记住它的实际路径名(`create` 操作的返回值)。

(2) 查询锁 znode 的子节点并且设置一个观察。

(3) 如果步骤 1 中所创建的 znode 在步骤 2 返回的所有子节点中具有最小的顺序号，则获取到锁，退出。

(4) 等待步骤 2 中所设观察的通知，转到步骤 2。

1. 羊群效应

虽然这个算法是正确的，但还是存在一些问题。第一个问题是这种实现会受到**羊群效应**(herd effect)的影响。在有成百上千客户端的情况，所有的客户端都在尝试获得锁，所以每个客户端都会在锁 znode 上设置一个观察，用于捕捉子节点的变化。每次锁被释放或一个新进程开始申请锁的时候，观察都会被触发并且每个客户端都会收到一个通知。“羊群效应”就是指这种大量客户端收到同一事件的通知，但实际上只有很少一部分需要处理这一事件。在这种情况下，只有一个客户端会成功地获取锁，但是维护的过程以及向所有客户端发送观察事件会产生峰值流量，这会对 ZooKeeper 服务器造成压力。

为了避免出现羊群效应，我们需要优化发送通知的条件。关键在于仅当前一个顺序号的子节点消失时才需要通知下一个客户端，而不是删除(或创建)任何子节点时都进行通知。在我们的例子中，如果客户端创建了 znode */leader/lock-1*、*/leader/lock-2* 和 */leader/lock-3*，那么只有当 */leader/lock-2* 消失时才需要通知 */leader/lock-3* 对应的客户端；*/leader/lock-1* 消失或有新的 znode */leader/lock-4* 加入时，不需要通知该客户端。

2. 可恢复的异常

这个申请锁的算法目前还存在另一个问题，就是不能处理因连接丢失而导致的 `create` 操作失败。如前所述，在这种情况下我们不知道操作是成功还是失败。由于创建一个顺序 znode 是非幂等操作，所以我们不能简单地进行重试。原因在于如果第一次创建已经成功，重试会使我们多出一个永远删不掉的孤儿 znode(至少到客户端会话结束前)。最不幸的结果是还将会出现死锁。

问题在于，在重新连接之后客户端不能够判断它是否已经创建过子节点。解决方案是在 znode 的名称中嵌入一个 ID，如果客户端出现连接丢失的情况，重新连接之后它便可以对锁节点的所有子节点进行检查，看看是否有子节点的名称中包含其 ID。如果有一个子节点的名称包含其 ID，它便知道自己的创建操作已经成功，不需要再创建子节点。如果没有子节点的名称中包含其 ID，则客户端可以安全地创建一个新的顺序子节点。

客户端会话的 ID 是一个长整数，并且在 ZooKeeper 服务中是唯一的，因此非常适合在连接丢失后用于重新识别客户端。可以通过调用 Java `ZooKeeper` 类的 `getSessionId()`方法来获得会话的 ID。

在创建短暂顺序 znode 时应当采用 *lock-<sessionId>-*这样的命名方式，ZooKeeper 在其尾部添加顺序号之后，znode 的名称会形如 *lock-<sessionId>-<sequenceNumber>*。由于顺序号对于父节点来说是唯一的，但对于子节点名并不唯一，因此采用这样的命名方式可以让子节点在保持创建顺序的同时能够确定自己的创建者。

3. 不可恢复的异常

如果一个客户端的 ZooKeeper 会话过期，那么它所创建的短暂 znode 将会被删除，已持有的锁会被释放，或者是放弃了申请锁的位置。使用锁的应用程序应当意识到它已经不再持有锁，应当清理它的状态，然后通过创建并尝试申请一个新的锁对象来重新启动。注意，这个过程是由应用程序控制的，而不是锁，因为锁是不能预知应用程序需要如何清理自己的状态。

4. 实现

正确地实现一个分布式锁是一件棘手的事，因为很难对所有类型的故障都进行正确的解释处理。ZooKeeper 带有一个 Java 语言写的生产级别的锁实现，名为 `WriteLock`，客户端可以很方便地使用它。

21.4.4 更多分布式数据结构和协议

使用 ZooKeeper 可以实现很多不同的分布式数据结构和协议，例如“屏障”(barrier)、队列和两阶段提交协议。有趣的是它们都是同步协议，但我们可以使用异步 ZooKeeper 基本操作(如通知)来实现它们。

ZooKeeper 网站(*http://zookeeper.apache.org*)提供了一些用于实现分布式数据结构和协议的伪代码。ZooKeeper 本身也带有一些标准方法的实现(包括锁、领导者选举和队列)，放在安装位置下的 *recipes* 目录中。

Curator 项目(*https://github.com/Netflix/curator*)提供了更多 ZooKeeper 方法的实现。

BookKeeper 和 Hedwig

BookKeeper 是一个具有高可用性和可靠性的日志服务。它可以用来实现预写式日志(write-ahead logging)，这是一项在存储系统中用于保证数据完整性的常用技术。在一个使用预写式日志的系统中，每一个写操作在被应用前都先要写入事务日志。使用这个技术，我们不必在每个写操作之后都将数据写到永久存储器上，因为即使出现系统故障，也可以通过重新执行事务日志中尚未应用的写操作来恢复系统的最后状态。

BookKeeper 客户端所创建的日志被称为 *ledger*，每一个添加到 ledger 的记录被称为 *ledger entry*，每个 ledger entry 就是一个简单的字节数组。ledger 由保存有 ledger 数据副本的 *bookie* 服务器组进行管理。注意，ledger 数据不存储在 ZooKeeper 中，只有元数据保存在 ZooKeeper 中。

传统上，为了让使用预写式日志的系统更加稳定，必须解决保存有事务日志的节点的故障问题，这通常是通过某种方式复制事务日志来解决这个问题。前面描述过的 HDFS 高可用性使用一组日志节点来提供高可用性编辑日志，这虽然与 BookKeeper 相似，但它是为 HDFS 编写的独立服务，并且不采用 ZooKeeper 作为协调引擎。

Hedwig 是利用 BookKeeper 实现的一个基于主题的发布-订阅系统。以 ZooKeeper 作为基础，Hedwig 提供了一个具有高可用性的服务，即使在订阅者长时间离线的情况下它也能够保证消息的传递。

BookKeeper 是 ZooKeeper 的一个子项目，可以访问 *http://zookeeper.apache.org/bookkeeper/*，找到它和 Hedwig 的更多相关用法。

21.5 生产环境中的 ZooKeeper

在生产环境中，应当以复制模式运行 ZooKeeper。在这里，我们将讨论使用 ZooKeeper 服务器的集合体时需要考虑的一些问题。但是本节的内容不够详尽，建议参考《ZooKeeper 管理员指南》(*http://bit.ly/admin_guide*)获得详细的最新操作指南，包括支持的平台、推荐的硬件、维护过程和配置属性。

21.5.1 可恢复性和性能

在安放 ZooKeeper 所用的机器时，应当考虑尽量减少机器和网络故障可能带来的影响。在实践过程中，一般是跨机架、电源和交换机来安放服务器，这样，这些设备中的任何一个出现故障都不会使集合体损失半数以上的服务器。

对于那些需要低延迟服务(毫秒级别)的应用来说，最好将所有的服务器都放在同一个数据中心的同一个集合体中。也有一些应用不需要低延迟服务，它们可以通过跨数据中心(每个数据中心至少两台服务器)安放服务器来获得更好的可恢复性，领导者选举和分布式粗粒度锁是这类应用的代表。这两个应用中的状态改变都相对较少，因此相对于整个服务来说，数据中心之间传递状态改变消息所需的几十毫秒开销是可以承受的。

ZooKeeper 中有一个**观察节点**(observer node)的概念，是指没有投票权的跟随者。由于观察节点不参与写请求过程中达成共识的投票，因此使用观察节点可以让 ZooKeeper 集群在不影响写性能的情况下提高读操作的性能。[①]使用观察节点可以让 ZooKeeper 集群跨越多个数据中心，同时不会增加正常投票节点的延迟。可以通过将投票节点安放在一个数据中心，将观察节点安放在另一个数据中心来实现这一点。

ZooKeeper 是具有高可用性的系统，对它来说，最关键的是能够及时地履行其职能。因此，ZooKeeper 应当运行在专用的机器上。如果有其他应用程序竞争资源，可能会导致 ZooKeeper 的性能明显下降。

通过对 ZooKeeper 进行配置，可以使它的事务日志和数据快照分别保存在不同的磁盘驱动器上。在默认情况下，两者都保存在 `dataDir` 属性所指定的目录中，但是通过为 `dataLogDir` 属性设置一个值，便可以将事务日志写在指定的位置。通过指定一个专用的设备(不只是一个分区)，一个 ZooKeeper 服务器可以以最大速率将日志记录写到磁盘，因为写日志是顺序写，并且没有寻址操作。由于所有的写操作都是通过领导者来完成的，增加服务器并不能提高写操作的吞吐量，所以提高性能的关键是写操作的速度。

如果写操作的进程被交换到磁盘上，则性能会受到不利的影响。这是可以避免

① 详情参见 Henry Robinson 的文章，标题为“Observers: Making ZooKeeper Scale Even Further”，2009 年 12 月发表于 Cloudera，网址为 *http://www.cloudera.com/blog/2009/12/observers-making-zooker-scale-even-further/*。

的，将 Java 堆的大小设置为小于机器上空闲的物理内存即可。ZooKeeper 脚本可以从它的配置目录中获取一个名为 *java.env* 的文件，这个文件被用来设置 JVMFLAGS 环境变量，包括设置 Java 堆的大小(和任何其他所需的 JVM 参数)。

21.5.2 配置

ZooKeeper 服务器的集合体中，每个服务器都有一个数值型的 ID，服务器 ID 在集合体中是唯一的，并且取值范围在 1～255 之间。可以通过一个名为 *myid* 的纯文本文件设定服务器的 ID，这个文件保存在 dataDir 参数所指定的目录中。

为每台服务器设置 ID 只完成了工作的一半。我们还需要将集合体中其他服务器的 ID 和网络位置告诉所有的服务器。在 ZooKeeper 的配置文件中必须为每台服务器添加下面这行配置：

```
server.n=hostname:port:port
```

n 是服务器的 ID。这里有两个端口设置：第一个是跟随者用来连接领导者的端口；第二个端口用于领导者选举。这里有一个包含三台机器的复制模式下 ZooKeeper 集合体的配置例子：

```
tickTime=2000
dataDir=/disk1/zookeeper
dataLogDir=/disk2/zookeeper
clientPort=2181
initLimit=5
syncLimit=2
server.1=zookeeper1:2888:3888
server.2=zookeeper2:2888:3888
server.3=zookeeper3:2888:3888
```

服务器在 3 个端口上进行监听：2181 端口被用于客户端连接；对于领导者来说，2888 端口被用于跟随者连接；3888 端口被用于领导者选举阶段的其他服务器连接。当一个 ZooKeeper 服务器启动时，它读取 *myid* 文件用于确定自己的服务器 ID，然后通过读取配置文件来确定应当在哪个端口进行监听，同时确定集合体中其他服务器的网络地址。

连接到这个 ZooKeeper 集合体的客户端在 ZooKeeper 对象的构造函数中应当使用 zookeeper1:2181、zookeeper2:2181 和 zookeeper3:2181 作为主机字符串。

在复制模式下，有两个额外的强制参数：initLimit 和 syncLimit，两者都是以滴答参数的倍数进行度量。

initLimit 参数设定了所有跟随者与领导者进行连接并同步的时间范围。如果在设定的时间段内，半数以上的跟随者未能完成同步，领导者便会宣布放弃领导地位，然后进行另外一次领导者选举。如果这种情况经常发生(可以通过日志中的记录发现这种情况)，则表明设定的值太小。

syncLimit 参数设定了允许一个跟随者与领导者进行同步的时间。如果在设定的时间段内，一个跟随者未能完成同步，会自己重启。所有关联到跟随者的客户端将连接到另一个跟随者。

这些是建立和运行一个 ZooKeeper 服务器集群所需的最少设置。《ZooKeeper 管理员指南》(*http://bit.ly/zookeeper_admin*)列出了更多的配置选项，特别是性能调优方面的。

21.6 延伸阅读

想要获取更多关于 ZooKeeper 的深度知识，请参阅 O'Reilly 在 2013 年出版的 *ZooKeeper* 一书，网址为 *http://shop.oreilly.com/product/0636920028901.do*)，作者 Flavio 和 Benjamin Reed。

第V部分

案例学习

第 22 章　医疗公司 Cerner 的可聚合数据
第 23 章　生物数据科学：用软件拯救生命
第 24 章　开源项目 Cascading

第 22 章

医疗公司塞纳(Cerner)的可聚合数据

(作者：Ryan Brush 与 Micah Whitacre)

一直以来，健康医疗信息技术所做的事情是将现有流程用自动化的方式来实现。然而这一切正在发生改变。随着人们对提高治疗质量和控制医疗成本的诉求日益增长，迫切需要有更好的系统来支撑这些目标。接下来，我们将看 Cerner 公司是如何用 Hadoop 生态系统来理解健康医疗的概念并构建方案来解决这些问题的。

22.1 从多 CPU 到语义集成

Cerner 公司长期致力于将技术应用于健康医疗，并在很长的一段时间里将重点放在电子医疗记录上。然而，新的问题需要一种更宽泛的方法，这驱使我们去研究 Hadoop。

2009 年时，我们需要为医疗记录建立更好的搜索索引。由此而引出的处理需求无法通过其他架构简单解决。搜索索引需要以高昂的代价处理临床文档：从文档中抽取术语，并解析其与其他术语的关系。例如，如果用户键入“心脏病”一词，我们希望返回的是探讨心肌梗塞的文档。这一处理，代价相当高，比如大文档，它需要占用数秒钟的 CPU 时间，而我们想要将其应用于数以百万计的文档。简而言之，我们需要投入许多 CPU 来处理这个问题，并且希望处理过程能够经济合算。

在其他可选方案中，我们曾经考虑过用一种基于分段式事件驱动架构(SEDA，staged

event-driven architecture)的方法来成规模地抽取文档。但是 Hadoop 在满足一项重要需求方面脱颖而出：我们需要在数小时或者更快的时间内频繁地反复处理数以百万计的文档。面向临床文档的知识抽取的逻辑在快速改进中，我们需要将这种改进迅速推向世界。在 Hadoop 中，这仅仅意味着在已有的数据之上运行一个新版本的 MapReduce 作业，然后处理文档被载入一个 Apache Solr 服务器集群以支持应用查询。

这些早期的成功为后期更多涉足的项目打好了基础。这一系统类型及其数据可被用作经验基础，以帮助控制成本以及改善全部人口的医疗水平。由于健康医疗数据经常以碎片化的形式分布在各系统和机构中，我们需要首先收集所有这些数据并理解其含义。

当有了大量的数据源和格式，甚至标准化的数据模型有待解析时，我们面临着一个庞大的语义集成问题。最大的挑战并非来自数据的规模——众所周知 Hadoop 可以根据需要扩展——而是来自于为满足我们的需求，对数据进行清理、管理和转换所带来的极端复杂性。我们需要更高级的工具来管理这一复杂性。

22.2 进入 Apache Crunch

收集并分析这些各不相干的数据集会引发出许多需求，其中以下几点最为突出。

- 需要将许多处理步骤分割成模块，这些模块可被轻易地组装进一条复杂的管线。
- 需要提供一个比原始 MapReduce 更高级的编程模型。
- 需要处理医疗记录的复杂结构，该结构具有数百个独一无二的字段和数级嵌套子结构。

在这一案例中，我们尝试过多种选择，包括 Pig、Hive 和 Cascading，每一种效果都不错。我们继续使用 Hive 进行一些特别的分析处理，然而当把任意逻辑应用于我们的复杂数据结构时，Hive 却变得很不灵便。然后我们听说了 Crunch(参见第 18 章)，Josh Wills[①]领导的一个项目，与 Google 的 FlumeJava 系统相类似。Crunch 提供了一个简单的基于 Java 的编程模型以及对记录的静态类型检查，完美贴合我们 Java 开发者社区的需求以及所处理的数据类型。

① 编注：前谷歌广告系统工程师，曾经专门写书讲述了工业界和学术界在机器学习方面的异同，现为著名信息聚合平台 Slack 的首席数据工程师。

22.3 建立全貌

规模化的理解和管理健康医疗需要大量干净、规范化和可靠的数据。不幸的是，这样的数据通常分布在许多数据源中，合并起来较为困难且易于出错。医院、医生办公室、诊所和药房各自拥有个人记录的一部分，以行业标准的格式，诸如CCDs(Continuity of Care Documents)、HL7(Health Level 7，一种健康医疗数据交换格式)、CSV 文件或专有格式存储。

我们面临的挑战是，如何获取这一数据，将其转换为一种干净、集成的表现形式，并使用它创建注册信息，以帮助患者管理明确的病情、度量健康医疗的可操作方面，并支持各种分析，如图 22-1 所示。

图 22-1. 可操作的数据流

一个至关重要的步骤是，创建一个我们可以依赖的干净的、语义集成的基础，这也是本章实例学习的聚焦点。我们先将数据规范化为一种通用结构。这一系统的早期版本使用了不同的模型，但是后来都已经迁移至 Avro 以便在各处理步骤之间存储和共享数据。范例 22-1 给出了一个简化的 Avro IDL 以描述我们提出的通用结构。

范例 22-1. 用于通用数据类型的 Avro IDL

```
@namespace("com.cerner.example")
protocol PersonProtocol {

  record Demographics {
  string firstName;
  string lastName;
  string dob;
  ...
```

```
  }
  record LabResult {
  string personId;
  string labDate;
  int labId;
  int labTypeId;
  int value;
  }
  record Medication {
  string personId;
  string medicationId;
  string dose;
  string doseUnits;
  string frequency;
  ...
  }
  record Diagnosis {
  string personId;
  string diagnosisId;
  string date;
  ...
  }
  record Allergy {
  string personId;
  int allergyId;
  int substanceId;
  ...
  }
  /**
  * Represents a person's record from a single source.
  */
  record PersonRecord {
  string personId;
  Demographics demographics;
  array<LabResult> labResults;
  array<Allergy> allergies;
  array<Medication> medications;
  array<Diagnosis> diagnoses;
  . . .
  }
}
```

注意，这里多种数据类型都嵌套在一条通用个人记录中，而不是分散在不同数据集中。这种结构支持对这类数据的最常见的使用模式(查看一条完整的记录)无需对数据集进行大量昂贵的连接操作。

将数据写入 PCollection<PersonRecord>使用了一系列 Crunch 管线，这样可以隐藏每个源的复杂性，并提供了一个与原始规范化记录数据进行交互的简单接口。在幕后，每个 PersonRecord 可以存储在 HDFS 中或者作为 HBase 的一行存储，其中每个人的数据元素分布在列簇(column families)和列描述(column qualifiers)中。聚合的结果看起来如表 22-1 所示。

表 22-1. 聚合数据

源	个人 ID	个人统计信息	数据
医生办公室	12345	Abraham Lincoln…	糖尿病诊断，实验室结果
医院	98765	Abe Lincoln ...	流感诊断
药房	98765	Abe Lincoln ...	过敏，药物
诊所	76543	A. Lincoln ...	实验室结果

希望从一些获得授权的源中检索数据的用户调用一个“retriever” API，就可以轻松地为被请求数据生成一个 Crunch `PCollection`：

```
Set<String> sources = ...;
PCollection<PersonRecord> personRecords =
  RecordRetriever.getData(pipeline, sources);
```

这种 retriever 模式允许用户载入数据集，而无需知道这些数据集物理上是如何及在哪里存储的。在本书编写的同时，这一模式的某些用途正被新兴的 Kite SDK(*http://kitesdk.org/*) 所取代，用于在 Hadoop 中管理数据。检索到的 `PCollection<PersonRecord>`中的每个条目代表了一个人在单一源中的完整医疗记录。

22.4 集成健康医疗数据

从原始数据到健康医疗相关的问题答案之间，有着许多处理步骤。这里我们讨论其中一个步骤：将来自多个源的、关于同一个人的数据集中起来。

不幸的是，在美国缺乏通用的病人标识符，再加上噪声数据的影响，例如不同系统中个人姓名和统计资料的变异，使得跨越多个源将一个人的数据精确地统一起来非常困难。分布在多个源中的信息可能看起来如表 22-2 所示。

表 22-2. 来自多个源的数据

源	个人ID	名	姓	地址	性别
医生办公室	12345	Abraham	Lincoln	1600 Pennsylvania Ave.	M
医院	98765	Abe	Lincoln	Washington, DC	M
医院	45678	Mary Todd	Lincoln	1600 Pennsylvania Ave.	F
诊所	76543	A.	Lincoln	Springfield, IL	M

在健康医疗领域，上述数据统一问题通常是通过一个称为“企业级患者主索引”(EMPI，Enterprise Master Patient Index)的系统来解决的。一个 EMPI 可以从多个系

统中获取数据，并确定哪些记录确实是对应了同一个人。有很多途径可以达到这种目的，例如，明确声明各自关系的人类，或是能够识别共性的复杂算法。

在某些情况下，我们可以从外部系统载入 EMPI 信息，而在另一些情况下，我们可以在 Hadoop 内对其进行计算。关键在于我们可以将这一信息提供给基于 Crunch 的管线使用。其结果便是 PCollection<EMPIRecord>，数据结构如下：

```
@namespace("com.cerner.example")
protocol EMPIProtocol {

  record PersonRecordId {
  string sourceId;
  string personId
  }
  /**
  * Represents an EMPI match.
  */
  record EMPIRecord {
  string empiId;
  array<PersonRecordId> personIds;
  }
}
```

给定这一结构中数据的 EMPI 信息，PCollection<EMPIRecord>将会包含类似于表 22-3 所示的数据。

表 22-3. EMPI 数据

EMPI 标识符	个人记录 ID(<源 ID，个人 ID>)
EMPI-1	<offc-135, 12345>
	<hspt-246, 98765>
	<clnc-791, 76543>
EMPI-2	<hspt-802, 45678>

为了将所提供的 PCollection<EMPIRecord>和 PCollection<PersonRecord>里属于同一个人的医疗记录组合起来，必须将所收集的数据转换成 PTable，并采用一个通用键做为键。在这种情况下，Pair<String,String>(其中第一个值为 sourceId，第二个值为 personId)将确保采用唯一的键值进行连接。

第一步是从所收集的数据中的每个 EMPIRecord 里提取通用键：

```
PCollection<EMPIRecord> empiRecords = ...;
PTable<Pair<String, String>, EMPIRecord> keyedEmpiRecords =
    empiRecords.parallelDo(
  new DoFn<EMPIRecord, Pair<Pair<String, String>, EMPIRecord>>() {
```

```
    @Override
    public void process(EMPIRecord input,
      Emitter<Pair<Pair<String, String>, EMPIRecord>> emitter) {
    for (PersonRecordId recordId: input.getPersonIds()) {
    emitter.emit(Pair.of(
        Pair.of(recordId.getSourceId(), recordId.getPersonId()), input));
      }
    }
  }, tableOf(pairs(strings(), strings()), records(EMPIRecord.class)
);
```

下一步，需要从每个 PersonRecord 里提取相同的键：

```
PCollection<PersonRecord> personRecords = ...;
PTable<Pair<String, String>, PersonRecord> keyedPersonRecords =
personRecords.by(
    new MapFn<PersonRecord, Pair<String, String>>() {
  @Override
  public Pair<String, String> map(PersonRecord input) {
    return Pair.of(input.getSourceId(), input.getPersonId());
  }
}, pairs(strings(), strings()));
```

连接这两个 PTable 对象将返回 PTable<Pair<String, String>,Pair<EMPIRecord, PersonRecord>>。此时，原键不再有用，于是我们用 EMPI 标识符作为该表的键：

```
PTable<String, PersonRecord> personRecordKeyedByEMPI = keyedPersonRecords
  .join(keyedEmpiRecords)
  .values()
  .by(new MapFn<Pair<PersonRecord, EMPIRecord>>() {
  @Override
  public String map(Pair<PersonRecord, EMPIRecord> input) {
    return input.second().getEmpiId();
  }
}, strings()));
```

最后一步是通过键将表进行分组，以保证所有要作为一个完整集合处理的数据被聚合在一起：

```
PGroupedTable<String, PersonRecord> groupedPersonRecords =
  personRecordKeyedByEMPI.groupByKey();
```

PGroupedTable 将包含类似表 22-4 中的数据。

这种将数据源进行统一的逻辑只是一个更大规模执行流程的第一步。其他一些 Crunch 下游功能基于这些步骤构建，以满足众多客户需求。在一个常见应用案例中，通过将统一的 PersonRecord 的内容载入一个基于规则的处理模型以发布新的临床诊断知识，这种方式解决了大量问题。例如，我们可以在那些记录之上运

行规则，从而判断一个糖尿病患者是否正在接受推荐疗法，并指出可以改进之处。相似的规则集为了各种需求而存在，从一般健康问题到复杂病情管理。这种逻辑可以是复杂的，并且对于不同的应用案例会存在许多变种，但逻辑的实现全都依赖于构成 Crunch 管线的各功能模块。

表 22-4. 已组合的 EMPI 数据

EMPI 标识符	可迭代的个人记录
EMPI-1	`{ "personId": "12345", "demographics": { "firstName": "Abraham", "lastName": "Lincoln", ... }, "labResults": [...] }, { "personId": "98765", "demographics": { "firstName": "Abe", "lastName": "Lincoln", ... }, "diagnoses": [...] }, { "personId": "98765", "demographics": { "firstName": "Abe", "lastName": "Lincoln", ... }, "medications": [...]}, { "personId": "76543", "demographics": { "firstName": "A.", "lastName": "Lincoln", ... } ... }`
EMPI-2	`{ "personId": "45678", "demographics": { "firstName": "Mary Todd", "lastName": "Lincoln", ... } ... }`

22.5 框架之上的可组合性

这里所描述的模式，呈现了以个人为中心的健康医疗的一个特定类别的问题。然而，这一数据也可用作理解健康医疗的可操作性和系统性的基础，这对我们转换

和分析它的能力提出了新的要求。

像 Crunch 这样的库能够帮助我们应对新兴的需求，因为它们有助于将我们的数据和处理逻辑变得可组合化。相比单一、静态的数据处理框架，我们能够将功能和数据集模块化，并且在新需求出现时对其进行重用。图 22-2 展示了组件如何以新的方式彼此连接，其中每个方框作为一个或多个 Crunch `DoFns` 来实现。这里，我们利用个人记录来识别糖尿病患者，推荐健康管理计划，同时采用那些可组合部件来整合可操作的数据并推动健康系统分析。

图 22-2. 可组合的数据集和功能

可组合性也使得对新问题空间的迭代变得更简单。当创建一种新的数据视图以回答一类新问题时，我们能够借助现有的数据集和转换过程，发布我们的新版本。随着问题变得更易于理解，该视图可以被迭代替换或更新。最终，这些新功能和数据集对历史资产做出贡献，并被用于新的需求。其结果便是一个不断增长的数据集目录，以支持不断增长的理解数据的需求。

处理过程需要与 Oozie 相互协调。每当新的数据到达，在 HDFS 中的一个明确定义的位置，将创建一个具有唯一标识符的新数据集。Oozie 协调器(Oozie coordinators)观察该位置，并简单地发起 Crunch 任务来创建下游数据集，该数据集随后可被其他协调器获得。在本书写作时，数据集和更新由 UUIDs 标识以保持其唯一性。然而，我们正在进行一项处理，即将新数据放置在基于时间戳的分区(timestamp-based partitions)中，以便更好地与 Oozie 的标称时间模型(Oozie's nominal time model)一起工作。

22.6 下一步

我们正在关注两个主要步骤，从而更高效地将这一系统的价值最大化。

首先，我们想要创建关于 Hadoop 生态系统及其支持库的规范化实践。本书和其他地方都定义了很多优秀的实践，但是经常需要重要的专业知识来有效实现这些实践。我们正在使用和建立库，从而使此类模式对于更大规模的受众来说变得明晰且易得。

通过将各种连接和处理模式构建进库，Crunch 提供了一些好的例子。

其次，我们不断增长的数据集目录引发了对简单且规范的数据管理的需求，这也是对 Crunch 提供的处理特性的一个补充。在一些使用案例中，我们已采纳 Kite SDK 来满足这一需求，并希望有朝一日扩展其用途。

最终目标为一个安全、可扩展的数据目录，以支持健康医疗领域的众多需求，包括尚未出现的问题。事实证明 Hadoop 能够满足我们对数据和处理的需求，而更高级的库正在使 Hadoop 广泛适用于更大规模的受众，以解决诸多问题。

第 23 章

生物数据科学：用软件拯救生命

(作者：Matt Massie)

难以相信，介绍 MapReduce 的论文在 OSDI'04 会议(*http://research.google.com/archive/mapreduce.html*)上出现至今已经过去了十年。这一论文对技术产业的影响同样难以夸大，MapReduce 范式向非专业人士开放了分布式编程领域，并使得在商用硬件构建的集群上进行大规模数据处理成为可能。作为回应，开源社区创建了基于 MapReduce 的开源系统，比如 Apache Hadoop 和 Spark，这些系统允许数据科学家和工程师们能够以之前无法想象的规模描述和解决问题。

当技术产业被基于 MapReduce 的系统改变时，生物学正在经历由第二代(或称为“下一代”)测序技术驱动的自我蜕变，如图 23-1 所示。测序机器是一种科学仪器，可以读取构成人类基因组的化学“字母”(A，C，T 和 G)，即人的整套遗传物质。在 MapReduce 论文发表的年代，对人的基因组测序需要花费约 2 千万美元并且要耗时好几个月；然而今天，这一过程仅需花几千美元及几天时间。第一个人类基因组耗费了数十年才创造出来，而 2014 年这一年，预计世界范围内会对 228,000 个基因组进行测序。[①] 这一预估意味着 2014 年全世界产生了大约 20 PB 的测序数据。

① 详情可以参见 2014 年 Antonio Regalado 发表于 EmTech 世界新兴技术峰会的文章，标题为“EmTech: Illumina Says 228 000 Human Genomes Will Be Sequenced This Year”，指 Illumina (宜曼达)公司将在当年完成 228 000 个人类基因组测序)，网址为 *http://bit.ly/genome_sequencing*。

图 23-1. 大数据技术的时间线和测序一个基因组的成本

骤然下降的测序成本，意味着未来几年基因数据将超线性增长。这种 DNA 数据的泛滥，使得生物数据科学家们要想使用当前的基因组软件及时并可扩展的处理数据非常费劲。AMPLab(*http://amplab.cs.berkeley.edu*)是加州大学伯克利分校计算机科学系的一个研究实验室，致力于开创新型的大数据系统和应用。例如，Apache Spark(参见第 19 章)便是一个由 AMPLab 孵化出的系统。最近，Spark 打破了 Daytona Gray Sort(一项数据排序比赛)的世界纪录，仅用时 23 分钟完成了 100 TB 数据的排序。打破纪录的 Databricks 公司(*http://databricks.com*)参赛队也演示了他们能够在 4 小时之内排序 1 PB 数据。

设想这一令人惊喜的可能性：我们现在拥有的技术，使我们能够利用几百台机器，在以天为数量级的时间里，分析 2014 年收集到的每一个基因组数据。

虽然 AMPLab 出于技术原因，将基因学视为理想的大数据应用领域，但也有更重要的富有同情心的原因：即对生物学数据的及时处理能够拯救生命。本章讨论的这一简短的应用案例将聚焦于我们所用的系统，这些系统是与合作伙伴和开源社区合作开发的，用于快速分析大型生物学数据集。

23.1 DNA 的结构

1953 年，利用罗莎琳德 · 富兰克林(Rosalind Franklin)[1]和马乌里斯 · 威尔金斯(Maurice Wilkins)采集的实验数据，Francis Crick 和 James D. Watson[2]发现 DNA 具有双螺旋结构，这是 20 世纪最伟大的科学发现之一。他们在《自然》杂志发表的题为“核酸的分子结构：脱氧核糖核酸结构”的文章，包含了科学界最深刻而又朴素的名句之一：“我们同时注意到，我们所构想的这一特殊的配对，直接暗示着遗传物质的一种可能的复制机制。”

这一“特殊的配对”指观察到的以下事实：碱基腺嘌呤(A)和胸腺嘧啶(T)总是一起配对，而鸟嘌呤(G)和胞嘧啶(C)总是一起配对，如图 23-2 所示。这一确定性的配对使“复制机制”成为可能：解开 DNA 双螺旋，互补碱基对相互对准位置，创造两个原始 DNA 链(strand)的精确复制品。

图 23-2. DNA 双螺旋结构

① 编注：罗莎琳德·富兰克林，英国女科学家，1920 年出生于英国伦敦一个富裕的犹太人家族，毕业于剑桥大学，专业物理化学。1950 年任职于伦敦大学国王学院，她非常擅长于拍摄晶体的 x 射线衍射图片。1958 年因卵巢症去世，英国皇家学会为了纪念她，专门设立了“富兰克林”奖章。

② 编注：1962 年诺贝尔生理学和医学奖得主，美国科学家，DNA 双螺旋结构发现者，2014 年拍卖其诺贝尔奖章 475.7 万美元，手稿 36.5 万美元，论文 24.5 万美元。

23.2 遗传密码：将 DNA 字符转译为蛋白质

没有蛋白质就没有生命。在创造蛋白质的过程中，DNA 相当于一个配方。蛋白质是由若干氨基酸组成的链，这些氨基酸折叠成特定的 3D 形状[①]，用作特别的结构或功能。总共有 20 种氨基酸[②]，而 DNA 字母表里只有 4 个字母(A，C，T，G)，大自然将这些字母组成词，称为密码子(codon)。每个密码子的长度为三个碱基(因为两个碱基仅支持 4^2=16 个氨基酸)。

1968 年，Har Gobind Khorana、Robert W. Holley 和 Marshall Nirenberg 因成功定位(mapping)与 64 个密码子中的每一个相关联的氨基酸，获得了诺贝尔生理学或医学奖。每个密码子对一个单独的氨基酸进行编码，或者指定起始和终止位置(详见表 23-1)。因为有 64 个可能的密码子而只有 20 种氨基酸，所以有一部分氨基酸对应着多个密码子。

表 23-1. 密码子表

氨基酸	密码子	氨基酸	密码子
丙氨酸	GC{U,C,A,G}	亮氨酸	UU{A,G} 或 CU{U,C,A,G}
精氨酸	CG{U,C,A,G} 或 AG{A,G}	赖氨酸	AA{A,G}
天冬酰胺	AA{U,C}	蛋氨酸	AUG
天冬氨酸	GA{U,C}	苯丙氨酸	UU{U,C}
半胱氨酸	UG{U,C}	脯氨酸	CC{U,C,A,G}
谷氨酸	GA{A,G}	苏氨酸	AC{U,C,A,G}
谷氨酰胺	CA{A,G}	丝氨酸	UC{U,C,A,G} 或 AG{U,C}
甘氨酸	GG{U,C,A,G}	色氨酸	UGG
组氨酸	CA{U,C}	酪氨酸	UA{U,C}
异亮氨酸	AU{U,C,A}	缬氨酸	GU{U,C,A,G}
起始!	AUG	终止!	UAA 或 UGA 或 UAG

因为地球上的每个生物都从相同的共同祖先进化而来，所以地球上的每个生物使用相同的遗传密码，仅有很少的变异。无论该生物是树、虫子、真菌或猎豹，密

① 编注：这一过程称为“蛋白质折叠”。Folding@home 网站(*http://folding.standford.edu/*)允许志愿者捐助 CPU 周期以帮助研究人员确定蛋白质折叠的机制。还有一个蛋白质折叠游戏 *Foldit* (看我怎么叠)。该游戏是由蛋白结构领域的科学家 David Bekei 找人设计的。2008 年，注册玩家很快达到 24 万人，困扰研究人员 5 年之久的蛋白结构 M-PMV 作为任务发布到该游戏应用之后，仅 10 天时间就被草根大神们破解。

② 也有一些非标准氨基酸未在表格中展示，它们的编码有所不同。

码子 UGG 都是色氨酸的编码。在过去的数十亿年中，大自然才是编码重用的终极实践者。

DNA 并不直接用于合成氨基酸。作为替代，一个称为转录(transcription)的过程，将为蛋白质编码的 DNA 序列复制到信使核糖核酸(mRNA，messenger RNA)。这些 mRNA 将信息从细胞核运送至周围的细胞质，进而在一个被称为转译(translation)的过程中创造蛋白质。

注意，这一查询表并不包含 DNA 字母 T(代表胸腺嘧啶)，并且还出现一个新的字母 U(代表尿嘧啶)。在转录期间，U 替代了 T：

```
$ echo "ATGGTGACTCCTACATGA" | sed 's/T/U/g' | fold -w 3
AUG
GUG
ACU
CCU
ACA
UGA
```

通过在密码子表中查询这些密码子，能够确定由这一特定的 DNA 链转译成的蛋白质将有着以下氨基酸组成的链：蛋氨基、缬氨酸、苏氨基、脯氨酸和苏氨基。这是一个人为的例子，但是它从逻辑上说明了 DNA 如何指导蛋白质的创造，从而使你成为独一无二的你。科学已允许我们理解 DNA 的语言，包括起始和终止标点，这真是一个奇迹。

22.3 将 DNA 想象成源代码

从细胞这一层面来看，可以把你的身体看作是一个完全的分布式系统，没有什么是集中式的。它就像一个 37.2 万亿(trillion)[①]个细胞的集群，执行着相同的代码：你的 DNA。

如果将你的 DNA 想象成源代码，就需要考虑下面这些事项。

源仅由四个字符组成：A，C，T 和 G。

源具有两个贡献者，你的母亲和父亲，每人贡献了 32 亿个字母。实际上，由基因

① 参见 Eva Bianconi 等人的文章，标题为“An estimation of the number of cells in the human body”(人体细胞数量估值)，网址为 *http://bit.ly/cell_estimate*，发表于 2013 年 11 月 12 月合刊《人体生物学年鉴》。

组参考合作组织(GRC：Genome Reference Consortium)提供的参考基因组只不过是一个有着 32 亿字符的 ASCII 文件。[①]

源被分割成 25 个称为“染色体”的独立文件，每个文件拥有源的不同片段。这些文件被编了号，并且文件规模随着编号逐渐减小，1 号染色体拥有约 2.5 亿个字符，而 22 号染色体 22 仅拥有约 5 千万个字符。还有 X 染色体、Y 染色体和线粒体染色体。术语“染色体”的基本含义是“着色的东西”，主要因为当初生物学家能够给它们染色，却不知道它们是什么。

源在你的生物机器上一次执行三个字母(即一个密码子)，使用之前解释过的遗传密码，这与一台读取化学字母而不是纸带的图灵机没有什么不同。

源有大约 20 000 个函数，称为基因(genes)，执行时，每个基因创造一个蛋白质。每个基因在源中的位置叫基因座(locus)。可以把基因想象为一个染色体上特定范围内的连续碱基位。例如，在 17 号染色体上从碱基位 41 196 312 到 41 277 500，可以找到与乳腺癌有牵连的 BRCA1 基因。基因就像“指针”或“地址”，而等位基因(alleles)，随后进行描述)则是真实的内容。每个人都有 BRCA1 基因，但并不是每个人都有使其有患病风险的等位基因。

单体型(haplotype)类似于面向对象编程语言中的对象，拥有继承下来的特定函数(基因)。

源对于每个基因有两个定义，称为等位基因，一个来自于母亲，一个来自于父亲。等位基因在配对染色体的相同位置被发现(当你体内的细胞为二倍体时，也就是说，每个基因有两个等位基因。有些生物是三倍体、四倍体等)。等位基因双方都被执行，生成的蛋白质相互作用创造出特定的表型(phenotype)。例如，造成或降低眼睛颜色色素的蛋白质会导致一种特别的表型，或显性特征(observable characteristic)(例如蓝眼睛)。如果从父母那里遗传到的等位基因完全相同，那么孩子的那个等位基因就是纯合的(homozygous)；否则，就是杂合的(heterozygous)。

单核多态性(SNP，single-nucleic polymorphism)，发音如“snip”，是源代码中单一字符的改变(例如从 ACTGACTG 变为 ACTTACTG)。

indel 为 insert-delete 的缩写，代表参考基因组的插入或缺失。例如，如果参考基

① 你可能以为有 64 亿个字母，但不管怎样，参考基因组其实是众多个体平均的单倍体(haploid)表示。

因是 CCTGACTG，而你的样本插入了四个字符，比如 CCTG**CCTA**ACTG，那么它就是一个 indel。

只有 0.5%的源得到转译，成为维持生命的蛋白质。这部分源被称为外显子(exome)。人类的外显子需要几 G(千兆)字节的存储空间，以存储在被压缩的二进制文件中。

另外 99.5%的源被注释掉，并用作填充字(内含子：introns)。它用来调节何时基因被打开和重复等[①]。完整基因组需要几百 G(千兆)字节的空间以存储在被压缩的二进制文件中。

你身体的每个细胞具有相同的源[②]，但是可以通过表观遗传(epigenetic)因子选择性地注释掉部分源，就像 DNA 甲基化(methylation)和组蛋白修饰(histone modification)那样。这如同对每个细胞类型的`#ifdef` 声明(例如，`#ifdef RETINA` 或`#ifdef LIVER`)。这些因子负责视网膜细胞与肝脏细胞以不同的方式运行。

变异检测(variant calling)的过程类似于在不同的 DNA 源之间执行 *diff* 操作。

这些类比不要以过于字面化地去理解，但是希望它们能够帮助你熟悉一些基因学的术语。

23.4 人类基因组计划和参考基因组

1953 年，沃森和克里克(Watson 和 Crick)发现了 DNA 的结构，1965 年尼伦伯格(Nirenberg)在其 NIH(美国国立卫生研究院)同事的帮助下，破解了遗传密码，揭示了将 DNA 或 mRNA 转译成蛋白质的规则。科学家们知道存在数百万种人类蛋白质，但却没有对人类基因组进行全面调查，这使得要想完全掌握负责蛋白质合成的基因基本是不可能的。例如，如果每个蛋白质由一个单独的基因创造，这将意味着有数百万蛋白质编码基因存在于人类基因组之中。

① 仅有约 28%的 DNA 被转录进新生的 RNA，而在 RNA 剪接之后，仅有约 1.5%的 RNA 留下用作蛋白质编码。进化选择发生在 DNA 这一级，通过大多数 DNA 向另外 0.5%DNA 提供支持或整体取消来实现选择过程（作为更合适的 DNA 进化）。可以这么说，有一些癌症似乎是由 DNA 的休眠区域被激活引起的。

② 实际上，平均每复制十亿个 DNA“字母”就会有 1 个错误。因此，每个细胞其实并不是完全相同的。

1990 年，“人类基因组计划”项目(the Human Genome Project)打算确定所有构成人类 DNA 的化学碱基对。这一国际合作研究项目于 2003 年 4 月发布了第一个人类基因组[①]，估计成本为 38 亿美元。人类基因组计划带来的经济影响估值约 7960 亿美元，相当于 141：1 的投资回报率[②]。人类基因组计划找到了大约 20 500 个基因——显著的低于根据简单的基因与蛋白质 1：1 模型所预估的数百万，这是因为蛋白质可由基因的组合、折叠期间的转译后加工、以及其他机制组装而成。

第一个人类基因组用了十年时间才得以建立，一旦建立，它使得引导随后的其他基因组测序变得更加容易。对于第一个基因组，科学家们是在黑暗中摸索。他们没有任何参考资料能够用作构建完整基因组的路线图。迄今为止，没有技术能够从头到尾读取整个基因组，但有很多技术在读取速度、精确度和读取 DNA 片段的长度上各有千秋。从事人类基因组计划的科学家们必须分片对基因组测序，不同的片(pieces)能够通过不同的技术更加轻松地测序。一旦拥有一个完整的人类基因组，随后的人类基因组构建将变得更加容易；你可以用第一个基因组作为第二个的参考。来自第二个基因组的片段可以与第一个进行模式匹配，类似于拼图盒子上的图片有助于告诉我们拼图碎片的位置。它非常有帮助，因为大多数编码序列高度稳定，1000 个基因座(loci)中仅有 1 个会发生变异。

在人类基因组计划完成后不久，基因组参考合作组织(GRC)(http://genomereference.org)得以建立，这是一个由学术和研究机构组成的国际性合作机构，致力于改进参考基因组的表征。GRC 发布了一个新的人类参考基因组，作用有点像一个通用坐标系或者地图，用以帮助分析新的基因组。最新的人类参考基因组，发布于 2014 年 2 月，命名为 GRCh38；它取代了五年前发布的 GRCh37。

22.5 DNA 测序和比对

随着众多硬件供应商以及大约每六个月便开发出的新测序方法，第二代测序技术发展迅速。然而，所有这些技术的一个共同特征是使用了大规模并行处理方法，同时能发生成千乃至数百万的反应。双链 DNA 从中间平分开来，单链被多次复制，而复制品被随机分割成不同长度的小片段，称为 *read*(译者注：*read* 是测序的最小单位，是高通量测序仪产生的测序数据)，这些片段被放置进测序仪(sequencer)。为了

① 特意选在沃森和克里克发现 DNA 3D 结构之后的 50 年。

② 详见 Jonathan Max Gitlin 在 2013 年 6 月发表的文章，标题为“Calculating the economic impact of the Human Genome Project”，网址为 *http://www.genome.gov/27544383*。

高通量，测序仪采用并行的方式读取每一个 read 中的“字母”，并输出一个原始 ASCII 文件，文件中包含每个 read(例如，AGTTTCGGGATC...)，以及对读取的每个字母的质量评估，这些将被用于下游的分析。

一个称为“比对器”的软件取得每个 read 并找到其在参考基因组里的位置(参见图 23-3)[①]。一个完整的人类基因组有大约 30 亿个碱基(A，C，T，G)对长[②]。参考基因组(例如 GRCh38)就像拼图盒子上的图片，呈现了人类基因组的整体轮廓和色彩。每个短的 read 就像一块拼图碎片，需要尽可能近的嵌入相应位置。一个通用的度量标准是“编辑距离”(edit distance)，它对一个字符串转换成另一个所必须的操作数量进行了量化。完全相同的两个字符串的编辑距离为 0；如果有字符串中有一个字母的插入缺失(indel)，则编辑距离为 1。由于从基因角度来看，人类彼此间有着 99.9%的相同度，因此大多数 read 都能很好地与参考基因组吻合并具有很小的编辑距离。建立一个好的比对器所面临的挑战在于处理异质(idiosyncratic)read。

图 23-3. 将 read 与参考基因组进行比对，来自 Wikipedia，网址为 *http://bit.ly/mapping_reads*

① 也有第二种方法，即重新组装，将 read 放进一个图形数据结构，这样无需建立其与参考基因组的定位就可以创建长序列。

② 每个碱基大约 3.4 埃(angstrom 是：长度单位，1 埃等于 0.1 纳米)，这样算来，自一个人类细胞的 DNA 首尾相连后延展开来将超过 2 米。

23.6 ADAM，一个可扩展的基因组分析平台

将 read 与参考基因组进行比对仅仅是生成可用于临床或研究的报告的一系列必要步骤中的第一步。这一处理管线的初级阶段看起来与其他提取转换加载(ETL，extract-transform-load)管线相类似，在分析之前需要对数据进行去重复和规范化。

测序过程复制基因组 DNA，这样一来，相同的 DNA read 便有可能被生成多次；这些复制品需要进行标记。测序仪也提供对其读取的每个 DNA“字母”的质量评估，这些字母存在需要进行调节的偏差，而这些偏差都是与特定的测序仪有关。比对器经常会将具有 indel(插入或缺失的序列)的 read 放错位置，需要在参考基因组上重新定位这些 read。目前，完成这一预处理是采用了由单个机器上的 shell 脚本启动的单一用途工具。这些工具需要很多天来完成整个基因组的处理。该过程受到磁盘的约束，每个阶段都要写入新的文件以供随后的阶段读取，这对于通用大数据技术应用来说是一个理想的应用案例。ADAM 能够在 2 小时之内完成相同的预处理任务。

ADAM 是一个基因组分析平台，专注于快速处理 PB(petabytes)量级的高覆盖、完整基因组数据。ADAM 依赖于 Apache Avro、Parquet 和 Spark。这些系统在一起使用时呈现出很多优势，因为它们有以下特征：

- 允许开发者专注于算法，无需担心分布式系统故障。
- 使作业能够本地运行在单个机器上、内部集群或云平台上，无需改变任务代码。
- 压缩遗留基因组格式，并且基于性能考虑，提供谓词下推和投影。
- 提供了一种定制和演进数据格式的敏捷方式。
- 被设计成易于扩展(仅使用商业硬件)。
- 在标准的 Apache 2.0 许可证(license)[①]下可以共享。

23.7 使用 Avro 接口描述语言进行自然语言编程

测序比对/定位(SAM，Sequence Alignment/Map)规范(*http://samtools.github.io/htsspecs/*

① 遗憾的是，一些更为流行的基因组软件具有不明确的或定制的、受限的许可证。从科学角度而言，干净的开源许可和源代码是必要的，能够使重现结果和理解结果更加容易。

SAMv1.pdf)定义了表 23-2 所列的强制性字段。

表 23-2. SAM 格式中的强制性字段

序号	字段	类型	正则表达式/范围	简要描述
1	QNAME	String	[!-?A-~]{1,255}	查询模板名称
2	FLAG	Int	[0, 216-1]	位标记
3	RNAME	String	*|[!-()+-<>-~][!-~]*	参考序列名称
4	POS	Int	[0,231-1]	基于 1 的最左边的定位位置
5	MAPQ	Int	[0,28-1]	定位质量
6	CIGAR	String	*|([0-9]+[MIDNSHPX=])+	CIGAR 字符串
7	RNEXT	String	*|=|[!-()+-><-~][!-~]*	配对物的参考名称/下一 read
8	PNEXT	Int	[0,231-1]	配对物的位置/下一 read
9	TLEN	Int	[-231+1,231-1]	观察到的模板长度
10	SEQ	String	*|[A-Za-z=.]+	分段序列
11	QUAL	String	[!-~]	表示序列质量信息的字符，使用 Phred+33 编码(即，质量字符的 ASCII 码值=质量得分+33)

任何想要实施这一规范的开发者，需要将这一英文规范翻译成他们所选择的计算机语言。在 ADAM 中，我们选择使用 Avro IDL 定义规范，并采用自然语言编程。例如，SAM 强制性字段能够方便地用一个简单的 Avro 记录表示如下：

```
record AlignmentRecord {
string qname;
int flag;
string rname;
int pos;
int mapq;
string cigar;
string rnext;
int pnext;
int tlen;
string seq;
string qual;
}
```

Avro 能够自动生成本地 Java(或 C++、Python 等)类用于读取和写数据，并提供标准接口(例如 Hadoop 的 `InputFormat`)从而易与众多系统集成。Avro 的设计也使得模式演进(schema evolution)更加容易。实际上，我们今天使用的 ADAM 模式(ADAM schema)(*http://bit.ly/bdg-formats*)已经演进得更加复杂、更有表现力且可定制性更强，能够表现多种基因组模型，比如结构性变异、基因型、变异检测标注、变异的影响等。

加州大学伯克利分校是全球基因组学与健康联盟的成员(Global Alliance for Genomics & Health，*http://genomicsandhealth.org*)，这是一个非政府的公私合作组织，包含了分布在 30 个国家的超过 220 个组织，其目标在于通过高效和可靠的数据共享，最大程度地发掘基因组医学的潜能。该联盟已经接受了这种自然语言编程方式，并发布了用 Avro IDL 描述的编程模式(schemas，网址为(*https://github.com/ga4gh/schemas*)。Avro 的使用使得全球各地的研究人员能够在逻辑层次上谈论数据，无需考虑计算机语言或磁盘格式(on-disk format)。

23.8 使用 Parquet 进行面向列的存取

SAM 和 BAM[①]文件格式是面向行的(row-oriented)：每条记录的数据作为单行文本或二进制记录存储在一起。详情参见 5.4.3 节关于面向行和面向列对比的讨论。SAM 文件中一个双末端 read 也许看起来如下所示：

```
read1 99 chrom1 7 30 8M2I4M1D3M = 37 39 TTAGATAAAGGATACTG *
read1 147 chrom1 37 30 9M        = 7 -39 CAGCGGCAT        * NM:i:1
```

一个典型的 SAM/BAM 文件包含数百万的行，每一行对应一个来自测序仪的 DNA read。先前的文本片段很容易翻译成表 23-3 中所示的视图。

表 23-3. SAM 片段的逻辑视图

Name	Reference	Position	MapQ	CIGAR	Sequence
read1	chromosome1	7	30	8M2I4M1D3M	TTAGA TAAAGGA TACTG
read1	chromosome1	37	30	9M	CAGCGG CAT

在这一例子中，read 标识为 `read1`，定位至参考基因组 `chromosome1` 的位置 7 和 37。这称为“双末端”读长，因为它代表了测序仪从每一末端读取的一个单链 DNA。打个比方，这就像从 `0..50` 和 `150..100` 读取长度为 150 的数组。

`MapQ` 分值代表序列正确定位至参考基因组的概率。`MapQ` 分值 20、30 和 40 分别代表 99%、99.9%和 99.99%的正确概率。从 `MapQ` 分值计算错误概率，采用表达式 $10^{(-MapQ/10)}$(例如 $10^{(-30/10)}$表示概率为 0.001)。

① BAM 是 SAM 格式的压缩二进制版本。

CIGAR 解释了 DNA 序列中的个体核苷酸是如何定位至参考基因组的[①]。当然，Sequence 就是定位至参考基因组的 DNA 序列。

基因组分析通用的面向列的存取模式和 SAM/BAM 面向行的磁盘格式之间存在很明显的不匹配。考虑下列情况。

- 一条范围查询，查找与乳腺癌相关联的特定基因(即 BRCA1)数据："寻找覆盖 17 号染色体位置 41 196 312 至 41 277 500 的所有 read"。
- 一个简单过滤操作，查找定位效果不佳的 read："查找所有 MapQ 小于 10 的 read"。
- 一个搜索操作，查找所有具有 indel(插入或缺失)的 read："查找所有 CIGAR 字符串中包含 I 或 D 的 read"。
- 计算不同的 *k*-mers 的数量："读取每个 Sequence，并生成所有可能的长度为 *k* 的子字符串"。

Parquet 的谓词下推(predicate pushdown)支持快速过滤 read 以用于分析(例如查找一个基因，忽略定位效果不佳的 read)。投影操作支持仅对感兴趣的列进行精确实体化(precise materialization)(例如，仅读取序列用于 *k*-mer 计算)。

此外，一些字段具有低基数(low cardinality)，使得它们对于像行程长度编码(RLE，run-length encoding)这样的数据压缩技术来说十分理想。例如，给定人类只有 23 对染色体，Reference 字段将只有很少的几个不同取值(例如 chromosome1、chromosome17 等)。我们发现在 Parquet 文件中存储 BAM 记录导致了约 20%的压缩。在 Parquet 中使用 PrintFooter 命令，我们发现可对质量评分进行行程长度编码和位填充从而压缩约 48%，但是它们仍占据约 70%的总空间。我们期待 Parquet2.0，这样便能够对质量评分使用增量编码，从而更大幅度地压缩文件大小。

23.9 一个简单例子：用 Spark 和 ADAM 做 *k*-mer 计数

k-mer 计数是一种字数统计，用于基因组学。术语 *k*-mers 指的是一个 read 的所有

① 第一条记录的 CIGAR(Compact Idiosyncratic Gap Alignment Report，紧凑异质型间隙比对报告)字符串被译作"8 个匹配(8M)，2 个插入(2I)，4 个匹配 (4M)，1 个缺失(1D)，3 个匹配(3M)"。

可能的长度为 k 的子序列。例如，一个 read 具有序列 AGATCTGAAG，该序列的 3-mers 将是['AGA', 'GAT', 'ATC','TCT', 'CTG', 'TGA', 'GAA', 'AAG']。虽然这是一个微不足道的例子，但 k-mer 在建立序列组装所用的如 De Bruijn 图这样的结构时是有用的。本例中，我们将为所给的 read 生成所有可能的 21-mer，对它们计数，然后将总数写入一个文本文件。

假设已经创建了一个名为 sc 的 SparkContext。首先，创建一个 AlignmentRecords 的 Spark RDD，采用谓词下推来移除低质量读长，并采用投影对每个读长中的 sequence 字段进行实体化：

```
// Load reads from 'inputPath' into an RDD for analysis
val adamRecords: RDD[AlignmentRecord] = sc.adamLoad(args.inputPath,

// Filter out all low-quality reads that failed vendor quality checks
predicate = Some(classOf[HighQualityReadsPredicate]),

// Only materialize the 'sequence' from each record
projection = Some(Projection(AlignmentRecordField.sequence)))
```

由于 Parquet 是一个面向列的存储格式，它能够快速地仅对 sequence 列进行实体化，并迅速跳过不需要的字段；接下来，用一个长度 $k = 21$ 的滑动窗口遍历每个 sequence，发出 1L 的计数；然后将 k-mer 子序列作为键调用 reduceByKey，以获得输入文件的总计数：

```
// The length of k-mers we want to count
val kmerLength = 21

// Process the reads into an RDD of tuples with k-mers and counts
val kmers: RDD[(String, Long)] = adamRecords.flatMap(read => {
read.getSequence
.toString
.sliding(kmerLength)
.map(k => (k, 1L))
}).reduceByKey { case (a, b) => a + b}

// Print the k-mers as a text file to the 'outputPath'
kmers.map { case (kmer, count) => s"$count,$kmer"}
.saveAsTextFile(args.outputPath)
```

当在数据集“1000 基因组计划(1000 Genomes project)”[①]中的样本 NA21144 的 11 号染色体上运行时，这一作业输出如下：

```
AAAAAAAAAAAAAAAAAAAAAA, 124069
```

① 可能是最流行的公开可用数据集，网址 *http://www.1000genomes.org*。

```
TTTTTTTTTTTTTTTTTTTTTTT, 120590
ACACACACACACACACACACAC, 41528
GTGTGTGTGTGTGTGTGTGTGT, 40905
CACACACACACACACACACACA, 40795
TGTGTGTGTGTGTGTGTGTGTG, 40329
TAATCCCAGCACTTTGGGAGGC, 32122
TGTAATCCCAGCACTTTGGGAG, 31206
CTGTAATCCCAGCACTTTGGGA, 30809
GCCTCCCAAAGTGCTGGGATTA, 30716
...
```

ADAM 能够做的可不仅仅是 *k*-mer 计数。除了已经提及的预处理阶段——对重复基因的标记(duplicate marking)、碱基质量评分重新校准、以及 indel 重新排列——它还可以做以下事情。

- 计算变异检测格式(VCF，Variant Call Format)文件中每个变异的 read 覆盖深度(coverage read depth)。
- 对 read 数据集进行 *k*-mers/*q*-mers 计数。
- 从基因转译格式(GTF，Gene Transfer Format)文件加载基因标注，并输出相应的基因模型。
- 打印 read 数据集之中所有 read 的统计信息(例如，定位至参考基因组的百分比、重复的数量、跨染色体定位的 read 等)。
- 启动遗留变异检测器，将 read 以管道方式输入 *stdin*，然后从 *stdout* 保存输出。

自带一个基础的基因组浏览器以便在 web 浏览器中查看 read

然而，ADAM 所提供的最重要的是一个开放的、可扩展的平台。所有产品都发布到 Maven Central(*http://search.maven.org*，搜索组 ID `org.bdgenomics`)，便于开发者从 ADAM 所提供的基础中受益。ADAM 数据存储在 Avro 和 Parquet 中，这样你也能使用像 SparkSQL、Impala、Apache Pig、Apache Hive 这样的系统或其他系统来分析数据。ADAM 也支持用 Scala、Java 和 Python 写成的作业，支持更多语言的产品正在开发中。

2014 年巴黎召开的 Scala.IO 大会上，Andy Petrella 和 Xavier Tordoir 在 1000 个基因组数据集上，基于 ADAM 平台并使用 Spark 的 MLlib kmeans 进行群体分层(population stratification，是指将个人基因组指定到祖先组的过程)。他们发现 ADAM/Spark 的使用可以将性能提升 150 倍。

23.10 从个性化广告到个性化医疗

ADAM 设计用于快速并可扩展地分析已比对的 read，它自己并不比对 read，而是依赖于标准的 short-reads 比对器。可扩展核苷酸比对项目(SNAP，Scalable Nucleotide Alignment Program，*http://snap.cs.berkeley.edu/*)是多家机构合作的项目，参与者包括微软研究院、加州大学旧金山分校(UCSF)、AMPLab 以及开源开发者，几家共享 Apache 2.0 许可证。SNAP 比对器和目前的顶级比对器(像 BWA-mem、Bowtie2 和 Novalign)一样精确，但是运行速度快 3～20 倍。在医生急于识别病原体时这一速度优势是十分重要的。

2013 年，一个男孩有脑炎的症状，发热和头疼，在四个月里三次去威斯康辛大学医院和诊所的急诊部就诊。在做了大量血液检查、脑部扫描和活组织检查之后，都无法确诊。最终住院治疗。五个星期之后，孩子开始发生癫痫，需要置于药物性昏迷(medically induced coma)状态。绝望之中，医生对其脊髓液取样，并送至加州大学旧金山分校(UCSF)医学实验室的华裔教授邱华彦(Charles Chiu)领导的一个实验项目，在那里对样本进行测序分析。SNAP 的速度和精确度使得 UCSF 快速过滤掉所有人类 DNA，并从剩余的 0.02%read 中，识别到一种稀有的传染性细菌 *Leptospira santarosai*。仅仅在送来样本两天之后，他们便将这一发现报告给了威斯康辛的医生。该男孩进行十天的抗生素治疗之后从昏迷中醒来，两周之后被允许出院[①]。

针对有兴趣深入了解邱教授实验室使用的系统“基于序列的超高速病原体识(SURPI，Sequence-based Ultra-Rapid Pathogen Identification(*http://chiulab.ucsf.edu/surpi*)的人，他们慷慨地通过一个 BSD 许可证分享其软件，并提供预装 SURPI 的 Amazon EC2 机器映像(AMI，Amazon EC2 Machine Image)。SURPI 从大量来源收集了 348 922 个独特的细菌序列和 1 193 607 个独特的病毒序列，并将它们保存在 29 个 SNAP 索引数据库中，每个将近 27 GB，用于快速搜索。

当前，数据分析用于个性化广告多于个性化医疗，但未来不会是这样。通过个性化医疗，人们将得到关注其自身特有 DNA 图谱(DNA profiles)的定制化健康医疗服务。随着测序价格的下降和更多人的基因组被测序，统计能力的增加将使得研

① 详见 Michael Wilson 等人的文章，标题为“Actionable Diagnosis of Neuroleptospirosis by Next-Generation Sequencing”，网址为 *http://www.nejm.org/doi/pdf/10.1056/NEJMoa1401268*，2014 年 6 月刊发于《新英格兰医学杂志》。

究人员能够理解基础疾病的遗传机制，并将这些发现结合进个性化医疗模型，以改进后续对患者的治疗。虽然今年全球仅生成了 25 PB 基因组数据，但明年这一数字将很可能为 100 PB。

23.11 联系我们

虽然我们已开启了一段伟大的旅程，但 ADAM 项目仍然是一个实验平台并需要进一步开发。如果你有兴趣学习更多关于 ADAM 上的编程或想贡献代码，请参阅 O'Reilly 在 2014 年出版的 Sandy Ryza 的 *Advanced Analytics with Spark: Patterns for Learning from Data at Scale* 一书，其中有一章专门讲如何用 ADAM 和 Spark 分析基因组数据。你可以通过以下方式联系我们：访问 *http://bdgenomics.org*、IRC 上使用#adamdev 或 Twitter 上是@bigdatagenomics。

第 24 章

开源项目 Cascading

(作者：Chris K. Wensel)

Cascading 是一个开源 Java 库和 API，它为 MapReduce 提供了一个抽象层，允许开发者建立可以在 Hadoop 集群上运行的复杂、任务关键型的数据处理应用。

Cascading 项目始于 2007 年之夏。首次公开发布的版本 0.1 版则诞生于 2008 年 1 月。版本 1.0 在 2009 年 1 月发布。可以从该项目网站(*http://www.cascading.com*)下载二进制文件、源代码和附加模块。

map 和 reduce 操作提供了功能强大的原语。但是，对于创建能够在不同开发者之间共享的复杂且高度可组合性代码来说，它们的粒度层级往往不符合要求。而且很多开发者发现，当面对现实世界的问题时，用 MapReduce 的技术概念进行“思考”是件困难的事。

为了解决第一个问题，Cascading 用简单的字段名和一个数据元组模型(data tuple model)来代替 MapReduce 中使用的键(keys)和值(values)，其中元组仅仅是一个值的列表。为了解决第二个问题，Cascading 通过引进更高级抽象作为替代而直接从 map 和 reduce 操作分离出来，这些抽象为 `Functions`，`Filters`，`Aggregators` 和 `Buffers`。

其他一些替代方案大约也是在 Cascading 项目最初公开发布的同时出现的。考虑到大部分替代架构强置要求了前置(pre-)和后置(post-)约束条件，或者其他一些预期，Cascading 被设计用作这些替代方案的补充。

例如，在其他几个 MapReduce 工具里，运行应用之前，必须将数据预格式化、过滤或导入到 HDFS 中，准备数据的步骤必须在编程抽象的外部执行。相比之下，在 Cascading 中，准备和管理数据则是编程抽象必不可少的一部分。

本实例研究从介绍 Cascading 主要概念开始，然后以概述 ShareThis (*http://www.sharethis.com*)[①]如何在其基础设施中使用 Cascading 作为结束。

为了获取更深入的有关 Cascading 处理模型的介绍，请查看项目网站上的“Cascading 用户指南”，网址为 *http://www. cascading.org/documentation/*。

24.1 字段、元组和管道

MapReduce 模型使用键和值实现输入数据和 map 函数、map 函数和 reduce 函数、reduce 函数和输出数据的连接。

但是众所周知，现实世界的 Hadoop 应用通常包含不止一个链接在一起的 MapReduce 作业。思考一下 MapReduce 中实现的标准字数统计范例。如果需要将数值型计数结果按降序排列，这是一个不太可能实现的要求，需要在第二个 MapReduce 作业中完成。

这样，在抽象层面，键和值不仅将 map 和 reduce 连接在一起，也连接 reduce 到下一个 map，然后再到下一个 reduce，以此类推(参见图 24-1)。也就是说，键/值对来源于输入文件，流经 map 和 reduce 操作组成的链，最终停留于一个输出文件。当实现了足够多的这些链接在一起的 MapReduce 应用，将会看到一个定义良好的键-值操作集合，它被反复使用以修改键-值数据流。

Cascading 简化了这一过程，它将键和值抽象出来并用具有相应字段名的元组来替代它们，类似于关系型数据库中表和列名的概念。在处理过程中，当这些字段和元组流经由管道连接在一起的用户自定义操作时，就会得到相应的操作，具体如图 24-2 所示。

① 译者注：ShareThis 是一个社会化分享应用开发平台，一个致力于互联网分享工具开发的创业平台。

图 24-1. MapReduce 中的计数和排序

图 24-2. 由字段和元组连接的管道

这样一来，Map Reduce 的键和值就被简化为以下形式。

- 字段(fields)
  一个字段是 `String` 名称(比如“first_name”)、数值型位置(比如 2 或-1，分别对应第三和最后的位置)，或两者的组合。这样，字段被用于声明元组中值的名称，以及从元组中通过名称选取值。后者类似于 SQL 的 `select` 调用。

- 元组(tuples)
  一个元组其实是一组 `java.lang.Comparable` 对象。一个元组非常像数据库的一行或一条记录。

而 map 和 reduce 操作被抽象于一个或多个管道实例之后(参见图 24-3)。

- Each
  `Each` 管道一次处理一个输入元组。它可以对输入元组应用 `Function` 或 `Filter` 操作(简短描述)。

- GroupBy
  GroupBy 管道依据分组字段(grouping fields)对元组进行分组。它的行为类似 SQL 中的 GROUP BY 语句。它也能将多个输入元组流合并为单个流，如果它们全都共享相同的字段名。

- CoGroup
  CoGroup 管道通过共同字段名将多个元组流连接在一起，并且也通过共同分组字段对元组进行分组。所有标准的连接类型(内部的、外部的等等)和自定义连接都可以对两个或更多个元组流实施。

- Every
  Every 管道一次处理一个元组分组，该分组是 GroupBy 或 CoGroup 管道操作的结果。Every 管道可对分组过程应用 Aggregator 或 Buffer 操作。

- SubAssembly
  SubAssembly 管道允许在单个管道内的组装嵌套(nesting of assemblies)，该管道可进而被嵌入更复杂的组装。

图 24-3. 管道类型

所有这些管道链接在一起被开发者装入“管道组装”(pipe assemblies)，其中每个组装可有很多输入元组流(sources)和很多输出元组流(sinks)。如图 24-4 所示。

表面上，这也许看起来比传统 MapReduce 模型更复杂。并且，不可否认这里有比 map，reduce，键和值更多的概念。但实际上，有更多的概念必须协同工作以提供不同的行为。

图 24-4.一个简单的管道组装

例如，一个开发者如果想要提供对 reducer 值的“辅助排序”，将需要实现一个 map、一个 reduce、一个“复合”(composite)键(两个键嵌套在一个父键中)、一个值、一个分区器(partitioner)、一个“输出值分组”比较器(comparator)以及一个“输出键”比较器，所有这些将以各种方式彼此耦合，并且有很大的可能性在随后的应用中不能重用。

在 Cascading 中，这将是一行代码：`new GroupBy(<previous>, <grouping fields>, <secondary sorting fields>)`，其中`<previous>`来自于之前的管道。

24.2 操作

正如之前所提到的，通过引入替代操作(operation)，Cascading 从 MapReduce 分离出来，这些替代操作被应用于个体元组或元组的分组(参见图 24-5)。

图 24-5. 操作类型

- Function
  Function 对单个输入元组进行操作，对于每一个输入可能返回 0 或更多输出元组。Function 操作由 Each 管道使用。

- Filter
  Filter 是一个特殊种类的函数，返回的是布尔值，指示当前输入元组是否要从元组流中移除。Function 也用作这一目的，但是 Filter 对于这一情况是经过优化的，并且很多过滤器可通过“逻辑”过滤器(如 AND、OR、XOR 和 NOT)进行分组，从而快速创建更复杂的过滤操作。

- Aggregator
  Aggregator 针对一组元组执行一些操作，这些元组通过共同的字段值集合进行分组(例如，所有具有相同“last-name”值的元组)。常见的 Aggregator 实现有 Sum，Count，Average，Max 和 Min。

- Buffer
  Buffer 与 Aggregator 类似，但为了在特定的分组过程中能用作“滑动窗口”，在所有的元组之间滑动，对 Buffer 进行了优化。当开发者需要在一个有序元组集合里高效插入缺失值(比如缺失的日期或持续时间)或创建一个滑动平均(running average)时，这是非常有用的。当处理元组分组时，Aggregator 通常是首选操作，因为很多 Aggregators 可以很高效的链接在一起，但有时 Buffer 是处理作业的最佳工具。

当一个管道组装被创建时，各项操作就和管道绑定在一起了，如图 24-6 所示。

图 24-6. 操作的组装

管道 Each 和 Every 提供了一种简单的机制，可以在值被传递到子操作之前，从一个输入元组中选出部分或全部值。并且有一种简单的机制用于将操作结果与原

始输入元组合并，以创建输出元组。无需深入细节，这种方式允许每个操作只关注作为参数的元组值和字段，而不用关注当前输入元组中的整个字段集。随后，操作能够以与 Java 方法重用同样的方式，在应用之间重用。

例如在 Java 中，一个被声明为 concatenate(String first, String second) 的方法比 concatenate(Person person) 更加抽象。第二种情形中，concatenate()函数必须“知道”Person 对象；而在第一种情形中，数据从何而来是不可知的。Cascading 操作展示了相同的特性。

24.3 Taps，Schemes 和 Flows

在先前的许多图中，提到了“source”和“sink”。在 Cascading 中，所有数据从 Tap 实例读取或写入 Tap 实例，但是通过 Scheme 对象与元组实例来回转换。

- Tap
  Tap 负责“如何”以及“到何处”访问数据。例如，数据是在 HDFS 还是在本地文件系统中？是通过 Amazon S3 还是通过 HTTP 获取？
- Scheme
  Scheme 负责读取原始数据并转换成元组和/或将元组写入原始数据之中，原始数据可以是文本行、Hadoop 二进制序列文件或某专有格式。

注意，Taps 不是管道组装的一部分，因此它们不是一种 Pipe。但如果它们要成为集群可执行的，就要和管道组装连接在一起。当一个管道组装与必要数量的 source 和 sink 的 Tap 实例相连接时，我们便得到了一个 Flow。这些 Tap 发出或捕获管道组装期望的字段名。也就是说，如果一个 Tap 发出一个带有字段名“line”的元组(通过从 HDFS 上的文件里读取数据)，管道组装的头部(head)一定会期待着一个“line”值。否则，连接管道组装和这些 Tap 的进程将立即因为错误而失效。

所以管道组装实际上是一种对数据处理的定义，并且其自身并不可“执行”。在能够在集群上运行之前，必须要与 source 和 sink 的 Tap 实例连接。这种 Tap 和管道组装的分离，是 Cascading 为什么如此强大的原因之一。

如果你把管道组装想作一个 Java 类，那么一个 Flow 就像一个 Java 对象实例，如图 24-7 所示。也就是说，在相同的应用中，相同的管道组装可被多次“实例化

(instantiated)”为新的 Flow，而无需担心它们之间会互相干扰。这使得能像标准 Java 库一样创建和共享管道组装。

图 24-7.一个 Flow 示例

24.4 Cascading 实践应用

既然我们知道了 Cascading 是什么，并且对它如何工作有了一个清楚的了解，那么按照 Cascading 写成的应用看起来是什么样的呢？详见范例 24-1。

范例 24-1. 字数统计和排序

```
Scheme sourceScheme =
new TextLine(new Fields("line"));①
Tap source =
new Hfs(sourceScheme, inputPath);②
Scheme sinkScheme = new TextLine();③
Tap sink =
new Hfs(sinkScheme, outputPath, SinkMode.REPLACE);④
Pipe assembly = new Pipe("wordcount");⑤
String regexString = "(?<!\\pL)(?=\\pL)[^ ]*(?<=\\pL)(?!\\pL)";
Function regex = new RegexGenerator(new Fields("word"), regexString);
assembly =
new Each(assembly, new Fields("line"), regex);⑥
assembly =
new GroupBy(assembly, new Fields("word"));⑦
Aggregator count = new Count(new Fields("count"));
assembly = new Every(assembly, count);⑧
assembly =
new GroupBy(assembly, new Fields("count"), new Fields("word"));⑨
FlowConnector flowConnector = new FlowConnector();
Flow flow =
flowConnector.connect("word-count", source, sink, assembly);⑩
flow.complete();⑪
```

① 创建一个新的 Scheme，读取简单的文本文件，并为名为“line”的字段中的每一行发布一个新的 Tuple，用 Fields 实例作为声明。

③ 创建一个新的 Scheme，写入简单的文本文件，允许一个 Tuple 具有任意数目字段/值。如果存在不止一个值，它们将在输出文件中通过制表符分隔。

②④ 创建 source 和 sink 的 Tap 实例，分别指向输入文件和输出目录。sink 的 Tap 实例将重写任何可能已经存在的文件。

⑤ 构建管道组装的头部，并将其命名为“wordcount”。这一名称被用来将 source 和 sink 的 Tap 与管道组装绑定。如果头部或尾部有多个，那么命名时需要将它们区分开。

⑥ 用一个函数构建一个 Each 管道，该函数将把遇到的每个字的“line”字段解析进一个新的 Tuple。

⑦ 构建一个 GroupBy 管道，它将为字段“word”中的每个唯一值创建一个新的 Tuple 分组。

⑧ 用一个 Aggregator 构建一个 Every 管道，该 Aggregator 将计算第 (7) 步每个分组中的 Tuple 数量。结果存储于名为“count”的字段中。

⑨ 构建一个 GroupBy 管道，它将为字段“count”中的每个唯一值创建一个新的 Tuple 分组，并根据字段“word”中的值进行辅助排序。结果将是一个“count”和“word”值的列表，其中“count”按升序排序。

⑩ ⑪ 在一个 Flow 中将管道组装连接至它的 source 和 sink，然后在集群上执行该 Flow。

在本例中，对输入文档中遇到的字数进行了统计，并按自然顺序(递增)对计数进行排序。如果一些字具有相同的“count”值，这些字将按它们的自然顺序(字母顺序)进行排序。

本例的一个明显问题在于某些情况下一些字可能具有大写字母——例如，“the”和出现在句首的“The”。可以考虑插入一个新的操作将所有字强制变为小写，但是我们认识到未来所有需要从文档解析字的应用都将具有相同的行为，因此创建一个名为 SubAssembly 的可重用管道作为解决方案，正如在一个传统应用中我们会通过创建一个子程序(subroutine)来解决该问题一样，详见范例 24-2。

范例 24-2. 创建一个 SubAssembly

```
public class ParseWordsAssembly extends SubAssembly ①
  {
  public ParseWordsAssembly(Pipe previous)
  {

  String regexString = "(?<!\\pL)(?=\\pL)[^ ]*(?<=\\pL)(?!\\pL)";
  Function regex = new RegexGenerator(new Fields("word"), regexString);
  previous = new Each(previous, new Fields("line"), regex);
  String exprString = "word.toLowerCase()";
  Function expression =
    new ExpressionFunction(new Fields("word"), exprString, String.class); ②
  previous = new Each(previous, new Fields("word"), expression);

  setTails(previous); ③
  }
}
```

(1) 对 SubAssembly 类划分子类，SubAssembly 类自身就是一种 Pipe 管道。

(2) 创建一个 Java 表达式函数，它将对“word”字段中的 String 值调用 toLowerCase()。调用时必须传入表达式期望“word”所属的 Java 类型，在本例中，该类型是 String。底层使用的是 Janino 编译器(*http://www.janino.net*)。

(3) 告诉 SubAssembly 超类管道子组装的尾端在哪里。

首先，创建一个 SubAssembly 管道来支持用于解析字词(parse words)的管道组装。由于这是一个 Java 类，因此只要有名为“word”的字段输入(参见范例 24-3)，它可被重用于任何其他应用中。注意，有一些方法能使这一函数更加通用，详情请查阅“Cascading 用户指南”(*http://www.cascading.org/documentation*)。

范例 24-3. 用一个 SubAssembly 来扩展字数统计和排序

```
Scheme sourceScheme = new TextLine(new Fields("line"));
Tap source = new Hfs(sourceScheme, inputPath);

Scheme sinkScheme = new TextLine(new Fields("word", "count"));
Tap sink = new Hfs(sinkScheme, outputPath, SinkMode.REPLACE);

Pipe assembly = new Pipe("wordcount");

assembly =
    new ParseWordsAssembly(assembly);  (1)

assembly = new GroupBy(assembly, new Fields("word"));

Aggregator count = new Count(new Fields("count"));
```

```
assembly = new Every(assembly, count);

assembly = new GroupBy(assembly, new Fields("count"), new Fields("word"));

FlowConnector flowConnector = new FlowConnector();

Flow flow = flowConnector.connect("word-count", source, sink, assembly);

flow.complete();
```

① 用 ParseWordsAssembly 管道代替来自之前范例中的 Each 管道。

最后，我们只是用新的 `SubAssembly` 替换了之前范例中所用的 `Each` 和字词解析函数。这一嵌套可以在必要时继续深入。

24.5 灵活性

让我们退后一步来看看新的模型给了我们什么，或者，它带走了什么。

可以看到，我们不再考虑 MapReduce 作业，或 `Mapper` 和 `Reducer` 接口实现，以及如何将后继 MapReduce 作业绑定或链接至前续作业。在运行期间，Cascading 规划软件“Planner”计算出将管道组装分割成 MapReduce 作业并管理它们之间联系的最佳方案，参见图 24-8。

图 24-8. 一个 Flow 如何转换为链接的 MapReduce 作业

正因如此，开发者能够构建任意粒度的应用。他们可以从一个仅过滤日志文件的小应用开始，然后根据需要将更多功能迭代构建到该应用中。

因为 Cascading 是一个 API 而不是像 SQL 字符串那样的语法，所以它更加灵活。首先，开发者能够使用他们喜欢的语言——比如 Groovy、JRuby、Jython、Scala 及其他，详见项目网站 *http://www.cascading.org/*)来创建特定领域语言(DSLs，

domain-specific languages)。其次，开发者能够扩展 Cascading 的不同部分，比如可以定制要读/写的 Thrift 或 JSON 对象，并允许它们通过元组流传递。

24.6 ShareThis 中的 Hadoop 和 Cascading

ShareThis(*http://www.sharethis.com*)是一个让分享任何在线内容变得简单的共享网络。通过点击网页上按钮或浏览器插件，ShareThis 使得用户只要在线，可以无缝访问他们的联系人和网络，并通过 email、IM、Facebook、Digg、移动 SMS 以及类似服务分享内容，甚至无需离开当前页面。发布者能够部署 ShareThis 按钮来挖掘该服务的全球共享能力，从而驱动流量、激发病毒式传播，以及追踪在线内容的共享。ShareThis 通过减少网页上的混乱以及提供跨越社交网络、会员组和社区的即时内容分发，也简化了社交媒体服务。

因为 ShareThis 用户通过在线小工具(widgets)分享页面和信息，一个持续的事件流便进入了 ShareThis 网络。这些事件首先被过滤和处理，然后交给不同的后端系统，包括 AsterData、Hypertable 和 Katta。

这些事件的量可能很巨大，以至于无法用传统系统处理。这一数据也有可能会由于来自欺诈系统、浏览器漏洞或有缺陷的小工具的“注入式攻击”而非常脏(dirty)。为此，ShareThis 的开发者选择在后端系统前部署 Hadoop 进行预处理和编排(orchestration)。他们也选择使用 Amazon Web Services，以便将服务器托管在弹性计算云(EC2，Elastic Computing Cloud)上，实现在简单存储服务(S3，Simple Storage Service)上提供长期存储，同时也期望利用弹性 MapReduce(EMR，Elastic MapReduce)。

在本章概述中，我们将关注于“日志处理管线”(log processing pipeline) (如图 24-9 所示。这一管线只是获取存储于 S3 存储段(bucket)中的数据，对其进行处理，并将结果存储进另一个存储段。简单队列服务(SQS：Simple Queue Service)用于协调那些标记数据处理运行开始和结束的事件。在下游，其他进程拉(pull)数据载入 AsterData，从 Hypertable 拉取 URL 列表来获取网络爬虫(web crawl)的源头，或拉取被抓取(crawled)的页面数据来创建 Katta 所使用的 Lucene 索引。注意，Hadoop 对于 ShareThis 架构极为重要，它被用于协调架构组件之间数据的处理和移动。

图 24-9. ShareThis 日志处理管线

有了 Hadoop 作为前端，所有事件日志在被载入 AsterData 集群或被任何其他组件使用之前，都能够根据一个规则集进行解析、过滤、清理和组织。AsterData 是一个集群化数据仓库，支持大的数据集，并允许基于标准 SQL 语法的复杂特定查询。ShareThis 选择在 Hadoop 集群上清理和准备输入的数据集，然后将数据载入 AsterData 集群用于特定分析和报告。尽管 AsterData 也能完成这一处理，但用 Hadoop 作为处理管线的第一环节仍然具有重要的意义，因为可以抵消主数据仓库的负荷。

为简化开发流程，Cascading 被选为主要的数据处理 API，制定数据在架构组件之间的协调规则，并提供面向开发者的组件接口。这代表了同更为“传统的”Hadoop 应用案例的分离，这些案例本质上只是查询存储的数据。Cascading 和 Hadoop 一起为完整的解决方案提供了更好且更简单的端到端结构，这样可以为用户提供更多价值。

对于开发者来说，Cascading 使得编程更简单，程序员可以从执行简单文本解析的一个简单单元测试(通过对 `cascading.ClusterTestCase` 子类化而创建)开始，然后逐层添加更多的处理规则，同时为便于维护，保持应用根据逻辑进行组织。Cascading 提供两种方式支持这种逻辑组织。第一种方式，独立的操作(`Functions`, `Filters` 等等)能够独立地被编写和测试。第二种方式，应用被分为多个阶段：一个用于解析，一个用于规则，而最后一个阶段用于装箱/整理(binning/collating)数据，所有这些都是通过之前描述的基类 `SubAssembly` 来完成的。

来自 ShareThis 日志记录器的数据看起来很像 Apache 日志，具有日期/时间戳、共享 URLs、推荐者 URLs 以及少量元数据。为了将数据用于下游的分析，URLs 需要被拆包(解析出查询字符串、域名等等)。为此，创建一个高层的 SubAssembly 来封装解析过程。如果一些特定字段十分复杂难以解析，那么可以在 SubAssembly 中嵌套子组装(child subassemblies)来处理这些字段解析。

当使用规则时，需要做同样的事。当每个 Tuple 通过规则 SubAssembly 传递时，一旦触发任何规则，将被标记为“bad”。与“bad”标签一起，还有一段关于记录为何标记为“bad”的描述被添加进 Tuple 用于后续查看。

最后，创建一个分解器(splitter)SubAssembly，主要用于做两件事。第一，它允许元组流分解成两部分：“good”数据流和“bad”数据流。第二，分解器将数据装箱(bin)成一个个区间，例如每小时一个区间。为了实现上述目标，仅有两个操作是必须的：首先，根据已经出现在流中的时间戳创建区间；其次，使用区间和 *good/bad* 元数据来创建一个目录路径(例如，*05/good/*，其中“05”表示 5a.m.，而“good”表示 Tuple 通过了所有规则)。该路径将由 Cascading 的 TemplateTap 使用，这是一个特殊的 Tap，它能够根据 Tuple 中的值动态地将元组流输出到不同位置。在本章中，TemplateTap 使用“path”值来创建最终的输出路径。

开发者也创建了第四个 SubAssembly。该 SubAssembly 在单元测试期间将使用 Cascading 的 Assertion。这些 Assertion 进行双重检查，以确保规则和解析子组装各司其职。

在范例 24-4 的单元测试中，我们看到分解器未被测试，实际上它被添加进另一个未在此介绍的集成测试中。

范例 24-4. 针对 Flow 的单元测试

```
public void testLogParsing() throws IOException
  {
  Hfs source = new Hfs(new TextLine(new Fields("line")), sampleData);
  Hfs sink =
    new Hfs(new TextLine(), outputPath + "/parser", SinkMode.REPLACE);

  Pipe pipe = new Pipe("parser");

  // split "line" on tabs
  pipe = new Each(pipe, new Fields("line"), new RegexSplitter("\t"));

  pipe = new LogParser(pipe);

  pipe = new LogRules(pipe);
```

```
// testing only assertions
pipe = new ParserAssertions(pipe);

Flow flow = new FlowConnector().connect(source, sink, pipe);
flow.complete(); // run the test flow

// Verify there are 98 tuples and 2 fields, and matches the regex pattern
// For TextLine schemes the tuples are { "offset", "line" }
validateLength(flow, 98, 2, Pattern.compile("^[0-9]+(\\t[^\\t]*){19}$"));
}
```

为了集成和部署，Cascading 中构建了很多功能，以方便和外部系统集成，且获得更大的处理宽容度(process tolerance)。

在生产过程中，所有子组装连接在一起并被规划进一个 Flow，在该 Flow 中，除了 source 和 sink 的 Tap 之外，还设计了陷阱(trap)Tap(参见图 24-10)。通常，当一个操作从一个远程 mapper 或 reducer 任务中抛出异常时，该 Flow 将失效并终结它管理的所有 MapReduce 作业。如果一个 Flow 包含陷阱，那么任何异常将被捕捉，而引发异常的数据将被存入与当前陷阱相关联的 Tap。随后会接着处理下一个 Tuple 而无需停止该 Flow。有时你希望 Flow 因错误而失效，但是在这种情况下，ShareThis 开发者知道他们能够退回去查看失效数据，并在数据生产系统保持运行的同时升级他们的测试单元。损失几个小时的处理时间要比损失几个坏记录更加糟糕。

图 24-10. ShareThis 日志处理流

使用 Cascading 的事件监听器，可以集成 Amazon SQS。当一个 Flow 结束时，将发送一条消息通知其他系统数据已准备就绪，可从 Amazon S3 获取。若出现失

效，则将发送一条不同的消息，向其他进程告警。

日志处理管线在不同的独立集群上运行结束后，都将由余下的下游进程接续。目前日志处理管线一天运行一次，由于毫无必要让一个 100 个节点的集群闲置 23 小时而无所事事，所以它将退出运作并在 24 小时之后重新投入运作。

将来根据业务需要，把更小集群上的这一任务执行间隔提升至每 6 个小时或 1 小时一次将很容易。其他集群将根据负责各组件的业务单元的需求，各自以不同间隔启动和关闭。例如，网络爬虫组件(采用 Bixo，由 EMI 和 ShareThis 开发的一个基于 Cascading 的网络爬虫工具包)可在 Hypertable 集群的配合下，在一个小集群上连续不断地运行。这一按需模型与 Hadoop 结合的效果很好，可以对每个集群根据其预期处理的工作量类型进行调优。

24.7 总结

对于处理和协调数据在多种架构组件之间的移动而言，Hadoop 是一个非常强大的平台。其唯一缺点在于主要计算模型为 MapReduce。

Cascading 旨在通过一个非常合理的 API 来帮助开发者快速而简单地构建强大的应用，而无需思考 MapReduce 的具体实现，并且将数据分发、复制、分布式进程管理和活性(liveness)等繁重工作留给了 Hadoop。

想要进一步了解 Cascading，请加入在线社区，访问项目网站 *http://www.cascading.org/*，下载示例应用。

附录 A

安装 Apache Hadoop

在单机上安装 Hadoop 来尝试一下是非常容易的，面向集群的安装方法，可以参见第 10 章。

本附录将介绍如何使用一个 Apache Software Foundation 发布的二进制压缩包安装 Hadoop Common、HDFS、MapReduce 和 YARN。本书提到的其他项目的安装方法已经包含在相应章的开始部分。

另一种方法是使用一个虚拟机(例如 Cloudera 的 Quick-Start 虚拟机)进行安装，虚拟机自带有预装和配置好的 Hadoop 服务。

接下来的安装指导方法适用于基于 Unix 的系统，包括 MAC OS X(虽然它不是一个产品平台，但是非常适用于开发)。

A.1 先决条件

必须确保安装的是合适版本的 Java。可以通过 Hadoop wiki (*http://wiki.apache.org/hsdoop/HadoopJavaVersions*)查看所安装版本的信息。以下命令可以帮助确认 Java 是正确安装的。

```
% java -version
java version "1.7.0_25"
Java(TM) SE Runtime Environment (build 1.7.0_25-b15)
Java HotSpot(TM) 64-Bit Server VM (build 23.25-b01, mixed mode)
```

A.2 安装

首先，决定以什么用户的身份来运行 Hadoop。如果只是想试用 Hadoop 或或开发 Hadoop 程序，可以用用户账号在单机上运行 Hadoop。

从 Apache Hadoop 发布页面(*http://hadoop.apache.org/common/releases.html*)下载一个稳定版的发布包(通常打包为一个 gzipped tar 文件)，再解压缩到本地文件系统：

```
% tar xzf hadoop-x.y.z.tar.gz
```

在运行 Hadoop 安装程序之前，需要指定 Java 在本地系统中的路径。如果系统的 JAVA_HOME 环境变量已经正确地指向一个 Java 安装路径，且用户也希望使用该 Java 安装路径，则无需进行其他配置。(这通常在一个 shell 启动文件中设置，例如~/*.bash_profile* 或~/*.bashrc*)。否则，仍然需要编辑 *conf/hadoop-env.sh* 文件来设置 JAVA_HOME 环境变量以指定 Java 安装路径。例如，在 Mac 机器上，可如下编辑以便指向安装的某个版本 Java：

```
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.7.0_25.jdk/Contents/Home
```

很容易创建指向 Hadoop 安装目录(依照惯例是 HADOOP_HOME)的环境变量；此外，还要将 Hadoop 的二进制目录添加到命令行路径上。例如：

```
% export HADOOP_HOME=~/sw/hadoop-x.y.z
% export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
```

注意，sbin 目录下有运行 Hadoop 守护进程的脚本，因此如果计划在本地机器上运行守护进程的话，需要将该目录包含进命令行路径中。

输入以下指令来判断 Hadoop 是否工作：

```
% hadoop version
Hadoop 2.5.1
Subversion https://git-wip-us.apache.org/repos/asf/hadoop.git -r
2e18d179e4a8065b6a9f29cf2de9451891265cce
Compiled by jenkins on 2014-09-05T23:11Z
Compiled with protoc 2.5.0
From source with checksum 6424fcab95bfff8337780a181ad7c78
This command was run using /Users/tom/sw/hadoop-2.5.1/share/hadoop/common/hadoop
-common-2.5.1.jar
```

A.3 配置

Hadoop 的各个组件均可利用 XML 文件进行配置。*core-site.xml* 文件用于配置通用属性，*hdfs-site.xml* 文件用于配置 HDFS 属性，*mapred-site.xml* 文件则用于配置 MapReduce 属性，*yarn-site.xml* 文件用于配置 YARN 属性。这些配置文件都放在 *etc/hadoop* 子目录中。

上述配置文件中的属性都有默认设置，分别保存在 Hadoop 安装路径的 `share/doc` 子目录下的四个 HTML 文件中，即 `core-default.html`、`hdfs-default.html`、`mapred-default.html` 和 `yarn-default.xml`。

Hadoop 有以下三种运行模式。

- **独立(或本地)模式** 无需运行任何守护进程，所有程序都在同一个 JVM 上执行。在独立模式下测试和调试 MapReduce 程序很方便，因此该模式在开发阶段比较合适。
- **伪分布模式** Hadoop 守护进程运行在本地机器上，模拟一个小规模的集群。
- **全分布模式** Hadoop 守护进程运行在一个集群上。此模式的设置请参见第 10 章。

在特定模式下运行 Hadoop 需要关注两个因素：正确设置属性和启动 Hadoop 守护进程。表 A-1 列举了配置各种模式所需要的最小属性集合。在独立模式下，将使用本地文件系统和本地 MapReduce 作业运行器；在分布模式下，将启动 HDFS 和 YARN 守护进程，此外还需配置 MapReduce 以便能够使用 YARN。

表 A-1. 不同模式的关键配置属性

组件名称	属性名称	独立模式	伪分布模式	全分布模式
Common	fs.defaultFS	file:///(默认)	hdfs://localhost/	*hdfs://namenode/*
HDFS	dfs.replication	N/A	1	3 (默认)
MapReduce	mapreduce.framework.name	local (默认)	Yarn	yarn
YARN	yarn.resourcemanager.hostname	N/A	Localhost	*resourcemanager*
	yarn.nodemanager.aux-services	N/A	mapreduce_shuffle	mapreduce_shuffle

可以阅读 10.3 节，了解更多配置信息。

A.4 独立模式

由于默认属性专为本模式所设定，且本模式无需运行任何守护进程，因此在独立模式下不需要更多操作。

A.5 伪分布模式

在伪分布模式下，使用如下内容创建配置文件，并将其放在 *etc/hadoop* 目录下。也可以把 *etc/hadoop* 目录复制到另一个位置，然后把*-*site.xml* 这些配置文件放在该目录下。这种方法的优点是，将配置设置和安装文件隔离开。如果是这么做的，那么需要将环境变量 HADOOP_CONF_DIR 设置成指向那个新目录，或确定在启动守护进程时使用--config选项)。

```
<?xml version="1.0"?>
<!-- core-site.xml -->
<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://localhost/</value>
  </property>
</configuration>

<?xml version="1.0"?>
<!-- hdfs-site.xml -->
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
</configuration>

<?xml version="1.0"?>
<!-- mapred-site.xml -->
<configuration>
  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
</configuration>

<?xml version="1.0"?>
<!-- yarn-site.xml -->
<configuration>
```

```
  <property>
    <name>yarn.resourcemanager.hostname</name>
    <value>localhost</value>
  </property>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
    </property>
</configuration>
```

1. 配置 SSH

如前所述，在伪分布模式下工作时必须启动守护进程，而启动守护进程的前提是使用需要提供的脚本成功安装 SSH。Hadoop 并不严格区分伪分布模式和全分布模式，它只是在集群内的(多台)主机(由 *slaves* 文件定义)上启动守护进程：SSH 连接到各个主机并启动一个守护进程。伪分布模式是全分布模式的一个特例。在伪分布模式下，(单)主机就是本地计算机(localhost)，因此需要确保用户能够用 SSH 连接到本地主机，并且可以不输入密码登录。

首先，确保 SSH 已经安装，且服务器正在运行。例如，在 Ubuntu 上可通过以下指令进行测试：

```
% sudo apt-get install ssh
```

在 Mac OS X 系统中，确保远程登录(在系统“首选项”|“共享”下)对当前用户(或所有用户)可用。

然后基于空口令生成一个新 SSH 密钥，以实现无密码登录。

```
% ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
% cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

如果正运行的是 *ssh-agent*，可能还需要运行 *ssh-add*。

用以下指令测试是否能够连接：

```
% ssh localhost
```

如果成功，则无需键入密码。

2. 格式化 HDFS 文件系统

在首次使用 Hadoop 前，必须格式化文件系统。只需键入如下指令：

```
% hdfs namenode -format
```

3. 启动和终止守护进程

为启动 HDFS、YARN 和 MapReduce 守护进程，键入如下指令：

```
% start-dfs.sh
% start-yarn.sh
% mr-jobhistory-daemon.sh start historyserver
```

如果配置文件并不位于默认的 *conf* 目录中，则或者在运行脚本前输出 HADOOP_CONF_DIR 环境变量，或者启动守护进程时使用--config 选项，该选项带有一条绝对路径指向配置目录：

```
% start-dfs.sh --config path-to-config-directory
% start-yarn.sh --config path-to-config-directory
% mr-jobhistory-daemon.sh --config path-to-config-directory
start historyserver
```

本地计算机将启动以下守护进程：一个 namenode、一个辅助 namenode、一个 datanode(HDFS)、一个资源管理器、一个节点管理器(YARN)和一个历史服务器(MapReduce)。可以浏览 *logs* 目录(在 Hadoop 安装目录下)中的日志文件来检查守护进程是否成功启动，或通过 Web 界面：在 *http://localhost:50070/* 查看 namenode、在 *http://localhost:8088/* 查看资源管理器、在 *http://localhost:19888/*查看历史服务器。此外，Java 的 jps 命令也能查看守护进程是否正在运行。

终止守护进程也很容易，示例如下：

```
% mr-jobhistory-daemon.sh stop historyserver
% stop-yarn.sh
% stop-dfs.sh
```

4. 创建用户目录

为自己创建一个主目录需键入以下指令：

```
% hadoop fs -mkdir -p /user/$USER
```

A.6 全分布模式

在集群上安装 Hadoop 还需要考虑更多因素，所以第 10 章专门对此模式进行了全面的描述。

附录 B

关于 CDH

CDH(全称 Cloudera's Distribution Including Apache Hadoop，即 Cloudera 公司发布的 Apache Hadoop)是一个集成的基于 Apache Hadoop 的栈，包含了系统开发所需的所有组件，这些组件已经经过测试、并且被打包在一起了。Cloudera 公司的这款发布包有多种形式，包括 Linux 包、虚拟机映像、tar 文件和在云上运行 CDH 的工具。CDH 是在 Apache 2.0 许可下发布的自由软件，用户可从 *http://www.cloudera.com/cdh* 获得。

以 CDH5 为例，它包含下列包，其中许多包在本书中做过介绍。

- *Apache Avro*
  一个跨语言的数据序列化库，包括丰富的数据结构、一种快速压缩的二进制格式和 RPC。
- *Apache Crunch*
  高层 Java API 库，用于创建可以在 MapReduce 或 Spark 上运行的数据处理管线。
- *Apache DataFu(孵化中)*
  非常有用的统计类用户定义函数集 (UDF)库，用于大规模分析。
- *Apache Flume*
  高可靠、可配置的数据流集合。
- *Apache Hadoop*
  高度可伸缩的数据存储(HDFS)、资源管理(YARN)和数据处理(MapReduce)。
- *Apache HBase*
  面向列的实时数据库，支持随机读/写访问。

- *Apache Hive*
  用于大数据集合的类 SQL 查询和表
- *Hue*
  提供 Web UI 界面，便于操作 Hadoop 数据。
- *Cloudera Impala*
  支持面向 HDFS 或 HBase 的交互式、低延迟 SQL 查询。
- *Kite SDK*
  API 库，范例和文档，用于构建 Hadoop 顶层应用。
- *Apache Mahout*
  可伸缩的机器学习和数据挖掘算法
- *Apache Oozie*
  用于相互依赖的 Hadoop 作业的工作流调度器。
- *Apache Parquet (孵化中)*
  一种有效的支持嵌套式数据的列式存储格式。
- *Apache Pig*
  支持大数据集开发的数据流语言。
- *Cloudera Search*
  自由文本、Google 风格的搜索方案，支持对 Hadoop 数据的搜索。
- *Apache Sentry (孵化中)*
  对 Hadoop 用户的基于角色的细粒度访问控制。
- *Apache Spark*
  支持 Scala、Java 和 Python 语言的集群计算框架，支持基于内存的大规模数据处理。
- Apache Sqoop
  支持结构化数据存储(如关系型数据库)和 Hadoop 之间的高效数据传输。
- Apache ZooKeeper
  面向分布式应用的高可用协调服务。

Cloudera 还提供 *Cloudera manager* 来支持部署和操作 CDH 的 Hadoop 集群。

要下载 CDH 和 Cloudera manager，请访问 *http://www.cloudera.com/ downloads/*。

附录 C

准备 NCDC 气象数据

这里首先简要介绍如何准备原始气象数据文件，以便我们能用 Hadoop 对它们进行分析。如果打算得到一份数据副本供 Hadoop 处理，可按照本书配套网站(网址为 *http://www.hadoopbook.com/*)给出的指导进行操作。接下来，首先说明如何处理原始的气象文件。

原始数据实际是一组经过 *bzip2* 压缩的 *tar* 文件。每个年份有价值的数据单独放在一个文件中。部分文件列举如下：

```
1901.tar.bz2
1902.tar.bz2
1903.tar.bz2
...
2000.tar.bz2
```

各个 *tar* 文件包含一个 *gzip* 压缩文件，描述某一年度所有气象站的天气记录。(事实上，由于在存档中的各个文件已经预先压缩过，因此再利用 `bzip2` 对存档压缩就稍显多余了)。示例如下：

```
% tar jxf 1901.tar.bz2
% ls -l 1901 | head
011990-99999-1950.gz
011990-99999-1950.gz
...
011990-99999-1950.gz
```

由于气象站数以万计，所以整个数据集实际上是由大量小文件构成的。鉴于 Hadoop 对少量的大文件的处理更容易、更高效(参见 8.2.1 节)，所以在本例中，我们将每个年度的数据解压缩到一个文件中，并以年份命名。上述操作可由一个 MapReduce 程序来完成，以充分利用其并行处理能力的优势。下面具体看看这个程序。

该程序只有一个 map 函数，无 reduce 函数，因为 map 函数可并行处理所有文件操作，无需整合步骤。这项处理任务能够用一个 Unix 脚本进行处理，因而在这里使用面向 MapReduce 的 Streaming 接口比较恰当。请看范例 C-1。

范例 C-1. 利用 bash 脚本来处理原始的 NCDC 数据文件并将其存储在 HDFS 中

```
#!/usr/bin/env bash

# NLineInputFormat gives a single line: key is offset, value is S3 URI
read offset s3file

# Retrieve file from S3 to local disk
echo "reporter:status:Retrieving $s3file" >&2
$HADOOP_HOME/bin/hadoop fs -get $s3file .

# Un-bzip and un-tar the local file
target=`basename $s3file .tar.bz2`
mkdir -p $target
echo "reporter:status:Un-tarring $s3file to $target" >&2
tar jxf `basename $s3file` -C $target

# Un-gzip each station file and concat into one file
echo "reporter:status:Un-gzipping $target" >&2
for file in $target/*/*
do
  gunzip -c $file >> $target.all
  echo "reporter:status:Processed $file" >&2
Done

# Put gzipped version into HDFS
echo "reporter:status:Gzipping $target and putting in HDFS" >&2
gzip -c $target.all | $HADOOP_HOME/bin/hadoop fs -put - gz/$target.gz
```

输入是一个小的文本文件(*ncdc_files.txt*)，列出了所有待处理文件(这些文件放在 S3 文件系统中，因此能够以 Hadoop 所认可的 S3 URI 的方式被引用)。示例如下：

```
s3n://hadoopbook/ncdc/raw/isd-1901.tar.bz2
s3n://hadoopbook/ncdc/raw/isd-1902.tar.bz2
...
s3n://hadoopbook/ncdc/raw/isd-2000.tar.bz2
```

由于输入格式被指定为 `NLineInputFormat`，每个 mapper 接受一行输入(包含必须处理的文件)。处理过程在脚本中解释，但简单说来，它会解压缩 *bzip2* 文件，然后将该年份所有文件整合为一个文件。最后，该文件以 *gzip* 进行压缩并复制至 HDFS 之中。注意，可以使用指令 `hadoop fs -put -`从标准输入中获得数据。

状态消息输出到“标准错误”并以 `reporter:status` 为前缀，这样可以解释为 MapReduce 状态更新。这告诉 Hadoop 该脚本正在运行，并未挂起。

运行 Streaming 作业的脚本如下：

```
% hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
  -D mapred.reduce.tasks=0 \
  -D mapred.map.tasks.speculative.execution=false \
  -D mapred.task.timeout=12000000 \
  -input ncdc_files.txt \
  -inputformat org.apache.hadoop.mapred.lib.NLineInputFormat \
  -output output \
  -mapper load_ncdc_map.sh \
  -file load_ncdc_map.sh
```

由此易知，这是一个“只有 map”的作业，因为 reduce 任务数为 0。以上脚本还关闭了推测执行(speculative execution)，因此重复的任务不会写相同的文件(7.4.3 节所讨论的方法也是可行的)。任务超时参数被设置为一个比较大的值，使得 Hadoop 不会杀掉那些运行时间较长的任务，例如，在解档文件或将文件复制到 HDFS 时，或者当进展状态未被报告时。

最后，调用 *distcp* 将文件从 HDFS 中复制出来，再存档到 S3 文件系统中。

附录 D

新版和旧版 Java MapReduce API

本书通篇使用的 Java MapReduce API 被称为“新 API”，它替代了功能上等同的旧 API。尽管 Hadoop 同时提供新旧 MapReduce API，但它们彼此并不兼容。如果愿意的话可以使用旧 API，因为对于本书中所有 MapReduce 范例，相应的使用旧 API 的范例代码都可以到本书的配套网站下载(在 `oldapi` 包中)。

新版旧版 API 有下面几个值得注意的差异。

- 新 API 在 `org.apache.hadoop.mapreduce` 包(子包)中。旧 API 仍然在 `org.apache.hadoop.mapred` 中。
- 新 API 偏好在接口上定义抽象类，因为这样更利于演进。该特性意味着可以在一个抽象类中添加一个方法(带有默认的实现)，而无需破坏该类中旧的实现[①]。例如，旧 API 中的 `Mapper` 和 `Reducer` 接口在新 API 中都变成了抽象类。
- 新 API 大量使用了 *context object*，使得用户代码能够与 MapReduce 系统通信。例如，新的 `Context` 本质上统一了旧 API 中的以下三种角色 `JobConf`，`OutputCollector` 和 `Reporter`。
- 在新版和旧版 API 中，键值对都是被推送给 mapper 和 reducer。但是，新 API 允许 mapper 和 reducer 通过重写 `run()`方法来控制执行流。例如，可以以批为单位处理记录，或者在所有记录都被处理前终止本次执

① 从技术角度而言，如果类中原有的方法和新增的方法签名相同，那么这种改变几乎必然会破坏原有方法的实现，正如 Jim des Rivières 在“Evolving Java-based APIs”一文(网址为 *http://bit.ly/adding_api_method*)中解释的那样。出于实用目的，我们将这种改变视为可兼容的。

行。在旧 API 中，mapper 可以通过写一个 MapRunnable 类来达到同样的目的，但对于 reducer 来说没有等价的手段。

- 新 API 通过 Job 类来完成作业控制，旧 API 中对应的是 JobClient，在新 API 中已经删除该类。
- 新 API 对配置功能进行了整合。旧 API 专门有一个用于作业配置的 JobConf 对象，是 Hadoop 的 vanilla 对象 Configuration(用于配置守护进程，详情参见 6.1 节)的一个扩展。新 API 中，作业配置通过 Configuration 完成,可能会用到 Job 类的一些辅助类(helper)方法。
- 输出文件的命名略有不同：旧 API 中，map 和 reduce 输出命名格式为 *part-nnnnn*，而在新 API 中，map 输出命名为 *part-m-nnnnn*，reduce 输出命名为 *part-r-nnnnn*(*nnnnn* 是个整数，表示该部分的编号，从 00000 开始)。
- 新 API 中，用户可重写的方法声明抛出的异常是 java.lang.InterruptedException。意味着用户可以自己写代码处理中断，这样，如果需要，系统框架可以友好地取消长时间运行的操作。[①]
- 新 API 中，reduce()方法用 java.lang.Iterable 传递值，而旧 API 使用 java.lang.Iterator 传递值。这种改变使得使用 Java 的 for-each 循环结构实现值的迭代更加容易。

```
for (VALUEIN value : values) { ... }
```

使用新 API 且在 Hadoop 1 上编译的程序，如果要在 Hadoop 2 上运行，则需重新编译。这是因为新 Mapreduce API 中的一些类在 Hadoop 1 和 Hadoop 2 两个版本之间改变了接口。现象就是一个运行时的错误，如下所示：

```
java.lang.IncompatibleClassChangeError: Found interface org.apache.hadoop.
mapreduce.TaskAttemptContext, but class was expected.
```

范例 D-1 给出了一个 MaxTemperature 应用，该程序来自 2.3.2 节，这里使用旧 API 对其进行了重写。新旧的差异之处用粗体予以标注。

① 详情可以参见 Brian Goetz 的文章“Java theory and practice: Dealing with InterruptedException”，网址为 *http://bit.ly/interruptedexception*。

把 Mapper 和 Reducer 类转换为新 API 时，不要忘记将 map()方法和 reduce()方法的签名修改成新的形式。为了扩展新 Mapper 或 Reducer 类而仅修改原有的类不会导致编译错误或警告错误，因为这些类分别提供了 map()方法和 reduce()方法的标识形式。但是，你的 mapper 或 reducer 代码将不会被调用，这会导致一些难以诊断的错误。

使用@override 注释 map()和 reduce()方法，这样一来，Java 编译器就会捕捉到这些错误。

范例 D-1. 这个应用程序在气象数据集中找出最高气温(使用旧的 MapReduce API)

```
public class OldMaxTemperature {

  static class OldMaxTemperatureMapper extends MapReduceBase
      implements Mapper<LongWritable, Text, Text, IntWritable> {

    private static final int MISSING = 9999;

    @Override
      public void map(LongWritable key, Text value,
        OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {

      String line = value.toString();
      String year = line.substring(15, 19);
      int airTemperature;
      if (line.charAt(87) == '+') { // parseInt doesn't like leading plus signs
        airTemperature = Integer.parseInt(line.substring(88, 92));
      } else {
        airTemperature = Integer.parseInt(line.substring(87, 92));
      }
      String quality = line.substring(92, 93);
      if (airTemperature != MISSING && quality.matches("[01459]")) {
        output.collect(new Text(year), new IntWritable(airTemperature));
    }
  }
}

  static class OldMaxTemperatureReducer extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {

    @Override
    public void reduce(Text key, Iterator<IntWritable> values,
      OutputCollector<Text, IntWritable> output, Reporter reporter)
      throws IOException {
      int maxValue = Integer.MIN_VALUE;
      while (values.hasNext()) {
        maxValue = Math.max(maxValue, values.next().get());
      }
      output.collect(key, new IntWritable(maxValue));
    }
  }
```

```
public static void main(String[] args) throws IOException {
  if (args.length != 2) {
    System.err.println("Usage: OldMaxTemperature <input path> <output path>");
    System.exit(-1);
    }

  JobConf conf = new JobConf(OldMaxTemperature.class);
  conf.setJobName("Max temperature");

  FileInputFormat.addInputPath(conf, new Path(args[0]));
  FileOutputFormat.setOutputPath(conf, new Path(args[1]));

  conf.setMapperClass(OldMaxTemperatureMapper.class);
  conf.setReducerClass(OldMaxTemperatureReducer.class);

  conf.setOutputKeyClass(Text.class);
  conf.setOutputValueClass(IntWritable.class);

  JobClient.runJob(conf);
 }
}
```

关于作者

Tom White 是最杰出的 Hadoop 专家之一。自 2007 年 2 月以来，Tom White 一直是 Apache Hadoop 的提交者(committer)，也是 Apache 软件基金会的成员。Tom 是 Cloudera 的软件工程师，他是 Cloudera 的首批员工，对 Apache 和 Cloudera 做出了举足轻重的贡献。在此之前，他是一名独立的 Hadoop 顾问，帮助公司搭建、使用和扩展 Hadoop。他是很多行业大会的专题演讲人，比如 ApacheCon、OSCON 和 Strata。Tom 在英国剑桥大学获得数学学士学位，在利兹大学获得科学哲学硕士学位。他目前与家人居住在威尔士。

关于译者

王海博士，解放军理工大学通信工程学院教授，博导，教研中心主任，长期从事无线自组网网络的设计与研发工作，主持国家自然科学基金、国家 863 计划课题等多项国家级课题，近 5 年获军队科技进步二等奖 1 项，三等奖 6 项，作为第一发明人申请国家发明专利十余项，发表学术论文 50 余篇。

华东博士，现任南京医科大学计算机教研室教师，一直致力于计算机辅助教学的相关技术研究，陆续开发了人体解剖学网络自主学习考试平台、诊断学自主学习平台和面向执业医师考试的预约化考试平台等系统，并在各个学科得到广泛的使用，获得全国高等学校计算机课件评比一等奖和三等奖各一项。主编、副主编教材两部，获发明专利一项、软件著作权多项。

刘喻博士，长期从事软件开发、软件测试和软件工程化管理工作，目前任教于清华大学软件所。

吕粤海，长期从事军事通信网络技术研究与软件开发工作，先后通过华为光网络高级工程师认证、思科网络工程师认证。

关于封面图片

《Hadoop 权威指南》封面上的动物是非洲象。非洲象属中的大象是地球上最大的陆地动物（体形略大于其表亲亚洲象），可以通过耳朵来辨认，它们的耳朵与亚洲大陆的形状相似。成年非洲雄象高 12 英尺（约 3 米），最高能到 4.1 米，重 12 000 磅（约 5.4 吨），但重的能到 15 000 磅（约 6 .8 吨），最高记录是 10 吨。成年非洲雌象身高 10 英尺，体重 8 000～11 000 磅。甚至小象也身形高大，刚出生时体重就接近 200 磅（约 90 公斤），身高 3 英尺（约 0.9 米）左右。

非洲象生活在撒哈拉以南的非洲地区。陆地上大部分非洲象生活在稀树大草原和干旱的矮树丛。有些非洲象生活在沙漠地区，有些则生活在山区。

非洲象这一物种在其生活的丛林和草原生态系统中扮演着非常重要的角色。许多植物种子要发芽，只能通过非洲象的消化道来完成。据估计，非洲西部近三分之一树种依赖于非洲象的消化方式来发芽。非洲象食用大量的草本植物，会对植被和丛林的结构产生影响。例如，在自然条件下，非洲象食用大量草本植物之后，热带雨林中会形成空隙，阳光可以投射到这些区域，从而促进植物的生长。非洲象对许多动植物都有影响，之所以称之为基础物种，是因为它们可以使自己的栖息地持续存在。

封面图片来自于 Dover Pictorial Archive。封面字体使用的是 Adobe ITC Garamond。正文字体是汉仪书宋一简；页眉字体是汉仪中等线简+MyriadPro-regular；代码字体是 LucasFont 的 Consolas。