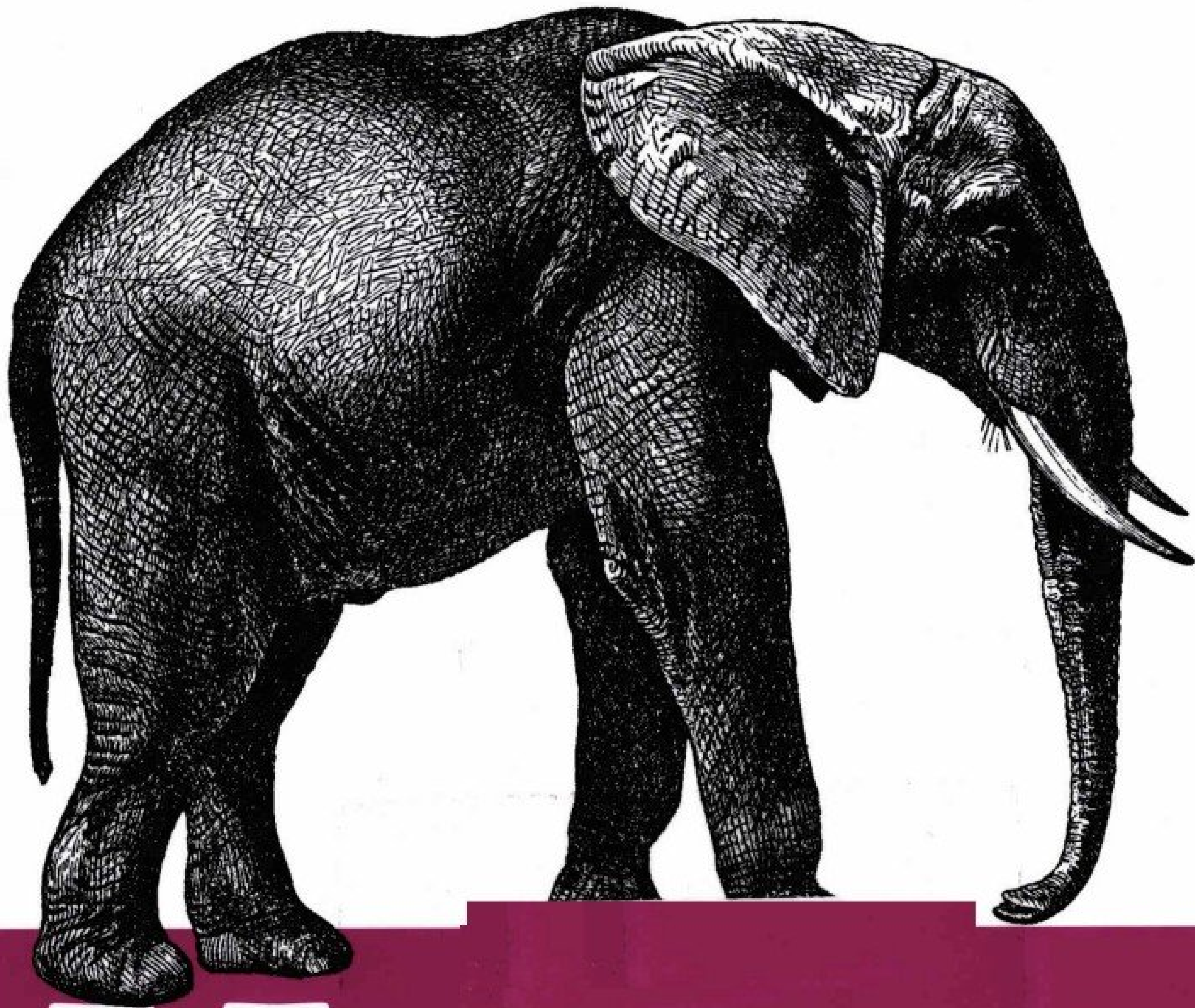
Hadoop: The Definitive Guide



Hadoop

权威指南(中文版)

Tom White 著

曾大聃 周傲英 译

周敏 审校

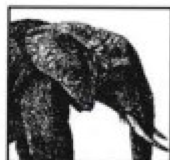
O'REILLY® | YAHOO! PRESS

清华大学出版社

Programming

O'REILLY®

Hadoop权威指南（中文版）



什么是谷歌帝国的基石？MapReduce算法是也！Apache Hadoop架构作为MapReduce算法的一种开源应用，是应对海量数据的理想工具。项目负责人Tom White透过本书详细阐述了如何使用Hadoop构建可靠、可伸缩的分布式系统，程序员可从中探索如何分析海量数据集，管理员可以从中了解如何安装和运行Hadoop集群。

本书结合丰富的案例来展示如何用Hadoop解决特殊问题，它将帮助您：

- 使用Hadoop分布式文件系统（HDFS）来存储海量数据集，通过MapReduce对这些数据集运行分布式计算
- 熟悉Hadoop的数据和I/O构件，用于压缩、数据集成、序列化和持久处理
- 洞悉编写MapReduce实际应用时的常见陷阱和高级特性
- 设计、构建和管理一个专用的 Hadoop集群或在云上运行Hadoop
- 使用高级查询语言Pig来处理大规模数据
- 利用Hadoop数据库HBase来保存和处理结构化/半结构化数据
- 学会使用ZooKeeper来构建分布式系统

如果您拥有海量数据，无论是GB级还是PB级，Hadoop都将是您的完美解决方案。

“恭喜您有此良机向大师学习Hadoop，在享用技术本身的同时，您还能领略到大师的睿智及其令人如沐春风的写作风格。”

——Hadoop创始人
Doug Cutting

Tom White从2007年2月以来，一直担任Apache Hadoop项目负责人。他是Apache软件基金会的成员之一，同时也是Cloudera的一名工程师。Tome为O'Reilly.com、Java.net及IBM的developerWorks撰写过大量文章，并经常在很多行业大会上举行Hadoop主题演讲。

Cloudera为Hadoop提供商业支持并志愿贡献社区，不收取任何费用。不管是打算在云中运行Hadoop，还是在自己的服务器上运行Hadoop，Cloudera都能使其轻松实现。详情访问*http://www.cloudear.com/hadoop*。

www.oreilly.com

O'Reilly Media, Inc.授权清华大学出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-302-22424-2

定价：79.00元

Hadoop权威指南
(中文版)

Tom White 著

曾大聃 周傲英 译

周 敏 审校

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • 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

本书从 Hadoop 的缘起开始，由浅入深，结合理论和实践，全方位地介绍 Hadoop 这一高性能处理海量数据集的理想工具。全书共 14 章，3 个附录，涉及的主题包括：Haddoop 简介；MapReduce 简介；Hadoop 分布式文件系统；Hadoop 的 I/O、MapReduce 应用程序开发；MapReduce 的工作机制；MapReduce 的类型和格式；MapReduce 的特性；如何安装 Hadoop 集群，如何管理 Hadoop；Pig 简介；Hbase 简介；ZooKeeper 简介，最后还提供了丰富的案例分析。

本书是 Hadoop 权威参考，程序员可从中探索如何分析海量数据集，管理员可以从中了解如何安装与运行 Hadoop 集群。

Copyright © 2009 Tom White. All rights reserved.
Authorized Simplified Chinese translation edition, by O'Reilly Media, Inc., is published by Tsinghua University Press, 2010. Authorized translation of the original English edition, 2009 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 2009 年出版。
本书中文简体版由 O'Reilly Media, Inc. 授权清华大学出版社出版 2010 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：01-2009-5152
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(中文版)/(美)怀特(White, T.)著；曾大聃，周傲英译；周敏审校. —北京：清华大学出版社，2010.5
书名原文：Hadoop: The Definitive Guide
ISBN 978-7-302-22424-2
Ⅰ.H… Ⅱ.①怀… ②曾… ③周… ④周… Ⅲ.数据处理—应用软件—指南 Ⅳ.TP274-62
中国版本图书馆 CIP 数据核字(2010)第 064977 号

责任编辑：文开琪
封面设计：Ellie Volckhausen 张 健
版式设计：北京东方人华科技有限公司
责任印制：杨 艳
出版发行：清华大学出版社 **地 址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn **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 刷 者：清华大学印刷厂
装 订 者：三河市新茂装订有限公司
经 销：全国新华书店
开 本：178×233 **印 张**：33 **字 数**：769 千字
版 次：2010 年 5 月第 1 版 **印 次**：2010 年 5 月第1 次印刷
印 数：1～4000
定 价：79.00 元

产品编号：034686-01

本页已使用福昕阅读器进行编辑。
福昕软件(C)2005-2007，版权所有，
仅供试用。

推荐序

Hadoop 起源于 Nutch。当时，我们少数几个人正在打算构建一个开源的网络搜索引擎，但受困于如何管理仅运行于几台计算机的计算。在 Google 发布 GFS 和 MapRduce 的论文后，我们解决这个问题的思路变得清晰起来。他们设计的系统准确解决了我们在 Nutch 中碰到的问题。因此，我们两个中途开始尝试重建这些系统，将其作为 Nutch 的一部分。

我们成功地让 Nutch 运行在 20 台计算机上，但很快我们意识到，要想处理 Web 的巨大规模，我们需要在上千台计算机上运行它，显然，这远远不是两个 half-time 开发人员能够对付的。。

在那段时间，Yahoo！对其产生兴趣，并迅速组建了一个团队，我也加入其中。我们将 Nutch 的分布式运算这部分独立出来，命名为 Hadoop。在 Yahoo！的帮助下，Hadoop 很快成为确实可扩展应用于 Web 的技术。

2006 年，Tom White 开始效力于 Hadoop。由于之前已通过他写的一篇有关 Nutch 的精彩论文认识他，所以，我知道他能够用清晰的语言表达复杂的理念。我也很快意识到，他还能编写体现其思想的软件。

首先，Tom 对 Hadoop 的贡献体现了他对用户和项目的关注。与大多开源工作者不同，Tom 并不怎么关心如何将系统调整到更符合他的需求，而是努力使它变得更方便所有人使用。

一开始，Tom 着重于使 Hadoop 在亚马逊的 EC2 和 S3 上顺畅运行。而后他转向各种各样的问题。包括改进 MapReduce 的 API，改善网站，设计对象序列化框架。在任何时候，Tom 都很清晰地表达他的构想。很快，Tom 赢得 Hadoop 项目负责人的身份并很快成为 Hadoop 项目管理委员会的成员。

本页已使用福昕阅读器进行编辑。
福昕软件(C)2005-2007，版权所有，
仅供试用。

现在，Tom 是 Hadoop 开发社区一名令人尊敬的高级成员。尽管他是这个项目的技术多面手，但他最大的专长还是让 Hadoop 更易于使用和理解。

因此，当我得知 Tom 有意写一本有关 Hadoop 的书时，我非常开心。还有谁更具备这个资格呢？现在您有此良机向大师学习 Hadoop，在享用技术本身的同时，品味他的睿智和清晰的文风。

Doug Cutting

于院棚，加州

笑看云卷云舒

(编者按：在国内，提起 Hadoop，圈内人物第一个想到就是邵铮。这位 Hadoop 社区积极的推动者和志愿者，有 Hadoop 的地方，就活跃着邵铮和他的朋友们和随随者们。在 Hadoop 社区，这批有理想、有抱负的互联网新生力量，为着自己的理想，为着开源社区的奉献、共享精神而无私贡献着自己的才华和精力。他们为技术而痴迷，为新技术的发扬光大群策群力，推动着 Web 3.0 的变革。在"Hadoop 中国 2009 云计算大会"上，热烈的现场，精彩的演讲，全神贯注的与会者，都让我们深切感受到他们的传道是卓有成效的，他们的号召力是有积极响应的。)

随着 Internet 数据的爆炸性增长，传统的技术架构已经越来越不适应当前海量数据处理的要求。Hadoop 就是在这样的环境下出现的。Hadoop 的出现代表着互联网软件发展的两个方向。

第一，海量数据处理的广泛应用。Hadoop 的设计思想来源于 Google 的 Google File System 和 Map-Reduce 这两篇学术文章。它最早的应用是为搜索引擎建立索引。目前先进的搜索引擎所抓取的网页数在百亿到千亿的数量级。单个搜索引擎索引的数据量可达 100TB 至 1PB。

然而，如果只有搜索引擎建立索引这样一个应用， Hadoop 是无法达到今天这样的普及程度的。目前，海量数据处理技术已经被广泛的应用于用户行为分析、广告效果分析、产品设计分析、商业智能分析和报表统计等各个环节，为 Internet 公司的发展方向和决策制定提供最重要的信息。在 Facebook 公司，每天处理的新数据量就高达 20 TB。Facebook 用户数量的飞速增长与这些数据的深层次分析是分不开的。

第二，开源软件的蓬勃发展。互联网公司最常用的软件套装 LAMP (Linux, Apache,

MySQL 和 PHP) 都是开源的。开源软件在互联网服务的底层架构上发挥着举足轻重的作用。而 Hadoop 无疑是互联网公司在开源社区中的新宠儿。

Hadoop 之所以能在短短 3 年多时间内就迅速崛起，获得如此广泛的应用，与其开源性质是息息相关的。互联网时代发展的最大特点是快。在如此短的时间之内研究和开发出的技术，如果没有一个很好的管理模式能够使其健康的演化发展，是很容易在同样短的时间之内被取代的。

由于开源技术的开放性，任何人都能很轻松的参与到 Hadoop 的开发中。这些 Hadoop 开发的志愿者不但帮助了 Hadoop 技术的进一步发展，同时也为 Hadoop 做了很好的宣传，吸引了更多的用户。而更多的用户也带来了更多的志愿者，从而形成一个良性循环和规模效应。开源社区的规模比起任何一家公司都要大很多。

互联网时代，知识更新的速度越来越快，软件和互联网从业人员的学习成本是非常大的。相比专利技术和公司内部的专用技术而言，开源的通用的技术能在将来更长的时间内发挥作用。在国外，Hadoop 已经成为了海量数据处理的事实标准。无数著名的互联网公司，如 Facebook, Yahoo, Amazon 等都在生产系统中使用 Hadoop。很多国内的互联网公司也在奋起直追。因此，学习和掌握 Hadoop 技术在今天，无异于数年前学习和掌握 LAMP 技术一样的重要。

我是 Hadoop 开源项目的一个志愿者。从 2008 年 3 月我加入 Facebook 公司以来，我在 Hadoop 的子项目 Hive 的开发中投入了大量的时间和精力，同时也积极参与 Hadoop 核心（HDFS 和 MAPREDUCE）的改进和完善。本书英文版作者 Tom White 是我的好友。目前我们都是 Hadoop 项目管理委员会的成员。

一个好的开源软件离不开一个活跃健康的开源社区。从 2008 年 11 月开始，在中科院计算所的查礼博士、何永强博士，雅虎北京全球研发中心的郑皓博士、韩轶平，百度公司的王守彦等人的指导以及大量志愿者的帮助下，中科院计算所在北京成功的举办了多次 Hadoop 开发者的交流活动。这些活动推动了 Hadoop 在中国的普及，促进了 Hadoop 开源社区在中国的发展。

同样，一个好的开源软件也离不开一本好书。在此，我想感谢原书作者 Tom White 和译者、编辑们的辛勤劳动，以及上述支持和参与到 Hadoop 开发者活动中的朋友们的努力和付出。希望这本书能给千千万万渴望了解 Hadoop 技术的朋友提供一些帮助，也希望更多的读者能加入我们的队伍，共同推动 Hadoop 技术和开源社区的发展。

邵铮

于帕罗拉多家中，美国加州

译者序

随着数据规模的急剧增加、应用类型的巨大丰富，企业和个人用户信息使用模式的变化已经远远超过了原有系统平台所提供的局限。越来越多的应用和平台，不论对企业级还是个人级用户都不堪重负，应接不暇。系统的大集中，企业应用平台的不断累加；个人用户桌面应用更是五花八门呈爆炸式增长；sars、5.12 和 H1N1，传统数据分析处理领域在面临新的重大问题时，需要更多领域数据的融合和协作。在这种巨大的潮流和趋势力的推动下，风起“云”涌。云计算被推上了计算机科学和应用的舞台，带来信息使用模式的巨大变革。

我也曾思考，为什么将其称为云，这是一个为用户屏蔽了底层异构的软硬件资源，为其提供服务和资源的平台，各种不同类型的资源经过层层的虚拟化技术之后，针对虚拟资源的分配、共享和使用。是分布式计算技术和信息处理技术，以及网络技术、Web 技术等，在架构层、应用层全面融合之后产生的必然结果。SaaS，PaaS，CaaS，IasS，DaaS，云上的应用种类繁多，仍在发展中，将其定义成云或者什么其他都是不重要的，重要的是我们在之上，将数据、软件和平台等等的复杂构建、安装和维护工作转嫁给云提供商，通过大型的用户池共享资源来降低基础设施成本，不同层面的用户将在云上轻松得到自己想要的，做到 thin client 和 on-demand service。

Google 的 App Engine 允许用户通过使用其提供的 API，在 Google 云上构建自己的应用；Amazon 的云平台 EC3，S3 等为用户提供了种类丰富的云计算服务；Google 和 IBM 联合宣布推广“云计算”的计划，包括卡内基梅隆大学、斯坦福、伯克利、华盛顿大学、MIT、清华大学都加入了这项计划。通过这项计划，高校的研究者能够更方便地利用 Google 和 IBM 的云计算资源，搭建出各种创新性的应用。未来的云计算平台中，用户个人维护的操作系统将被浏览器所取代。这使信息工业界面临一次重新洗牌的机会，我国的软件技术企业应抓住这个机会为云计算的

本土化市场占领先机，同时学术界也将面临新的挑战和机遇。Microsoft 感到了来自 Google 的威胁，微软 MSRA 的 WebStudio 就已经能够提供在 Web 规模上快速搭建应用。国内外各种云计算相关会议和论坛更是不计其数。在这其中，他们都无一例外的将 Hadoop 作为云计算中的重要技术之一。

Apache Hadoop 作为一个开源项目，克隆了 Google 运行系统的主要框架，包括文件系统 HDFS、计算架构 MapReduce 及对于结构化数据处理的 HBase 等。基于此的其他开源项目，比如 Pig，Zookeeper，HIVE 等，为 Hadoop 的使用和系统架构也带来了更多的福音。目前正在进行 Hadoop Avro 等，各种工作也将推进 Hadoop 在云计算的实现中扮演越来越重要的角色。利用 Hadoop，对底层，可以实现对集群的控制和管理；对上层，可以更加便捷的构建企业级的应用。Hadoop 实现海量数据的管理和分布式数据处理，使传统的分布式计算中的数据分割和错误管理等复杂问题屏蔽在于系统本身，从而取得更好的系统伸缩性。使用者可以更多地关注于数据处理本身和对应用问题本身的分析。

本书的作者 Tom White 是 Hadoop 开发团体的重要高级成员，是 Hadoop 项目中许多技术方向的专家，参与了多项 Hadoop 主要技术方向的设计、改进和实施。他对 Hadoop 的卓越贡献，使他成为项目管理委员会成员，并且他是推广 Hadoop 开发和使用的专家。

本书内容组织得很好，思路清晰，紧密结合实际问题，阅读并在实际中实践本书内容将是一个愉快的、充满收获的过程。对于 Hadoop 的开发使用者深刻全面理解其内部原理、使用以及二次开发，将很有帮助。

Cloudera 作为一个商业公司致力于推广和培训 Hadoop 的使用，这是它出版的第一本 Hadoop 书籍，颇具代表性和全面性。国外使用这门技术已经比较成熟了，并且发挥了较高经济效益。

为了推动该技术在国内的普及，让更多读者更早受益，本书经过我们精心雕琢而成。在此期间，文开琪编辑为该书的出版付出了非常多的努力和辛勤工作，令人敬佩。由于时间有限、工作繁重，译著不足纰漏难免，欢迎广大读者批评、指正和交流。

尾声：在云得到了蓬勃的发展和广泛的机会的同时，我们也该警醒。这时候更加欢迎负面和反对的声音。关注于云安全及相关规范的制订，相关技术的合理使用，以及配套应急措施的实施。只有这样才能使云以更低的代价受益于更多的人，对于云在未来的扩展和推广，更具有普遍的意义。

曾大聃

二零一零年五月

前言

马丁·加德纳(数学家和科学作家)，曾经在一次采访中说道：

“没有微积分，我的生命就失去了意义。这是我成功的秘诀。我花了如此长的时间了解我在写什么，所以我知道如何写作才能让大多数读者明白我的意思。”

在许多方面，这就是我对 Hadoop 的感觉。它的内部工作机制是复杂的、相互依赖的，因为它运行在分布式系统的理论、实用技术和技术常识这些复杂的基础之上。对于门外汉来说，Hadoop 就像是异形一样难以理解。

但事实上并不是这样的。剥离其核心，Hadoop 提供给组件分布式系统的工具——如数据存储、数据分析和协调——是十分简单的。如果有一个共同的主题，那么它将与提高抽象水平相关的——为程序员创建用于处理这些事情的基础架构，这些程序员中，或者正好有大量数据需要存储，或者有大量数据需要分析，或者有大量机器需要协调，或者没有时间、技能或兴趣成为分布式系统专家。

借由这样一个简单的、普遍适用的功能组合，在开始使用这个理当被广泛普及的 Hadoop 的时候，我的想法逐渐清晰起来。然而，在当时(2006 年初)，设置、配置和编写程序来使用 Hadoop 称得上是一门艺术。幸运的是，此后有了明显的进步，因为有更多的文件，更多的例子，一旦有疑问，还有那么多邮件地址可以发过去帮助你解惑。但对大多数新手来说，最大的障碍是理解这项技术能做什么，它的长处何在，如何使用它。这就是我写这本书的原因。

Apache Hadoop 社区已经走过了漫长的道路。在三年的过程中，Hadoop 项目已经拓展并分成许多子项目。在这个时候，软件已在性能、可靠性、可扩展性和可管理性上有了很大的飞跃。然而为了获得更为广泛地应用，我相信我们需要让 Hadoop

变得更容易使用。这将涉及三方面的工作：编写更多的工具；与更多的系统集成；编写新的改进后的 API。我期待成为完成这项工作的一员，并且我希望这本书也能鼓励和帮助其他人完成这些事情。

写作风格的说明

在文中特定 Java 类的讨论中，我往往会省略其包名以减少混乱。如果需要知道一个类是放在哪个包里的，可以轻松地从 Hadoop 的 Java API 文档中查找到有关的子项目，这些条目与 Apache Hadoop 的主页 *http://hadoop.apache.org/*链接。或者，如果使用的是 IDE，它可以帮助使用其自动完成机制。

同样，虽然它偏离一贯的教学规则，但是从相同的使用星号通配符的包中导入不同的类还是能节省空间的(例如：import org.apache.hadoop.io .*)。

本书的示例程序可以从本书相关网站下载，网址为 *http://www.hadoopbook.com/*。还可以在此找到获得本书所用数据库的指令以及更多运行本书程序的注解、更新链接、其他资源以及我的 blog 。

本书内容

本书其余部分内容如下。第 2 章介绍 MapReduce。第 3 章着眼于 Hadoop 的文件系统，特别深入地讲解 HDFS。第 4 章涵盖 Hadoop 基础的 I/O 输入和输出，主题包括：数据的完整性、压缩、序列化和基于文件的数据结构。

接下来的 4 章更深入地涉及 MapReduce。第 5 章讲述全程开发一个 MapReduce 应用程序的实际步骤。第 6 章着眼于从用户的观点来看 MapReduce 如何在 Hadoop 上实现。第 7 章涉及 MapReduce 编程模型及 MapReduce 可以处理的各种数据格式。第 8 章的主题为如何改进 MapReduce，包括排序和联接(JOIN)数据。

第 9 章和第 10 章是写给 Hadoop 管理员看的，阐述如何建立和维持在 Hadoop 集群上运行 HDFS 和 MapReduce。

第 11 章、第 12 章和第 13 章分别提供了 Pig，HBase 和 ZooKeeper 应用示例。

最后，第 14 章提供 Apahe Hadoop 社区成员贡献的综合案例研究。

本书所用约定

本书采用如下印刷约定。

斜体

表示新名词，URL，电子邮件地址，文件名，文件扩展名，路径名，目录和 Unix 实用程序。

`等宽字体`

表示命令、选项、开关、变量、属性、键值、函数、类型、类、命名空间、方法、模块、参数、参数、值、对象、事件、事件句柄、XML 标签、HTML 标签、文件内容或者命令输出。

`等宽粗体`

显示需要用户逐字输入的命令或者其他文字。也用于代码中的强调。

`等宽斜体`

显示应该被用户输入值代替的文字。

[...]

表示引用参考文献。

注意：该图标表示一个技巧，建议或一般注解。

警告：该图标表示警告或注意事项。

示例代码的使用

本书的目的是帮助你完成任务。一般来说，可以使用在本书的程序和文档中的代码。不需要联络我们即可获得许可，除非是要复制代码的重要部分。例如，写一个使用本书几大块代码的程序并不需要经过我们许可。销售或分发含有来自 O'Reilly 书籍中的例子的光盘需要许可。回答问题援引本书和引用示例代码也不需要许可。把本书中的大量示例代码纳入到你的产品文档需要经过我们许可。

我们对提供版权说明的说法表示赞赏，但不强求。版权说明通常包括书名、作者、

出版商和 ISBN。例如："Hadoop：The Definltive Guide，Tom 著。版权所有 2009 Tom White，978-0-596-52197-4。"

如果觉得自己使用示例代码使用上或以上的要求有失公允，请随时联系我们，电子邮件地址为 permissions@oreilly.com。

Safari 图书在线

看到 Safari 图书在线标识出现在你最喜欢的技术书籍封面上时，便意味着此书(英文版)是可以通过 O'Reilly 在线书店购买。

Safari 提供了一个比电子书更好的解决方案。它提供一个虚拟的图书馆，让你轻松地搜索数千本顶尖的科技图书，剪切和粘贴代码示例，下载章节，并按自己的需要最准确、最及时的信息时，能够寻找到快速解答。免费试用请访问 *http://my.safaribooksonline.com*。

联系我们

对于本书，如果有任何意见或疑问，请按照以下地址联系本书出版商：

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。你可以访问：

http://www.oreilly.com/catalog/9780596521974(英文版)
http://www.oreilly.com.cn/book.php?bn=978-7-302-22424-2(中文版)

对本书做出评论或者询问技术问题，请发送 E-mail 至：

bookquestions@oreilly.com

希望获得关于本书、会议、资源中心和 O'Reilly 网络的更多信息，请访问：

http://www.oreilly.com

http://www.oreilly.com.cn

鸣谢

在写这本书的时候，我间接或直接地请教了很多人。我要感谢 Hadoop 社区，我在此学到了很多。我特别要感谢撰写 HBase 相关内容的 Michael Stack 和 Jonathan Gray。同时要感谢 Adrian Woddhead，Marc de Palol，Joydeep Sen Sarma，Ashish Thusoo，Andrzei Bialecki，Stu Hood，Chris K Wensel，和 Owen O'Malley 为第 14 章案例研究所做出的贡献。我非常感谢 Mattress Massie 和 ToddlerLipcon 撰写了附录 B。

我还要感谢那些为修改草稿提出许多有益建议的人，他们是：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 让审阅过程变得非常顺利。Philip(“flip”)Kromer 曾帮助我收集和处理本书中 NCDC 气象数据集相关例子。特别感谢 Owen O'Malley 和 Arun C Murthy 为我解释 MapReduce shuffle 的工作原理。对于仍然存在的任何错误，如蒙指正，我将感激不尽。

特别感谢 Doug Cutting 对我的鼓励、支持和友谊，以及对本书所做的贡献。

同时感谢在我写书过程中曾与我交流或发邮件进行讨论的人。

在写作本书的过程中，我加入了 Cloudera，在这里我要感谢我的同事对我的大力支持，让我能够有充足的时间迅速完成本书。

我还要感谢编辑 Mike Loukides 及其 O'Reilly 的同事对本书的帮助。Mike 一直都在回答我的问题，阅读我的初稿，并帮助我掌握进度。

最后，写作本书工作量很大，没有家庭持之以恒的支持，我不可能完成写作。我的妻子 Eliane，不仅操持家务，还帮助我审查、编辑和校对本书的案例。我的女儿，Emilia 和 Lottie，一直都十分体谅我，我期待着能够花更多时间陪伴她们。

目　　录

第 1 章　初识 Hadoop 1

1.1　数据！数据 1

1.2　数据的存储和分析 3

1.3　相较于其他系统 4

1.3.1　关系型数据库管理系统 5

1.3.2　网格计算 6

1.3.3　志愿计算 8

1.4　Hadoop 发展简史 9

1.5　Apache Hadoop 项目 12

第 2 章　MapReduce 简介 15

2.1　一个气象数据集 15

2.2　使用 Unix Tools 来分析数据 17

2.3　使用 Hadoop 进行数据分析 19

2.3.1　map 和 reduce 19

2.3.2　Java MapReduce 20

2.4　分布化 30

2.4.1　数据流 30

2.4.2　具体定义一个 combiner 34

2.4.3　运行分布式 MapReduce 作业 35

2.5　Hadoop 流 35

2.5.1　Ruby 语言 36

2.5.2　Python 38

2.6　Hadoop 管道 40

第 3 章 Hadoop 分布式文件系统 44
3.1 HDFS 的设计 44
3.2 HDFS 的概念 45
3.2.1 块 45
3.2.2 名称节点与数据节点 47
3.3 命令行接口 48
3.4 Hadoop 文件系统 50
3.5 Java 接口 54
3.5.1 从 Hadoop URL 中读取数据 54
3.5.2 使用 FileSystem API 读取数据 56
3.5.3 写入数据 59
3.5.4 目录 62
3.5.5 查询文件系统 62
3.5.6 删除数据 67
3.6 数据流 68
3.6.1 文件读取剖析 68
3.6.2 文件写入剖析 71
3.6.3 一致模型 73
3.7 通过 distcp 进行并行复制 75
3.8 Hadoop 归档文件 77
3.8.1 使用 Hadoop Archives 77
3.8.2 不足 79
第 4 章 Hadoop 的 I/O 80
4.1 数据完整性 80
4.1.1 HDFS 的数据完整性 81
4.1.2 本地文件系统 82
4.1.3 ChecksumFileSystem 82
4.2 压缩 83
4.2.1 编码/解码器 84
4.2.2 压缩和输入分割 89

4.2.3 在 MapReduce 中使用压缩 90
4.3 序列化 92
4.3.1 Writable 接口 93
4.3.2 Writeable 类 96
4.3.3 实现自定义的 Writable 104
4.3.4 序列化框架 109
4.4 基于文件的数据结构 111
4.4.1 SequenceFile 类 112
4.4.2 MapFile 120
第 5 章 MapReduce 应用开发 125
5.1 API 的配置 126
5.1.1 合并资源 127
5.1.2 各种扩展形式 128
5.2 配置开发环境 128
5.2.1 配置的管理 129
5.2.2 GenericOptionsParser，Tool 和 ToolRunner 131
5.3 编写单元测试 134
5.3.1 Mapper 135
5.3.2 reducer 137
5.4 本地运行测试数据 138
5.4.1 在本地作业运行器上运行作业 139
5.4.2 测试驱动程序 142
5.5 在集群上运行 144
5.5.1 打包 144
5.5.2 启动作业 144
5.5.3 MapReduce 网络用户界面 146
5.5.4 获取结果 150
5.5.5 调试作业 151
5.5.6 使用远程调试器 157
5.6 作业调优 159
5.7 MapReduce 的工作流 162

5.7.1 将问题分解成 MapReduce 作业 163
5.7.2 运行独立的作业 164

第 6 章 MapReduce 的工作原理 166

6.1 运行 MapReduce 作业 166
6.1.1 提交作业 166
6.1.2 作业的初始化 168
6.1.3 任务的分配 168
6.1.4 任务的执行 169
6.1.5 进度和状态的更新 170
6.1.6 作业的完成 171
6.2 失败 172
6.2.1 任务失败 172
6.2.2 tasktracker 失败 174
6.2.3 jobtracker 失败 174
6.3 作业的调度 174
6.4 shuffle 和排序 175
6.4.1 map 端 176
6.4.2 reduce 端 177
6.4.3 配置的调整 178
6.5 任务的执行 181
6.5.1 推测式执行 181
6.5.2 任务 JVM 重用 183
6.5.3 跳过坏记录 184
6.5.4 任务执行环境 185

第 7 章 MapReduce 的类型与格式 188

7.1 MapReduce 类型 188
7.2 输入格式 198
7.2.1 输入分片与记录 198
7.2.2 文本输入 210
7.2.3 二进制输入 214
7.2.4 多种输入 215

7.2.5 数据库格式的输入/输出 216
7.3 输出格式 217
7.3.1 文本输出 217
7.3.2 二进制输出 218
7.3.3 多个输出 218
7.3.4 延迟输出 226
7.3.5 数据库输出 226
第 8 章 MapReduce 特性 227
8.1 计数器 227
8.1.1 内置计数器 227
8.1.2 用户自定义 Java 计数器 230
8.1.3 用户自定义流计数器 235
8.2 排序 235
8.2.1 准备 235
8.2.2 部分排序 237
8.2.3 全局排序 242
8.2.4 二次排序 246
8.3 联接 252
8.3.1 map 端联接 253
8.3.2 reduce 端联接 254
8.4 次要数据的分布 258
8.4.1 使用作业配置 258
8.4.2 分布式缓存 258
8.5 MapReduce 的类库 263
第 9 章 Hadoop 集群的安装 264
9.1 集群说明 264
9.2 集群的建立和安装 268
9.2.1 安装 Java 268
9.2.2 创建 Hadoop 用户 269
9.2.3 安装 Hadoop 269
9.2.4 测试安装 270

9.3 SSH 配置 270
9.4 Hadoop 配置 271
9.4.1 配置管理 271
9.4.2 环境设置 274
9.4.3 重要的 Hadoop 后台程序属性 278
9.4.4 Hadoop 后台程序地址和端口 283
9.4.5 其他 Hadoop 属性 284
9.5 安装之后 286
9.6 Hadoop 集群基准测试 286
9.6.1 Hadoop 基准测试 287
9.6.2 用户作业 290
9.7 云计算中的 Hadoop 290

第 10 章 Hadoop 的管理 293

10.1 HDFS 293
10.1.1 持久化的数据结构 293
10.1.2 安全模式 298
10.1.3 审计日志 300
10.1.4 工具 300
10.2 监控 306
10.2.1 日志 306
10.2.2 度量 307
10.2.3 Java 管理扩展 310
10.3 维护 313
10.3.1 例行管理程序 313
10.3.2 委托节点和撤消节点 314
10.3.3 升级 317

第 11 章 Pig 简介 321

11.1 安装和运行 Pig 322
11.1.1 执行类型 322
11.1.2 运行 Pig 程序 324
11.1.3 Grunt 324

11.1.4 Pig Latin 编辑器 325
11.2 实例 325
11.3 与数据库比较 329
11.4 Pig Latin......330
11.4.1 结构 330
11.4.2 语句 331
11.4.3 表达式 334
11.4.4 类型 335
11.4.5 模式 337
11.4.6 函数 341
11.5 用户定义函数 343
11.5.1 过滤 UDF......343
11.5.2 求值 UDF......347
11.5.3 加载 UDF......348
11.6 数据处理操作符 353
11.6.1 加载和存储数据 353
11.6.2 过滤数据 353
11.6.3 数据的分组和联接 356
11.6.4 数据的排序 361
11.6.5 数据的合并和分割 362
11.7 Pig 实践提示与技巧 363
11.7.1 并行 363
11.7.2 参数替换 364
第 12 章 Hbase 简介 366
12.1 HBase 基础 366
12.2 概念 367
12.2.1 数据模型速览 367
12.2.2 实现 368
12.3 安装 371
12.4 客户端 374
12.4.1 Java......374

12.4.2 REST 和 thrift 376
12.5 示例 377
12.5.1 架构 378
12.5.2 加载数据 379
12.5.3 Web 查询 382
12.6 HBase 与 RDBMS 的比较 385
12.6.1 成功的服务 386
12.6.2 HBase 387
12.6.3 用例：streamy.com 的 Hbase 388
12.7 实践 390
12.7.1 版本 390
12.7.2 Hbase 和 HDFS 的爱与恨 390
12.7.3 用户界面 392
12.7.4 度量 392
12.7.5 架构设计 392
第 13 章 ZooKeeper 简介 394
13.1 ZooKeeper 的安装和运行 395
13.2 范例 396
13.2.1 ZooKeeper 中的组成员制 397
13.2.2 创建组 397
13.2.3 加入组 400
13.2.4 列出组成员 401
13.2.5 删除一个组 404
13.3 ZooKeeper 服务 405
13.3.1 数据模型 405
13.3.2 操作 407
13.3.3 执行 411
13.3.4 一致性 412
13.3.5 会话 414
13.3.6 状态 416
13.4 使用 ZooKeeper 建立应用程序 417

13.4.1 配置服务 417
13.4.2 可恢复的 ZooKeeper 应用 421
13.4.3 锁服务 425
13.4.4 更多分布式数据结构和协议 427
13.5 工业界中的 ZooKeeper 428
13.5.1 恢复力及性能 428
13.5.2 配置 429

第 14 章 案例研究 431

14.1 Hadoop 在 Last.fm 的应用 431
14.1.1 Last.fm：社会音乐革命 431
14.1.2 使用 Hadoop 生成排行榜 432
14.1.3 单曲统计程序 433
14.1.4 小结 440
14.2 Hadoop 和 Hive 在 Facebook 的应用 441
14.2.1 简介 441
14.2.2 Hadoop 在 Facebook 的应用 441
14.2.3 虚拟案例研究 444
14.2.4 Hive 446
14.2.5 存在的问题及未来的工作 450
14.3 Hadoop 在 Nutch 搜索引擎 451
14.3.1 背景 451
14.3.2 数据结构 453
14.3.3 Nutch 中 Hadoop 数据处理精选实例 455
14.3.4 小结 465
14.4 Hadoop 用于 Rackspace 的日志处理 466
14.4.1 需求/存在的问题 466
14.4.2 简史 467
14.4.3 选择 Hadoop 467
14.4.4 收集和存储 467
14.4.5 日志的 MapReduce 468
14.5 Cascading 项目 474

14.5.1 字段、元组和管道 475
14.5.2 操作 477
14.5.3 Tap、Sheme 和 Flow 479
14.5.4 Cascading 实践 480
14.5.5 灵活性 483
14.5.6 Hadoop 和 Cascading 在 ShareThis 的应用 484
14.5.7 小结 487
14.6 Apache Hadoop 的 1 TB 排序 488
附录 A Apache Hadoop 的安装 491
附录 B Cloudera 的 Hadoop 分发包 497
附录 C 预备 NCDC 气象资料 502

第 1 章

初识 Hadoop

古时候，人们用牛来拉重物，当一头牛拉不动一根圆木的时候，他们不曾想过培育个头更大的牛。同样，我们也不需要尝试更大的计算机，而是应该开发更多的计算系统。

——格蕾斯·霍珀

1.1 数据！数据

我们生活在数据时代！很难估计全球存储的电子数据总量是多少，但是据 IDC 估计 2006 年“数字全球”项目(digital universe)的数据总量为 0.18 ZB，并且预测到 2011 年这个数字将达到 1.8 ZB，为 2006 年的 10 倍[①]。1 ZB 相当于 10 的 21 次方字节的数据，或者相当于 1000 EB，1 000 000 PB，或者大家更熟悉的 10 亿 TB 的数据！这相当于世界上每个人一个磁盘驱动器的数量级。

这一数据洪流有许多来源。考虑下文：[②]

① 来源：“The Diverse and Exploding Digital Universe,” 2008 年 3 月 Gantz 等人(*http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf*)。

② 来源：*http://www.intelligententerprise.com/showArticle.jhtml?articleID=207800705*
http://mashable.com/2008/10/15/facebook-10-billion-photos/
http://blog.familytreemagazine.com/insider/Inside+Ancestrycoms+TopSecret+Data+Center.aspx
http://www.archive.org/about/faqs.php, http://www.interactions.org/cms/?pid=1027032。

- 纽约证券交易所每天产生 1 TB 的交易数据。
- 著名社交网站 Facebook 的主机存储着约 100 亿张照片，占据 PB 级存储空间。
- Ancestry.com，一个家谱网站，存储着 2.5 PB 数据。
- 互联网档案馆(The Internet Archive)存储着约 2 PB 数据，并以每月至少 20 TB 的速度增长。
- 瑞士日内瓦附近的大型强子对撞机每年产生约 15 PB 的数据。

此外还有大量数据。但是你可能会想它对自己有何影响。大部分数据被锁定在最大的网页内容里面(如搜索引擎)或者是金融和科学机构，对不对？是不是所谓的“大数据”的出现会影响到较小的组织或个人？

我认为是这样的。以照片为例，我妻子的祖父是一个狂热的摄影爱好者，并且他成人之后，几乎一直都在拍照片。他的所有照片(中等格式、幻灯片和 35 mm 胶片)，在扫描成高解析度照片时，占了大约 10 GB 的空间。相比之下，我家去年一年用数码相机拍摄的照片就占用了 5 GB 的空间。我家产生照片数据的速度是我妻子祖父的 35 倍！并且，随着拍摄更多的照片变得越来越容易，这个速度还在增加中。

更常见的情况是，个人数据的产生量正在快速地增长。微软研究院的 MyLifeBits 项目(*http://research.microsoft.com/en-us/projects/mylifebits/default.aspx*)显示，在不久的将来，个人信息档案将可能成为普遍现象。MyLifeBits 是这样的一个实验：一个人与外界的联系(电话、邮件和文件)被抓取和存储供以后访问。收集的数据包括每分钟拍摄的照片等，导致整个数据量达到每月 1 GB 的大小。当存储成本下降到使其可以存储连续的音频和视频时，服务于未来 MyLifeBits 项目的数据量将是现在的许多倍。

个人数据的增长的确是大势所趋，但更重要的是，计算机所产生的数据可能比人所产生的数据更大。机器日志、RFID 读取器、传感器网络、车载 GPS 和零售交易数据等，这些都会促使“数据之山越来越高”。

公开发布的数据量也在逐年增加。作为组织或企业，再也不能只管理自己的数据，未来的成功在很大程度上取决于它是否能从其他组织的数据中提取出价值。

这方面的先锋(如亚马逊网络服务器、Infochimps.org 或者 andtheinfo.org)的公共数据集，它们的存在就在于促进“信息共享”，任何人都可以共享并自由(或以 AWS 平台的形式，或以适度的价格)下载和分析这些数据。不同来源的信息混合处理后会带来意外的效果和至今难以想像的应用。

以 Astrometry.net 项目为例，这是一个研究 Flickr 网站上天体爱好者群中新照片的项目。它分析每一张上传的照片，并确定它是天空的哪一部分，或者是否是有趣的

天体，如恒星或者星系。虽然这只是一个带实验性质的新服务，但是它显示了数据(这里特指摄影照片)的可用性并且被用来进行某些活动(图像分析)，而这些活动很多时候并不是数据创建者预先能够想像到的。

有句话是这么说的："算法再好，通常也难敌更多的数据。"意思是说对于某些问题(譬如基于既往偏好生成的电影和音乐推荐)，不论你的算法有多么猛，它们总是会在更多的数据面前无能为力(更不用说没有优化过的算法了)。[①]

现在，我们有一个好消息和一个坏消息。好消息是有海量数据！坏消息是我们正在为存储和分析这些数据而奋斗不息。

1.2 数据的存储和分析

问题很简单：多年来硬盘存储容量快速增加的同时，访问速度——数据从硬盘读取的速度——却未能与时俱进。1990 年，一个普通的硬盘驱动器可存储 1370 MB 的数据并拥有 4.4 MB/s 的传输速度[②]，所以，只需五分钟的时间就可以读取整个磁盘的数据。20 年过去了，1 TB 级别的磁盘驱动器是很正常的，但是数据传输的速度却在 100 MB/s 左右。所以它需要花两个半小时以上的时间读取整个驱动器的数据。

从一个驱动器上读取所有的数据需要很长的时间，写甚至更慢。一个很简单的减少读取时间的办法是同时从多个磁盘上读取数据。试想一下，我们拥有 100 个磁盘，每个存储百分之一的数据。如果它们并行运行，那么不到两分钟我们就可以读完所有的数据。

只使用一个磁盘的百分之一似乎很浪费。但是我们可以存储 100 个数据集，每个 1 TB，并让它们共享磁盘的访问。我们可以想像，此类系统的用户会很高兴看到共享访问可以缩短分析时间，并且，从统计角度来看，他们的分析工作会分散到不同的时间点，所以互相之间不会有太多干扰。

尽管如此，现在更可行的是从多个磁盘并行读写数据。

第一个需要解决的问题是硬件故障。一旦开始使用多个硬件设施，其中一个会出故障的概率是非常高的。避免数据丢失的常见做法是复制：通过系统保存数据的冗余

---

① 引自 Anand Rajaraman 对于 Netflix Challeng 的报告，网址如下：*http://anand.typepad.com/datawocky/2008/03/more-data-usual.html*

② 这些描述基于 Seagate ST-41600n。

副本，在故障发生时，可以使用数据的另一份副本。这就是冗余磁盘阵列的工作方式。Hadoop 的文件系统 HDFS(Hadoop Distributed Filesystem)也是一个例子，虽然它采取的是另一种稍有不同的方法，详见后文描述。

第二个问题是大部分分析任务需要通过某种方式把数据合并起来，即从一个磁盘读取的数据可能需要和另外 99 个磁盘中读取的数据合并起来才能使用。各种不同的分布式系统能够组合多个来源的数据，但是如何保证正确性是一个非常难的挑战。MapReduce 提供了一个编程模型，其抽象出上述磁盘读写的问题，将其转换为计算一个由成对键/值组成的数据集。这种模型的具体细节将在后面的章节讨论。但是目前讨论的重点是，这个计算由两部分组成：Map 和 Reduce。这两者的接口就是“整合”之地。就像 HDFS 一样，MapReduce 是内建可靠性这个功能的。

简而言之，Hadoop 提供了一个稳定的共享存储和分析系统。存储由 HDFS 实现，分析由 MapReduce 实现。纵然 Hadoop 还有其他功能，但这些功能是它的核心所在。

1.3 相较于其他系统

MapReduce 似乎采用的是一种蛮力方法。即，针对每个查询，每一个数据集——至少是很大一部分——都会被处理。但这正是它的能力。MapReduce 可以处理一批查询，并且它针对整个数据集处理即席查询并在合理时间内获得结果的能力也是具有突破性的。它改变了我们对数据的看法，并且解放了以前存储在磁带和磁盘上的数据。它赋予我们对数据进行创新的机会。那些以前需要很长时间才能获得答案的问题现在已经迎刃而解，但反过来，这又带来了新的问题和见解。

例如，Rackspace 的邮件部门 Mailtrust，用 Hadoop 处理邮件的日志。他们写的一个查询是找到其用户的地理分布。他们是这样说的：

“随着我们的壮大，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哪些 Rackspace 数据中心需要添加新的邮件服务器。”[1]

通过将数百 GB 的数据整合，借助于分析工具，Rackspace 的工程师得以了解这些数据，否则他们永远都不会了解，并且他们可以运用这些信息去改善他们为用户提供的服务。第 14 章将详细介绍 Rackspace 公司是如何运用 Hadoop 的。

---

① 来源：*http://blog.racklabs.com/?p=66*。

1.3.1 关系型数据库管理系统

为什么我们不能使用数据库加上更多磁盘来做大规模的批量分析？为什么我们需要 MapReduce?

这个问题的答案来自于磁盘驱动器的另一个发展趋势：寻址时间的提高速度远远慢于传输速率的提高速度。寻址就是将磁头移动到特定位置进行读写操作的工序。它的特点是磁盘操作有延迟，而传输速率对应于磁盘的带宽。

如果数据的访问模式受限于磁盘的寻址，势必会导致它花更长时间(相较于流)来读或写大部分数据。另一方面，在更新一小部分数据库记录的时候，传统的 B 树(关系型数据库中使用的一种数据结构，受限于执行查找的速度)效果很好。但在更新大部分数据库数据的时候，B 树的效率就没有 MapReduce 的效率高，因为它需要使用排序/合并来重建数据库。

在许多情况下，MapReduce 能够被视为一种 RDBMS(关系型数据库管理系统)的补充。(两个系统之间的差异见表 1-1)。MapReduce 很适合处理那些需要分析整个数据集的问题，以批处理的方式，尤其是 Ad Hoc(自主或即时)分析。RDBMS 适用于点查询和更新(其中，数据集已经被索引以提供低延迟的检索和短时间的少量数据更新。MapReduce 适合数据被一次写入和多次读取的应用，而关系型数据库更适合持续更新的数据集。

表 1-1：关系型数据库和 MapReduce 的比较

	传统关系型数据库	MapReduce
数据大小	GB	PB
访问	交互型和批处理	批处理
更新	多次读写	一次写入多次读取
结构	静态模式	动态模式
集成度	高	低
伸缩性	非线性	线性

MapReduce 和关系型数据库之间的另一个区别是它们操作的数据集中的结构化数据的数量。结构化数据是拥有准确定义的实体化数据，具有诸如 XML 文档或数据库表定义的格式，符合特定的预定义模式。这就是 RDBMS 包括的内容。另一方面，半结构化数据比较宽松，虽然可能有模式，但经常被忽略，所以它只能用作数据结构指南。例如，一张电子表格，其中的结构便是单元格组成的网格，尽管其本身可能保存任何形式的数据。非结构化数据没有什么特别的内部结构，例如纯文本

或图像数据。MapReduce 对于非结构化或半结构化数据非常有效，因为它被设计为在处理时间内解释数据。换句话说：MapReduce 输入的键和值并不是数据固有的属性，它们是由分析数据的人来选择的。

关系型数据往往是规范的，以保持其完整性和删除冗余。规范化为 MapReduce 带来问题，因为它使读取记录成为一个非本地操作，并且 MapReduce 的核心假设之一就是，它可以进行(高速)流的读写。

Web 服务器日志是记录集的一个很好的非规范化例子(例如，客户端主机名每次都以全名来指定，即使同一客户端可能会出现很多次)，这也是 MapReduce 非常适合用于分析各种日志文件的原因之一。

MapReduce 是一种线性的可伸缩的编程模型。程序员编写两个函数—— map 函数和 Reduce 函数——每一个都定义一个键/值对集映射到另一个。这些函数无视数据的大小或者它们正在使用的集群的特性，这样它们就可以原封不动地应用到小规模数据集或者大的数据集上。更重要的是，如果放入两倍的数据量，运行的时间会少于两倍。但是如果是两倍大小的集群，一个任务任然只是和原来的一样快。这不是一般的 SQL 查询的效果。

随着时间的推移，关系型数据库和 MapReduce 之间的差异很可能变得模糊。关系型数据库都开始吸收 MapReduce 的一些思路(如 ASTER DATA 的和 GreenPlum 的数据库)，另一方面，基于 MapReduce 的高级查询语言(如 Pig 和 Hive)使 MapReduce 的系统更接近传统的数据库编程人员。[①]

1.3.2 网格计算

高性能计算(High Performance Computing，HPC)和网格计算社区多年来一直在做大规模的数据处理，它们使用的是消息传递接口(Message Passing Interface，MPI)这样的 API。从广义上讲，高性能计算的方法是将作业分配给一个机器集群，这些机

① 2007 年 1 月，David J. Dewitt 和 Michael Stonebraker 因为发表"MapReduce：A major step backwards"而名噪一时，这篇文章的网址为 *http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html*。其中，他们对关系型数据库的雙脚替代品 MapReduce 加以批评。许多评论家认为这是一个错误的比较(例如，Mark C. Chu-Carroll 称"如果数据库是锤，MapReduce 则是螺丝刀，网址为 *http://scienceblogs.com/goodmath/2008/01/databases_are_hammers_mapreduc.php*，DeWitt 和 Stonebraker 紧跟着发表了"MapReduce Ⅱ"(网址为 *http://www.databasecolumn.com/2008/01/MapReduce-continued.html*)，回答了其他人提出的问题。

器访问共享文件系统，由一个存储区域网络(Storage Area Network，SAN)进行管理。这非常适用于以主计算密集型为主的作业，但当节点需要访问的大数据量(数百 GB 的数据，这是 MapReduce 实际开始“发光”的起点)时，这会成为一个问题，因为网络带宽成为“瓶颈”，所以计算节点闲置下来了。

MapReduce 尝试在计算节点本地存储数据，因此数据访问速度会因为它是本地数据而比较快。[①]这项“数据本地化”功能，成为 MapReduce 的核心功能并且也是它拥有良好性能的原因之一。意识到网络带宽在数据中心环境是最有价值的资源(到处复制数据会很容易的把网络带宽饱和)之后，MapReduce 便通过显式网络拓扑结构不遗余力地加以保护。请注意，这种安排不会排除 MapReduce 中的高 CPU 使用分析。

MPI 赋予程序员很大的控制，但也要求显式控制数据流机制，需要使用传统的 C 语言的功能模块完成(例如 socket)，以及更高级的算法来进行分析。而 MapReduce 却是在更高层面上完成任务，即程序员从键/值对函数的角度来考虑，同时数据流是隐含的。

在一个大规模分布式计算平台上协调进程是一个很大的挑战。最困难的部分是恰当的处理失效与错误——在不知道一个远程进程是否已经失败的时候——仍然需要继续整个计算。MapReduce 将程序员从必须考虑失败任务的情况中解放出来，它检测失败的 map 或者 reduce 任务，在健康的机器上重新安排任务。MapReduce 能够做到这一点，因为它是一个无共享的架构，这意味着各个任务之间彼此并不依赖。(这里讲得稍微简单了一些，因为 mapper 的输出是反馈给 reducer 的，但这由 MapReduce 系统控制。在这种情况下，相对于返回失败的 map，应该对返回 reducer 给予更多关注，因为它必须确保它可以检索到必要的 map 输出，如果不行，必须重新运行相关的 map 从而生成必要的这些输出。)因此，从程序员的角度来看，执行任务的顺序是无关紧要的。相比之下，MPI 程序必须显式地管理自己的检查点和恢复机制，从而把更多控制权交给程序员，但这样会加大编程的难度。

MapReduce 听起来似乎是一个相当严格的编程模型，而且在某种意义上看的确如此：我们被限定于键/值对的类型(它们按照指定的方式关联在一起)，mapper 和 reducer 彼此间的协作有限，一个接一个地运行(mapper 传输键/值对给 reducer)。对此，一个很自然的问题是：你是否能用它做点儿有用或普通的事情？

答案是肯定的。MapReduce 作为一个建立搜索索引产品系统，是由 Google 的工程师们开发出来的，因为他们发现自己一遍又一遍地解决相同的问题(MapReduce 的灵感来自传统的函数式编程、分布式计算和数据库社区)，但它后来被应用于其他

① Jim Gray 是把计算结合到数据的早期倡导者之一。见“Distributed Computing Economics”，2003 年 3 月，网址为 *http://research.microsoft.com/apps/pubs/default.aspx?id=70001*。

行业的其他许多应用。我们惊喜地看到许多算法的变体在 MapReduce 中得以表示，从图像图形分析，到基于图表的问题，再到机器学习算法[①]。它当然不能解决所有问题，但它是一个很普遍的数据处理工具。

第 14 章将介绍一些 Hadoop 应用范例。

1.3.3 志愿计算

人们第一次听说 Hadoop 和 MapReduce 的时候，经常会问："和 SETI@home 有什么区别？"SETI，全称为 Search for Extra-Terrestrial Intelligence(搜寻外星人)，运行着一个称为 SETI@home 的项目(*http://setiathome.berkeley.edu*)。在此项目中，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外星智慧生命信号。SETI@home 是最有名的拥有许多志愿者的项目，其他的还有 Great Internet Mersenne Prime Search(搜索大素数)与 Folding@home 项目(了解蛋白质构成及其与疾病之间的关系)。

志愿计算项目通过将他们试图解决的问题分为几个他们成为工作单元的块来工作，并将它们送到世界各地的电脑上进行分析。例如，SETI@home 的工作单元大约是 0.35 MB 的无线电望远镜数据，并且一个典型的计算机需要数小时或数天来分析。完成分析后，结果发送回服务器，客户端获得的另一项工作单元。作为防止欺骗的预防措施，每个工作单元必须送到三台机器上并且需要有至少两个结果相同才会被接受。

虽然 SETI@home 在表面上可能类似于 MapReduce(将问题分为独立的块，然后进行并行计算)，但差异还是显著的。SETI@home 问题是 CPU 高度密集型的，使其适合运行于世界各地成千上万台计算机上，因为相对于其计算时间而言，传输工作单元的时间微不足道[②]。志愿者捐献的是 CPU 周期，而不是带宽。

MapReduce 被设计为用来运行那些需要数分钟或数小时的作业，这些作业在一个聚集带宽很高的数据中心中可信任的专用硬件设备上运行。相比之下，SETI@home 项目是在接入互联网的不可信的计算机上运行，这些计算机的网速不同，而且数据也不在本地。

---

① Apache Mahout(*http://lucene.apache.org/mahout/*)是一个项目，目的是构建可运行于 Hadoop 之上的机器学习库(如分类和群集算法)。

② 2008 年 1 月，有报道(网址为 *http://www.planetary.org/programs/projects/setiathome/setiathome_20080115.html*)称，SETI@home 每天处理 300 千兆字节，使用 320 000 台电脑(其中大部分并没有专用于 SETI@home 项目，它们还用于其他用途)。

1.4 Hadoop 发展简史

Hadoop 是 Doug Cutting—— Apache Lucene 创始人—— 开发的使用广泛的文本搜索库。Hadoop 起源于 Apache Nutch，后者是一个开源的网络搜索引擎，本身也是由 Lucene 项目的一部分。

> ### Hadoop 名字的起源
>
> Hadoop 这个名字不是一个缩写，它是一个虚构的名字。该项目的创建者，Doug Cutting 如此解释 Hadoop 的得名："这个名字是我孩子给一头吃饱了的棕黄色大象命名的。我的命名标准就是简短，容易发音和拼写，没有太多的意义，并且不会被用于别处。小孩子是这方面的高手。Googol 就是由小孩命名的。"

Hadoop 及其子项目和后继模块所使用的名字往往也与其功能不相关，经常用一头大象或其他动物主题(例如："Pig")。较小的各个组成部分给与更多描述性(因此也更俗)的名称。这是一个很好的原则，因为它意味着可以大致从其名字猜测其功能，例如，jobtracker[1]的任务就是跟踪 MapReduce 作业。

从头开始构建一个网络搜索引擎是一个雄心勃勃的目标，不只是要编写一个复杂的、能够抓取和索引网站的软件，还需要面临着没有专有运行团队支持运行它的挑战，因为它有那么多独立部件。同样昂贵的还有：据 Mike Cafarella 和 Doug Cutting 估计，一个支持此 10 亿页的索引需要价值约 50 万美元的硬件投入，每月运行费用还需要 3 万美元。[2]不过，他们相信这是一个有价值的目标，因为这会开放并最终使搜索引擎算法普及化。

Nutch 项目开始于 2002 年，一个可工作的抓取工具和搜索系统很快浮出水面。但他们意识到，他们的架构将无法扩展到拥有数十亿网页的网络。在 2003 年发表的一篇描述 Google 分布式文件系统(简称 GFS)的论文为他们提供了及时的帮助，文中称 Google 正在使用此文件系统。[3]GFS 或类似的东西，可以解决他们在网络抓取和索引过程中产生的大量的文件的存储需求。具体而言，GFS 会省掉管理所花

---

① 在本书中，我们一般使用小写形式的"jobtracker"来表示实体，用 JobTracker 来表示实现它的 Java 类。

② Mike Cafarella 和 Doug Cutting，"Building Nutch：Open Source Search"，ACM Queue，2004 年 4 月，网址为 *http://queue.acm.org/detail.cfm?id=988408*。

③ Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung："The Google File System"，2003 年 10 月，网址为 *http://labs.google.com/papers/gfs.html*。

的时间，如管理存储节点。在 2004 年，他们开始写一个开放源码的应用，即 Nutch 的分布式文件系统(NDFS)。

2004 年，Google 发表了论文，向全世界介绍了 MapReduce。[①]2005 年初，Nutch 的开发者在 Nutch 上有了一个可工作的 MapReduce 应用，到当年年中，所有主要的 Nutch 算法被移植到使用 MapReduce 和 NDFS 来运行。

Nutch 中的 NDFS 和 MapReduce 实现的应用远不只是搜索领域，在 2006 年 2 月，他们从 Nutch 转移出来成为一个独立的 Lucene 子项目，称为 Hadoop。大约在同一时间，Doug Cutting 加入雅虎，Yahoo 提供一个专门的团队和资源将 Hadoop 发展成一个可在网络上运行的系统(见后文的补充材料)。在 2008 年 2 月，雅虎宣布其搜索引擎产品部署在一个拥有 1 万个内核的 Hadoop 集群上。[②]

2008 年 1 月，Hadoop 已成为 Apache 顶级项目，证明它是成功的，是一个多样化、活跃的社区。通过这次机会，Hadoop 成功地被雅虎之外的很多公司应用，如 Last.fm、Facebook 和《纽约时报》。(一些应用在第 14 章的案例研究和 Hadoop 维基有介绍，Hadoop 维基的网址为 *http://wiki.apache.org/hadoop/PoweredBy*。)

有一个良好的宣传范例，《纽约时报》使用亚马逊的 EC2 云计算将 4 TB 的报纸扫描文档压缩，转换为用于 Web 的 PDF 文件。[③]这个过程历时不到 24 小时，使用 100 台机器运行，如果不结合亚马逊的按小时付费的模式(即允许《纽约时报》在很短的一段时间内访问大量机器)和 Hadoop 易于使用的并行程序设计模型，该项目很可能不会这么快开始启动。

2008 年 4 月，Hadoop 打破世界纪录，成为最快排序 1TB 数据的系统。运行在一个 910 节点的群集，Hadoop 在 209 秒内排序了 1 TB 的数据(还不到三分半钟)，击败了前一年的 297 秒冠军。同年 11 月，谷歌在报告中声称，它的 MapReduce 实现执行 1TB 数据的排序只用了 68 秒。[④]在 2009 年 5 月，有报道宣称 Yahoo 的团队使用 Hadoop 对 1 TB 的数据进行排序只花了 62 秒时间。

---

① “Yahoo！ Launches World’s Largest Hadoop Production Application”，2008 年 2 月 19 日，网址为 *http://developer.yahoo.net/blogs/hadoop/2008/02/yahoo-worlds-largest-production-hadoop.html*。

② Jeffrey Dean Sanja Ghemawat 合作完成，“MapReduce:Simplified Data Processing Large Clusters”，2004 年 12 月，网址为 *http://labs.google.com/papers/mapreduce.html*。

③ Derek Gottfrid，“Self-service，Prorated Super Computing Fun!”，2007 年 11 月 1 日，网址为 *http://open.blogs.nytimes .com/2007/11/01/self-service-prorated-super-computing-fun/*。

④ “Sorting IPB with MapReduce”，20008 年 11 月 21 日，网址为 *http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html*。

Hadoop@Yahoo!

构建互联网规模的搜索引擎需要大量的数据，因此需要大量的机器来进行处理。Yahoo! Search 包括四个主要组成部分：Crawler，从因特网下载网页；WebMap，构建一个网络地图；Indexer，为最佳页面构建一个反向索引；Runtime(运行时)，回答用户的查询。WebMap 是一幅图，大约包括一万亿条边(每条代表一个网络链接)和一千亿个节点(每个节点代表不同的网址)。创建和分析此类大图需要大量计算机运行若干天。在 2005 年初，WebMap 所用的基础设施名为 Dreadnaught，需要重新设计以适应更多节点的需求。Dreadnaught 成功地从 20 个节点扩展到 600 个，但需要一个完全重新的设计，以进一步扩大。Dreadnaught 与 MapReduce 有许多相似的地方，但灵活性更强，结构更少。具体说来，每一个分段(fragment)，Dreadnaught 作业可以将输出发送到此作业下一阶段中的每一个分段，但排序是在库函数中完成的。在实际情形中，大多数 WebMap 阶段都是成对存在的，对应于 MapReduce。因此，WebMap 应用并不需要为了适应 MapReduce 而进行大量重构。

Eric Baldeschwieler(Eric14)组建了一个小团队，我们开始设计并原型化一个新的框架(原型为 GFS 和 MapReduce，用 C++语言编写)，打算用它来替换 Dreadnaught。尽管当务之急是我们需要一个 WebMap 新框架，但显然，标准化对于整个 Yahoo! Search 平台至关重要，并且通过使这个框架泛化，足以支持其他用户，我们才能够充分运用对整个平台的投资。

与此同时，我们在关注 Hadoop(当时还是 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，我们决定放弃我们的原型，转而使用 Hadoop。相较于我们的原型和设计，Hadoop 的优势在于它已经在 20 个节点上实际应用过。这样一来，我们便能在两个月内搭建一个研究集群，并着手帮助真正的客户使用这个新的框架，速度比原来预计的快许多。另一个明显的优点是 Hadoop 已经开源，较容易(虽然远没有那么容易！)从雅虎法务部门获得许可在开源方面进行工作。因此，我们在 2006 年初设立了一个 200 个节点的研究集群，我们将 WebMap 的计划暂时搁置，转而为研究用户支持和发展 Hadoop。

Hadoop 大事记

- 2004 年——最初的版本(现在称为 HDFS 和 MapReduce)由 Doug Cutting 和 Mike Cafarella 开始实施。
- 2005 年 12 月——Nutch 移植到新的框架，Hadoop 在 20 个节点上稳定运行。
- 2006 年 1 月——Doug Cutting 加入雅虎。
- 2006 年 2 月——Apache Hadoop 项目正式启动以支持 MapReduce 和 HDFS 的独立发展。
- 2006 年 2 月——雅虎的网格计算团队采用 Hadoop。
- 2006 年 4 月——标准排序(10 GB 每个节点)在 188 个节点上运行 47.9 个小时。
- 2006 年 5 月——雅虎建立了一个 300 个节点的 Hadoop 研究集群。
- 2006 年 5 月——标准排序在 500 个节点上运行 42 个小时(硬件配置比 4 月的更好)。
- 06 年 11 月——研究集群增加到 600 个节点。
- 06 年 12 月——标准排序在 20 个节点上运行 1.8 个小时，100 个节点 3.3 小时，500 个节点 5.2 小时，900 个节点 7.8 个小时。
- 07 年 1 月——研究集群到达 900 个节点。
- 07 年 4 月——研究集群达到两个 1000 个节点的集群。
- 08 年 4 月——赢得世界最快 1 TB 数据排序在 900 个节点上用时 209 秒。
- 08 年 10 月——研究集群每天装载 10 TB 的数据。
- 09 年 3 月——17 个集群总共 24 000 台机器。
- 09 年 4 月——赢得每分钟排序，59 秒内排序 500 GB(在 1400 个节点上)和 173 分钟内排序 100 TB 数据(在 3400 个节点上)。

1.5 Apache Hadoop 项目

今天，Hadoop 是一个分布式计算基础架构这把“大伞”下的相关子项目的集合。这些项目属于 Apache 软件基金会(*http://hadoop.apache.org*)，后者为开源软件项目

社区提供支持。虽然 Hadoop 最出名的是 MapReduce 及其分布式文件系统(HDFS，从 NDFS 改名而来)，但还有其他子项目提供配套服务，其他子项目提供补充性服务。这些子项目的简要描述如下，其技术栈如图 1-1 所示。

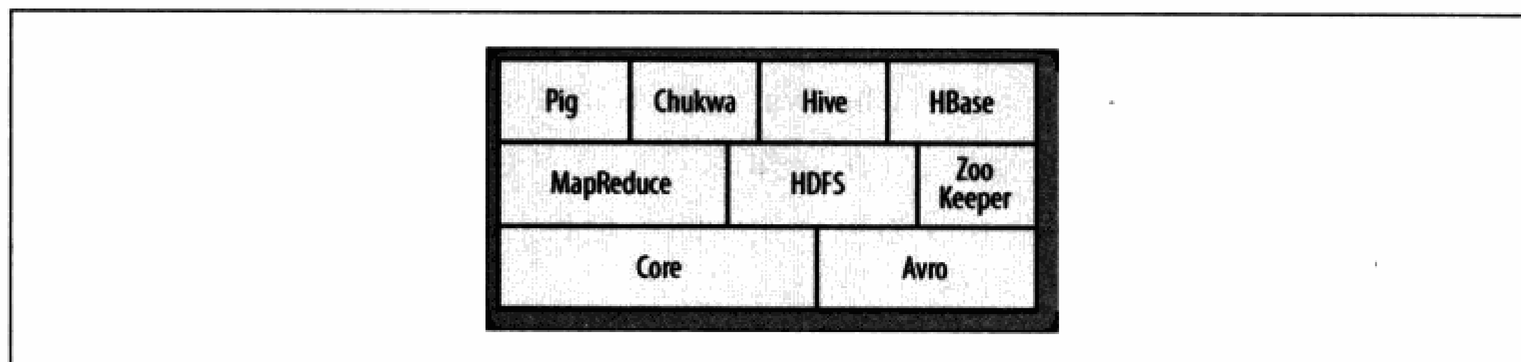


图 1-1：Hadoop 的子项目

Core

一系列分布式文件系统和通用 I/O 的组件和接口(序列化、Java RPC 和持久化数据结构)。

Avro

一种提供高效、跨语言 RPC 的数据序列系统，持久化数据存储。(在本书写作期间，Avro 只是被当作一个新的子项目创建，而且尚未有其他 Hadoop 子项目在使用它。)

MapReduce

分布式数据处理模式和执行环境，运行于大型商用机集群。

HDFS

分布式文件系统，运行于大型商用机集群。

Pig

一种数据流语言和运行环境，用以检索非常大的数据集。Pig 运行在 MapReduce 和 HDFS 的集群上。

Hbase

一个分布式的、列存储数据库。HBase 使用 HDFS 作为底层存储，同时支持 MapReduce 的批量式计算和点查询(随机读取)。

ZooKeeper

一个分布式的、高可用性的协调服务。ZooKeeper 提供分布式锁之类的基本服务用于构建分布式应用。

Hive

分布式数据仓库。Hive 管理 HDFS 中存储的数据，并提供基于 SQL 的查询语言(由运行时引擎翻译成 MapReduce 作业)用以查询数据。

Chukwa

分布式数据收集和分析系统。Chukwa 运行 HDFS 中存储数据的收集器，它使用 MapReduce 来生成报告。(在写作本书期间，Chukwa 刚刚从 Core 中的“contrib”模块分离出来独立成为一个独立的子项目。)

第 2 章

MapReduce 简介

MapReduce 是一种用于数据处理的编程模型。该模型非常简单。同一个程序 Hadoop 可以运行用各种语言编写的 MapReduce 程序。在本章中，我们将看到用 Java，Ruby，Python 和 C++这些不同语言编写的不同版本。最重要的是，MapReduce 程序本质上是并行的，因此可以将大规模的数据分析交给任何一个拥有足够多机器的运营商。MapReduce 的优势在于处理大型数据集，所以下面首先来看一个例子。

2.1 一个气象数据集

在我们这个例子里，要编写一个挖掘气象数据的程序。分布在全球各地的气象传感器每隔一小时便收集当地的气象数据，从而积累了大量的日志数据。它们是适合用 MapReduce 进行分析的最佳候选，因为它们是半结构化且面向记录的数据。

数据的格式

我们将使用 National Climatic Data Center(国家气候数据中心，NCDC，网址为 *http://www.ncdc.noaa.gov/*)提供的数据。数据是以面向行的 ASCII 格式存储的，每一行便是一个记录。该格式支持许多气象元素，其中许多数据是可选的或长度可变的。为简单起见，我们将重点讨论基本元素(如气温)，这些数据是始终都有且有固定宽度的。

例 2-1 显示了一个简单的示例行，其中一些重要字段加粗显示。该行已被分成多行以显示出每个字段，在实际文件中，字段被整合成一行且没有任何分隔符。

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 × 1000)
+028783     # longitude (degrees × 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 × 10)
1           # quality code
-0139       # dew point temperature (degrees Celsius × 10)
1           # quality code
10268       # atmospheric pressure (hectopascals × 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，每一个目录都包含一个打包文件，文件中的每一个气象站都带有当年的数据。例如，1990 年的前面的数据项如下：

```
% 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
```

因为实际生活中有成千上万个气象台，所以整个数据集由大量较小的文件组成。通常情况下，我们更容易、更有效地处理数量少的大型文件，因此，数据会被预先处理而使每年记录的读数连接到一个单独的文件中。(具体做法请参见附录 C)

2.2 使用 Unix Tools 来分析数据

在全球气温数据中每年记录的最高气温是多少？我们先不用 Hadoop 来回答这一问题，因为答案中需要提供一个性能标准(baseline)和一种检查结果的有效工具。

对于面向行的数据，传统的处理工具是 awk。例 2-2 是一个小的程序脚本，用于计算每年的最高气温。

例 2-2：一个用于从 NCDC 气象记录中找出每年最高气温的程序

```
#!/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
```

该脚本循环遍历压缩文件，首先显示年份，然后使用 awk 处理每个文件。awk 脚本从数据中提取两个字段：气温和质量代码。气温值通过加上一个 0 变成一个整数。接下来，执行测试，从而判断气温值是否有效(值 9999 代表在 NCDC 数据集缺少值)，质量代码显示的读数是有疑问还是根本就是错误的。如果读数是正确的，那么该值将与目前看到的最大值进行比较，如果该值比原先的最大值大，就替

换掉目前的最大值。当文件中所有的行都已处理完并打印出最大值后，END 块中的代码才会被执行。

下面是某次运行结果的开始部分：

```
%./max_temperature.sh
1901 317
1902 244
1903 289
1904 256
1905 283
...
```

由于源文件中的气温值按比例增加到 10 倍，所以结果 1901 年的最高气温是 31.7°C(在本世纪初几乎没有多少气温读数会被记录下来，所以这是可能的)。为完成对跨越一世纪这么长时间的查找，程序在 EC2 High-CPU Extra Large Instance 机器上一共运行了 42 分钟。

为加快处理，我们需要并行运行部分程序。从理论上讲，这很简单：我们可以通过使用计算机上所有可用的硬件线程来处理在不同线程中的各个年份的数据。但是这之中存在一些问题。

首先，划分成大小相同的作业块通常并不容易或明显。在这种情况下，不同年份的文件，大小差异很大，所以一些线程会比其他线程更早完成。即使它们继续下一步的工作，但是整个运行中占主导地位的还是那些运行时间很长的文件。另一种方法是将输入数据分成固定大小的块，然后把每块分配到各个进程。

其次，独立线程运行结果在合并后，可能还需要进一步的处理。在这种情况下，每年的结果是独立于其他年份，并可能通过连接所有结果和按年份排序这两种方式来合并它们。如果使用固定大小的块这种方法，则此类合并会更紧凑。对于这个例子，某年的数据通常被分割成几个块，每个进行独立处理。我们将最终获得每个数据块的最高气温，所以最后一步是寻找这些每年气温值中的最大值。

最后，我们仍然受限于一台计算机的处理能力。如果手中所有的处理器都使用上都至少需要 20 分钟，那就只能这样了。我们不能使它更快。另外，一些数据集的增长会超出一台计算机的处理能力。当我们开始使用多台计算机时，整个大环境中的许多其他因素将发挥作用，可能由于协调性和可靠性的问题而出现当机等错误。谁运行整个作业？我们如何处理失败的进程？

因此，尽管并行处理可行，但实际上它非常复杂。使用 Hadoop 之类的框架非常有助于处理这些问题。

2.3 使用 Hadoop 进行数据分析

为了更好地发挥 Hadoop 提供的并行处理机制的优势，我们必须把查询表示成 MapReduce 作业。经过一些本地的小规模测试，我们将能够在机器集群上运行它。

2.3.1 map 和 reduce

MapReduce 的工作过程分为两个阶段：map 阶段和 reduce 阶段。每个阶段都有键/值对作为输入和输出，并且它们的类型可由程序员选择。程序员还具体定义了两个函数：map 函数和 reduce 函数。

我们在 map 阶段输入的是原始的 NCDC 数据。我们选择的是一种文本输入格式，以便数据集的每一行都会是一个文本值。键是在文件开头部分文本行起始处的偏移量，但我们没有这方面的需要，所以将其忽略。

map 函数很简单。我们使用 map 函数来找出年份和气温，因为我们只对它们有兴趣。在本例中，map 函数只是一个数据准备阶段，通过这种方式来建立数据，使得 reducer 函数能在此基础上进行工作：找出每年的最高气温。map 函数也是很适合去除已损记录的地方：在这里，我们将筛选掉缺失的、不可靠的或错误的气温数据。

为了全面了解 map 的工作方式，我们思考下面几行示例的输入数据(考虑到页面篇幅，一些未使用的列已被去除，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/值对的方式来表示 map 函数：

```
(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...)
```

键是文件中的行偏移量，而这往往是我们在 map 函数中所忽视的。map 函数的功

能仅仅提取年份和气温(以粗体显示)，并将其作为输出被发送。(气温值已被解释为整数)

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

map 函数的输出先由 MapReduce 框架处理，然后再被发送到 reduce 函数。这一处理过程根据键来对键/值对进行排序和分组。因此，继续我们的示例，reduce 函数会看到如下输入：

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

每年的年份后都有一系列气温读数。所有 reduce 函数现在必须重复这个列表并从中找出最大的读数：

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

这是最后的输出：全球气温记录中每年的最高气温。

整个数据流如图 2-1 所示。在图的底部是 Unix 的管道，模拟整个 MapReduce 的流程，其中的内容我们将在以后讨论 Hadoop 数据流时再次提到。

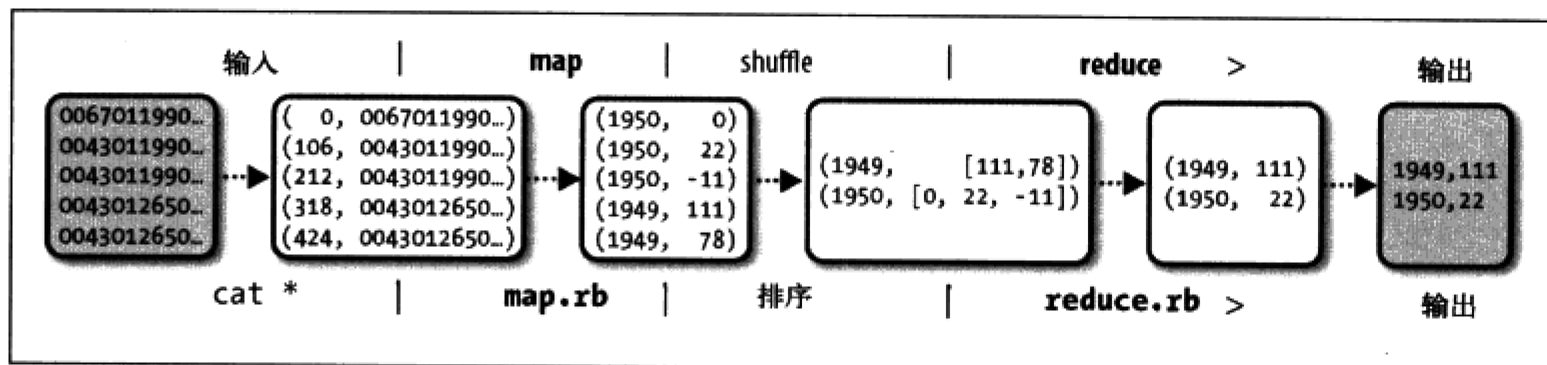


图 2-1：MapReduce 的逻辑数据流

2.3.2 Java MapReduce

在明白 MapReduce 程序的工作原理之后，下一步就是要用代码来实现它。我们需要三样东西：一个 map 函数、一个 reduce 函数和一些来运行作业的代码。map 函数是由一个 Mapper 接口来实现的，其中声明了一个 map()方法。例 2-3 显示了我

们的 map 函数的实现。

例 2-3：最高气温示例的 Mapper 接口

```
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.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;

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

  private static final int MISSING = 9999;

  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));
  }
 }
}
```

该 Mapper 接口是一个泛型类型，它有 4 个形式参数类型，由它们来指定 map 函数的输入键、输入值、输出键和输出值的类型。就目前的示例来说，输入键是一个长整数偏移量，输入的值是一行文本，输出的键是年份，输出的值是气温(整数)。Hadoop 规定了自己的一套可用于网络序列优化的基本类型，而不是使用内置的 Java 类型。这些都可以在 org.apache.hadoop.io 包中找到。现在我们使用的是

LongWritable 类型(相当于 Java 的 Long 类型)、Text 类型(相当于 Java 的 String 类型)和 IntWritable 类型(相当于 Java 的 Integer 类型)。

map()方法需要传入一个键和一个值。我们将一个包含 Java 字符串输入行的 Text 值转换成 Java 的 String 类型，然后利用其 substring()方法提取我们感兴趣的列。

map()方法还提供了一个 OutputCollector 实例来写入输出内容。在这种情况下，我们写入年份作为一个 Text 对象(因为我们只使用一个键)，用 IntWritable 类型包装气温值。我们只有在气温显示出来后并且它的质量代码表示的是正确的气温读数时才写入输出记录。

reduce 函数同样在使用 Reducer 时被定义，如例 2-4 所示。

例 2-4：最高气温示例的 Reducer

```
import java.io.IOException;
import java.util.Iterator;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;

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

  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));
  }
}
```

同样，四个形式参数类型用于指定 reduce 函数的输入和输出类型。reduce 函数的输入类型必须与 map 函数的输出类型相匹配：Text 类型和 IntWritable 类型。在这种情况下，reduce 函数的输出类型是 Text 和 IntWritable 这两种类型，前者是

年份的类型而后者是最高气温的类型，在这些输入类型之中，我们遍历所有气温，并把每个记录进行比较直到找到一个最高的为止。

第三部分代码运行的是 MapReduce 作业(请参见例 2-5)。

例 2-5：在气象数据集中找出最高气温的应用程序

```
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.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;

public class MaxTemperature {

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

    JobConf conf = new JobConf(MaxTemperature.class);
    conf.setJobName("Max temperature");

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

    conf.setMapperClass(MaxTemperatureMapper.class);
    conf.setReducerClass(MaxTemperatureReducer.class);

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

    JobClient.runJob(conf);
  }
}
```

JobConf 对象指定了作业的各种参数。它授予你对整个作业如何运行的控制权。当我们在 Hadoop 集群上运行这个作业时，我们把代码打包成一个 JAR 文件(Hadoop

会在集群分发这个包)。我们没有明确指定 JAR 文件的名称，而是在 JobConf 构造函数中传送一个类，Hadoop 会找到这个包含此类的 JAR 文件。

在创建 JobConf 对象后，我们将指定输入和输出的路径。通过调用 FileInputFormat 内的静态方法 addInputPath()来定义输入的路径，它可以是单个文件、目录(本例中，输入的内容组成此目录下所有文件)或文件模式的路径。同时，addInputPath()可被调用多次从而实现使用多路径输入。

输出路径(其中只有一个)是在 FileOutputFormat 内的静态方法 setOutputPath()来指定的。它指定了 reduce 函数输出文件写入的目录。在运行作业前该目录不应该存在，否则 Hadoop 会报错并且拒绝运行任务。这种预防措施是为了防止数据丢失(一个长时间的任务可能非常恼人地被另一个意外覆盖)。

接下来，通过 setMapperClass()和 setReducerClass()这两个方法来指定要使用的 map 和 reduce 类型。

setOutputKeyClass()和 setOutputValueClass()方法控制 map 和 reduce 函数的输出类型，正如本例所示，这两个方法往往是相同的。如果它们不同，那么 map 的输出类型可设置成使用 setMapOutputKeyClass()和 setMapOutputValueClass()方法。

输入的类型通过输入格式来控制，我们没有设置，因为我们使用的是默认的 TextInputFormat(文本输入格式)。

在设置了定义 map 和 reduce 函数的类之后，运行作业的准备工作就算完成了。JobClient 内的静态方法 runJob()会提交作业并等待它完成，把进展情况写入控制台。

运行测试

写完 MapReduce 作业之后，拿一个小型的数据集进行测试以排除与代码直接有关的问题，这是常规做法。首先，以独立模式安装 Hadoop(详细说明请参见附录 A)。在这种模式下，Hadoop 运行中使用本地带 job runner(作业运行程序)的文件系统。让我们用前面讨论过的五行代码的例子来测试它(考虑到页面，这里已经对输出稍做修改和重新排版)：

```
% export HADOOP_CLASSPATH=build/classes
% hadoop MaxTemperature input/ncdc/sample.txt output
09/04/07 12:34:35 INFO jvm.JvmMetrics: Initializing JVM Metrics
with processName=Job
```

```
Tracker, sessionId=
09/04/07 12:34:35 WARN mapred.JobClient: Use
GenericOptionsParser for parsing the
arguments. Applications should implement Tool for the same.
09/04/07 12:34:35 WARN mapred.JobClient: No job jar file set.
User classes may not
be found. See JobConf(Class) or JobConf#setJar(String).
09/04/07 12:34:35 INFO mapred.FileInputFormat: Total input
paths to process : 1
09/04/07 12:34:35 INFO mapred.JobClient: Running job:
job_local_0001
09/04/07 12:34:35 INFO mapred.FileInputFormat: Total input
paths to process : 1
09/04/07 12:34:35 INFO mapred.MapTask: numReduceTasks: 1
09/04/07 12:34:35 INFO mapred.MapTask: io.sort.mb = 100
09/04/07 12:34:35 INFO mapred.MapTask: data buffer =
79691776/99614720
09/04/07 12:34:35 INFO mapred.MapTask: record buffer =
262144/327680
09/04/07 12:34:35 INFO mapred.MapTask: Starting flush of map output
09/04/07 12:34:36 INFO mapred.MapTask: Finished spill 0
09/04/07 12:34:36 INFO mapred.TaskRunner:
Task:attempt_local_0001_m_000000_0 is
done. And is in the process of commiting
09/04/07 12:34:36 INFO mapred.LocalJobRunner:
file:/Users/tom/workspace/htdg/input/n
cdc/sample.txt:0+529
09/04/07 12:34:36 INFO mapred.TaskRunner: Task
'attempt_local_0001_m_000000_0' done.
09/04/07 12:34:36 INFO mapred.LocalJobRunner:
09/04/07 12:34:36 INFO mapred.Merger: Merging 1 sorted segments
09/04/07 12:34:36 INFO mapred.Merger: Down to the last merge-
pass, with 1 segments
left of total size: 57 bytes
09/04/07 12:34:36 INFO mapred.LocalJobRunner:
09/04/07 12:34:36 INFO mapred.TaskRunner:
Task:attempt_local_0001_r_000000_0 is done
. And is in the process of commiting
09/04/07 12:34:36 INFO mapred.LocalJobRunner:
09/04/07 12:34:36 INFO mapred.TaskRunner: Task
attempt_local_0001_r_000000_0 is
```

```
allowed to commit now
09/04/07 12:34:36 INFO mapred.FileOutputCommitter: Saved output
of task
'attempt_local_0001_r_000000_0' to
file:/Users/tom/workspace/htdg/output
09/04/07 12:34:36 INFO mapred.LocalJobRunner: reduce > reduce
09/04/07 12:34:36 INFO mapred.TaskRunner: Task
'attempt_local_0001_r_000000_0' done.
09/04/07 12:34:36 INFO mapred.JobClient: map 100% reduce 100%
09/04/07 12:34:36 INFO mapred.JobClient: Job complete:
job_local_0001
09/04/07 12:34:36 INFO mapred.JobClient: Counters: 13
09/04/07 12:34:36 INFO mapred.JobClient: FileSystemCounters
09/04/07 12:34:36 INFO mapred.JobClient: FILE_BYTES_READ=27571
09/04/07 12:34:36 INFO mapred.JobClient:
FILE_BYTES_WRITTEN=53907
09/04/07 12:34:36 INFO mapred.JobClient: Map-Reduce Framework
09/04/07 12:34:36 INFO mapred.JobClient: Reduce input groups=2
09/04/07 12:34:36 INFO mapred.JobClient: Combine output
records=0
09/04/07 12:34:36 INFO mapred.JobClient: Map input records=5
09/04/07 12:34:36 INFO mapred.JobClient: Reduce shuffle bytes=0
09/04/07 12:34:36 INFO mapred.JobClient: Reduce output
records=2
09/04/07 12:34:36 INFO mapred.JobClient: Spilled Records=10
09/04/07 12:34:36 INFO mapred.JobClient: Map output bytes=45
09/04/07 12:34:36 INFO mapred.JobClient: Map input bytes=529
09/04/07 12:34:36 INFO mapred.JobClient: Combine input
records=0
09/04/07 12:34:36 INFO mapred.JobClient: Map output records=5
09/04/07 12:34:36 INFO mapred.JobClient: Reduce input records=5
```

如果 Hadoop 命令是以类名作为第一个参数，它就会启动一个 JVM 来运行这个类。使用命令比直接使用 Java 更方便，因为前者把类的路径(及其依赖关系)加入 Hadoop 的库中，并获得 Hadoop 的配置。要添加应用程序类的路径，我们需要定义一个 HADOOP_CLASSPATH 环境变量，Hadoop 脚本会来执行相关操作。

注意：以本地(独立)模式运行时，本书所有程序希望都以这种方式来设置 HADOOP_CLASSPATH。命令必须在示例代码所在的文件夹下被运行。

运行作业所得到的输出提供了一些有用的信息。(无法找到作业 JAR 文件的相关信

息是意料之中的，因为我们是在本地模式下没有 JAR 的情况下运行的。在集群上运行时，不会看到此警告。)例如，我们可以看到，这个作业被给予了一个 ID job_local_0001，并且它运行了一个 map 任务和一个 reduce 任务(使用 attempt_local_0001_m_000000_0 和 attempt_local_0001_r_000000_0 两个 ID)。在调试 MapReduce 作业时，知道作业和任务的 ID 是非常有用的。

输出的最后一部分叫“计数器”(Counter)，显示了在 Hadoop 上运行的每个作业产生的统计信息。这些对检查处理的数据量是否符合预期非常有用。例如，我们可以遵循整个系统中记录的数目：5 个 map 输入产生了 5 个 map 的输出，然后 5 个 reduce 输入产生两个 reduce 输出。

输出被写入 output 目录，其中每个 reducer 包括一个输出文件。作业包含一个 reducer，所以我们只能找到一个文件，名为 part-00000：

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

这个结果和我们之前手动寻找的结果一样。我们可以把前面这个结果解释为在 1949 年的最高气温记录为 11.1℃，而在 1950 年为 2.2℃。

新的 Java Mapreduce API

Hadoop 最新版 Java MapReduce Release 0.20.0 的 API 包括一个全新的 MapReduce Java API，有时也称为“context object”(上下文对象)，旨在使 API 在未来更容易扩展。新的 API 类型上不兼容以前的 API，所以，以前的应用程序需要重写才能使新的 API 发挥其作用[①]。

新的 API 和旧的 API 之间有下面几个明显的区别。

- 新的 API 倾向于使用抽象类，而不是接口，因为这更容易扩展。例如，你可以添加一个方法(用默认的实现)到一个抽象类而不需修改类之前的实现方法。在新的 API 中，Mapper 和 Reducer 是抽象类。
- 新的 API 是在 org.apache.hadoop.mapreduce 包(和子包)中的。之前版本的 API 则是放在 org.apache.hadoop.mapred 中的。

① 本书写作期间，Hadoop 中并非所有的 MapReduce 库都可被移植到新的 API 中。这就是本书适用旧版的 API 的原因。然而，本书的一些例子将被重写来使用新的 API，这些可以在本书网站上获得。

- 新的 API 广泛使用 context object(上下文对象)，并允许用户代码与 MapReduce 系统进行通信。例如，MapContext 基本上充当着 JobConf 的 OutputCollector 和 Reporter 的角色。
- 新的 API 同时支持“推”和“拉”式的迭代。在这两个新老 API 中，键/值记录对被推 mapper 中，但除此之外，新的 API 允许把记录从 map()方法中拉出，这也适用于 reducer。“拉”式的一个有用的例子是分批处理记录，而不是一个接一个。
- 新的 API 统一了配置。旧的 API 有一个特殊的 JobConf 对象用于作业配置，这是一个对于 Hadoop 通常的 Configuration 对象的扩展(用于配置守护进程，请参见 5.1 节)。在新的 API 中，这种区别没有了，所以作业配置通过 Configuration 来完成。
- 作业控制的执行由 Job 类来负责，而不是 JobClient，它在新的 API 中已经荡然无存。

例 2-6 使用新 API 重写了 MaxTemperature 的代码，不同之处用黑体字突出显示。

例 2-6：使用新的 context object(上下文对象)MapReduce API 在气象数据集中查找最高气温

```
public class NewMaxTemperature {
  static class NewMaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

    private static final int MISSING = 9999;

    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));
    }
  }
}

static class NewMaxTemperatureReducer
  extends Reducer<Text, IntWritable, Text, IntWritable> {

  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));
  }
}

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

  Job job = new Job();
  job.setJarByClass(NewMaxTemperature.class);

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

  job.setMapperClass(NewMaxTemperatureMapper.class);
  job.setReducerClass(NewMaxTemperatureReducer.class);

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

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

```
    }
}
```

2.4 分布化

前面展示了 MapReduce 针对小量输入的工作方式，现在是时候整体了解系统并进入大数据流作为输入了。为简单起见，我们的例子到目前为止都使用本地文件系统中的文件。然而，为了分布化，我们需要把数据存储在分布式文件系统中，典型的如 HDFS(详情参见第 3 章)，以允许 Hadoop 把 MapReduce 的计算移到承载部分数据的各台机器。下面我们就来看看这是如何工作的。

2.4.1 数据流

首先是一些术语的说明。MapReduce 作业(job)是客户端执行的单位：它包括输入数据、MapReduce 程序和配置信息。Hadoop 通过把作业分成若干个小任务(task)来工作，其包括两种类型的任务：map 任务和 reduce 任务。

有两种类型的节点控制着作业执行过程：jobtracker 和多个 tasktracker。jobtracker 通过调度任务在 tasktracker 上运行，来协调所有运行在系统上的作业。Tasktracker 运行任务的同时，把进度报告传送到 jobtracker，jobtracker 则记录着每项任务的整体进展情况。如果其中一个任务失败，jobtracker 可以重新调度任务到另外一个 tasktracker。Hadoop 把输入数据划分成等长的小数据发送到 MapReduce，称为输入分片(input split)或分片。Hadoop 为每个分片(split)创建一个 map 任务，由它来运行用户自定义的 map 函数来分析每个分片中的记录。

拥有许多分片就意味着处理每个分片的时间与处理整个输入的时间相比是比较小的。因此，如果我们并行处理每个分片，且分片是小块的数据，那么处理过程将有一个更好的负载平衡，因为更快的计算机将能够比一台速度较慢的机器在作业过程中处理完比例更多的数据分片。即使是相同的机器，没有处理的或其他同时运行的作业也会使负载平衡得以实现，并且在分片变得更细时，负载平衡质量也会更佳。

另一方面，如果分片太小，那么管理分片的总时间和 map 任务创建的总时间将决定作业的执行的总时间。对于大多数作业，一个理想的分片大小往往是一个 HDFS 块的大小，默认是 64 MB，虽然这可以根据集群进行调整(对于所有新建文件)或在新建每个文件时具体进行指定。

map 任务的执行节点和输入数据的存储节点是同一节点，Hadoop 的性能达到最佳。这就是所谓的 data locality optimization(数据局部性优化)。现在我们应该清楚为什么最佳分片的大小与块大小相同：它是最大的可保证存储在单个节点上的数据量。如果分区跨越两个块，那么对于任何一个 HDFS 节点而言，基本不可能同时存储这两数据块，因此此分布的某部分必须通过网络传输到节点，这与使用本地数据运行 map 任务相比，显然效率更低。

map 任务把输出写入本地硬盘，而不是 HDFS。这是为什么？因为 map 的输出作为中间输出：而中间输出则被 reduce 任务处理后产生最终的输出，一旦作业完成，map 的输出就可以删除了。因此，把它及其副本存储在 HDFS 中，难免有些小题大做。如果该节点上运行的 map 任务在 map 输出给 reduce 任务处理之前崩溃，那么 Hadoop 将在另一个节点上重新运行 map 任务以再次创建 map 的输出。

reduce 任务并不具备数据本地读取的优势——一个单一的 reduce 任务的输入往往来自于所有 mapper 的输出。在本例中，我们有一个单独的 reduce 任务，其输入是由所有 map 任务的输出组成的。因此，有序 map 的输出必须通过网络传输到 reduce 任务运行的节点，并在那里进行合并，然后传递到用户定义的 reduce 函数中。为增加其可靠性，reduce 的输出通常存储在 HDFS 中。如第 3 章所述，对于每个 reduce 输出的 HDFS 块，第一个副本存储在本地节点上，其他副本存储在其他机架节点中。因此，编写 reduce 的输出确实十分占用网络带宽，但是只和正常的 HDFS 写管线的消耗一样。

一个单一的 reduce 任务的整个数据流如图 2-2 所示。虚线框表示节点，虚线箭头表示数据传输到一个节点上，而实线的箭头表示节点之间的数据传输。

reduce 任务的数目并不是由输入的大小来决定，而是单独具体指定的。在第 7 章的 7.1 节中，将介绍如何为一个给定的作业选择 reduce 任务数量。

如果有多个 reducer，map 任务会对其输出进行分区，为每个 reduce 任务创建一个分区(partition)。每个分区包含许多键(及其关联的值)，但每个键的记录都在同一个分区中。分区可以通过用户定义的 partitioner 来控制，但通常是用默认的分区工具，它使用的是 hash 函数来形成“木桶”键/值，这种方法效率很高。

一般情况下，多个 reduce 任务的数据流如图 2-3 所示。此图清楚地表明了 map 和 reduce 任务之间的数据流为什么要称为“shuffle”(洗牌)，因为每个 reduce 任务的输入都由许多 map 任务来提供。shuffle 其实比此图所显示的更复杂，并且调整它可能会对作业的执行时间产生很大的影响，详见 6.4 节。

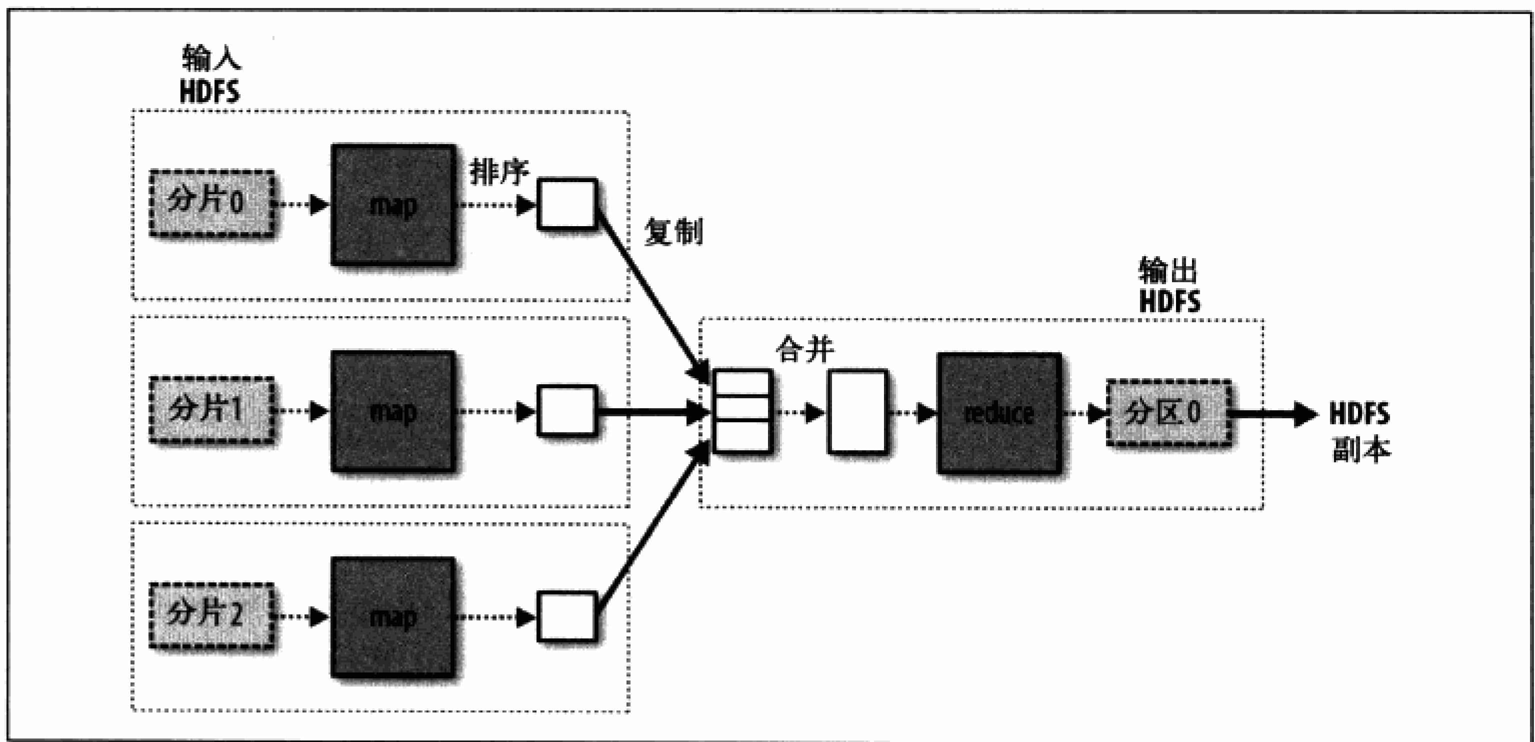


图 2-2：MapReduce 中单一 reduce 任务的数据流图

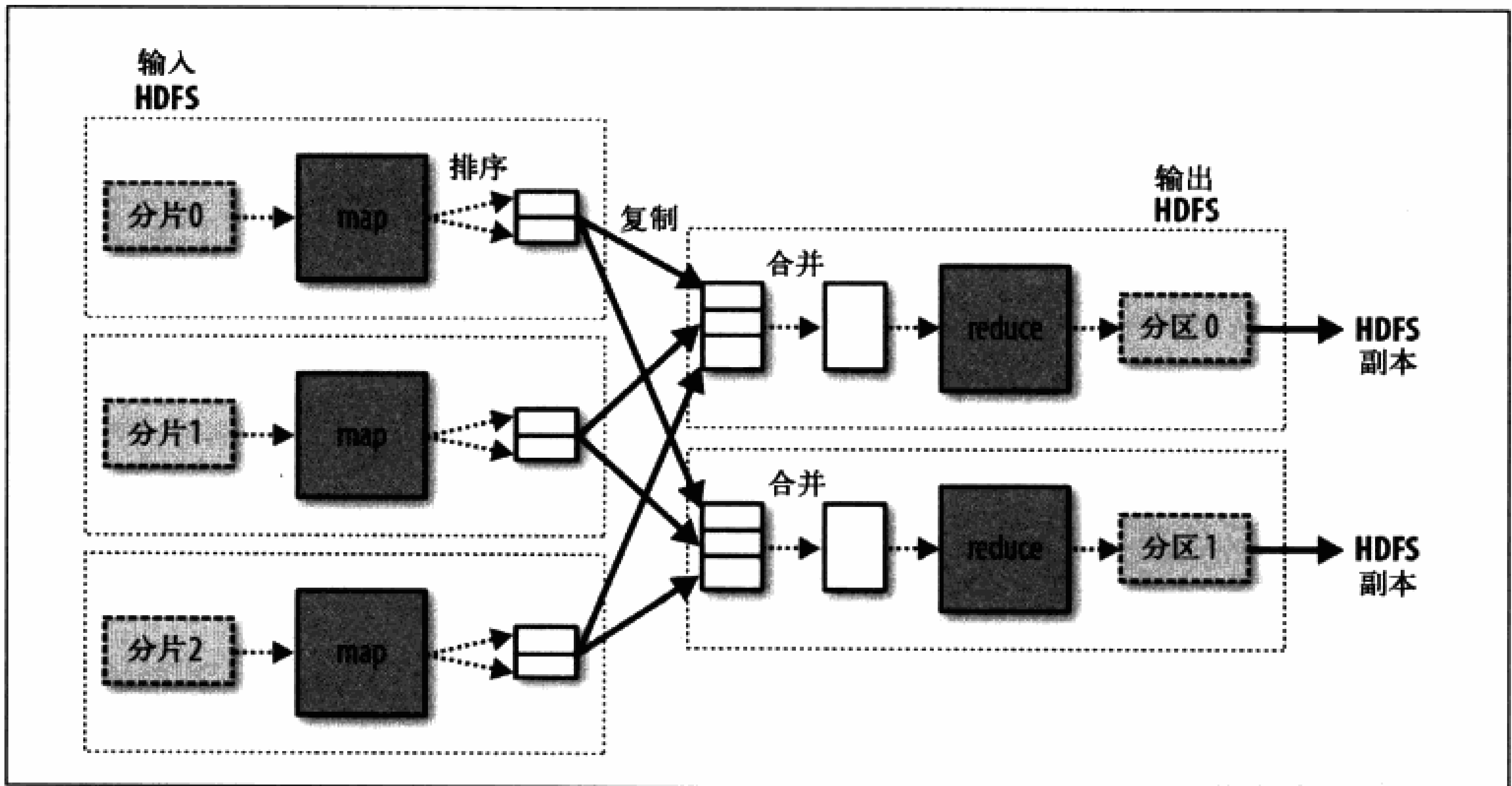


图 2-3：多个 reduce 任务的 MapReduce 数据流

最后，也有可能不存在 reduce 任务，不需要 shuffle 的时候，这样的情况是可能的，因为处理可以并行进行(第 7 章有几个例子讨论了这个问题)。在这种情况下，唯一的非本地节点数据传输是当 map 任务写入到 HDFS 中(见图 2-4)。

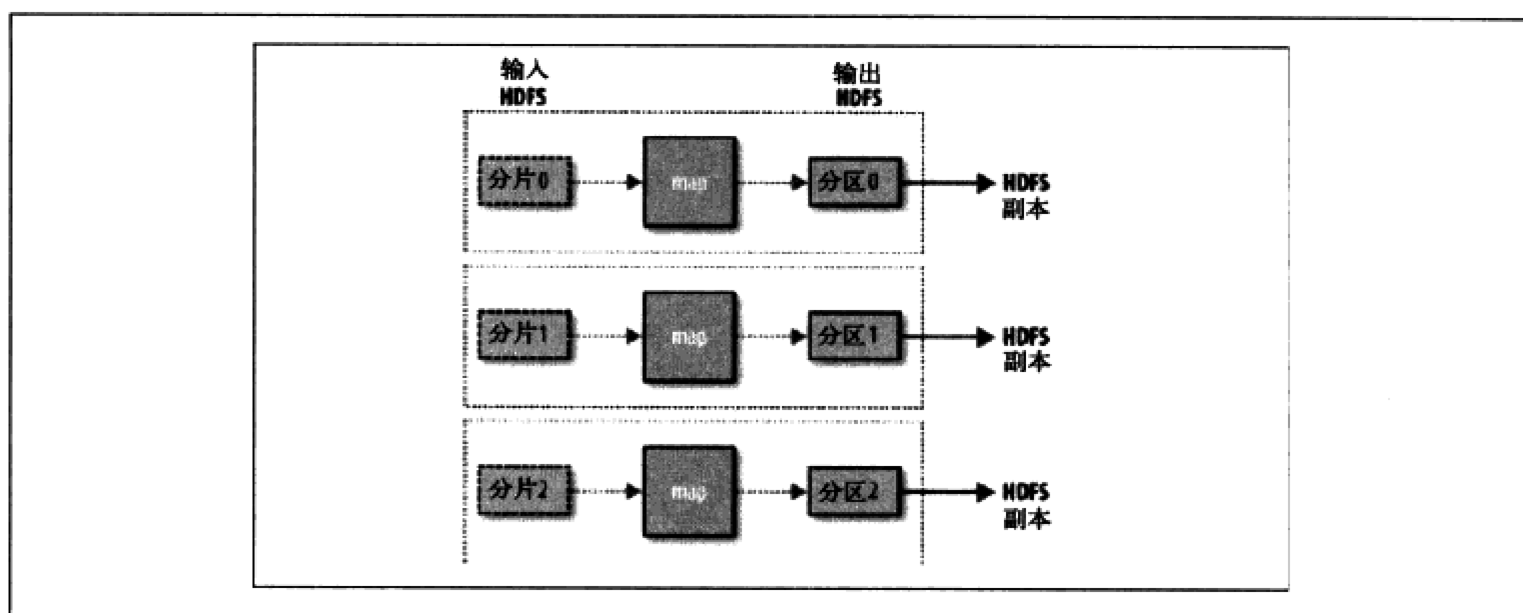


图 2-4：MapReduce 中没有 reduce 任务的数据流

集群的可用带宽限制了 MapReduce 作业的数量，因此 map 和 reduce 任务之间数据传输的代价是最小的。Hadoop 允许用户声明一个 combiner，运行在 map 的输出上——该函数的输出作为 reduce 函数的输入。由于 combiner 是一个优化方法，所以 Hadoop 不保证对于某个 map 的输出记录是否调用该方法，调用该方法多少次。换言之，不调用该方法或者调用该方法多次，reducer 的输出结果都一样。

combiner 的规则限制着可用的函数类型。我们将用一个例子来巧妙地加以说明。以前面的最高气温例子为例，1950 年的读数由两个 map 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 函数再调用时被传入以下数字：

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

因为 25 是输入值中的最大数，所以输出如下：

```
(1950, 25)
```

我们可以用 combiner，像 reduce 函数那样，为每个 map 输出找到最高气温。reduce 函数被调用时将被传入如下数值：

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

然而，reduce 输出的结果和以前一样。更简单地说，我们可以像下面这样，对本例中的气温值进行如下函数调用：

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

并非所有函数都有此属性。例如，如果我们计算平均气温，便不能用 mean 作为 combiner，因为：

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

但是：

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

combiner 并不能取代 reduce 函数。(为什么呢？reduce 函数仍然需要处理来自不同的 map 给出的相同键记录。)但它可以帮助减少 map 和 reduce 之间的数据传输量，而正因为此，是否在 MapReduce 作业中使用 combiner 是需要慎重考虑的。

2.4.2 具体定义一个 combiner

让我们回到 Java MapReduce 程序，combiner 是用 `reducer` 接口来定义的，并且该应用程序的实现与 `MaxTemperatureReducer` 中的函数相同。唯一需要强调的不同是如何在 `JobConf` 上设置 `combiner` 类(见例 2-7)。[①]

例 2-7：使用 combiner 高效查找最高气温

```
public class MaxTemperatureWithCombiner {

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

① 在 Gray 等 1995 年发表的论文“Data Cube：A Relational Aggregation OperatorGeneralizingGroup-By，Cross-Tab, and Sub-Totals”中，具有此属性的函数被称为是可分布的(distributive)。

```java
    JobConf conf = new JobConf(MaxTemperatureWithCombiner.class);
    conf.setJobName("Max temperature");

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

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

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

    JobClient.runJob(conf);
  }
}
```

2.4.3 运行分布式 MapReduce 作业

同一个程序将在一个完整的数据集中直接运行而不做更改。这是 MapReduce 的优势之一：它扩充数据大小和硬件规模。这里有一个运行结果：在一个 10 节点 EC2 High-CPU Extra Large Instance，程序只用 6 分钟就运行完了。[1]

我们将在第 5 章分析程序在集群上运行的机制。

2.5 Hadoop 流

Hadoop 提供了一个 API 来运行 MapReduce，并允许你用除 java 以外的语言来编写自己的 map 和 reduce 函数。Hadoop 流使用 Unix 标准流作为 Hadoop 和程序之间的接口，所以可以使用任何语言，只要编写的 MapReduce 程序能够读取标准输入，并写入到标准输出。

流适用于文字处理(尽管 0.21.0 版本也可以处理二进制流)，在文本模式下使用时，它有一个面向行的数据视图。map 的输入数据把标准输入流传输到 map 函数，其

① 此速度是在单机上使用 awk 串行运行的七倍(译注：前面 P18 指出单机 awk 使用 42 分钟，此处 10 节点 MapReduce 使用 6 分钟，所以 7 倍。但没有达到线性的 10 倍)，因为输入的数据不是平均分割的。为方便起见，输入文件用年份来压缩，从而造成随着气象记录数量越来越多，在数据集中后面年份文件变得越来越大。

中是一行一行的传输，然后再把行写入标准输出。一个 map 输出的键/值对是以单一的制表符分隔的行来写入的。reduce 函数的输入具有相同的格式——通过制表符来分隔的键/值对——传输标准输入流。reduce 函数从标准输入流读入行，然后为保证结果的有序性用键来排序，最后将结果写入标准输出。

下面使用流来重写按年份寻找最高气温的 MapReduce 程序。

2.5.1 Ruby 语言

例 2-8 中的 map 函数是用 ruby 来写的。

例 2-8：用于查找最高气温的 map 函数(ruby 版)

```
#!/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
```

程序通过执行 STDIN(IO 类型的一个全局常量)中的每行，不断从标准输入中反复查找。它把相关的字段从每行中取出，如果气温有效，就过一个 tab 字符\t 把年份和气温通隔开，然后输入标准输出(使用 puts)。

注意：指出流和 Java MapReduce API 之间的差异是十分值得的。Java 的 API 是主要面向一次处理一个 map 函数的记录。该框架调用 Mapper 的 map()方法来处理输入中的每条记录，然而 map 程序可以决定如何处理输入流，例如，它可以轻松地读取和同一时间处理多行，因为它控制着多行读取。用户的 Java map 实现是压栈记录，但它仍可以考虑处理多行，具体做法是将 mapper 中实例变量中之前的行汇聚在一起。[①] 在这种情况下，需要实现 close()方法，以便了解最后的记录何时被读取，从而完成对最后一组记录行的处理。

由于该脚本只运行在标准输入和输出上，所以只需使用 Unix 管道而不使用 Hadoop 来进行测试，这是不具有说服力的：

```
% cat input/ncdc/sample.txt | src/main/ch02/ruby/max_
```

① 此外，也可以在新的 MapReduce API 中使用“拉”式处理方式。

```
temperature_map.rb
1950 +0000
1950 +0022
1950 -0011
1949 +0111
1949 +0078
```

例 2-9 显示的 reduce 函数稍微复杂一些。

例 2-9：用于查找最高气温的 Reduce 函数(Ruby 版本)

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

last_key, max_val = nil, 0
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，但是当我们处理每组键时，我们要存储一些状态。在这种情况下，这个键是气象站的标识符，我们把看到的最后一个键和迄今为止见到的最高气温存储到那个键组中。MapReduce 框架确保键是有序的，以便让我们知道，如果一个键与其上一个键不同，可将其它移入一个新的键组。相比之下，Java API 则提供每一个键组的迭代器，我们只能在流中使用程序来找到键组的边界。

我们从每行中取出键和值，随后，如果刚刚完成一组键(`last_key` & `last_key =` 键)，则在将新值更新为最高气温之前，写入键和该组的最高气温，它们由一个制表符来分隔。如果我们没有结束一个组，只能更新当前键的最高气温。

程序的最后一行保证了输入的最后一组键会有一行输出。

现在可以用 UNIX 管线(来模拟整个 MapReduce 的管线，它与图 2-1 显示的 Unix 管线是相同的)。

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

输出结果和 Java 程序一样，所以下一步是用 Hadoop 来运行它。

Hadoop 命令不支持 Streaming 函数，因此，我们需要指定 JAR 文件流与 jar 选项。流程序的选项指定了输入和输出路径选项，map 和 reudce 脚本，如下所示：

```
% hadoop jar $HADOOP_INSTALL/contrib/streaming/hadoop-*-
  streaming.jar \
  -input input/ncdc/sample.txt \
  -output output \
  -mapper src/main/ch02/ruby/max_temperature_map.rb \
  -reducer src/main/ch02/ruby/max_temperature_reduce.rb
```

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

从 0.21.0 版开始，combiner 可以是任何一个流命令。对于早期版本的 combiner，只能用 Java 来编写，所以作为变通的方法，一般都在 mapper 中手动合并，从而避开 Java 语言。在这种情况下，我们可以把 mapper 改成管线：

```
% hadoop jar $HADOOP_INSTALL/contrib/streaming/hadoop-*-
  streaming.jar \
  -input input/ncdc/all \
  -output output \
  -mapper "ch02/ruby/max_temperature_map.rb | sort |
   ch02/ruby/max_temperature_reduce.rb" \
  -reducer src/main/ch02/ruby/max_temperature_reduce.rb \
  -file src/main/ch02/ruby/max_temperature_map.rb \
  -file src/main/ch02/ruby/max_temperature_reduce.rb
```

还要注意`-file`的使用，在集群上运行流程序把脚本传输到集群上时，可使用此选项。

2.5.2 Python

Hadoop 流支持任何可以从标准输入读取和写入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 脚本参见例 2-10，reduce 脚本参见例 2-11。

例 2-10：用于查找最高气温的 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], val[87:92], val[92:93])
  if (temp != "+9999" and re.match("[01459]", q)):
    print "%s\t%s" % (year, temp)
```

例 2-11　用于查找最高气温的 Reduce 函数(Python 版本)

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

import sys

(last_key, max_val) = (None, 0)
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)
```

我们可以测试程序且用在 Ruby 中用过的相同的方法来运行作业。比如，运行一个测试：

```
% cat input/ncdc/sample.txt |
```

① 作为流的备用方案之一，Python 程序员可以考虑 Dumbo(*http://www.last.fm/dumbo*)，这将使 MapReduce 流的接口更接近 Python，更便于使用。

```
src/main/ch02/python/max_temperature_map.py | \
sort | src/main/ch02/python/max_temperature_reduce.py
1949 111
1950 22
```

2.6 Hadoop 管道

Hadoop 管道是 Hadoop MapReduce 的 C++接口的代称。与流不同，流使用标准输入和输出让 map 和 reduce 节点之间相互交流，管道使用 sockets 作为 tasktracker 与 C++编写的 map 或者 reduce 函数的进程之间的通道。JNI 未被使用。

我们将用 C++重写贯穿本章的示例，然后，我们将看到如何使用管道来运行它。例 2-12 显示了用 C++语言编写的 map 函数和 reduce 函数的源代码。

例 2-12：用 C++语言编写的最高气温程序

```
#include <algorithm>
#include <limits>
#include <string>

#include "hadoop/Pipes.hh"
#include "hadoop/TemplateFactory.hh"
#include "hadoop/StringUtils.hh"

class MaxTemperatureMapper : public HadoopPipes::Mapper {
public:
  MaxTemperatureMapper(HadoopPipes::TaskContext& context) {
  }
  void map(HadoopPipes::MapContext& context) {
    std::string line = context.getInputValue();
    std::string year = line.substr(15, 4);
    std::string airTemperature = line.substr(87, 5);
    std::string q = line.substr(92, 1);
    if (airTemperature != "+9999" &&
        (q == "0" || q == "1" || q == "4" || q == "5" || q == "9")) {
      context.emit(year, airTemperature);
    }
  }
};
```

```
class MapTemperatureReducer : public HadoopPipes::Reducer {
public:
  MapTemperatureReducer(HadoopPipes::TaskContext& context) {
  }
  void reduce(HadoopPipes::ReduceContext& context) {
    int maxValue = INT_MIN;
    while (context.nextValue()) {
      maxValue = std::max(maxValue,
        HadoopUtils::toInt(context.getInputValue()));
    }
    context.emit(context.getInputKey(),
      HadoopUtils::toString(maxValue));
  }
};

int main(int argc, char *argv[]) {
  return HadoopPipes::runTask(HadoopPipes::TemplateFactory
                      <MaxTemperatureMapper,MapTemperatureReducer>());
}
```

此应用程序连接到 Hadoop C++库，后者是一个用于与 tasktracker 子进程进行通信的轻量级的封装器。通过扩展在 HadoopPipes 命名空间的 Mapper 和 Reducer 类且提供 map()和 reduce()方法的实现，我们便可定义 map 和 reduce 函数。这些方法采用了一个上下文对象(MapContext 类型或 ReduceContext 类型)，后者提供读取输入和写入输出，通过 JobConf 类来访问作业配置信息。本例中的处理过程非常类似于 Java 的处理方式。

与 Java 接口不同，C++接口中的键和值是字节缓冲，表示为标准模板库(Standard Template Library，STL)的字符串。这使接口变得更简单，尽管它把更重的负担留给了应用程序的开发人员，因为开发人员必须将字符串 convert to and from 表示 to 和 from 两个逆操作。开发人员必须在字符串及其他类型之间进行转换。这一点在 MapTemperatureReducer 中十分明显，其中，我们必须把输入值转换为整数的输入值(使用 HadoopUtils 中的便利方法)，在最大值被写出之前将其转换为字符串。在某些情况下，我们可以省略这个转化，如在 MaxTemperatureMapper 中，它的 airTemperature 值从来不用转换为整数，因为它在 map()方法中从来不会被当作数字来处理。

main()方法是应用程序的入口点。它调用 HadoopPipes::runTask，连接到从 Mapper 或 Reducer 连接到 Java 的父进程和 marshals 数据。runTask()方法被传入一个 Factory 参数，使其可以创建 Mapper 或 Reducer 的实例。它创建的其中

一个将受套接字连接中 Java 父进程控制。我们可以用重载模板 factory 方法来设置一个 combiner(combiner)、partitioner(partitioner)、记录读取函数(record reader)或记录写入函数(record writer)。

编译运行

现在我们可以用 Makerfile 编译连接例 2-13 的程序。

例 2-13：C++版本的 MapReduce 程序的 Makefile

```
CC = g++
CPPFLAGS = -m32 -I$(HADOOP_INSTALL)/c++/$(PLATFORM)/include

max_temperature: max_temperature.cpp
    $(CC) $(CPPFLAGS) $< -Wall
    -L$(HADOOP_INSTALL)/c++/$(PLATFORM)/lib -lhadooppipes \
    -lhadooputils -lpthread -g -O2 -o $@
```

在 Makefile 中应当设置许多环境变量。除了 HADOOP_INSTALL(如果遵循附录 A 的安装说明，应该已经设置好)，还需要定义平台，指定操作系统、体系结构和数据模型(例如，32 位或 64 位)。我在 32 位 Linux 系统的机器编译运行了如下内容：

```
% export PLATFORM=Linux-i386-32
% make
```

在成功完成之前，当前目录中便有 max_temperature 可执行文件。

要运行管道作业，我们需要在伪分布式(pseudo_distrinuted)模式下(其中所有守护进程运行在本地计算机上)运行 Hadoop，其中的安装步骤参见附录 A。管道不在独立模式(本地运行)中运行，因为它依赖于 Hadoop 的分布式缓存机制，仅在 HDFS 运行时才运行。

Hadoop 守护进程开始运行后，第一步是把可执行文件复制到 HDFS，以便它们启动 map 和 reduce 任务时，它能够被 tasktracker 取出：

```
% hadoop fs -put max_temperature bin/max_temperature
```

示例数据也需要从本地文件系统复制到 HDFS：

```
% hadoop fs -put input/ncdc/sample.txt sample.txt
```

现在可以运行这个作业。为了使其运行，我们用 Hadoop 管道命令，使用-program 参数来传递在 HDFS 中可执行文件的 URI。

```
% hadoop pipes\
  -D hadoop.pipes.java.recordreader\
  -D hadoop.pipes.java.recordwriter\
  inpit sample.txt\
  output output
  program bin/max_temperature
```

我们使用-D 选项来指定两个属性：`hadoop.pipes.java.recordreader` 和 `hadoop.pipes.java.recordwriter`，这两个属性都被设置为 true，表示我们并没有指定一个 C+++记录读取函数或记录写入函数，但我们要使用默认的 Java 设置(用来设置文本输入和输出)。管道还允许你设置一个 Java mapper，reducer，combiner 或 partitioner。事实上，在任何一个作业中，都可以混合使用 Java 类或 C++类。

结果和用其他语言编写的程序所得的结果一样。

第3章

Hadoop 分布式文件系统

当数据集超过一个单独的物理计算机的存储能力时，便有必要将它分布到多个独立的计算机。管理着跨计算机网络存储的文件系统称为分布式文件系统。因为它们是基于网络的，所有网络编程的复杂性都会随之而来，所以分布式文件系统比普通磁盘文件系统更复杂。举例来说，使这个文件系统能容忍节点故障而不损失数据就是一个极大的挑战。

Hadoop 有一个被称为 HDFS 的分布式系统，全称为 Hadoop Distributed Filesystem。(有时可能简称为 DFS，在非正式情况或是文档和配置中，其实是一样的。) HDFS 是 Hadoop 的旗舰级文件系统，也是本章的重点，但 Hadoop 实际上有一个综合性的文件系统抽象，因而接下来我们将看看 Hadoop 如何与其他文件系统集成(如本地文件系统或 Amazon S3)。

3.1 HDFS 的设计

HDFS 是为以流式数据访问模式存储超大文件而设计的文件系统，在商用硬件的集群上运行。让我们仔细看看下面的明。

超大文件

“超大文件”在这里指几百 MB，几百 GB 甚至几百 TB 大小的文件。目前已经有 Hadoop 集群存储 PB(petabytes)级的数据了。[1]

---

① 在 Yahoo!，Hadoop 已经扩充到 4000 个节点，参见以下网址：*http://developer.yahoo.net/blogs/hadoop/2008/09/scaling-hadoop_to_4000_nodes_a.html*

流式数据访问

HDFS 建立在这样一个思想上：一次写入、多次读取模式是最高效的。一个数据集通常由数据源生成或复制，接着在此基础上进行各种各样的分析。每个分析至少都会涉及数据集中的大部分数据（甚至全部），因此读取整个数据集的时间比读取第一条记录的延迟更为重要。

商用硬件

Hadoop 不需要运行在昂贵并且高可靠性的硬件上。它被设计运行在商用硬件(在各种零售店都能买到的普通硬件)的集群上，因此至少对于大的集群来说，节点故障的几率还是较高的。HDFS 在面对这种故障时，被设计为能够继续运行而让用户察觉不到明显的中断。

同时，那些并不适合 HDFS 的应用也是值得研究的。在目前，HDFS 还不太适用于某些领域，不过日后可能会有所改进。

低延迟数据访问

需要低延迟访问数据在毫秒范围内的应用并不适合 HDFS。HDFS 是为达到高数据吞吐量而优化的，这有可能会以延迟为代价。目前，对于低延迟访问，HBase(参见第 12 章)是更好的选择。

大量的小文件

名称节点(namenode)存储着文件系统的元数据，因此文件数量的限制也由名称节点的内存量决定。根据经验，每个文件，索引目录以及块占大约 150 个字节。因此，举例来说，如果有一百万个文件，每个文件占一个块，就至少需要 300 MB 的内存。虽然存储上百万的文件是可行的，十亿或更多的文件就超出目前硬件的能力了。

多用户写入，任意修改文件

HDFS 中的文件只有一个写入者，而且写操作总是在文件的末尾。它不支持多个写入者，或是在文件的任意位置修改。(可能在以后这些会被支持，但它们也相对不那么高效。)

3.2 HDFS 的概念

3.2.1 块

一个磁盘有它的块大小，代表着它能够读写的最小数据量。文件系统通过处理大小

为一个磁盘块大小的整数倍数的数据块来运作这个磁盘。文件系统块一般为几千字节，而磁盘块一般为 512 个字节。这些信息，对于仅仅在一个文件上读或写任意长度的文件系统用户来说是透明的。但是，有些工具会维护文件系统，如 df 和 fsck， 它们都在系统块级上操作。

HDFS 也有块的概念，不过是更大的单元，默认为 64 MB。与单一磁盘上的文件系统相似，HDFS 上的文件也被分为以块为大小的分块，作为单独的单元存储。但与其不同的是，HDFS 中小于一个块大小的文件不会占据整个块的空间。如果没有特殊指出，“块”在本书中就指代 HDFS 中的块。

为何 HDFS 中的一个块那么大？

HDFS 的块比磁盘的块大，目的是为了减小寻址开销。通过让一个块足够大，从磁盘转移数据的时间能够远远大于定位这个块开始端的时间。因此，传送一个由多个块组成的文件的时间就取决于磁盘传输送率。

我们来做一个速算，如果寻址时间在 10 毫秒左右，传输速率是 100 兆/秒，为了使寻址时间为传输时间的 1%，我们需要 100 MB 左右的块大小。而默认的大小实际为 64 MB，尽管很多 HDFS 设置使用 128 MB 的块。这一数字将在以后随着新一代磁盘驱动带来的传输速度加快而继续调整。

当然这种假定不应该如此夸张。MapReduce 过程中的 map 任务通常是在一个时间内运行操作一个块，因此如果任务数过于少(少于集群上的节点数量)，作业的运行速度显然就比预期的慢。

在分布式文件系统中使用抽象块会带来很多好处。第一个最明显的好处是，一个文件可以大于网络中任意一个磁盘的容量。文件的分块(block，后文有些地方也简称为“块”)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。其实，虽然不常见，但对于 HDFS 集群而言，也可以存储一个其分块占满集群中所有磁盘的文件。

第二个好处是，使用块抽象单元而不是文件会简化存储子系统。简单化是所有系统的追求，但对于故障种类繁多的分布式系统来说尤为重要的。存储子系统控制的是块，简化了存储管理。(因为块的大小固定，计算一个磁盘能存多少块就相对容易)，也消除了对元数据的顾虑(块只是一部分存储的数据—而文件的元数据，如许可信息，不需要与块一同存储，这样一来，其他系统就可以正交地管理元数据。)

不仅如此，块很适合于为提供容错和实用性而做的复制操作。为了应对损坏的块以及磁盘或机器的故障，每个块都在少数其他分散的机器(一般为 3 个)进行复制。如果一个块损坏了，系统会在其他地方读取另一个副本，而这个过程是对用户透明的。一个因损坏或机器故障而丢失的块会从其他候选地点复制到正常运行的机器上，以保证副本的数量回到正常水平。(参见第 4 章的“数据的完整性”小节，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热门的文件块设置更高的副本数量以提高集群的读取负载量。

与磁盘文件系统相似，HDFS 中 fsck 指令会显示块的信息。例如，执行以下命令将列出文件系统中组成各个文件的块(参见第 10 章的“文件系统查看(fsck)”小节)：

```
% hadoop fsck / -files -blocks
```

3.2.2 名称节点与数据节点

HDFS 集群有两种节点，以管理者-工作者的模式运行，即一个名称节点(管理者)和多个数据节点(工作者)。名称节点管理文件系统的命名空间。它维护着这个文件系统树及这个树内所有的文件和索引目录。这些信息以两种形式将文件永久保存在本地磁盘上：命名空间镜像和编辑日志。名称节点也记录着每个文件的每个块所在的数据节点，但它并不永久保存块的位置，因为这些信息会在系统启动时由数据节点重建。

客户端代表用户通过与名称节点和数据节点交互来访问整个文件系统。客户端提供一个类似 POSIX(可移植操作系统界面)的文件系统接口，因此用户在编程时并不需要知道名称节点和数据节点及其功能。

数据节点是文件系统的工作者。它们存储并提供定位块的服务(被用户或名称节点调用时)，并且定时的向名称节点发送它们存储的块的列表。

没有名称节点，文件系统将无法使用。事实上，如果运行名称节点的机器被毁坏了，文件系统上所有的文件都会丢失，因为我们无法知道如何通过数据节点上的块来重建文件。因此，名称节点能够经受故障是非常重要的，Hadoop 提供了两种机制来确保这一点。

第一种机制就是复制那些组成文件系统元数据持久状态的文件。Hadoop 可以通过配置使名称节点在多个文件系统上写入其持久化状态。这些写操作是具同步性和原子性的。一般的配置选择是，在本地磁盘上写入的同时，写入一个远程 NFS 挂载(mount)。

另一种可行的方法是运行一个二级名称节点，虽然它不能作为名称节点使用。这个

二级名称节点的重要作用就是定期的通过编辑日志合并命名空间镜像，以防止编辑日志过大。这个二级名称节点一般在其他单独的物理计算机上运行，因为它也需要占用大量 CPU 和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，在名称节点失效后就可以使用。但是，二级名称节点的状态是比主节点滞后的，所以主节点的数据若全部丢失，损失仍在所难免。在这种情况下，一般把存在 NFS 上的主名称节点元数据复制到二级名称节点上并将其作为新的主名称节点运行。

详情请参见第 10 章的“文件系统镜像与编辑日志”小节。

3.3 命令行接口

现在我们将通过命令行与 HDFS 交互。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。

我们将在一台机器上运行 HDFS，所以请先参照附录 A 中在伪分布模式下设置 Hadoop 的说明。稍后将介绍如何在集群上运行 HDFS 从而为我们提供伸缩性与容错性。

在我们设置伪分布配置时，有两个属性需要进一步解释。首先是 `fs.default.name`，设置为 hdfs://localhost/，用来为 Hadoop 设置默认文件系统。文件系统是由 URI 指定的，这里我们已使用了一个 `hdfs` URI 来配置 HDFS 为 Hadoop 的默认文件系统。HDFS 的守护程序将通过这个属性来决定 HDFS 名称节点的宿主机和端口。我们将在 `localhost` 上运行，默认端口为 8020。这样一来，HDFS 用户将通过这个属性得知名称节点在哪里运行以便于连接到它。

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制 3 份。在单独一个数据节点上运行时，HDFS 无法将块复制到 3 个数据节点上，所以会持续警告块的副本不够。此设置可以解决这个问题。

基本文件系统操作

文件系统已经就绪，我们可以执行所有其他文件系统都有的操作，例如，读取文件，创建目录，移动文件，删除数据，列出索引目录，等等。输入 `hadoop fs -help` 命令即可看到所有命令详细的帮助文件。

首先将本地文件系统的一个文件复制到 HDFS：

```
% hadoopfs -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`，提供一系列的子命令。在这里，我们执行的是`-copyFromLocal`。本地文件 quangle.txt 被复制到运行在 localhost 上的 HDFS 实体中的/user/tom/quangle.txt 文件。其实我们可以省略 URI 的格式与主机而选择默认设置，即省略 `hdfs://localhost`，就像 core-site.xml 中指定的那样。

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

也可以使用相对路径，并将文件复制到 home 目录，即/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，看看是否一样：

```
% hadoop fs -copyToLocal quangle.txt quangle.copy.txt
% md5 input/docs/quangle.txt quangle.copy.txt
MD5 (input/docs/quangle.txt) = a16f231da6b05e2ba7a339320e7dacd9
MD5 (quangle.copy.txt) = a16f231da6b05e2ba7a339320e7dacd9
```

MD5 分析结果是一样的，表明这个文件在 HDFS 之旅中得以幸存并完整。

最后，我们看一下 HDFS 文件列表。我们创建一个目录来看看它在列表中如何显示：

```
% hadoop fs -mkdir books
% hadoop fs -ls .
Found 2 items
drwxr-xr-x   - tom supergroup          0 2009-04-02 22:41
/user/tom/books
-rw-r--r--   1 tom supergroup        118 2009-04-02 22:29
/user/tom/quangle.txt
```

返回的信息结果与 Unix 命令 `ls -l` 的输出非常相似，仅有细微差别。第一列显示的是文件格式。第二列是这个文件的副本数(这在 Unix 文件系统是没有的)。由于我们设置的默认副本数在网站范围内为 1，所以这里显示的也都是 1。这一列的开头目录栏是空的，因为副本的概念并没有应用——目录是作为元数据并存在名称节点中的，而非数据节点。第三列和第四列显示文件的所属用户和组别。第五列是文件的大小，以字节显示，目录大小为 0。第六列和第七列是文件的最后修改日期与时间。最后的第八列是文件或目录的绝对路径。

HDFS 中的文件许可

HDFS 对于文件及目录有与 POSIX 非常相似的许可模式。

共有三种形式的许可：读取许可(r)、写入许可(w)和执行许可(x)。读取文件或列出目录内容时需要读取许可。写入一个文件，或是在一个目录上创建或删除文件或目录，需要写入许可。对于文件而言执行许可可以忽略因为 HDFS 中不能执行文件(与 POSIX 不同)，但在访问一个目录的子项时是需要的。

每个文件和目录都有一个所属用户、所属组别和模式。这个模式是由所属用户的许可、组内其他成员的许可及其他用户的许可组成。

客户端的标识是通过它正在运行的进程的 username(名称)和 groups(组别)来确定的。由于客户端是远程的，任何人都可以简单地在远程系统上创建一个账户来进行访问。因此，许可只能在一个合作的团体中的用户中使用，作为共享文件系统资源和防止数据意外损失的机制，而不能在一个敌意的环境中保护资源。但是，除去这些缺点，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，使用许可还是值得的(这也是默认的配置，参见 `dfs.permissions` 属性)。

如果启用了许可检查，所属用户许可与组别许可都会被检查，以确认用户的用户名与所属用户许可是否相同，确认他是否属于此用户组的成员；若不符，则检查其他许可。

这里有一个超级用户的概念，超级用户是名称节点进程的标识。对于超级用户，系统不会执行任何许可检查。

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` 展示了 Hadoop 的一个文件系统，而且有几个具体实现，如表 3-1 所示。

文件系统	URI 方案	Java 实现(全部在 org.apache.hadoop)	描述
Local	file	fs.LocalFileSystem	针对有客户端校验和的本地连接磁盘使用的文件系统。针对没有校验和的本地文件系统使用 RawLocalFileSystem。详情参见第 4 章
HDFS	hdfs	hdfs.Distributed-FileSystem	Hadoop 的分布式文件系统。HDFS 被设计为结合使用 Map-Reduce 实现高效工作
HFTP	hftp	hdfs.HftpFileSystem	一个在 HTTP 上提供对 HDFS 只读访问的文件系统(虽然其名称为 HFTP，但它与 FTP 无关)。通常与 distcp 结合使用(参见第 3 章)，在运行不同版本 HDFS 的集群间复制数据
HSFTP	hsftp	hdfs.Hsftp-FileSystem	在 HTTPS 上提供对 HDFS 只读访问的文件系统(同上，与 FTP 无关)
HAR	har	fs.HarFileSystem	一个构建在其他文件系统上来存档文件的文件系统。Hadoop 存档一般在 HDFS 中的文件存档时使用，以减少名称节点内存的使用
KFS(Cloud-Store)	kfs	fs.kfs.Kosmos-FleSystem	cloudstore(其前身是 Kosmos 文件系统)是相似于 HDFS 或是 Google 的 GFS 的文件系统，用 C++ 编写。详情可参见 *http://kosmosfs.sourceforge.net/*
FTP	ftp	fs.ftp.FTP-FileSystem	由 FTP 服务器支持的文件系统
S3(本地)	s3n	fs.s3native.Native-S3FileSystem.	由 Amazon S3 支持的文件系统。可参见 *http://wiki.apache.org/hadoop/AmazonS3*
S3(基于块)	s3	fs.s3.S3FileSystem	由 Amazon S3 支持的文件系统，以块格式存储文件(与 HDFS 很相似)来解决 S3 的 5 GB 文件大小限制

Hadoop 提供了许多文件系统的接口，它一般使用 URI 方案来选取合适的文件系统实例交互。举例来说，我们在前一小节中研究的文件系统 shell 可以操作所有的

Hadoop 文件系统。列出本地文件系统根目录下的文件，输入以下命令：

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

尽管运行那些可访问任何文件系统的 MapReduce 程序是可行的(有时也很方便)，但在处理大量数据时，仍然需要选择一个有最优本地数据的分布式文件系统，如 HDFS 或者 KFS(参见第 1 章)。

接口

Hadoop 是用 Java 编写的，所有 Hadoop 文件系统间的相互作用都是由 Java API 调解的。[1]举个例子，文件系统的 shell 就是一个 Java 应用，它使用 Java 文件系统类来提供文件系统操作。这些接口在 HDFS 中被广泛应用，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(FTP 的 FTP 客户，S3 的 S3 工具等)，但它们大多数都能和任意一个 Hadoop 文件系统协作。

Thrift

因为 Hadoop 的文件系统接口是 Java API，所以其他非 Java 应用访问 Hadoop 文件系统会比较麻烦。在“Thriftfs”分类单元中的 Thrift API 通过将 Hadoop 文件系统展示为一个 Apache Thrift 服务来弥补这个不足，使得任何有 Thrift 绑定的语言都能轻松地与 Hadoop 文件系统互动，如 HDFS。

使用 Thrift API，需要运行提供 Thrift 服务的 Java 服务器，以代理的方式访问 Hadoop 文件系统。你的应用程序在访问 Thrift 服务时，后者实际上就和它运行在同一台机器上。

Thrift API 包含很多其他语言的预生成 stub，包含 C++，Perl，PHP，Python 及 Ruby。Thrift 支持不同版本，因此我们可以从同一个客户代码中访问不同版本的 Hadoop 文件系统(不过必须运行针对每个版本的代理)。

关于安装与使用教程，请参阅 src/contrib/thriftfs 目录中关于 Hadoop 分布的文档。

---

[1] Hadoop 的 RPC 接口是以 Hadoop Writable 接口为基础的，这个 Writable 接口是面向 Java 的。在将来，Hadoop 将会采用其他跨语言的 RPC 序列化格式，使本地 HDFS 客户可由其他非 Java 的语言编写。

C 语言库

Hadoop 提供了反映 Java 文件系统接口的名为 libhdfs 的 C 语言库(它被编写为一个访问 HDFS 的 C 语言库，但其实可以访问任意 Hadoop 文件系统)。它会使用 Java 本地接口(JNI)调用一个 Java 文件系统客户。

C API 与 Java 的非常相似，但它一般比 Java 的滞后，因此目前还不支持一些新特征。相关资料可参见 libhdfs/docs/api 目录中关于 Hadoop 分布的 C API 文档。

Hadoop 中有预先建好的 32 位 Linux 的 libhdfs 二元码，但对于其他平台，需要使用 *http://wiki.apache.org/hadoop/LibHDFS* 的教程自己编写。

FUSE

用户空间文件系统(Filesystem in Userspace，FUSE)允许一些文件系统整合为一个 Unix 文件系统在用户空间中执行。通过使用 Hadoop 的 Fuse-DFS 分类模块，任意一个 Hadoop 文件系统(不过一般为 HDFS)都可以作为一个标准文件系统进行挂载。我们随后便可以使用 Unix 的工具(如 `ls` 和 `cat`)与这个文件系统交互，还可以通过任意一种编程语言使用 POSIX 库来访问文件系统。

Fuse-DFS 是用 C 语言实现的，使用 libhdfs 作为与 HDFS 的接口。要想了解如何编译和运行 Fuse-DFS，可参见 src/contrib./fuse-dfs 中的 Hadoop 分布目录。

WebDAV

WebDAV 是一系列支持编辑和更新文件的 HTTP 的扩展。在大部分操作系统中，WebDAV 共享都可以作为文件系统进行挂载，因此借由 WebDAV 来向外提供 HDFS(或其他 Hadoop 文件系统)，可以将 HDFS 作为一个标准文件系统进行访问。

在本书写作期间，Hadoop 中的 WebDAV 支持(通过对 Hadoop 调用 Java API 来实现)仍在开发中，要想了解最新动态，可访问 *https://issues,apache.org/jira/browse/HADOOP-496*。

其他 HDFS 接口

对于 HDFS 有两种特定的接口。

HTTP

HDFS 定义了一个只读接口用来在 HTTP 上检索目录列表和数据。目录列表由

名称节点的嵌入式 Web 服务器(运行在 50070 端口)以 XML 格式提供服务，文件数据由数据节点通过它们的 Web 服务器 (运行在 50075 端口)传输。这个协议并不拘泥于某个 HDFS 版本，因此用户可以编写使用 HTTP 从运行不同版本 Hadoop 的 HDFS 集群中读取数据的客户端应用。HftpFileSystem 就是其中一种：一个通过 HTTP 与 HDFS 交流的 Hadoop 文件系统(HsftpFileSystem 是 HTTPS 的变体)。

FTP

尽管本书写作期间尚未完成(*https://issues.apache.org/jira/browse/HADOOP-3199*)，但我们还是要提一下，还有一个对 HDFS 的 FTP 接口，它允许使用 FTP 协议与 HDFS 交互。这个接口很方便，它使用现有 FTP 客户端与 HDFS 进行数据的传输。

对 HDFS 的 FTP 接口与 FTPFileSystem 不可混为一谈，此接口的目的是将任意 FTP 服务器向外暴露为 Hadoop 文件系统。

3.5 Java 接口

在本小节，我们要深入探索 Hadoop 的 `Filesystem` 类：与 Hadoop 的文件系统交互的 API。[①]虽然我们主要关注的是 HDFS 的实现 `DistributedFileSystem`，但总体来说，还是应该努力编写不同于 `FileSsytem` 抽象类的代码，以保持其在不同文件系统中的可移植性。这是考验编程能力的最佳手段，因为我们很快就可以使用存储在本地文件系统中的数据来运行测试了。

3.5.1 从 Hadoop URL 中读取数据

要从 Hadoop 文件系统中读取文件，一个最简单的方法是使用 `java.net.URL` 对象来打开一个数据流，从而从中读取数据。一般的格式如下：

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

① 有一个关于 Java 改进文件系统接口的 Java 规范需求(JSR 203，“NIO.2” *http://jcp.org/en/jsr/detail?id=203*)，即以提供可插式文件系统实现为目标。从长远来看，这个新接口(提名将在 Java 7 中出现)有可能替代 Hadoop 的文件系统抽象。

```
    } finally {
        IOUtils.closeStream(in);
    }
```

这里还需要一点工作来让 Java 识别 Hadoop 文件系统的 URL 方案，就是通过一个 FsUrlStreamHandlerFactory 实例来调用在 URL 中的 setURLStreamHandler-Factory 方法。这种方法在一个 Java 虚拟机中只能被调用一次，因此一般都在一个静态块中执行。这个限制意味着如果程序的其他部件(可能是不在你控制中的第三方部件)设置一个 URLStreamHandlerFactory，我们便无法再从 Hadoop 中读取数据。下一节将讨论另一种方法。

例 3-1 展示了以标准输出显示 Hadoop 文件系统的文件的程序，它类似于 Unix 的 cat 命令。

例 3-1：用 URLStreamHandler 以标准输出格式显示 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);
    }
  }
}
```

我们使用 Hadoop 中简洁的 IOUtils 类在 finally 子句中关闭数据流，同时复制输入流和输出流之间的字节(本例中是 System.out)。copyBytes 方法的最后两个参数，前者是要复制的缓冲的大小，后者表示复制结束后是否关闭数据流。这里是将输入流关掉了，而 System.out 不需要关闭。

下面是一个运行示例：[1]

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

① 引自 Edward Lear 的 *The Quangle Wangle's 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.
```

3.5.2 使用 FileSystem API 读取数据

如前一小节所解释的，有时不能在应用中设置 `URLStreamHandlerFactory`。这时，我们需要用 `FileSystem API` 来打开一个文件的输入流。

文件在 Hadoop 文件系统中显示为一个 Hadoop Path 对象(不是一个 java.io.File 对象，因为它的语义与本地文件系统关联太紧密)。我们可以把一个路径视为一个 Hadoop 文件系统 URI，如 *hdfs://localhost/user/tom/quangle.txt*。

`FileSystem` 是一个普通的文件系统 API，所以首要任务是检索我们要用的文件系统实例，这里是 HDFS。取得 FileSystem 实例有两种静态工厂方法：

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

Configuration 对象封装了一个客户端或服务器的配置，这是用从类路径读取而来的配置文件(如 *conf/core-site.xml*)来设置的。第一个方法返回的是默认文件系统(在 *conf/core-site.xml* 中设置的，如果没有设置过，则是默认的本地文件系统)。第二个方法使用指定的 URI 方案及决定所用文件系统的权限，如果指定 URI 中没有指定方案，则退回默认的文件系统。

有了 `FileSystem` 实例后，我们调用 `open()` 来得到一个文件的输入流：

```
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 4 KB 的缓冲大小。

将它们合在一起，我们可以在例 3-2 中重写例 3-1。

例 3-2：直接使用 FileSystem 以标准输出格式显示 Hadoop 文件系统的文件

```
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

FileSystem 中的 open()方法实际上返回的是一个 FSDataInputStream，而不是标准的 java.io 类。这个类是 java.io.DataInputStream 的一个子类，支持随机访问，这样就可以从流的任意位置读取数据了。

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

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

Seekable 接口允许在文件中定位，并提供一个查询方法，用于查询当前位置相对于文件开始处的偏移量(getPos())：

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

调用 seek()来定位大于文件长度的位置会导致 IOException 异常。与

java.io.InputStream 中的 skip()不同，seek()并没有指出数据流当前位置之后的一点，它可以移到文件中任意一个绝对位置。

应用程序开发人员并不常用 seekToNewSource()方法。此方法一般倾向于切换到数据的另一个副本并在新的副本中寻找 targetPos 指定的位置。HDFS 内部就采用这样的方法在数据节点故障时为客户端提供可靠的数据输入流。

例 3-3 是例 3-2 的简单扩展，它将一个文件两次写入标准输出：在写一次后，定位到文件的开头再次读入数据流。

例 3-3：通过使用 seek 两次以标准输出格式显示 Hadoop 文件系统的文件

```
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得到以下结果：

```
% 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 也实现了 PositionedReadable 接口，从一个指定位置读取

一部分数据：

```
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()方法从指定 position 读取指定长度的字节放入缓冲 buffer 的指定偏离量 offset。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长度。readFully()方法会读出指定字节由 length 指定的数据到 buffer 中或在只接受 buffer 字节数组的版本中，再读取 buffer.length 字节(这儿指的是第三个函数)，若已经到文件末，将会抛出 EOFException。

所有这些方法会保留文件当前位置并且是线程安全的，因此它们提供了在读取文件(可能是元数据)的主要部分时访问其他部分的便利方法。其实，这只是使用 Seekable 接口的实现，格式如下：

```
long oldPos = getPos();
try {
  seek(position);
  // read data
} finally {
  seek(oldPos);
}
```

最后务必牢记，seek()是一个相对高开销的操作，需要慎重使用。我们需要依靠流数据构建应用访问模式(如使用 MapReduce)，而不要大量执行 seek 操作。

3.5.3 写入数据

FileSystem 类有一系列创建文件的方法。最简单的是给拟创建的文件指定一个路径对象，然后返回一个用来写的输出流：

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

这个方法有重载的版本允许我们指定是否强制覆盖已有的文件、文件副本数量、写入文件时的缓冲大小、文件块大小以及文件许可。

注意：create()方法为需要写入的文件而创建的父目录可能原先并不存在。虽然这样很方便，但有时并不希望这样。如果我们想在父目录不存在时不执行写入，就必须在调用exists()首先检查父目录是否存在。

还有一个用于传递回调接口的重载方法 Progressable，如此一来，我们所写的应用就会被告知数据写入数据节点的进度：

```
package org.apache.hadoop.util;
public interface Progressable {
      public void progress();
}
```

新建文件的另一种方法是使用 append()在一个已有文件中追加(也有一些其他重载版本)：

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

这个操作允许一个写入者打开已有文件并在其末尾写入数据。有了这个 API，会产生无边界文件的应用，以日志文件为例，就可以在重启后在已有文件上继续写入。此添加操作是可选的，并不是所有 Hadoop 文件系统都有实现。HDFS 支持添加，但 S3 就不支持了。

例 3-4 展示了如何将本地文件复制到 Hadoop 文件系统。我们在每次 Hadoop 调用 progress()方法时，也就是在每 64 KB 数据包写入数据节点管道后打印一个句号来展示整个过程。(注意，这个动作并不是 API 指定的，因此在 Hadoop 后面的版本中大多被改变了。API 仅仅是让我们注意到“发生了一些事”。)

例 3-4：将本地文件复制到 Hadoop 文件系统并显示进度

```
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);
  }
}
```

典型用途：

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

目前，其他 Hadoop 文件系统在写入时都不会调用 `progress()`。通过后面几章的描述，我们会感到进度之于 MapReduce 应用的重要性。

FSDataOutputStream

`FileSystem` 中的 `create()` 方法返回了一个 `FSDataOutputStream`，与 `FSDataInputStream` 类似，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
}
```

但是，与 `FSDataInputStream` 不同，`FSDataOutputStream` 不允许定位。这是因为 HDFS 只允许对一个打开的文件顺序写入，或向一个已有文件添加。换句话说，它不支持除文件尾部的其他位置的写入，这样一来，写入时的定位就没有什么意义。

3.5.4 目录

fIlesystem 提供了一个创建目录的方法：

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

这个方法会创建所有那些必要但不存在的父目录，就像 java.io.File 的 mkdirs()。如果目录(以及所有父目录)都创建成功，它会返回 true。

我们常常不需要很确切地创建一个目录，因为调用 create()写入文件时会自动生成所有的父目录。

3.5.5 查询文件系统

文件元数据：Filestatus

任何文件系统的一个重要特征是定位其目录结构及检索其存储的文件和目录信息的能力。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、副本、修改时间、所有者以及许可信息。

FileSystem 的 getFileStatus()提供了获取一个文件或目录的状态对象的方法。例 3-5 展示了它的用法。

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(conf, 1, true, null);
    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.isDir(), is(false));
    assertThat(stat.getLen(), is(7L));
    assertThat(stat.getModificationTime(),
        is(lessThanOrEqualTo(System.currentTimeMillis())));
    assertThat(stat.getReplication(), is((short) 1));
    assertThat(stat.getBlockSize(), is(64 * 1024 * 1024L));
    assertThat(stat.getOwner(), is("tom"));
    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.isDir(), 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("tom"));
```

```
    assertThat(stat.getGroup(), is("supergroup"));
    assertThat(stat.getPermission().toString(), is("rwxr-xr-x"));
  }

}
```

如果文件或目录不存在，即会抛出 FileNotFoundException 异常。如果你只对文件或目录是否存在有兴趣，exists()方法会更方便：

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

列出文件

查找一个文件或目录的信息很实用，但有时我们还需要能够列出目录的内容。这就是 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
```

传入参数是一个文件时，它会简单地返回长度为 1 的 FileStatus 对象的一个数组。当传入参数是一个目录时，它会返回 0 或者多个 FileStatus 对象，代表着此目录所包含的文件和目录。

重载方法允许我们使用 PathFilter 来限制匹配的文件和目录，示例参见后文。如果把路径数组作为参数来调用 listStatus 方法，其结果是与依次对每个路径调用此方法，再将 FileStatus 对象数组收集在一个单一数组中的结果是相同的，但是前者更为方便。这在建立从文件系统树的不同部分执行的输入文件的列表时很有用。例 3-6 是这种思想的简单示范。注意 FIleUtil 中 stat2Paths()的使用，它将一个 FileStatus 对象数组转换为 Path 对象数组。

例 3-6：显示一个 Hadoop 文件系统中一些路径的文件信息

```
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);
    }
  }
}
```

我们可以使用这个程序得到一个路径集的整个目录列表。

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

文件格式

在一步操作中处理批量文件，这个要求很常见。举例来说，处理日志的MapReduce作业可能会分析一个月的文件，这些文件被包含在大量目录中。Hadoop有一个通配的操作，可以方便地使用通配符在一个表达式中核对多个文件，不需要列举每个文件和目录来指定输入。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()`返回了其路径匹配于所供格式的`FileStatus`对象数组，按路径排序。可选的`PathFilter`命令可以进一步指定限制匹配。

Hadoop支持的一系列通配符与Unix bash相同(见表3-2)。

表3-2：通配符及其作用

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]	字符类别	匹配{a,b}中的一个字符

续表

通配符	名称	匹配
[^ab]	非字符类别	匹配不是{a,b}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 中的一个的语句
\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，便可以假设 2007 年最后一天的日志文件就会以/2007/12/31 的命名存入目录。假设整个文件列表如下：

- /2007/12/30
- /2007/12/31
- /2008/01/01
- /2008/01/02

以下是一些文件通配符及其扩展。

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

PathFilter 对象

通配格式不是总能够精确地描述我们想要访问的文件集合。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，使我们能够通过编程方式控制匹配：

```
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。

例 3-7 展示了一个 PathFilter，用于排除匹配一个正则表达式的路径。

例 3-7：一个 PathFilter，用于排除匹配一个正则表达式的路径

```
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);
  }
}
```

这个过滤器只留下与正则表达式不同的文件。我们将它与预先剔除一些文件集合的通配配合：过滤器用来优化结果。例如：

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

就会扩大到/2007/12/30。

过滤器只能作用于 Path 表示的文件名。它们无法使用文件的属性(如创建时间)作为过滤依据。不过，它们可以实现通配格式和正则表达式无法做到的匹配。比如，如果我们按日期设计目录结构(如上节所述)，这样我们就能写一个路径过滤来挑出日期在一定范围内的文件。

3.5.6 删除数据

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

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

如果传入的 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，一个非空目录以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 文件读取剖析

为了了解客户端及与之交互的 HDFS、名称节点和数据节点之间的数据流是怎样的，我们可参考图 3-1，其中显示了在读取文件时一些事件的主要顺序。

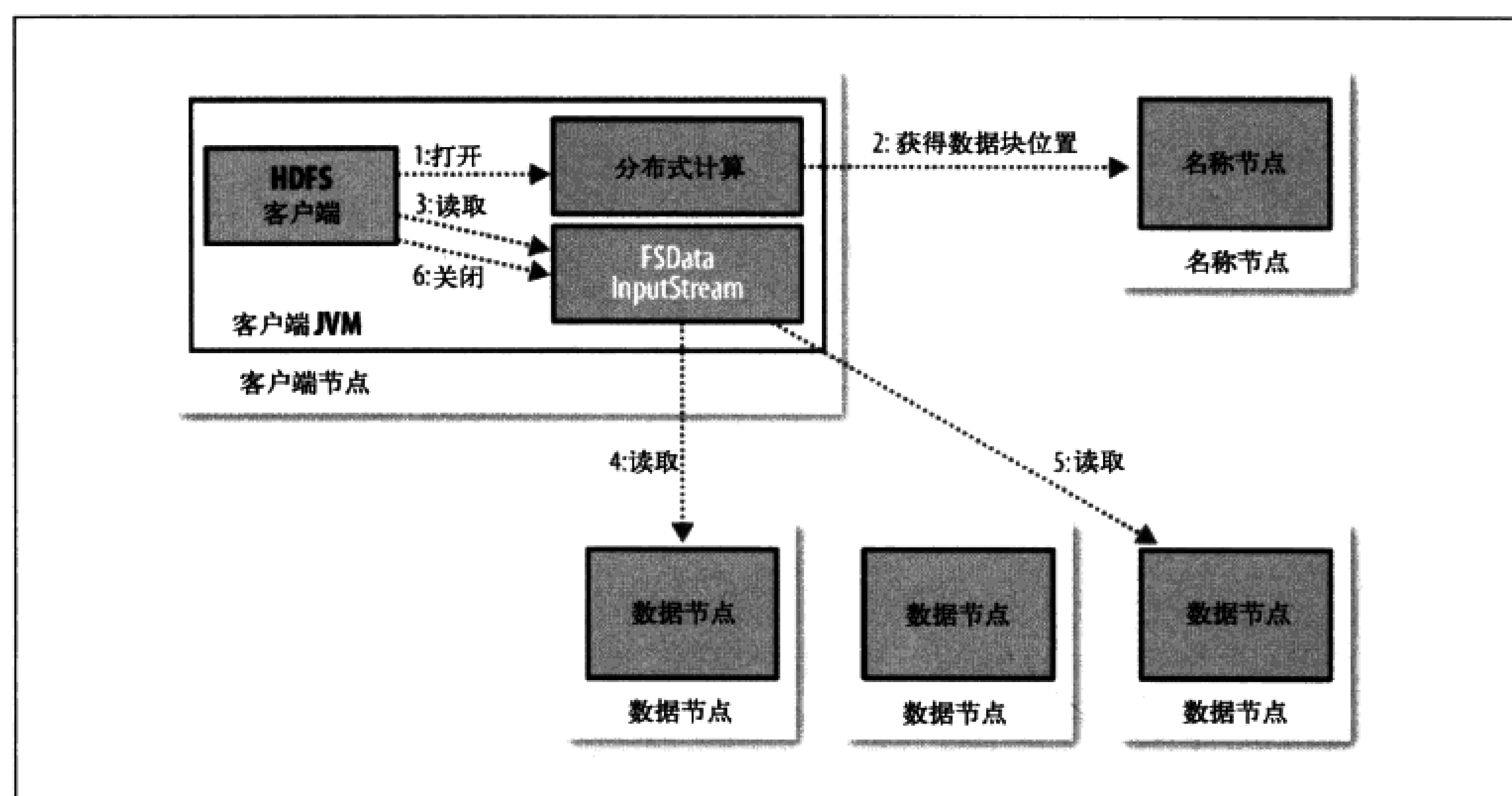


图 3-1：客户端从 HDFS 中读取数据

客户端通过调用 FileSystem 对象的 open()来读取希望打开的文件，对于 HDFS 来说，这个对象是分布式文件系统(图 3-1 中的步骤 1)的一个实例。DistributedFileystem 通过使用 RPC 来调用名称节点，以确定文件开头部分的块的位置(步骤 2)。对于每一个块，名称节点返回具有该块副本的数据节点地址。此外，这些数据节点根据它们与客户端的距离来排序(根据网络集群的拓扑；参见后文补充材料“网络拓扑与 Hadoop”)。如果该客户端本身就是一个数据节点(比如在一个 MapReduce 任务中)，便从本地数据节点中读取。

DistributedFilesystem 返回一个 FSDataInputStream 对象(一个支持文件定位的输入流)给客户端读取数据。FSDataInputStream 转而包装了一个 DFSInputStream 对象。

接着，客户端对这个输入流调用 read()(步骤 3)。存储着文件开头部分的块的数据节点地址的 DFSInputStream 随即与这些块最近的数据节点相连接。通过在数据流中重复调用 read()，数据会从数据节点返回客户端(步骤 4)。到达块的末端时，DFSInputStream 会关闭与数据节点间的联系，然后为下一个块找到最佳的数据节点(步骤 5)。客户端只需要读取一个连续的流，这些对于客户端来说都是透明的。

客户端从流中读取数据时，块是按照 DFSInputStream 打开与数据节点的新连接的顺序读取的。它也会调用名称节点来检索下一组需要的块的数据节点的位置。一旦客户端完成读取，就对文件系统数据输入流调用 close()(步骤 6)。

在读取的时候，如果客户端在与数据节点通信时遇到一个错误，那么它就会去尝试对这个块来说下一个最近的块。它也会记住那个故障的数据节点，以保证不会再对之后的块进行徒劳无益的尝试。客户端也会确认从数据节点发来的数据的校验和。如果发现一个损坏的块，它就会在客户端试图从别的数据节点中读取一个块的副本之前报告给名称节点。

这个设计的一个重点是，客户端直接联系数据节点去检索数据，并被名称节点指引到每个块中最好的数据节点。因为数据流动在此集群中是在所有数据节点分散进行的，所以这种设计能使 HDFS 可扩展到最大的并发客户端数量。同时，名称节点只不过是提供块位置请求(存储在内存中，因而非常高效)，不是提供数据。否则如果客户端数量增长，名称节点会快速成为一个“瓶颈”。

网络拓扑与 Hadoop

两个节点在一个本地网络中被称为“彼此的近邻”是什么意思？在高容量数据处理中，限制因素是我们在节点间传送数据的速率——带宽很稀缺。这个想法便是将两个节点间的带宽作为距离的衡量标准。

衡量节点间的带宽，实际上很难实现(它需要一个稳定的集群，并且在集群中成对的节点的数量的增长要是节点数量的平方)，不及 Hadoop 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距离它们最近的共同祖先的总和。该树中的等级是没有被预先设定的，但是它对于相当于数据中心、框架和一直在运

行的节点的等级是共同的。这个想法是，对于以下每个场景，可用带宽依次减少：

- 相同节点中的进程
- 同一机架上的不同节点
- 同一数据中心的不同机架上的节点
- 不同数据中心的节点[①]

例如，假设节点 n1 在数据中心 d1 中的机架 r1 上。这被表示成/d1/r1/n1。利用这种标记，这里给出四种描述的距离：

- 距离(/d1/r1/n1, /d1/r1/n1)=0(相同节点中的进程)
- 距离(/d1/r1/n1, /d1/r1/n2)=2(同一机架上的不同节点)
- 距离(/d1/r1/n1, /d1/r2/n3)=4(同一数据中心的不同机架上的节点)
- 距离(/d1/r1/n1, /d2/r3/n4)=6(不同数据中心的节点)

这在图 3-2 中用图示形式表达(数学爱好者会注意到这是一个距离公制的例子)。

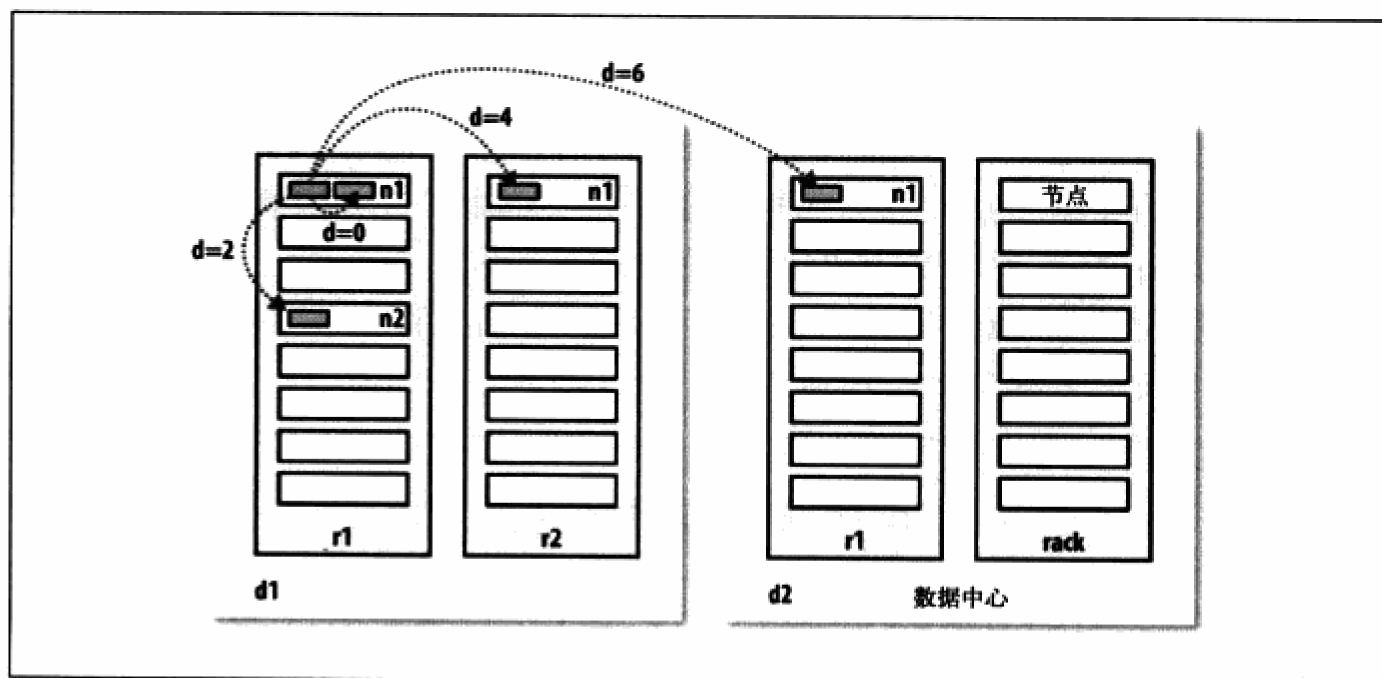


图 3-2：Hadoop 中的网络距离

我们必须意识到，Hadoop 无法预测网络拓扑结构。它需要一定帮助，我们将在第 9 章讨论如何配置拓扑。不过在默认情况下，假设网络是平的(一个单层的等级制)，或者换句话说，所有节点都在同一数据中心的同一机架。小的集群可能如此，所以不需要进一步的配置。

① 在本书写作期间，Hadoop 尚不适合跨数据中心运行。

3.6.2 文件写入剖析

接下来我们要看文件是如何写入 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 的连贯模型(coherency model)。

我们考虑的情况是创建一个新的文件，向其写入数据后关闭该文件。参见图 3-3。

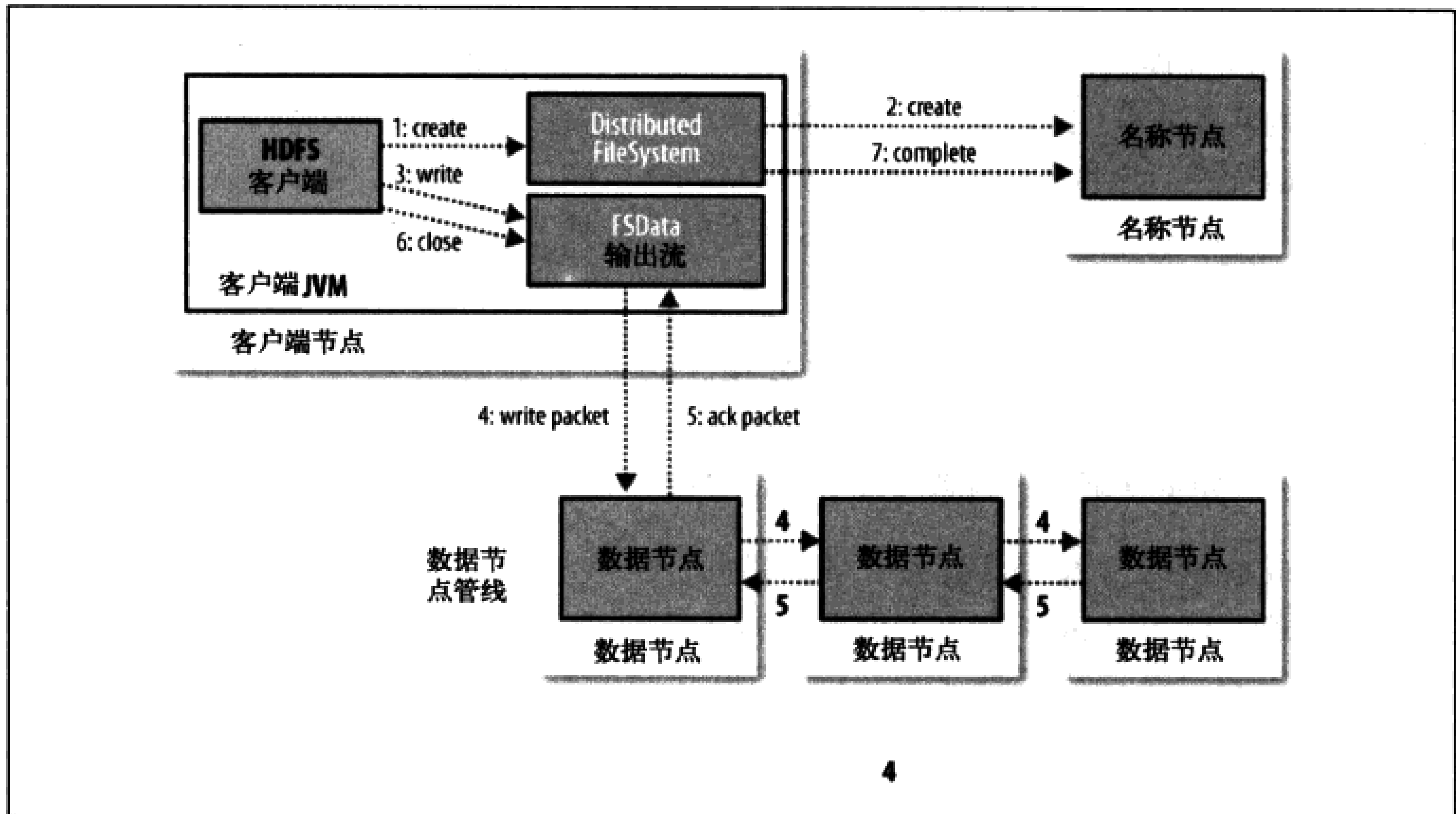


图 3-3：客户端对 HDFS 写入数据

客户端通过在 `DistributedFilesystem` 中调用 `create()`来创建文件(图 3-3 的步骤 1)。`DistributedFilesystem` 一个 RPC 去调用名称节点，在文件系统的命名空间中创建一个新的文件，没有块与之相联系(步骤 2)。名称节点执行各种不同的检查以确保这个文件不会已经存在，并且客户端有可以创建文件的适当的许可。如果这些检查通过，名称节点就会生成一个新文件的记录；否则，文件创建失败并向客户端抛出一个 `IOException` 异常。分布式文件系统返回一个文件系统数据输出流，让客户端开始写入数据。就像读取事件一样，文件系统数据输出流控制一个 `DFSoutPutstream`，负责处理数据节点和名称节点之间的通信。

在客户端写入数据时(步骤 3)，`DFSoutPutstream` 将它分成一个个的包，写入内部的队列，称为数据队列。数据队列随数据流流动，数据流的责任是根据适合的数据节点的列表来要求这些节点为副本分配新的块。这个数据节点的列表形成一个管线——我们假设这个副本数是 3，所以有 3 个节点在管线中。数据流将包分流给管线

中第一个的数据节点，这个节点会存储包并且发送给管线中的第二个数据节点。同样地，第二个数据节点存储包并且传给管线中第三个(也是最后一个)数据节点(步骤 4)。

`DFSoutputStream` 也有一个内部的包队列来等待数据节点收到确认，称为确认队列。一个包只有在被管线中所有节点确认后才会被移出确认队列(步骤 5)。

如果在有数据写入期间，数据节点发生故障，则会执行下面的操作，当然这对写入数据的客户端而言，是透明的。首先管线被关闭，确认队列中的任何包都会被添加回数据队列的前面，以确保数据节点从失败的节点处是顺流的，不会漏掉任意一个包。当前的块在正常工作的数据节点中被给予一个新的身份并联系名称节点，以便能在故障数据节点后期恢复时其中的部分数据块会被删除。故障数据节点会从管线中删除并且余下块的数据会被写入管线中的两个好的数据节点。名称节点注意到块副本不足时，会在另一个节点上安排创建一个副本。随后，后续的块会继续正常处理。

在一个块被写入期间多个数据节点发生故障的可能性虽然有但很少见。只要 `dfs.replication.min` 的副本(默认为 1)被写入，写操作就是成功的，并且这个块会在集群中被异步复制，直到满足其目标副本数(`dfs.replication` 的默认设置为 3)。

客户端完成数据的写入后，就会在流中调用 `close()`(步骤 6)。在向名称节点发送完信息之前，此方法会将余下的所有包放入数据节点管线并等待确认(步骤 7)。名称节点已经知道文件由哪些块组成(通过 `Data streamer` 询问块分配)，所以它只需在返回成功前等待块进行最小量的复制。

副本的放置

名称节点如何选择哪个数据节点来保存副本？我们需要在可靠性与写入带宽和读取带宽之间进行权衡。例如，因为副本管线都在单独一个节点上运行，所以把所有副本都放在一个节点基本上不会损失写入带宽，但这并没有实现真的冗余(如果节点发生故障，那么该块中的数据会丢失)。同样，离架读取的带宽是很高的。另一个极端，把副本放在不同的数据中心会最大限度地增大冗余，但会以带宽为代价。即使在相同的数据中心(所有的 Hadoop 集群到目前为止都运行在同一个数据中心)，也有许多不同的放置策略。其实，Hadoop 在发布的 0.17.0 版中改变了放置策略来帮助保持块在集群间有相对平均的分布(第 10 章详细说明了如何保持集群的平衡)。

Hadoop 的策略是在与客户端相同的节点上放置第一个副本(若客户端运行在

集群之外，就可以随机选择节点，不过系统会避免挑选那些太满或太忙的节点)。

第二个副本被放置在与第一个不同的随机选择的机架上(离架)。第三个副本被放置在与第二个相同的机架上，但放在不同的节点。更多的副本被放置在集群中的随机节点上，不过系统会尽量避免在相同的机架上放置太多的副本。

一旦选定副本放置的位置，就会生成一个管线，会考虑到网络拓扑。副本数为 3 的管道看起来如图 3-4 所示。

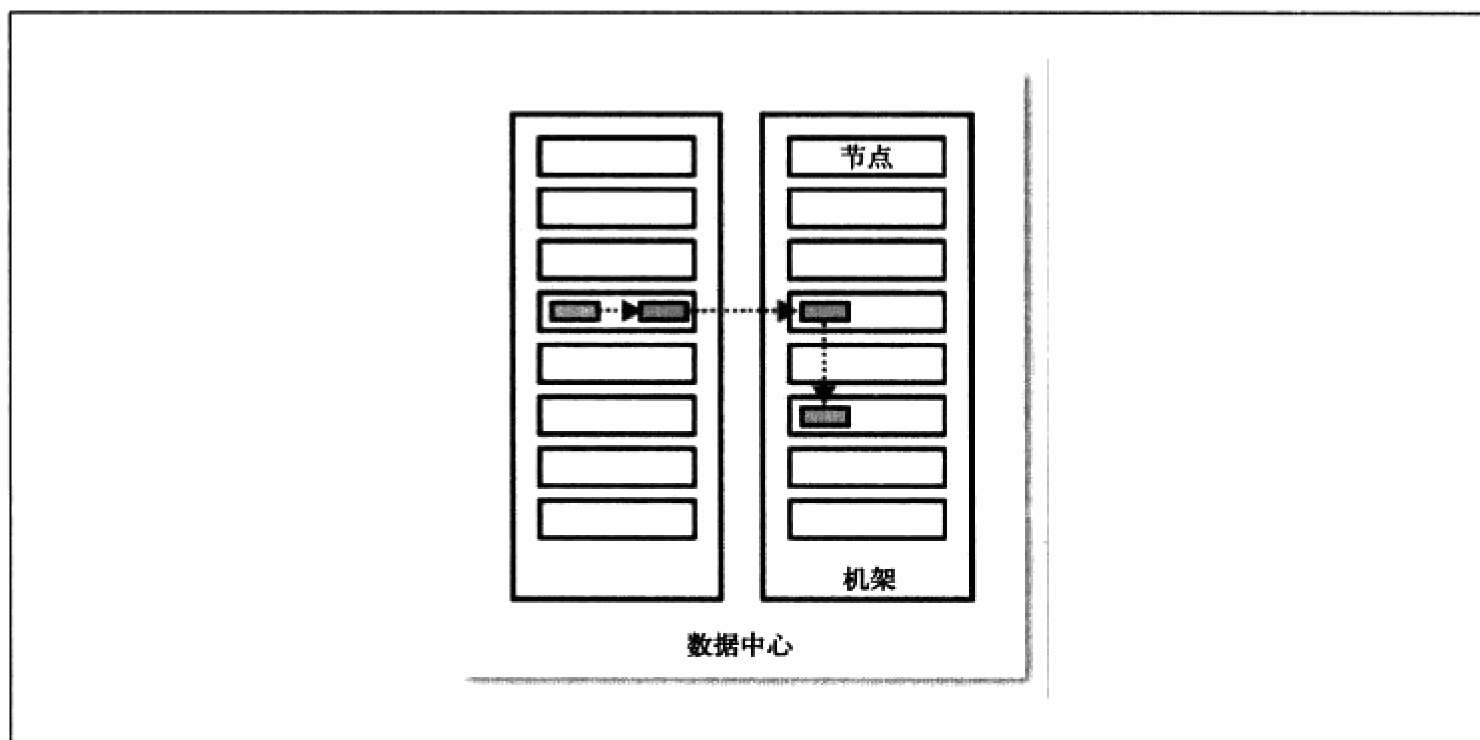


图 3-4：一个典型的副本管线

总的来说，这样的方法在稳定性(块存储在两个机架中)、写入带宽(写入操作只需要做一个单一网络转换)、读取性能(选择从两个机架中进行读取)和集群中块的分布(客户端只在本地机架写入一个块)之间，进行了较好的平衡。

3.6.3 一致模型

文件系统的一致模型描述了对文件读写的数据可见性。HDFS 为性能牺牲了一些 POSIX 请求，因此一些操作可能比想像的困难。

在创建一个文件之后，在文件系统的命名空间中是可见的，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被看见，即使数据流已经被刷新。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));
```

一旦写入的数据超过一个块的数据，新的读取者就能看见第一个块。对于之后的块也是这样。总之，它始终是当前正在被写入的块，其他读取者是看不见它的。

HDFS 提供一个方法来强制所有的缓存与数据节点同步，即在文件系统数据输出流使用 sync()方法。在 sync()返回成功后，HDFS 能保证文件中直至写入的最后的数据对所有新的读取者而言，都是可见且一致的。万一发生冲突(与客户端或 HDFS)，也不会造成数据丢失：

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

此行为类似于 Unix 中的 fsync 系统调用——为一个文件描述符提交缓冲数据。例如，利用 Java API 写入一个本地文件，我们肯定能够看到刷新流和同步之后的内容：

```
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 中关闭一个文件其实还执行了一个隐含的 sync()：

```
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())));
```

应用设计的重要性

这个一致模型与具体设计应用程序的方法有关。如果不调用 sync()，那么一旦客户端或系统发生故障，就可能失去一个块的数据。对很多应用来说，这是不可接受的，所以我们应该在适当的地方调用 sync()，例如在写入一定的记录或字节之后。尽管 sync()操作被设计为尽量减少 HDFS 负载，但它仍然有开销，所以在数据健壮性和吞吐量之间就会有所取舍。应用依赖就比较能接受，通过不同的 sync()频率来衡量应用程序，最终找到一个合适的平衡。

3.7 通过 distcp 进行并行复制

前面的 HDFS 访问模型都集中于单线程的访问。例如通过指定文件通配，我们可以对一部分文件进行处理，但是为了高效，对这些文件的并行处理需要新写一个程序。Hadoop 有一个叫 distcp(分布式复制)的有用程序，能从 Hadoop 的文件系统并行复制大量数据。

distcp 一般用于在两个 HDFS 集群中传输数据。如果集群在 Hadoop 的同一版本上运行，就适合使用 hdfs 方案：

```
% hadoop distcp hdfs://namenode1/foo hdfs://namenode2/bar
```

这将从第一个集群中复制/foo 目录(和它的内容)到第二个集群中的/bar 目录下，所以第二个集群会有/bar/foo 目录结构。如果/bar 不存在，则新建一个。我们可以指定多个源路径，并且所有的都会被复制到目标路径。源路径必须是绝对路径。

默认情况下，distcp 会跳过目标路径已经有的文件，但可以通过提供的-overwrite 选项进行覆盖。也可以用-update 选项来选择只更新那些修改过的文件。

注意：使用-overwrite 和-update 中任意一个(或两个)选项会改变源路径和目标路径的含义。这可以用一个例子清楚说明。如果改变先前例子中第一个集群的子树/foo 下的一个文件，就能通过运行对第二个集群的改变进行同步：

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

目标路径需要末尾这个额外的子目录/foo，因为源目录下的内容已被复制到目标目录下。(如果熟悉 rsync，你可以想像-overwrite 或-update 项对源路径而言，如同添加一个隐含的斜杠。)

如果对 discp 操作不是很确定，最好先对一个小的测试目录树进行尝试。

有很多选项可以控制分布式复制行为，包括预留文件属性，忽略故障和限制复制的文件或总数据的数量。运行时不带任何选项，可以看到使用说明。

distcp 是作为一个 MapReduce 作业执行的，复制工作由集群中并行运行的 map 来完成。这里并没有 reducer。每个文件都由一个单一的 map 进行复制，并且 distcp 通过将文件分成大致相等的文件来为每个 map 数量大致相同的数据。

map 的数量是这样确定的。通过让每一个 map 复制数量合理的数据以最小化任务建立所涉及的开销，是一个很好的想法，所以每个 map 的副本至少为 256 MB。(除非输入的总大小较少，否则一个 map 就足以操控全局。)例如，1 GB 的文件会被分成 4 个 map 任务。如果数据很大，为限制带宽和集群的使用而限制映射的数量就变得很有必要。map 默认的最大数量是每个集群节点(tasktracker)有 20 个。例如，复制 1000 GB 的文件到一个 100 个节点的集群，会分配 2000 个 map(每个节点 20 个 map)，所以平均每个会复制 512 MB。通过对 distcp 指定`-m` 参数，会减少映射的分配数量。例如，`-m 1000` 会分配 1000 个 map，平均每个复制 1 GB。

如果想在两个运行着不同版本 HDFS 的集群上利用 distcp，使用 hdfs 协议是会失败的，因为 RPC 系统是不兼容的。想要弥补这种情况，可以使用基于 HTTP 的 HFTP 文件系统从源中进行读取。这个作业必须运行在目标集群上，使得 HDFS RPC 版本是兼容的。使用 HFTP 重复前面的例子：

```
% hadoop distcp hftp://namenode1:50070/foo hdfs://namenode2/bar
```

注意，需要在 URI 源中指定名称节点的 Web 端口。这是由 dfs.http.address 的属性决定的，默认值为 50070。

保持 HDFS 集群的平衡

向 HDFS 复制数据时，考虑集群的平衡相当重要。文件块在集群中均匀地分布时，HDFS 能达到最佳工作状态。回顾前面 1000 GB 数据的例子，通过指定-m 选项为 1，即由一个单一的 map 执行复制工作，它的意思是，不考虑速度变慢和未充分利用集群资源，每个块的第一个副本会存储在运行 map 的节点上(直到磁盘被填满)。第二和第三个副本分散在集群中，但这一个节点并不会平衡。通过让 map 的数量多于集群中节点的数量，我们便可避免这个问题。鉴于此，最好首先就用默认的每个节点 20 个 map 这个默认设置来运行 distcp。

然而，这也并不总能阻止一个集群变得不平衡。也许想限制 map 的数量以便一些节点可以被其他作业使用。若是这样，可以使用 balancer 工具(参见第 10 章)继续

改善集群中块的分布。

3.8 Hadoop 归档文件

每个文件以块方式存储，块的元数据存储在名称节点的内存里，此时存储一些小的文件，HDFS 会较低效。因此，大量的小文件会耗尽名称节点的大部分内存。(注意，相较于存储文件原始内容所需要的磁盘空间，小文件所需要的空间不会更多。例如，一个 1 MB 的文件以大小为 128 MB 的块存储，使用的是 1 MB 的磁盘空间，而不是 128 MB。)

Hadoop Archives 或 HAR 文件，是一个更高效的将文件放入 HDFS 块中的文件存档设备，在减少名称节点内存使用的同时，仍然允许对文件进行透明的访问。具体说来，Hadoop Archives 可以被用作 MapReduce 的输入。

3.8.1 使用 Hadoop Archives

Hadoop Archives 通过使用 archive 工具根据一个文件集合创建而来。这些工具运行一个 MapReduce 作业来并行处理输入文件，因此我们需要一个 MapReduce 集群去运行使用它。HDFS 中有一些我们希望归档的文件：

```
% hadoop fs -lsr /my/files
-rw-r--r--   1 tom supergroup     1 2009-04-09 19:13
/my/files/a
drwxr-xr-x   - tom supergroup     0 2009-04-09 19:13
/my/files/dir
-rw-r--r--   1 tom supergroup     1 2009-04-09 19:13
/my/files/dir/b
```

现在我们可以运行 archive 指令：

```
% hadoop archive -archiveName files.har /my/files /my
```

第一个选项是归档文件名称，这里是 file.har。HAR 文件总是有一个.har 扩展名，这是必需的，具体理由见后文描述。接下来把文件放入归档文件。这里我们只归档一个源树，即 HDFS 下/my/files 中的文件，但事实上，该工具接受多个源树。最后一个参数是 HAR 文件的输出目录。让我们看看这个归档文件是怎么创建的：

```
% hadoop fs -ls /my
Found 2 items
```

```
drwxr-xr-x   - tom supergroup      0 2009-04-09 19:13 /my/files
drwxr-xr-x   - tom supergroup      0 2009-04-09 19:13
/my/files.har
% hadoop fs -ls /my/files.har
Found 3 items
-rw-r--r--  10 tom supergroup    165 2009-04-09 19:13
/my/files.har/_index
-rw-r--r--  10 tom supergroup     23 2009-04-09 19:13
/my/files.har/_masterindex
-rw-r--r--   1 tom supergroup      2 2009-04-09 19:13
/my/files.har/part-0
```

这个目录列表展示了一个 HAR 文件的组成部分：两个索引文件和部分文件的集合(本例中只有一个)。这些部分文件包含已经链接在一起的大量原始文件的内容，并且索引使我们可以查找那些包含归档文件的部分文件，包括它的起始点和长度。但所有这些细节对于使用 har URI 方案与 HAR 文件交互的应用都是隐藏的，HAR 文件系统是建立在基础文件系统上的(本例中是 HDFS)。以下命令以递归方式列出了归档文件中的文件：

```
% hadoop fs -lsr har:///my/files.har
drw-r--r--   - tom supergroup      0 2009-04-09 19:13
/my/files.har/my
drw-r--r--   - tom supergroup      0 2009-04-09 19:13
/my/files.har/my/files
-rw-r--r--  10 tom supergroup      1 2009-04-09 19:13
/my/files.har/my/files/a
drw-r--r--   - tom supergroup      0 2009-04-09 19:13
/my/files.har/my/files/dir
-rw-r--r--  10 tom supergroup      1 2009-04-09 19:13
/my/files.har/my/files/dir/b
```

如果 HAR 文件所在的文件系统是默认的文件系统，这就非常直观易懂。但如果想使用在其他文件系统中的 HAR 文件，就需要使用一个不同于正常情况的 URI 路径格式。以下两个指令作用相同，例如：

```
% hadoop fs -lsr har:///my/files.har/my/files/dir
% hadoop fs -lsr har://hdfs-
localhost:8020/my/files.har/my/files/dir
```

注意第二个格式，仍以 har 方案表示一个 HAR 文件系统，但是是由 hdfs 指定基础的文件系统方案，后面加上一个横杠和 HDFS host(localhost)和端口(8020)。我们现在算是明白为什么 HAR 文件必须要有.har 扩展名了。通过查看权限和路径及.har

扩展名的组成，HAR 文件系统将 har URI 转换成为一个基础文件系统的 URI。在本例中是 hdfs://localhost:8020/user/tom/files.har。路径的剩余部分是文件在归档文件中的路径：/user/tom/files/dir。

要想删除一个 HAR 文件，需要使用删除的递归格式，因为对于基础文件系统来说，HAR 文件是一个目录。

```
% hadoop fs -rmr /my/files.har
```

3.8.2 不足

对于 HAR 文件，还需要了解它的一些不足。创建一个归档文件会创建原始文件的一个副本，因此需要与要归档(尽管创建了归档文件后可以删除原始文件)的文件同样大小的磁盘空间。虽然归档的文件能被压缩(HAR 文件在这方面像 tar 文件)，但是目前还不支持档案压缩。

一旦创建，Archives 便不可改变。要增加或移除文件，必须重新创建归档文件。事实上，这对那些写后便不能改的文件来说不是问题，因为它们可以定期成批归档，比如每日或每周。

如前所述，HAR 文件可以用作 MapReduce 的输入。然而，没有归档 InputFormat 可以打包多个文件到一个单一的 MapReduce，所以即使在 HAR 文件中处理许多小文件，也仍然低效的。第 7 章讨论了解决此问题的另一种方法。

第 4 章

Hadoop 的 I/O

Hadoop 附带了一个基本类型的数据集的 I/O。其中某些技术比 Hadoop 本身更具普遍意义，例如数据的完整性和压缩，当然在处理 MB 级数据集的时候需要特殊考虑。其他还有 Hadoop 工具或 API，可以用于开发分布式系统时的模块构建，如序列化框架和磁盘数据结构。

4.1 数据完整性

Hadoop 的用户当然希望数据在存储或处理过程中不会丢失和损坏。但是，由于每个磁盘或者网络上的 I/O 操作可能会对正在读写的数据不慎引入错误，如果通过的数据流量非常大，数据发生损坏的几率是很高的。

检测损坏数据的常用方法是在第一次进入系统时计算数据的校验和，然后只要数据是在一个不可靠的通道上传输，就可能会被损坏。如果新生成的校验和不完全匹配原始的校验和，那么数据就会被认为是被损坏了。这项技术不提供任何方式来修复数据，仅仅是错误检测。(这也是不使用低端硬件的原因之一。具体说来，一定要使用 ECC 内存。)这里要注意一点，校验和可能有错，数据却是正确的，但这种可能性不大，因为校验和远小于数据。

一个常用的错误检测代码是 CRC-32(cyclic redundancy check，循环冗余检查)，计算一个 32 位的任何大小输入的整数校验和。

4.1.1 HDFS 的数据完整性

HDFS 以透明方式校验所有写入它的数据，并在默认设置下，会在读取数据时验证校验和。针对数据的每个 io.bytes.per.checksum 字节，都会创建一个单独的校验和。默认值为 512 字节，因为 CRC-32 校验和是 4 字节长，存储开销小于 1%。

数据节点负责在存储数据及其校验和前验证它们收到的数据。这适用于它们从客户端和其他数据节点复制过来得到的数据。客户端写入数据并且将它发送到一个数据节点的管线中(详细说明请参见第 3 章)，在管线上的最后一个数据节点验证校验和。如果此节点检测到错误，客户端便会收到一个 Checksum Exception，这是 IOException 的一个子类。

客户端读取数据节点上的数据时，会验证校验和，将其与数据节点上存储的校验和进行对比。每个数据节点维护一个连续的校验和验证日志，因此它知道每个数据块最后验证的时间。客户端成功验证数据块之后，便会告诉数据节点，后者便随之更新日志。保持这种统计，它对检测损坏磁盘是很有价值的。

除了对客户端读取数据进行验证，每个数据节点还会在后台线程运行一个 DataBlockScanner(数据块检测程序)，定期验证存储在数据节点上的所有块。这是为了防止物理存储介质中位衰减所造成的数据损坏。要想进一步了解如何访问 DataBlockScanner 生成的报告，请参见第 10 章。

由于 HDFS 存储着块的副本，它可以通过复制完整的副本来产生一个新的，无错的副本来“治愈”那些出错的数据块。它的工作方式是，如果一客户端读取数据块时检测到错误，它在抛出 Checksum Exception 前报告该坏块以及它试图从名称节点中要读取的数据节点。名称节点将这个块标记为损坏的，因此它不会直接复制给客户端，或复制此副本到另一个数据节点。它会从其他的副本复制一个新的副本，这样它的副本数就会回归到预期的数量。 一旦出现这种情况，损坏的副本将被删除。

我们可以在使用 open()方法来读取文件前，通过将 false 传给 FilesyStem 中的 setVerifyChecksum()方法来禁用校验和验证。我们可以使用 shell 命令来达到同样的效果，如在-get 或其等效的-copyToLocal 命令中使用-ignoreCrc 选项。此功能非常有用，如果有损坏的文件要检查，就可以决定如何处理。例如，在删除它之前看看是否可以挽救。

4.1.2 本地文件系统

Hadoop 的本地文件系统执行客户端校验。这意味着，在写一个名为 filename 的文件时，文件系统的客户端以透明方式创建了一个隐藏的文件.filename.crc，在同一个文件夹下包含每个文件块的校验和。像 HDFS，该块的大小是受 io.bytes.per.checksum 属性控制的，默认为 512 字节。块的大小作为元数据存储在.crc 文件中，因此即使块大小设置改变仍然可以正确读回该文件。读取文件过程中会对校验和进行验证，而且如果检测到错误，本地文件系统会抛出 ChecksumException。

校验和是代价相当小的计算(在 Java 中，它们在本地代码中实现)，通常只增加百分之几的时间开销来读取或写入文件。对于大多数应用程序，这是保持数据完整性可以接受的代价。然而，我们也可能禁用校验和：这里的情况是底层文件系统原生支持校验和。这是通过使用 RawLocalFileSystem 来代替 LocalFileSystem 完成的。要在一个应用中全局使用，只需通过设置属性 fs.file.impl 值为 org.apache.hadoop.fs.RawLocalFileSystem 来重新 map 执行文件的 URI。或者，如果只是想对某些读取禁用校验和检验，可以直接创建一个 RawLocalFileSystem 实例，例如：

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

4.1.3 ChecksumFileSystem

LocalFileSystem 使用 ChecksumFileSystem(校验和文件系统)为自己工作，这个类可以很容易添加校验和功能其他(无校验和)文件系统，因为 CheckSumFileSystem 也包含于文件系统中。一般语句如下：

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

底层文件系统被称为原始文件系统，可使用通过在 ChecksumFileSystem 中使用 getRawFileSystem()方法来获得。ChecksumFileSystem 有几个非常有用的方法来使用校验和，比如，getChecksumFile()可以为任何文件找到校验和文件的路径。检查其他文件的记录。

读取文件时，如果 ChecksumFileSystem 检测到错误，便会调用其 report-

`ChecksumFailure()`方法。默认执行什么事情也不做，但 `LocalFileSystem` 会将问题文件和其校验和移动到同一设备的次级目录下，名为 bad_files。管理员应定期检查这些坏文件并采取相应措施。

4.2 压缩

文件压缩带来了两大好处：它减少了存储文件所需的空间且加快了数据在网络上或者从磁盘上或到磁盘上的传输速度。在处理大容量的数据时，这些都是很重要的，因此对于在 Hadoop 中如何使用压缩，值得仔细考虑。

有许多不同的压缩格式、工具和算法，每一个都有不同的特性。表 4-1 列出了一些常见的可以用于 Hadoop 的。[①]

表 4-1：压缩格式小结

压缩格式	工具	算法	文件扩展名	多文件	可分割性
DEFLATE*	无	DEFLATE	.deflate	不	不
Gzip	gzip	DEFLATE	.gz	不	不
ZIP	zip	DEFLATE	.zip	是	是，在文件范围内
bzip2	bzip2	bzip2	.bz2	不	是
LZO	lzop	LZO	.lzo	不	不

* DEFLATE 是一种压缩算法，它的实现标准是 zlib。没有一个可普遍使用的命令行工具用于生成 DEFLATE 格式的文件，但 gzip 压缩却是普遍使用的。(注意， gzip 文件格式压缩是有额外的头部和尾部的。)Hadoop 使用的是.deflate 的文件扩展名。

所有的压缩算法都显示出一种时间空间的权衡：更快的压缩和解压速度通常会耗费更多的空间。表 4-1 列出的所有选项有利于我们对此平衡进行调整，这是通过 9 种不同的选项来实现的：-1 表示速度最优，-9 表示空间最优。例如，下面的命令使用最快压缩方法创建一个压缩文件 file.gz：

```
gzip -1 file
```

不同的工具有非常不同的压缩特性。gzip 和 ZIP 是通用的压缩工具，在空间/时间处理上相对平衡。bzip2 压缩比 gzip 或 ZIP 更有效，但速度较慢。bzip2 压缩的解

① 在我写作的时候，Hadoop 的 ZIP 压缩仍未完成，可参见 *https://issues.apache.org/jira/browse/HADOOP-1824*。

压速度快于它的压缩速度，但仍比其他格式慢。另一方面，LZO 使用速度最优算法，所以它比 gzip 或 ZIP 更快(或者其他压缩解压工具[①])，但压缩效率稍低。

表 4-1 的“可分割性”这一列表示压缩格式是否支持分割，也就是说，你是否可以要求寻找到数据流中的任何一个点并且从某些点上开始读取。可分割压缩格式特别适合 MapReduce。

4.2.1 编码/解码器

编码/解码器用以执行压缩解压算法。在 Hadoop 里，编码/解码器是通过一个压缩解码器接口实现的。因此，例如，`GzipCodec` 封装了 gzip 压缩的压缩和解压算法。表 4-2 列出了 Hadoop 可用的编码/解码器。

表 4-2：Hadoop 压缩编码/解码器

压缩格式	Hadoop 压缩编码/解码器
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

LZO 格式是基于 GPL 许可的，不能通过 Apache 来分发许可，基于此，它的 hadoop 编码/解码器必须单独下载，地址是 *http://code.google.com/p/hadoop-gpl-compression/*。lzop 编码/解码器兼容于 lzop 工具，它其实就是 LZO 格式，但额外还有头部，它正是我们想要的。还有一个纯 LZO 格式的编码/解码器 `LzoCodec`，它使用.lzo_deflate 作为扩展名(根据 DEFLATE 类推，是没有头部的 gzip 格式)。

CompressionCodec 对流进行压缩和解压缩

`CompressionCodec` 有两个方法可以用于轻松地压缩或解压缩数据。要想对正在被写入一个输出流的数据进行压缩，我们可以使用 `createOutputStream (OutputStreamout)`方法创建一个 `CompressionOutputStream`(未压缩的数据将被写到此)，将其以压缩格式写入底层的流。相反，要想对从输入流读取而来的数据进行解压缩，则调用 `createInputStream(InputStream in)`函数，从而获得一个

① Jeff 的实验比较报告(*http://compression.ca/act/act-summary.html*)包含各种工具的压缩和解压的速度和压缩比的基准测试。

CompressionInputStream，从而从底层的流读取未压缩的数据。

CompressionOutputSteam 和 CompressionInputStream 类似于 java.util.zip.DeflaterOutputStream 和 java.util.zip.DeflaterInputStream，前两者还可以提供重置其底层压缩和解压缩功能，当把数据流中的 section 压缩为单独的块时，这比较重要。比如 SequenceFile，详情参见后面的“序列文件”。

例 4-1 说明了如何使用 API 来压缩从标准输入读取的数据及如何将它写到标准输出。

例 4-1：压缩从标准输入读取的数据并将它写到标准输出

```
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();
  }
}
```

此应用需要压缩 CompressionCodec 的合法全名来作为命令行的第一个参数。我们使用 ReflectionUtils 来建立一个新的实例，然后获得一个压缩好的 System.out。然后我们调用 IOUtils 上的公共方法 copyBytes()将输入复制到经过 CompressionOutputStream 压缩的输出。最后，调用 CompressionOutputStream 的 finish()方法，从而向压缩程序表明结束向压缩流写入数据，但不关闭流。我们可以试试以下命令行，使用 StreamCompressor 程序与 GzipCodec 压缩字符串“Text”，然后使用 gunzip 从标准输入对它进行解压缩操作：

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

用 CompressionCodecFactory 方法来推断 CompressionCodec

在阅读一个压缩文件时，我们通常可以从其扩展名来推断出它的编码/解码器。以.gz 结尾的文件可以用 GzipCodec 来阅读，如此类推。每个压缩格式的扩展名均可参见表 4-1。

CompressionCodecFactory 提供了 getCodec()方法，从而将文件扩展名映射到相应的 CompressionCodec，此方法接受一个 Path 对象。例 4-2 显示了一个应用程序，此程序便使用这个功能来解压缩文件。

例 4-2：此程序根据文件的扩展名，利用编码/解码器对压缩文件进行解压

```
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);
    }
  }
}
```

编码/解码器一旦找到，就会被用来去掉文件名后缀生成输出文件名(通过 `CompressionCodecFactory` 的静态方法 `removeSuffix()` 来实现)。这样，如下调用程序便把一个名为 file.gz 的文件解压缩为 file 文件：

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

`CompressionCodecFactory` 从 `io.compression.codecs` 配置属性定义的列表中找到编码/解码器。默认情况下，这个列表列出了 Hadoop 提供的所有编码/解码器(见表 4-3)，如果你有一个希望要注册的编码/解码器(如外部托管的 LZO 编码/解码器)你可以改变这个列表。每个编码/解码器知道它的默认文件扩展名，从而使 `CompressionCodecFactory` 可以通过搜索这个列表来找到一个给定的扩展名相匹配的编码/解码器(如果有的话)。

表 4-3：压缩编码/解码器的属性

属性名	类型	默认值	描述
io.compression.codecs	逗号分隔的类名	org.apache.hadoop.io.compress.DefaultCodec, org.apache.hadoop.io.compress.GzipCodec, org.apache.hadoop.io.compress.Bzip2Codec	用于压缩/解压的 CompressionCodec 的列表

本地库

考虑到性能，最好使用一个本地库(native library)来压缩和解压。例如，在一个测试中，使用本地 gzip 压缩库减少了解压时间 50%，压缩时间大约减少了 10%(与内置的 Java 实现相比较)。表 4-4 展示了 Java 和本地提供的每个压缩格式的实现。并不是所有的格式都有本地实现(例如 bzip2 压缩)，而另一些则仅有本地实现(例如 LZO)。

表 4-4：压缩库实现

压缩格式	Java 实现	本地实现
DEFLATE	是	是
Gzip	是	是
bzip2	是	否
LZO	否	是

Hadoop 带有预置的 32 位和 64 位 Linux 的本地压缩库，位于库/本地目录。对于其他平台，需要自己编译库，具体请参见 Hadoop 的维基百科 *http://wiki.apache.org/hadoop/NativeHadoop*。

本地库通过 Java 系统属性 java.library.path 来使用。Hadoop 的脚本在 bin 目录中已经设置好这个属性，但如果不使用该脚本，则需要在应用中设置属性。

默认情况下，Hadoop 会在它运行的平台上查找本地库，如果发现，就自动加载。这意味着不必更改任何配置设置就可以使用本地库。在某些情况下，可能希望禁用本地库，比如在调试压缩相关问题的时候。为此，将属性 `hadoop.native.lib` 设置为 `false`，即可确保内置的 Java 等同内置实现被使用(如果它们可用的话)。

CodecPool(压缩解码池) 如果要用本地库在应用中大量执行压缩解压任务，可以考虑使用 `CodecPool`，从而重用压缩程序和解压缩程序，节约创建这些对象的开销。

例 4-3 所用的 API 只创建了一个很简单的压缩程序，因此不必使用这个池。

例 4-3：此应用程序使用一个压缩池程序来压缩从标准输入读入然后将其写入标准输出的数据

```
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);
    }
  }
}
```

我们从缓冲池中为指定的 `CompressionCodec` 检索到一个 `Comprssor` 实例，codec

的重载方法 `createOutputStream()` 中使用的便是它。通过使用 `finally` 块，我们便可确保此压缩程序会被返回缓冲池，即使在复制数据流之间的字节期间抛出了一个 `IOException`。

4.2.2 压缩和输入分割

在考虑如何压缩那些将由 MapReduce 处理的数据时，考虑压缩格式是否支持分割是很重要的。考虑存储在 HDFS 中的未压缩的文件，其大小为 1 GB。HDFS 的块大小为 64 MB，所以该文件将被存储为 16 块，将此文件用作输入的 MapReduce 作业会创建 16 个输入分片(split，也称为“分块”。对于 block，我们在本书中统一称为“块”。)，每个分片都被作为一个独立 map 任务的输入单独进行处理。

现在假设，该文件是一个 gzip 格式的压缩文件，压缩后的大小为 1 GB。和前面一样，HDFS 将此文件存储为 16 块。然而，针对每一块创建一个分块是没有用的，因为不可能从 gzip 数据流中的任意一点开始读取，map 任务也不可能独立于其他分块只读取一个分块中的数据。gzip 格式使用 DEFLATE 来存储压缩过的数据，DEFLATE 将数据作为一系列压缩过的块进行存储。问题是，每块的开始没有指定用户在数据流中任意点定位到下一个块的起始位置，而是其自身与数据流同步。因此，gzip 不支持分割(块)机制。

在这种情况下，MapReduce 不分割 gzip 格式的文件，因为它知道输入是 gzip 压缩格式的（通过文件扩展名得知），而 gzip 压缩机制不支持分割机制。这样是以牺牲本地化为代价：一个 map 任务将处理 16 个 HDFS 块，大都不是 map 的本地数据。与此同时，因为 map 任务少，所以作业分割的粒度不够细，从而导致运行时间变长。

在我们假设的例子中，如果是一个 LZO 格式的文件，我们会碰到同样的问题，因为基本压缩格式不为 reader 提供方法使其与流同步。但是，bzip2 格式的压缩文件确实提供了块与块之间的同步标记(一个 48 位的 π 近似值)，因此它支持分割机制。(表 4-1 列出了每种压缩格式是否支持分割。)

对于文件的收集，这些问题会稍有不同。ZIP 是存档格式，因此它可以将多个文件合并为一个 ZIP 文件。每个文件单独压缩，所有文档的存储位置存储在 ZIP 文件的尾部。这个属性表明 ZIP 文件支持文件边界处分割，每个分片中包括 ZIP 压缩文件中的一个或多个文件。在本书写作期间，Hadoop 还不支持 ZIP 文件作为输入格式。

> **应该使用哪种压缩格式？**
>
> 根据应用的具体情况来决定应该使用哪种压缩格式。就个人而言，更趋向于使用最快的速度压缩，还是使用最优的空间压缩？一般来说，应该尝试不同的策略，并用具有代表性的数据集进行测试，从而找到最佳方法。
>
> 对于那些大型的、没有边界的文件，如日志文件，有以下选项。
>
> - 存储未压缩的文件。
> - 使用支持分割机制的压缩格式，如 bzip2。
> - 在应用中将文件分割成几个大的数据块，然后使用任何一种支持的压缩格式单独压缩每个数据块(可不用考虑压缩格式是否支持分割)。在这里，需要选择数据块的大小使压缩后的数据块在大小上相当于 HDFS 的块。
> - 使用支持压缩和分割的 Sequence File(序列文件)。详情参见后面的“序列文件”小节。
>
> 对于大型文件，不要对整个文件使用不支持分割的压缩格式，因为这样会损失本地性优势，从而使降低 MapReduce 应用的性能。
>
> 对于存档，可考虑 Hadoop 的存档格式(参见前面的“Hadoop 存档”小节)，不过它也不支持压缩。

4.2.3 在 MapReduce 中使用压缩

如前所述，如果输入的文件是压缩过的，那么在被 MapReduce 读取时，它们会被自动解压，根据文件扩展名来决定应该使用哪一个压缩解码器。

如果要压缩 MapReduce 作业的输出，请在作业配置文件中将 `mapred.output.compress` 属性设置为 `true`，将 `mapred.output.compression.codec` 属性设置为自己打算使用的压缩编码/解码器的类名，如例 4-4 所示。

例 4-4：此应用程序运行最高气温作业从而产生压缩的输出结果

```
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);
    }

    JobConf conf = new JobConf(MaxTemperatureWithCompression.class);
    conf.setJobName("Max temperature with output compression");

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

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

    conf.setBoolean("mapred.output.compress", true);
    conf.setClass("mapred.output.compression.codec", GzipCodec.class,
        CompressionCodec.class);

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

    JobClient.runJob(conf);
  }
}
```

我们使用压缩过的输入来运行此应用程序(其实不必像它一样使用和输入相同的格式压缩输出)，如下所示：

```
% hadoop MaxTemperatureWithCompression input/ncdc/sample.txt.gz
output
```

最终输出的每部分都是压缩过的。在本例中，只有一部分：

```
% gunzip -c output/part-00000.gz
1949 111
1950 22
```

如果为输出使用了一系列文件，可以设置 mapred.output.compression.type 属性来控制压缩类型。默认为 RECORD，它压缩单独的记录。将它改为 BLOCK，则可以压缩一组记录，由于它有更好的压缩比，所以推荐使用。(详情参见后面的“序列文件格式”小节)

map 作业输出结果的压缩

即使 MapReduce 应用使用非压缩的数据来读取和写入，我们也可以受益于压缩 map 阶段的中间输出。因为 map 作业的输出会被写入磁盘并通过网络传输到 reducer 节点，所以如果使用 LZO 之类的快速压缩，能得到更好的性能，因为传输的数据量大大减少了。表 4-5 显示了启用 map 输出压缩和设置压缩格式的配置属性。

表 4-5：map 输出压缩属性

属性名称	类型	默认值	描述
mapred.compress.map.output	布尔	false	压缩 map 输出
mapred.map.output.compression.codec	类	org.apache.hadoop.io.compress.DefaultCodec	map 输出使用的压缩编码/解码器

下面几行代码用于在 map 作业中启用 gzip 格式来压缩输出结果：

```
conf.setCompressMapOutput(true);
conf.setMapOutputCompressorClass(GzipCodec.class);
```

4.3 序列化

序列化(serialization)指的是将结构化对象转为字节流以便于通过网络进行传输或写入持久存储的过程。反序列化指的是将字节流转为一系列结构化对象的过程。

序列化用于分布式数据处理中两个截然不同的领域：进程间通信和持久存储。

Hadoop 中，节点之间的进程间通信是用远程过程调用(RPC，remote procedures call)来实现的。 RPC 协议使用序列化将消息编码为二进制流(将被发送到远程节点)，此后，二进制流被反序列化为原始消息。一般情况下，可用的 RPC 序列化格式特点如下。

紧凑的

一个紧凑的格式使网络带宽得到充分利用，带宽是数据中心中最稀缺的资源。

快速

进程间通信是分布式系统的骨干，因此它必须尽量减少序列化和反序列化开销。

可扩展

协议随时间而变以满足新的要求，因此它应该直接演变为客户端和服务器端的

控制协议。例如，它应该可以加入一个新的参数方法调用，并且有新的服务器端接收来自老客户端的旧格式消息(不包括新的参数)。

互操作性

对于某些系统，最好能够支持用不同语言编写的客户端被写入服务器端，所以需要为此而精心设计文件格式。

在表面看来，数据持久性存储格式对序列化框架有不同的选择要求。毕竟，一个 RPC 的寿命不到一秒钟，而持久性数据可以写入后数年再读取。事实证明，RPC 序列化格式的四个可用特性对持久存储格式而言，相当重要。我们希望存储格式紧凑(高效使用存储空间)、快速(读取或写入 TB 级数据的开销变得极小)、可扩展(我们可以透明方式读取用旧格式写的数据)和互操作性(我们可以使用不同的语言读取或写入持久化数据)。

Hadoop 使用自己的序列化格式 Writables，它紧凑、快速(但不容易扩展或 Java 之外的语言)。由于 Writables 是 Hadoop 的核心(MapReduce 程序使用它来序列化键/值对)，所以在转而简要讨论其他几个有名的序列化框架(比如 Apache 的 Thrift 和谷歌的 Google Protocol Buffers)之前，我们先深入探讨一下。

4.3.1 Writable 接口

Writable 接口定义了两个方法：一个用于将其状态写入二进制格式的 `DataOutput` 流，另一个用于从二进制格式的 `DataInput` 流读取其态。

```
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;
}
```

让我们来看一个特别的 `Writeable`，看看可以对它进行哪些操作。我们要使用 `IntWritable`，这是一个 Java 的 `int` 对象的封装。可以使用 `set()` 函数来创建和设置它的值：

```
IntWritable writable = new IntWritable();
```

```
writable.set(163);
```

类似地，我们也可以使用构造函数：

```
IntWritable writable = new IntWritable(163);
```

为了检查 IntWritable 的序列化形式，我们写一个小的辅助方法，它把一个 java.io.ByteArrayOutputStream 封装到 java.io.DataOutputStream 中(java.io.DataOutput 的一个实现)，以此来捕获序列化的数据流中的字节：

```
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();
}
```

整数用四个字节写入(我们使用 Junit 4 断言)：

```
byte[] bytes = serialize(writable);
assertThat(bytes.length, is(4));
```

字节使用大端顺序写入(所以，最重要的字节写在数据流的开始处，这是由 java.io.DataOutput 接口规定的)，我们可以使用 Hadoop 的 StringUtils 方法看到它们的十六进制表示：

```
assertThat(StringUtils.byteToHexString(bytes), is("000000a3"));
```

让我们再来试试反序列化。我们创建一个帮助方法来从一个字节数组读取一个 Writeable 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;
}
```

我们构造一个新的、缺值的 IntWritable，然后调用 deserialize()方法来读取刚写入的输出流。然后发现它的值(使用 get()方法检索得到)还是原来的值 163：

```
IntWritable newWritable = new IntWritable();
deserialize(newWritable, bytes);
assertThat(newWritable.get(), is(163));
```

WritableComparable 和 comparator

IntWritable 实现了 WritableComparable 接口，后者是 Writable 和 java.lang.Comparable 接口的子接口：

```
package org.apache.hadoop.io;

public interface WritableComparable<T> extends Writable,
Comparable<T> {
}
```

类型的比较对 MapReduce 而言至关重要，键和键之间的比较是在排序阶段完成的。Hadoop 提供的一个优化方法是从 Java Comparator 的 RawComparator 扩展：

```
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);
}
```

这个接口允许执行者比较从流中读取的未被反序列化为对象的记录，从而省去了创建对象的所有开销。例如，IntWritables 的 comparator 使用原始的 compare()方法从每个字节数组的指定开始位置(S1 和 S2)和长度(L1 和 L2)读取整数 b1 和 b2 然后直接进行比较。

WritableComparator 是 RawComparator 对 WritableComparable 类的一个通用实现。它提供两个主要功能。首先，它提供了一个默认的对原始 compare()函数的调用，对从数据流对要比较的对象进行反序列化，然后调用对象的 compare()方法。其次，它充当的是 RawComparator 实例的一个工厂方法(Writable 方法已经注册)。例如，为获得 IntWritable 的 comparator，我们只需使用：

```
RawComparator<IntWritable> comparator=
  WritableComparator.get(IntWritable.class);
```

comparator 可以用来比较两个 IntWriteable 对象：

```
IntWritable w1 = new IntWritable(163);
IntWritable w2 = new IntWritable(67);
assertThat(comparator.compare(w1, w2), greaterThan(0));
```

或者它们的序列化描述；

```
byte[] b1 = serialize(w1);
byte[] b2 = serialize(w2);
assertThat(comparator.compare(b1, 0, b1.length, b2, 0,
    b2.length),greaterThan(0));
```

4.3.2 Writeable 类

Hadoop 将许多 Writeable 类归入 org.apache.hadoop.io 包。它们形成了图 4-1 所示的类层次结构。

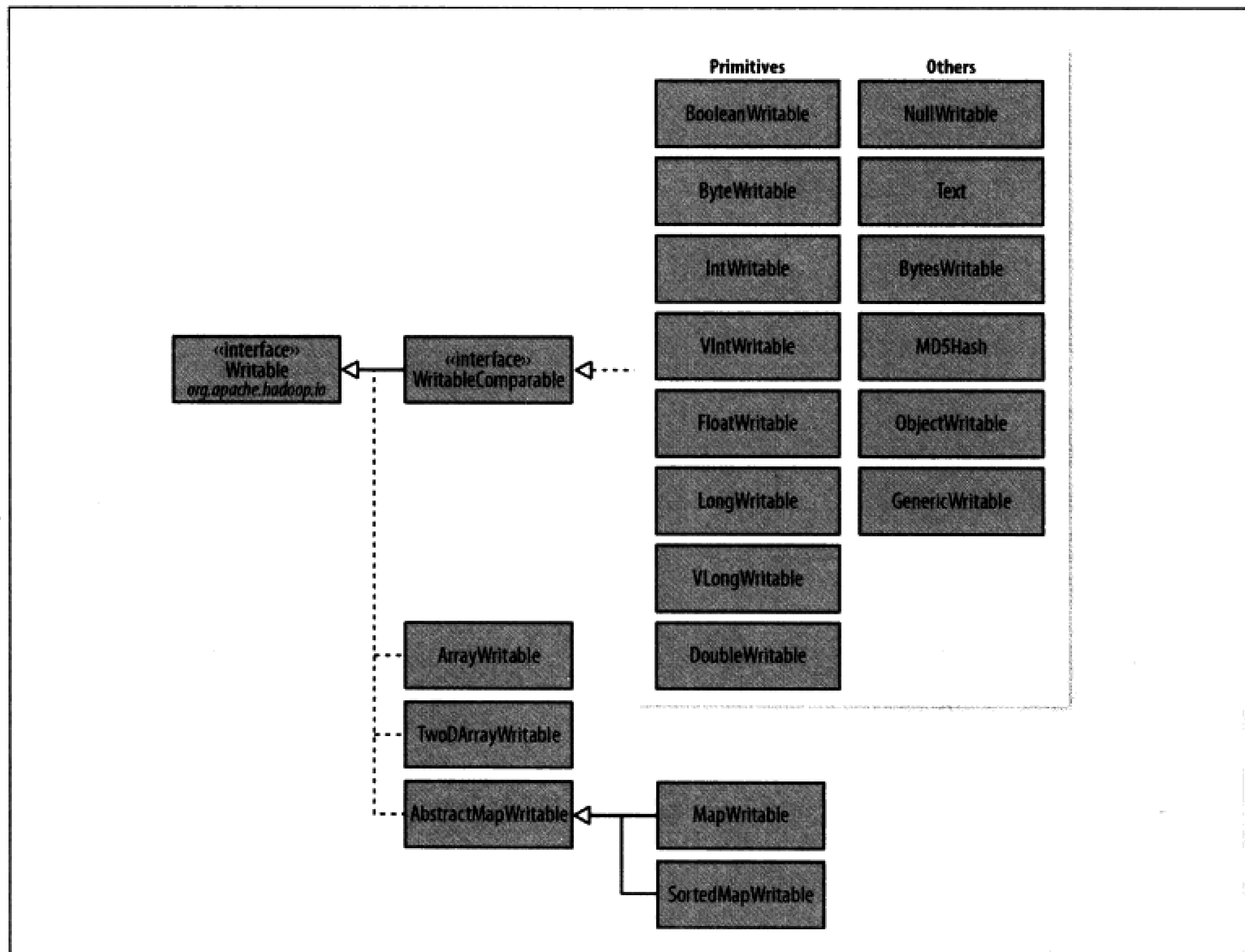


图 4-1：Writeable 类的层次结构

Writeable 的 Java 基本类封装

这是目前所有 Writeable 的 Java 基本类的封装(见表 4-6)，此外还有 short 和 char 类型(两者均可存储在 IntWritable 中)。它们都有用于检索和存储封装值的 get()和 set()方法。

表 4-6：Writeable 的 Java 基本类封装

Java 基本类型	Writeable 使用	序列化大小(字节)
布尔型	BooleanWritable	1
字节型	ByteWritable	1
整型	IntWritable	4
	VIntWritable	1-5
浮点型	FloatWritable	4
长整型	LongWritable	8
	VLongWritable	1-9
双精度浮点型	DoubleWritable	8

在对整数进行编码时，在固定长度格式(IntWritable 和 LongWritable)和可变长度格式(VIntWritable 和 VLongWritable)之间，有一个选择。如果值足够小(-112 和 127 之间，包含这两个值)，可变长度格式就只用一个字节来对值进行编码；否则，使用第一字节来表示值为正还是负，以及后面还有多少字节。例如，163 需要两个字节：

```
byte[] data = serialize(new VIntWritable(163));
assertThat(StringUtils.byteToHexString(data), is("8fa3"));
```

如何在固定长度和可变长度编码之间进行选择？固定长度编码的好处在于值比较均匀地分布在整个值空间中，就像(精心设计)的散列函数。大多数数字变量往往分布，所以可变长度编码往往更节省空间。可变长度编码的另一个好处是可以将 VintWritable 变为 VLongWritable，因为它们的编码实际上是相同的。因此，通过选择可变长度的编码方式，使空间可以增长，而不是一开始就占用 8 字节的空间。

Text 类

Text 类是一种 UTF-8 格式的 Writable。可以将它理解为一种与 java.lang.String 相类似的 Writable。Text 类代替了 UTF8 类，UTF8 类不支持编码大于 32 767 个字节的字符串，使用了 Java 改进过的 UTF-8。

Text 使用 int 型(使用一个可变长度的编码方案)在字符串编码中存储字节数，最大值是 2 GB。此外，Text 使用标准的 UTF-8，使其更易于与理解 UTF-8 的其他工具协同工作。

编制索引 由于强调使用标准的 UTF-8，所以 Text 和 Java 的 String 类之间还是有一些区别的。Text 类的索引位于编码后的字节系列中，而不是字符串中的 Unicode 字符，或 Java 的 char 编码单元(如同 String 一样)。对于 ASCII 字符串，索引位置的三个概念是一致的。下面的例子展示了 charAt()方法的用法：

```
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，charAt()返回了一个 int 类型来表示 Unicode 代码点，而不是像 String 变量那样返回一个 char 类型。Text 也有 find()方法，后者相当于 String 的 indexOf()：

```
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 在开始使用一个以上字节进行编码的字符时，Text 和 String 之间的区别是很明显的。我们来看看表 4-7 所示的 Unicode 字符。

表 4-7：Unicode 字符

统一编码代码点	名称	UTF-8 编码单元	Java 表示
U+0041	LATIN CAPITAL LETTER A	41	\u0041
U+00DF	LATIN SMALL LETTER SHARP S	c3 9f	\u00DF
U+6771	N/A (一种统一的汉字标准)	e6 9d b1	\u6771
U+10400	DESERET CAPITAL LETTER LONG I	f0 90 90 80	\uuD801\uDC00

除表中最后一个字符(U+10400)以外，所有字符都可以用一个 Java 字符来表示。U+10400 是补充的字符，可以用两个 Java 字符来表示，称为代理对。例 4-5[①]中的测试显示了 String 类和 Text 类在处理表 4-7 所示的由 4 字符组成的字符串的不同。

例 4-5：此测试演示了 String 类和 Text 类的不同之处

```
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));
```

① 本例基于 Java 平台的补充字符文章，网址为 *http://java.sun.com/developer/technicalArticles/Intl/Supplementary/*。

```
    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));
  }
}
```

测试结果证实，String 的长度是它包括的字符个数(5，3 个来自该字符串中的前 3 个字符，后面则是最后的代理对)，但 Text 对象的长度是其 UTF-8 编码的字节数(10 = 1 +2 +3 +4)。同样，indexOf()方法返回一个 char 类型的编码单元的索引，find()方法是字节偏移量。

在代理对不能表示整个 Unicode 字符的情况下，String 的 charAt()方法返回指定索引的 char 编码单元。在检索 int 类型的单独的 Unicode 字符时，我们需要使用 PointAt()方法(由字符编码单元索引)。事实上，Text 的 chatAt()方法比 String 的同名方法更像 codePointAt()方法。唯一的区别是，它由字节偏移量来索引。

迭代 使用索引的字节偏移对 Text 中的 Unicode 字符进行迭代是很复杂的，因为你不能只增加索引。迭代的定义有点模糊(见例 4-6)：将 Text 对象变成 java.nio.ByteBuffer，然后对缓冲的 Text 反复调用 bytesToCodePoint()静态方法。这个方法提取下一个代码点作为 int 然后更新缓冲中的位置。当 bytesToCodePoint()返回-1 时，检测到字符串结束。

例 4-6：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));
    }
  }
}
```

运行程序打印字符串中四个字符的代码点。

```
% hadoop TextIterator
41
df
6771
10400
```

可修改性　String 和 Text 的另一个区别在于可修改性(像 Hadoop 中的所有 Writable 实视一样，但 NullWritable 除外，后者是单实例对象)。我们可以通过对它调用 set()函数来重用 Text 实例。示例如下：

```
Text t = new Text("hadoop");
t.set("pig");
assertThat(t.getLength(), is(3));
assertThat(t.getBytes().length, is(3));
```

注意：在某些情况下，getBytes()方法返回的字节数组可能长于 getLength()返回的：

```
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));
```

这表明了为什么总是需要在调用 getBytes()时调用 getLength()，这样一来即可知道字节数组有多少有效数据。

转为字符串　Text 不像 java.lang.String 一样有一个可以处理字符串的 API，所以在许多情况下，需要将 Text 对象转化为 String 对象。这通常用 toString()方法来完成。

```
assertThat(new Text("hadoop").toString(), is("hadoop"));
```

BytesWritable

BytesWritable 是一个二进制数据数组封装。它的序列化格式是一个 int 字段(4 字节)，指定的是字节数及字节本身。例如，一个长度为 2，值为 3 和 5 的字节数组序列化为一个 4 字节的整数(00000002)加上两个来自数组的字节(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，其值可通过调用 set()方法来改变。对于 Text，从 getBytes()方法返回的字节数组大小可能并没有反映出存储在 BytesWritable 的

数据的实际大小。可以通过调用 getLength()方法来确定 BytesWritable 的长度，例如：

```
b.setCapacity(11);
assertThat(b.getLength(), is(2));
assertThat(b.getBytes().length, is(11));
```

NullWritable

NullWritable 是一种特殊的 Writable 类型，因为它的序列化是零长度的。没有字节被写入流或从流中读出。它被用作占位符。例如，在 MapReduce 中，在不需要这个位置的时候，键或值可以被声明为 NullWritable——它有效存储了一个不变的空值。NullWritable 也可以很有用，在打算存储一系列值的时候，作为 SequenceFile 的一个键，而不是键/值对。它是一个不变的单实例，其实例可以通过调用 NullWritable.get()方法来检索。

ObjectWritable 和 GenericWritable

针对 Java 基本类型、字符串、枚举、Writable、空值或任何一种此类类型的数组，ObjectWritable 是一种多用途的封装。它使用 Hadoop 的 RPC 来封送(marshal)和反封送(unmarshall)方法参数和返回类型。

ObjectWritable 适用于字段可以使用多种类型时。例如，如果在一个 SequenceFile 中的值有多种类型，就可以将值类型声明为 ObjectWritable 并把每个类型封装到一个 ObjectWritable 中。作为一个通用机制，这是相当浪费空间的，因为每次它被序列化时，都要写入被封装类型的类名。如果类型的数量不多并且事先可知，那么可以使用一个静态类型数组来提高效率，使用数组的索引来作为类型的序列化引用。这是 GenericWritable 使用的方法，我们必须继承它以指定支持的类型。

Writable 集合

org.apache.hadoop.io 包中有四种 Writable 集合类型，分别是 ArrayWritable, TwoDArrayWritable，MapWritable 和 SortedMapWritable。

ArrayWritable 和 TwoDArrayWritable 是 Writable 针对数组和二维数组(数组的数组)实例的实现。所有对 ArrayWritable 或者 TwoDArrayWritable 的使用都必须实例化相同的类，这是在构造时指定的，如下所示：

```
ArrayWritable writable = new ArrayWritable(Text.class);
```

在上下文中，Writable 由类型来定义，如在 SequenceFile 中的键或值，或一般作为 MapReduce 的输入，需要继承 ArrayWritable(或恰当使用 TwoDArrayWritable)以静态方式来设置类型。例如：

```
public class TextArrayWritable extends ArrayWritable {
  public TextArrayWritable() {
    super(Text.class);
  }
}
```

ArrayWritable 和 TwoDArrayWritable 都有 get() 和 set() 方法，也有 toArray() 方法，后者用于创建数组(或者二维数组)的浅拷贝(shallow copy)。

MapWritable 和 SortedMapWritable 分别是 java.util.Map(Writable, Writable) 和 java.util.SortedMap(WritableComparable, Writable) 的实现。每个键/值字段的类型都是此字段序列化格式的一部分。类型保存为单字节，充当一个数组类型的索引。数组是用 org.apache.hadoop.io 包中的标准类型来填充的，但自定义的 Writable 类型也是可以的，编写一个头，为非标准类型编码类型数组。正如它们所实现的那样，MapWritable 和 SortedMapWritable 使用正值 byte 值来表示自定义类型，因此最大值为 127 的非标准 Writable 类可以用于任何 MapWritable 或 SortedMapWritable 实例。下面是 MapWritable 的用法示例，针对不同的键/值对，使用不同的类型：

```
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)));
```

很显然 Writable 没有对集合和列表的实现。集合可以使用 NullWritable 值 MapWritable(或对一个排序集使用 SortedMapWritable)来模拟。对于 Writable 单类型的列表，ArrayWritable 足够了，但是存储不同的类型在一个单列表中，可以使用 GenericWritable 封装到 ArrayWritable 中。同时，也可以用 MapWritable 的思路写一个通用的 ListWritable。

4.3.3 实现自定义的 Writable

Hadoop 自带一系列有用的 Writable 实现，可以满足绝大多数用途。但有时，我们需要编写自己的自定义实现。通过自定义 Writable，我们能够完全控制二进制表示和排序顺序。Writables 是 MapReduce 数据路径的核心，所以调整二进制表示对其性能有显著影响。现有的 Hadoop Writable 应用已得到很好的优化，但为了对付更复杂的结构，最好创建一个新的 Writable 类型，而不是使用已有的类型。

为了演示如何创建一个自定义的 Writable，我们编写了一个表示一对字符串的实现，名为 TextPair。例 4-7 展示了基本的实现。

例 4-7：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);
  }
}
```

此实现的第一部分直观易懂：有两个 Text 实例变量(first 和 second)和相关的构造函数、get 方法和 set 方法。所有的 Writable 实现都必须有一个默认的构造函数，以便 MapReduce 框架能够对它们进行实例化，进而调用 readFields()方法来填充它们的字段。Writable 实例是易变的、经常重用的，所以我们应该尽量避免在 write()或 readFields()方法中分配对象。

通过委托给每个 Text 对象本身，TextPair 的 write()方法依次序列化输出流中的每一个 Text 对象。同样，也通过委托给 Text 对象本身，readFields()反序列化输入流中的字节。DataOutPut 和 DataInput 接口有丰富的整套方法用于序列化和反序列化 Java 基本类型，所以在一般情况下，我们能够完全控制 Writable 对象的数据传输格式。

正如为 Java 写的任意值对象一样，我们会重写 java.lang.Object 的 hashCode()方法，equals()方法和 toString()方法。HashPartitioner 使用 hashCode()方法来选择 reduce 分区，所以应该确保写一个好的哈希函数来确保 reduce 函数的分区在大小上是相当的。

注意： 如果想过在 TextOutputFormat 中使用自定义的 Writable，则必须实现 Writable 的 toString()方法。TextOutputFormat 调用 toString()来表现键/值对的输出形式。对于 TextPair，我们写入基本的 Text 对象作为由制表位分隔的字符串。

TextPair 是 WritableComparable 的实现，所以它提供了 compareTo()方法的实现，加入我们希望的顺序：它通过一个一个 String 逐个排序。请注意，TextPair 不同于前面小节中的 TextArrayWritable 类(除了它可以存储 Text 对象数之外)，因为 TextArrayWritable 只是一个 Writable，而不是 WritableComparable。

实现一个快速的 RawComparator

例 4-7 所示代码能够有效工作，但还可以进一步优化。正如前面所述，在 MapReduce 中，TextPair 被用作键时，它必须被反序列化为要调用的 compareTo()方法的对象。是否可以通过查看其序列化表示的方式来比较两个 TextPair 对象？

事实证明，我们可以这样做，因为 TextPair 由两个 Text 对象连接而成，二进制

Text 对象表示是一个可变长度的整型，包含 UTF-8 表示的字符串中的字节数，后跟 UTF-8 字节本身。关键在于读取开始的长度，从而得知第一个 Text 对象的字节表示有多长，然后可以委托 Text 对象的 RawComparator，然后利用第一或者第二个字符串的偏移量来调用它。例 4-8 给出了具体方法(注意，该代码嵌套在 TextPair 类中)。

例 4-8：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，我们一般都是继承 WritableComparator，而不是直接实现 RawComparator，因为它提供了一些便利的方法和默认实现。这段代码的精妙之处在于计算 firstL1 和 firstL2，每个字节流中第一个 Text 字段的长度。每个都由可变长度的整型(由 WritableUtils 的 decodeVIntSize()返回)和它的编码值(由 readVInt()返回)组成。

静态代码块注册原始的 comparator 以便 MapReduce 每次看到 TextPair 类，就知道使用原始 comparator 作为其默认 comparator。

自定义 comparator

从 TextPair 可知，编写原始的 comparator 比较费力，因为必须处理字节级别的细节。如果需要编写自己的实现，org.apache.hadoop.io 包中 Writable 的某些前瞻性实现值得研究研究。WritableUtils 的有效方法也比较非常方便。

如果可能，还应把自定义 comparator 写为 RawComparators。这些 comparator 实现的排序顺序不同于默认 comparator 定义的自然排序顺序。例 4-9 显示了 TextPair 的 comparator，称为 First Comparator，只考虑了一对 Text 对象中的第一个字符串。请注意，我们重写了 compare()方法使其使用对象进行比较，所以两个 compare()方法的语义是相同的。

我们将在第 8 章使用这种 comparator，届时再见识 MapReduce 中的连接和二次排序。

例 4-9：自定义的 RawComparator，用于比较 TextPair 字节表示中的第一个字段

```
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);
  }
}
```

4.3.4 序列化框架

虽然很多 MapReduce 程序都使用 Writable 键/值类型，但这并不是 MapReduce 的 API 指定的。事实上，任何类型都可以用，只要每种类型都可以转换为二进制表示并从二进制表示中转换出来。

为提供相应支持，Hadoop 有一个简便的序列化框架 API。序列化框架由 Serialization 实现(在 org.apache.hadoop.io.serializer 包中)来表示。例如 WritableSerialization 就是 Writable 类型的 Serialization 实现。

Serialization 定义了从类型到 Serializer 实例(一个对象转为一个字节流)和 Deserializer 实例(从一个字节流转为一个对象)的一个映射。

将 io.serializations 属性设置为逗号分隔的类名来注册序列化实现。它的默认值是 org.apache.hadoop.io.serializer.WritableSerialization，这意味着只有 Writable 对象才能在外部序列化或者反序列化。

Hadoop 包括一个名为 JavaSerialization 的类，此类使用的是 Java Object Serialization。虽然在 MapReduce 程序中，我们能够很方便地使用标准的 Java 类型，如 Interger 或 String，但 Java Object Serialization 不如 Writables 对象有效，所以不值得折腾(详见后文的补充内容)。

为什么不用 Java Object Serialization？

Java 拥有自己的序列化机制，名为 Java Object Serialization (经常有人简称为“Java Serialization”)，它与语言紧密结合在一起，所以很自然有人会问为什么不在 Hadoop 中使用它？对此，Doug Cutting 如是回答：

> 为什么在开始 Hadoop 时我不使用 Serialization？因为它看起来非常大、复杂，我感觉我们需要精确严密地控制对象的读写方式，因为它是 Hadoop 的灵魂。通过 Serialization，我们可以获得一些控制，但其复杂的本性，会让人烦恼不断。
>
> 不使用 RMI 的逻辑与此类似。对于 Hadoop 而言，有效、高性能的进程间通信至关重要。我们认为我们需要精确的控制连接、超时、缓冲区等的处理方式，对此，RMI 所给予的控制权少之甚少。

问题是 Java Serializtion 不符合前述序列化格式的条件：紧凑、快速、可扩展和可互操作。

Java Serializtion 并不紧凑。它将每个对象的类名写入流中——对于实现 `java.io.Serializable` 或者 `java.io.Externalizable` 的类的确如此。同一个类继承的实例在第一次使用时加入一个引用处理，占 5 字节的空间。但对于随机访问，引用控制的效果并不好，因为被引用的类出现在前面数据流中的某一点，即数据流中存储着它的状态。更糟的是，引用控制会破坏存储在序列化流中的记录排序。因为特定类的第一个记录都是不同的且必须被视为特殊情况进行处理。

所有这些问题都是可以避免的，具体做法是根本不要向流写入类名，这正是 Writable 采用的方法。这实际已经假设客户端知道自己预期的类型。结果是这种格式比 Java Serialization、随机访问和排序更紧凑得多，因为每个记录间都是独立的(所以它们没有流状态)。

Java Serialization 对于序列化对象的图是一个通用机制。所以一些序列化和反序列化操作的重写变得很有必要。此外，反序列化的过程为每个序列化的对象从流中创建了一个新的实例。另一方面，`Writable` 对象可以(或者经常)重用。例如，对于一个 MapReduce 作业，在它的核心是数十亿计的序列化和反序列化的各种类型的记录，不创建新的对象所获得的节省的开销非常大。

对于可扩展性，Java 序列化对于扩展一个类型有一定的支持，但非常脆弱并且事实上很难使用(Writable 没有支持，程序员不得不自己管理)。

原则上，别的语言可以解释 Java Serialization 的流协议(由 Java Object Serialization Specification 定义)，但实际°上，它们没有在其他语言的实现中广泛应用，所以这只是一种支持 Java 的解决方案。Writable 的情况也一样。

序列化 IDL

还有其他许多序列化框架，它们通过另一种方式来解决这个问题：不在代码中定义类型，而是使用的是接口描述语言(interface description language，IDL)，通过语言中立、声明式方式来定义它们。然后，系统便可以生成不同语言的类型，这对互操作性非常有用。它们通常还会定义版本控制方案，使类型演进更直观。

Hadoop 自身的记录 I/O(见 `org.apache.hadoop.record` 包)拥有一个 IDL，后者被编译到 Writable 对象，因此它能够更方便地生成兼容于 MapReduce 的类型。但即便如此，记录 I/O 也并不常用。

Apache Thrift(*http://indubator.apache.org/thrift*)和 Google Protocol Buffers(*http://code.google.com/p/protobuf/*)都是流行的序列化框架，并且它们常用作持久化二进制数据的格式。作为 MapReduce 的格式，它的支持有限。[①]但 Thrift 用在 Hadoop 的某些部分，用于提供跨语言的 API，如“thriftfs” contrib 模块，用于向 Hadoop 的文件系统提供 API。(详见第 3 章)

最后，在本书写作期间，Avro 还是一个新的 Hadoop 子项目，它定义了一个序列化格式。目的是迁移 Hadoop 的 RPC 机制以使用 Avro。Avro 将来还适合用作大型文件的数据格式。

4.4 基于文件的数据结构

对于某些应用，需要一个特殊的数据结构来存储数据。针对运行基于 MapReduce 的进程，将每个二进制数据块放入它自己的文件，这样做不易扩展，所以 Hadoop 为此开发了一系列高级容器。

---

① Thrift Serialization 框架的最新动态可以从 *https://issues.apache.org/jira/browse/HADOOP-3787* 获悉。Protocol Buffers Serialization 的最新动态则可以从 *https://issues.apache.org/jira/browse/HADOOP-3788* 获悉。

4.4.1 SequenceFile 类

假设有一个日志文件，它的每一个日志记录都是一行新的文本。如果想记录二进制类型，纯文本并不是一个合适的格式。Hadoop 的 SequenceFile 类很适合这个情况，它的做法是为二进制键/值对提供一个持久化的数据结构。如果想将它用作日志文件格式，需要选择一个键(如 LongWriteable 表示的时间戳)和一个值(是一个 Writable，表示日志记录的数量)。

SequenceFile 类作为小型文件的容器也不错。HDFS 和 MapReduce 是大型文件的利器，所以将文件打包到一个 SequenceFile 类中，使得我们能够更高效地对小型文件进行存储和处理(第 7 章包含一个程序，用于将文件打包到一个 SequenceFile 类中)。

编写 SequenceFile 类

要想新建一个 SequenceFile 类，请使用它的其中一个 createWriter()静态方法，该方法会返回一个 SequenceFile.Writer 实例。有几个重载版本，但是它们都需要指定一个要写入的流(FSDataOutputStream 或成对的文件系统和路径)、一个 Configuration 对象和键/值类型。可选参数包括压缩的类型和编码/解码器、一个将由写进度来唤醒的 Progressable 回调和一个将存储在 SequenceFile 类头部的 Metadata 实例。

存储在 SequenceFile 类中的键和值不一定必须是 Writable。可以被 SequenceFile 类序列化和反序列的任何类型都可以使用。

有 SequenceFile.Writer 之后，就用 append()方法写入键/值对。然后在结束的时候调用 close()方法。(SequenceFile.Writer 实现了 java.io.Closeable)。

例 4-10 显示了一个小程序，此程序将一些键/值对写入 SequenceFile 类，使用的是前面提到的 API。

例 4-10：编写一个 SequenceFile 类

```
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);
    }
  }
}
```

序列文件中的键/值是一个 100 到 1 递减的整型值，由 `IntWritable` 对象表示。值是 `Text` 对象。在每个记录加到 `SequenceFile.Writer` 之前，我们调用 `getLength()` 方法来了解文件中的当前位置。(我们将在下一小节按非序列方式读取文件时使用记录边界相关信息。)我们将位置信息写到控制台，随同键/值对一起。运行结果如下：

```
% 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
```

读取 SequenceFile 类

从头到尾读取序列文件，需要创建一个 SequenceFile.Reader 实例，反复调用 next()方法之一遍历记录。使用哪一个方法取决于所用的序列化框架。如果使用 Writable 类型，则可以使用取一个键和一个值作为参数的 next()方法，然后将数据流中的下一个键/值对读入这些变量：

```
public boolean next(Writable key, Writable val)
```

如果读取的是一个键/值对，则返回值为 true，如果读取的是文件末尾，则返回值为 false。

对其他非 Writable 类型的序列化框架(如 Apache Thrift)，可考虑使用下面两种方法：

```
public Object next(Object key) throws IOException
public Object getCurrentValue(Object val) throws IOException
```

在这里，必须确定希望使用的序列化方法已经设置了 io.serializations 属性。

如果 next()方法返回一个非 null 对象，则可以从数据流中读取一个键/值对，并用 getCurrentValue()方法来检索当前值。否则，如果 next()返回 null，则表明已经到达文件末尾。

例 4-11 中的程序说明了如何读取一个拥有 Writable 类型键/值对的序列文件。注意类型是如何通过调用 getKeyClass()和 getValueClass()方法从 SequenceFile.Reader 中找到的，然后 ReflectionUtils 被用于建立一个键的实

例和一个值的实例。通过使用这种技术，此程序可用于处理任何有 Writable 类型键/值对的序列文件。

例 4-11：读取一个序列文件

```
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);
    }
  }
}
```

此程序的另一个特征是它显示了序列文件中同步点的位置。同步点是流中的一个点，如果 reader“迷失”，同步点就可用于重新同步记录边界，例如在查找流中任意一个位置之后。同步点由 SequenceFile.Reader 来记录，当序列文件被写入的时候，它会每隔几个记录就插入一个特殊的项来标记此同步点。插入的这种项非常小，通常开销小于存储大小的 1%。同步点通常与记录边界重合。

运行例 4-11 中的例子，结果表明同步点在序列文件中显示为星号。第一个出现在位置 2021(第二个出现的位置在 4075，但输出中未显示)：

```
% 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
```

有两种方法可以查找序列文件中的指定位置。第一种是 seek()方法，它将 reader 定位在文件中的指定点。例如，如预期的那样寻找一个记录边界：

```
reader.seek(359);
assertThat(reader.next(key, value), is(true));
assertThat(((IntWritable) key).get(), is(95));
```

但如果文件中的指定位置不是记录边界，reader 会在调用 next()方法时失败。

```
reader.seek(360);
reader.next(key, value); // fails with IOException
```

第二种查找记录边界的方法用到了同步点。SequenceFile.Reader 上的 sync(long position)方法把 reader 定位到下一个同步点。(如果在这之后没有同步点，那么此 reader 会定位到文件末尾)。因此，我们可以用流中的任何位置来调用 sync()——如一个没有记录的边界——且 reader 将自己重定位到下一个同步点继续读取：

```
reader.sync(360);
assertThat(reader.getPosition(), is(2021L));
```

```
assertThat(reader.next(key, value), is(true));
assertThat(((IntWritable) key).get(), is(59));
```

注意：SequenceFile.Reader 有一个名为 sync()的方法，用于在流中的当前位置插入一个同步点。这并没有引起名称上的混淆，但由 Syncable 接口定义的无关联的 sync()方法用来对缓冲区和底层设备进行同步。

在使用序列文件作为 MapReduce 输入的时候，同步点开始发挥作用，因为它们允许文件分割，所以它的不同部分通过独立的 map 任务得以单独处理。详情参见第 7 章。

用命令行接口显示序列文件

Hadoop 文件系统命令有一个-text 选项显示文本格式的序列文件。它看起来像是文件的魔法数字，使其能够尝试检测文件类型并相应地将其转换为文本。它可以识别 gzip 压缩文件和序列文件，否则便假设输入是纯文本。

对于序列文件，在键/值对有一个有意义的字符串表示时，此命令非常有用(如 toString()方法定义的那样)。如果有自己的键/值对类，必须确定它们在 Hadoop 的类路径上。

对前一小节创建的序列文件运行它，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
```

排序和合并序列文件

排序和合并一个或多个序列文件，最强的方式是使用 MapReduce。MapReduce 固有的并行方式，允许我们指定使用多少个 reduce(此数量决定桌输出分区的个数)。例如，通过指定一个 reducer，我们会得到一个输出文件。通过指定输入和输出为序列文件并通过设置键/值类型，我们可以使用 Hadoop 自带的排序例子：

```
% hadoop jar $HADOOP_INSTALL/hadoop-*-examples.jar sort -r 1 \
  -inFormat org.apache.hadoop.mapred.SequenceFileInputFormat \
  -outFormat org.apache.hadoop.mapred.SequenceFileOutputFormat\
  -outKey org.apache.hadoop.io.IntWritable \
  -outValue org.apache.hadoop.io.Text \
numbers.seq sorted
% hadoop fs -text sorted/part-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
```

排序在第 8 章有详细描述。

作为使用 MapReduce 进行排序/合并的替代方案，`SequenceFile.Sorter` 类有 `sort()` 和 `merge()` 方法。这些函数的出现早于 MapReduce，但性能低于 MapReduce(例如，为了实现并行计算，需要对数据进行手动分区)，所以一般情况下，MapReduce 是排序、合并序列文件的首选方法。

序列文件的格式

序列文件由一个头部和一个或多个记录组成(参见图 4-2)。

序列文件的前三位字节是 `SEQ` 字节，作为幻数，后紧接一个字节代表版本号。头部包括其他字段(包含键/值类的名称、压缩细节、用户定义的元数据和同步标记)[1]。调用同步标记可以允许一个 reader 同步一个字段中的任何一个记录边界。每个文件有一个随机产生的同步记号，它的值存储在头部中。同步记号出现在序列文件的记录中，它们设计数量不超过存储的 1%，所以它们并没有必要出现每一对记录中(比如在一个很短的记录中)。

---

① 对于这些字段的格式，完整的详细信息可参见 `SequenceFile` 的文档(*http://hadoop.apache.org/core/docs/current/api/org/apache/hadoop/io/SequenceFile.html*)和源代码。

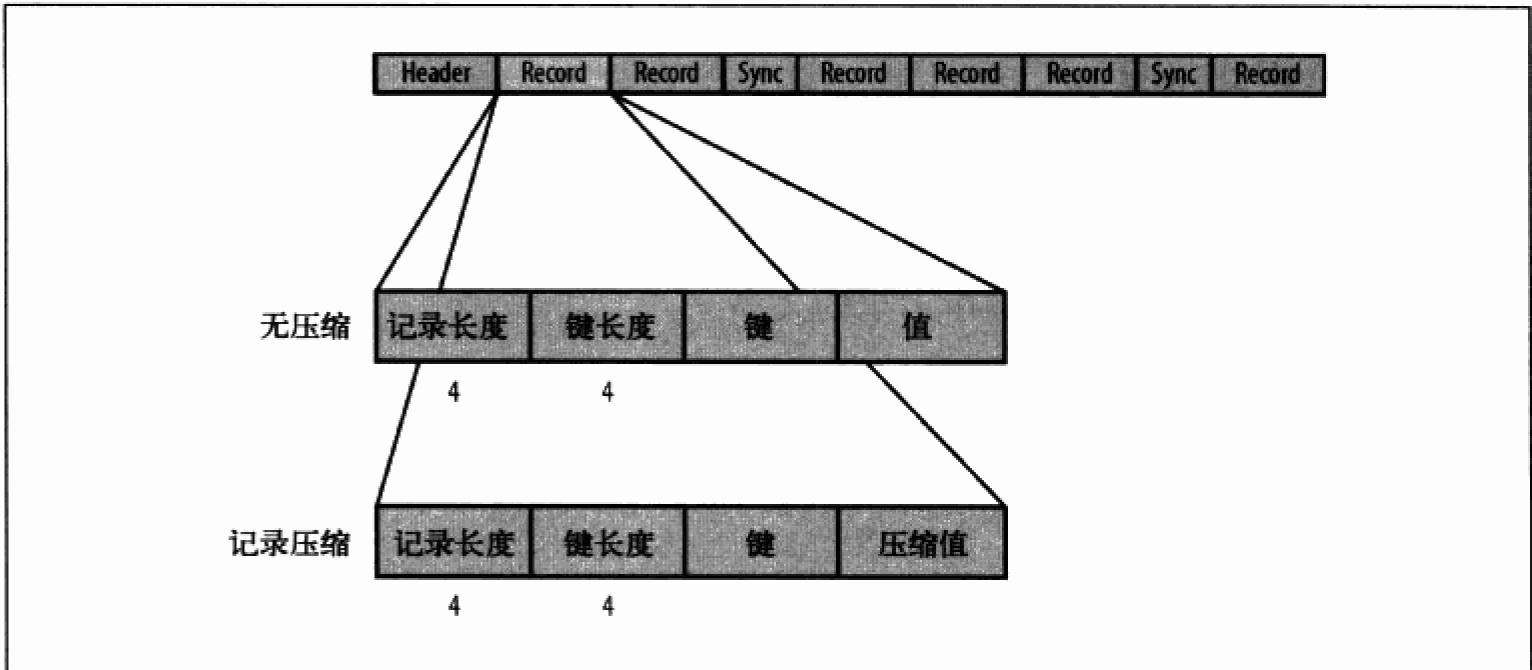


图 4-2：无压缩和记录压缩的序列文件的内部结构

记录的内部格式取决于是否启用压缩，如果是，则要么记录压缩，要么块压缩。

如果没有启用压缩(默认设置)，那么每个记录都由它的记录长度(字节数)、键的长度、键和值组成。长度字段被作为四字节整型值写入，遵循 `java.io.DataOutput` 中 `writeInt()`方法的规范。键/值的序列化是通过为正被写入序列文件中的类而定义的 `Serialization` 来完成的。

记录压缩格式与无压缩基本相同，不同的是值字节是用定义在头部的编码/解码器来压缩的。注意，键是不压缩的。

块压缩一次压缩多个记录，因此它比记录压缩更紧凑，且一般应优先选择，因为它有机会利用记录之间的相似之处(参见图 4-3)。记录直到到达字节的最小大小才会被添加到块，该最小值是由 `io.seqfile.compress.blocksize` 中的属性定义的，它的默认值是 1 000 000 字节。同步标记写在每个块开始之前。块的格式是一个字段(指出块中的记录数)，后跟四个字段(分别为键长度、键、值长度和值)。

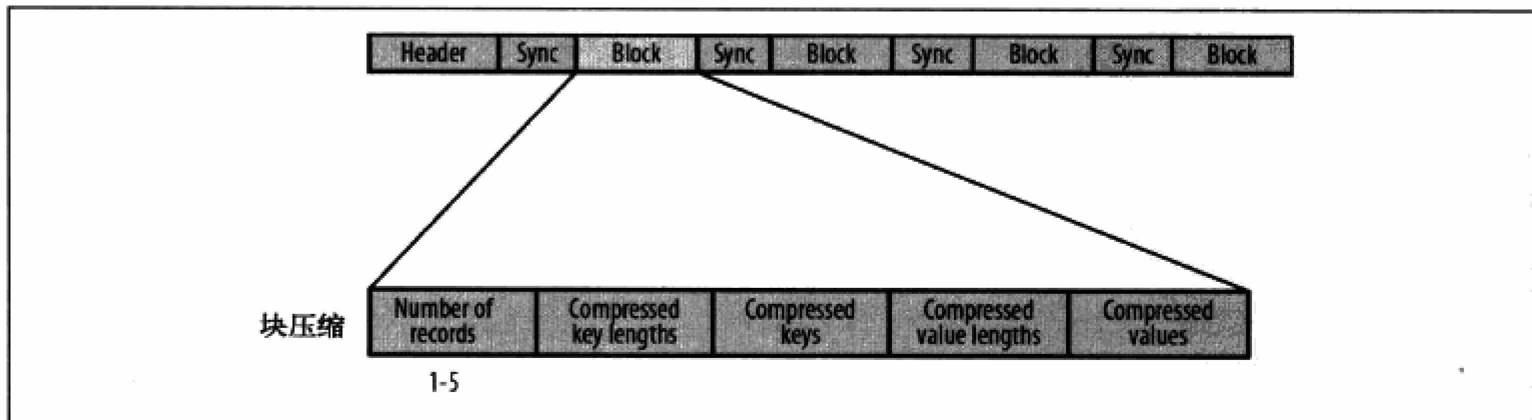


图 4-3：块压缩的序列文件的内部结构

4.4.2 MapFile

MapFile 是经过排序的带索引的 SequenceFile，可以根据键进行查找。MapFile 可以被认为是 java.util.Map 的一种持久化形式(虽然它没有实现这个接口)，它会增长，乃至超过 Map 在内存中占用的大小。

写一个 MapFile

写一个 MapFile 和写一个 SequenceFile 差不多：创建一个 MapFile.Writer 的实例，然后调用 append()方法，按顺序添加条目。(如果不按顺序添加，会抛出 IO 异常)键必须是 WritableComparable 的一个实例，值必须是 Writable 类型。相对于 SequenceFile，后者可以为其条目使用任何序列化框架。

例 4-12 的程序创建了一个 MapFile 文件，并向它写入了一些条目。它和例 4-10 创建 SequenceFile 的程序非常相似。

例 4-12：写一个 MapFile

```
public class MapFileWriteDemo {

  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);

    IntWritable key = new IntWritable();
    Text value = new Text();
    MapFile.Writer writer = null;
    try {
      writer = new MapFile.Writer(conf, fs, uri,
        key.getClass(), value.getClass());

      for (int i = 0; i < 1024; i++) {
```

```
        key.set(i + 1);
        value.set(DATA[i % DATA.length]);
        writer.append(key, value);
      }
    } finally {
      IOUtils.closeStream(writer);
    }
  }
}
```

使用此程序来创建一个 MapFile：

```
% hadoop MapFileWriteDemo numbers.map
```

看看这个 MapFile，可以看出它确实是一个目录，其中包含两个文件，分别名为 data 和 index：

```
% ls -l numbers.map
total 104
-rw-r--r-- 1 tom tom 47898 Jul 29 22:06 data
-rw-r--r-- 1 tom tom 251 Jul 29 22:06 index
```

两个文件都是 SequenceFile。数据文件包括所有的输入，按顺序的：

```
% hadoop fs -text numbers.map/data | head
1 One, two, buckle my shoe
2 Three, four, shut the door
3 Five, six, pick up sticks
4 Seven, eight, lay them straight
5 Nine, ten, a big fat hen
6 One, two, buckle my shoe
7 Three, four, shut the door
8 Five, six, pick up sticks
9 Seven, eight, lay them straight
10 Nine, ten, a big fat hen
```

index 文件包括一小部分键，并且包括键到 data 文件中键偏移量的映射：

```
% hadoop fs -text numbers.map/index
1 128
129 6079
257 12054
385 18030
513 24002
641 29976
```

```
769 35947
897 41922
```

从输出可以看出，默认情况下，只有每 128 个键被包括在索引中，不过可以改变这个数，具体做法是设置 io.map.index.interval 属性或调用 MapFile.Writer 实例的 setIndexInterval()方法。增加索引间隔的理由之一是减少 MapFile 存储索引所需要的内存大小。反之，在牺牲内存的情况下，可以减少索引间隔以改善随机时间(因为平均说来，需要跳过的记录更少)。

因为索引只是键的部分索引，所以 MapFile 不能提供方法来枚举或计数它包含的所有键。执行这些操作的唯一方法是读取整个文件。

读取一个 MapFile

按顺序遍历一个 MapFile 的过程和读取 SequenceFile 的过程相似：创建一个 MapFile.Reader，然后调用 next()函数直到它返回 false，表示没有条目可以再读取因为已经到达文件末尾。

```
public boolean next(WritableComparable key, Writable val)
throws IOException
```

随机存取查找可以调用 get()方法来执行：

```
public Writable get(WritableComparable key, Writable val)
throws IOException
```

返回值用于是否在 MapFile 中找到一个条目；如果返回值为 null，则指定值没有对应的值。如果找到键，则此键的对应值被读取到 val，就像从调用方法返回一样。

理解它的实现方式可能有所帮助。这里是一小段代码，从前一小节创建的 MapFile 进行检索：

```
Text value = new Text();
reader.get(new IntWritable(496), value);
assertThat(value.toString(), is("One, two, buckle my shoe"));
```

对于这个操作，MapFile.Reader 将 index 文件读取到内存中(这里是缓存的，使后面的随机存储调用能使用相同的内存索引)。然后，reader 对内存内的索引执行二叉搜索，试图在索引文件中找到小于或等于搜索键 496 的那些键。在这个例子中，找到的索引键是 385，对应的值是 18030，后者是在 data 文件中的偏移量。接着，reader 在 data 文件中找到这个偏移量，然后读取数据直到这个键大于或者等于搜索

键 496。在这里，找到了匹配且从 data 文件中读取了数据。总体来说，查找选择的是一次磁盘寻址和一次遍历磁盘上 128 项的磁盘扫描。对于随机存储读取，这实际上是非常高效的。

getClosest()方法和 get()相似，不同的是它返回“更靠近”指定键的匹配，而不是在没有匹配时返回 null。更精确地说，如果 MapFile 包括指定的键，那么它就是返回值；否则返回这个 MapFile 中直接在该指定键之后的键(或者之前，看布尔参数的设置)。

大型 MapFile 的索引会占很大的内存。相较于重建索引改变索引间隔，我们可以通过设置 io.map.index.skip 属性将 MapFile 中的小部分键读入内存。这个属性一般为 0，它意味着不跳过索引键；如果值为 1，意味着隔一个键跳过一个键(所以留下一半的键在索引中)；如果值为 2，表示读取一个键跳过两个键(所以三分之一的键留在索引中)，以此类推。更大的跳过值可以节省更多内存空间，但消耗查找时间，因为平均而言，需要在磁盘中扫描更多条目。

将 SequenceFile 转换为 MapFile

对于 MapFile，一种方式是将其视为经过索引和排序的 SequenceFile，所以我们自然会想到把 SequenceFile 转换为 MapFile。前面讨论了如何对 SequenceFile 进行排序，所以这里来看如何创建 SequenceFile 的索引。例 4-13 的程序使用了类似 MapFile 的静态实用方法 fix()，后者重建 MapFile 的索引。

例 4-13：重建 MapFile 的索引

```
public class MapFileFixer {

  public static void main(String[] args) throws Exception {
    String mapUri = args[0];

    Configuration conf = new Configuration();

    FileSystem fs = FileSystem.get(URI.create(mapUri), conf);
    Path map = new Path(mapUri);

    Path mapData = new Path(map, MapFile.DATA_FILE_NAME);

    // Get key and value types from data sequence file
    SequenceFile.Reader reader = new SequenceFile.Reader(fs, mapData,
      conf);
    Class keyClass = reader.getKeyClass();
```

```
    Class valueClass = reader.getValueClass();
    reader.close();

    // Create the map file index file
    long entries = MapFile.fix(fs, map, keyClass, valueClass, false,
      conf);
    System.out.printf("Created MapFile %s with %d entries\n", map,
      entries);
  }
}
```

fix()方法经常用于重建被损坏的索引，但因为它从零开始创建新的索引，所以它确实正是这里需要的方法。具体方法如下。

1. 将序列文件 numbers.seq 归入一个名为 number.map 的新建目录，后者将变成 MapFile。(如果这个序列文件已排好序，则可以跳过这个步骤。将其复制到一个名为 number.map/data 的文件，然后执行步骤 3。)

   ```
   % hadoop jar $HADOOP_INSTALL/hadoop-*-examples.jar sort -r 1 \
   -inFormat org.apache.hadoop.mapred.SequenceFileInputFormat \
   -outFormat org.apache.hadoop.mapred.SequenceFileOutputFormat\
   -outKey org.apache.hadoop.io.IntWritable \
   -outValue org.apache.hadoop.io.Text \
   numbers.seq numbers.map
   ```

2. MapReduce 的输出被重命名为 data 文件。

   ```
   % hadoop fs -mv numbers.map/part-00000 numbers.map/data
   ```

3. 创建 index 文件。

   ```
   % hadoop MapFileFixer numbers.map
   Created MapFile numbers.map with 100 entries
   ```

   现在，便有了名为 numbers.Map 的 MapFile，且该文件已经可用了。

第 5 章

MapReduce 应用开发

在第 2 章，我们介绍了 MapReduce 模型。在第 5 章，我们来看看用 Hadoop 开发 MapReduce 应用这一实用层面。

MapReduce 应用开发包含特定的流程。首先，编写 map 和 reduce 函数，最好能进行单元测试以保证它们能如期运行。然后，写一个驱动程序来运行作业，可以使用数据集中的少量数据从 IDE 运行，看它是否能够正常运行。如果失败，可以使用 IDE 的调试器来查找问题的根源。有了这些信息，便可扩展单元测试覆盖这些用例，然后改进 mapper 或 reducer 使其能够正确处理此类输入。

针对少量数据，程序如果能正常运行，就可以在集群上运行它了。运行整个数据集会暴露出更多问题，这些问题都可以像之前那样修复，具体做法是扩展测试与 mapper 或 reducer 来处理这些新的问题。在集群中调试失败的程序是一种挑战，但 Hadoop 为此提供了一些工具，如 `IsolationRunner`，必要时，我们可以用它附带的调试器在运行失败时对同样的输入重新运行任务。

在程序运行期间，可能希望做一些调整，首先按某些能使 MapReduce 程序更快运行的标准来运行 MapReduce 程序，然后再执行任务概要分析。对分布式程序进行概要分析可不是一件简单的事，但值得庆幸的是，Hadoop 为此提供了相应的工具。

在准备写 MapReduce 程序前，需要安装和配置开发环境。为此，需要了解如何配置 Hadoop。

5.1 API的配置

Hadoop 中的组件是按照 Hadoop 自己的 API 配置来进行配置的。一个 Configuration 类的实例(在 org.apche.hadoop.conf 包)包括配置属性及其值的集合。每个属性是一个 String 类型，值的类型可能是以下多种类型之一，包括 Java 基本类型(如 boolean、int、long、float)和其他一些有用的类型(如 String、Class、java.io.File 和 String collection)。

配置从给定资源中读取它们的属性，通常是名-值对构成的结构简单的 XML 文件。如例 5-1 所示。

例 5-1：一个简单的配置文件 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>
```

假设配置文件在 configuration-1.xml 文件中，我们就可以用下面的代码得到它的属性：

```
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"));
```

这里有一些地方需要注意：类型的信息并不存储在 XML 文件中；反之，在被读入时，属性可以自动识别成一个给定的类型。并且，`get()`方法允许定义一个默认的值，当属性没有在 XML 文件中定义时，系统就默认使用这个值，就像这里 `breadth` 的情况一样。

5.1.1 合并资源

当多个资源被用来定义一个配置的时候，问题将变得十分有趣。在 Hadoop 中，这用来区分 core-default.xml 文件内部定义的系统默认属性和 core-site.xml 中定义的是特定重写。例 5-2 中，文件定义了 `size` 和 `weight` 两个属性。

例 5-2：另一个配置文件 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>
```

资源按顺序添加到 Configuration 中：

```
Configuration conf = new Configuration();
conf.addResource("configuration-1.xml");
conf.addResource("configuration-2.xml");
```

后来加入的这些资源中定义的属性将覆盖先前定义的属性。因此，`size` 属性将从第二个配置文件 configuration-2.xml 中得到它的值：

```
assertThat(conf.getInt("size", 0), is(12));
```

然而，标记为 final 的属性不能被后来的定义所覆盖。在第一个配置文件中，weight 属性被标记为 final，所以在第二个文件中重写失败，而是从第一个文件中获取值：

```
assertThat(conf.get("weight"), is("heavy"));
```

尝试重写标记为 final 的属性通常会报告配置错误，所以其结果是有警告信息会被记录下来以便于诊断。管理员将守护进程地址文件中的属性标记为 final，以防止用户在客户端配置文件或作业提交参数中改变它。

5.1.2 各种扩展形式

配置的属性可以由其他属性或者系统属性来定义。例如，在第一个配置文件中的 size-weight 属性被定义为${size},${weight}，并且这些属性使用 configuration 中的值得以扩展：

```
assertThat(conf.get("size-weight"), is("12,heavy"));
```

系统属性的优先级比资源文件中所定义属性的优先级更高：

```
System.setProperty("size", "14");
assertThat(conf.get("size-weight"), is("14,heavy"));
```

在命令行中使用 JVM 参数-Dproperty=value 来重写属性时，此特性非常有用。

注意，配置属性可以通过系统属性来定义，但是除非系统属性用配置属性来重定义，否则不能通过配置 API 来访问这些系统属性。因此：

```
System.setProperty("length", "2");
assertThat(conf.get("length"), is((String) null));
```

5.2 配置开发环境

首先下载打算使用的 Hadoop 版本，然后将其解压到开发机器上(详见附录 A)。接着，在你喜欢的 IDE 中创建一个新的工程，将解压后文件夹根目录下的 JAR 文件和 lib 目录下的 JAR 文件都加入 classpath。随后就可以编译 Java Hadoop 程序，以本地模式在 IDE 中运行它。

注意：对于 Eclipse 用户，可以使用一个插件来浏览 HDFS 和运行 MapReduce 程序。详细的步骤可在 Hadoop 维基查询，网址为 *http://wiki.apache.org/hadoop/EclipsePlugIn*。

5.2.1 配置的管理

当我们开发 Hadoop 应用时，经常在本地或集群上切换运行程序。实际中，可能使用几个集群来工作，或者有一个本地的伪分布式集群来测试程序(伪分布式集群是一个其守护进程都在本地运行的集群，设置这种模式的详情请参见附录 A)。

适应这种多样性的一种办法是，使用多组 Hadoop 配置文件，其中每组包含一种运行集群的连接方式。在运行 Hadoop 应用或工具时，再指定使用哪组配置文件。一个最有效的办法是让这些文件远离 Hadoop 的安装目录树，以便在转换 Hadoop 版本时，配置不会重叠或丢失。

在本书中，假设有一个目录叫 conf，其中有三个配置文件：hadoop-local.xml、hadoop-localhost.xml 和 hadoop-cluster.xml(在本书示例代码中)。它们的名字并没有什么特殊之处——只是为了方便存储一些配置设置。(请与附录 A 中的表 A-1 相比，该表设置了相同的服务器端配置)

hadoop-local.xml 文件包含 Hadoop 默认文件系统和 jobtracker 的默认配置：

hadoop-localhost.xml 文件中的设置指向在本地运行的两个节点和 jobtracker：

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

  <property>
    <name>fs.default.name</name>
    <value>file:///</value>
  </property>

  <property>
    <name>mapred.job.tracker</name>
    <value>local</value>
  </property>

</configuration>
```

hadoop-localhost.xml 文件的设置表明在本地同时运行的名称节点和 jobtracker：

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

```
<configuration>

  <property>
    <name>fs.default.name</name>
    <value>hdfs://localhost/</value>
  </property>

  <property>
    <name>mapred.job.tracker</name>
    <value>localhost:8021</value>
  </property>

</configuration>
```

最后，hadoop-cluster.xml 文件包含集群的名称节点和 jobtracker 的地址详细信息。实际中，可以用集群的名字来命名文件，而不是像这里用 cluster 来命名。

```
<?xml version="1.0"?>
<configuration>
<property>
<name>fs.default.name</name>
<value>hdfs://namenode/</value>
</property>
<property>
<name>mapred.job.tracker</name>
<value>jobtracker:8021</value>
</property>
</configuration>
```

如果需要，可以在这些文件中加入其他配置属性。例如，如果想在一个特定的集群中的机器设定 Hadoop 用户名，则可以在相关的文件中进行设置。

设置用户身份

Hadoop 用作 HDFS 的访问权限的用户身份，它是由客户端系统上运行 whoami 命令来确定的。类似地，组名来自 groups 命令的输出。

然而，如果 Hadoop 用户身份和客户端机器上的用户账户名不同，可以通过设置 hadoop.job.ugi 属性，显式地设置 Hadoop 用户名和组名。用户名和组名以一列由逗号分隔的字符串来表示。如 preston,director,inventor 就是将用户名设置成 preston，组名设置成 director 和 inventor。

> 可以使用相同的语法规则设置 dfs.web.ugi，以设置运行 HDFS web 接口的用户身份。默认情况下，一般是 webuser 和 webgroup，而不是一个超级用户，系统文件不能通过网络接口访问。
>
> 注意，系统没有身份验证，这也是 Hadoop 未来的版本中将来要完善的。

有了这个设置，可以很方便地利用带有-conf 选项的命令行切换每种配置。例如，下面的命令显示了在伪分布式模式下运行的 HDFS 服务器的目录列表：

```
% hadoop fs -conf conf/hadoop-localhost.xml -ls .
Found 2 items
drwxr-xr-x - tom supergroup 0 2009-04-08 10:32 /user/tom/input
drwxr-xr-x - tom supergroup 0 2009-04-08 13:09 /user/tom/output
```

如果忘记-conf 选项，可以从 conf 的子目录下$HADOOP_INSTALL 的 conf 子目录中找到 Hadoop 配置。如何设置决定着这是一个独立环境还是伪分布式集群。

Hadoop 自带工具支持-conf 选项，也可以使用 Tool 接口，直接使程序(例如运行 MapReduce 作业的程序)支持此选项。

5.2.2 GenericOptionsParser，Tool 和 ToolRunner

Hadoop 提供一些辅助类，旨在简化在命令行运行作业。GenericOptionParser 是一个解释普通 Hadoop 命令行选项的类，可以根据应用需要在 Configuration 对象中进行设置。通常不直接使用 GenericOptionsParser，因为实现 Tool 接口，然后用 ToolRunner(内部调用 GenericOptionsParser)来运行应用更方便。

```
public interface Tool extends Configurable {
  int run(String [] args) throws Exception;
}
```

例 5-3 是一个非常简单的 Tool 实现，用来打印 Tool 的 Configuration 对象中所有属性的键和值。

例 5-3： Tool 实现示例，打印 Configuration 对象的属性

```
public class ConfigurationPrinter extends Configured implements Tool {

  static {
    Configuration.addDefaultResource("hdfs-default.xml");
    Configuration.addDefaultResource("hdfs-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，ConfigurationPrinter 是 Configurable 接口的一个实现。Tool 接口的所有实现都需要实现 Configurable(因为 Tool 继承于它)，而 Configured 的子类一般是最简单的实现方法。使用 Configurable 的 getConf()方法，run()方法可获得 Configuration，然后重复执行，将每个属性打印到标准输出。

静态代码块保证了 HDFS 和 MapReduce 的配置和核配置已被选取(Configuration 已经取得核的配置)。

ConfigurationPrinter 的 main()方法不会直接引用自身的 run()方法。相反，我们调用 ToolRunner 的静态 run()方法，它负责在调用自身的 run()方法之前为 Tool 新建一个 Configuration 对象。ToolRunner 还使用 GenericOptionParser 来得到在命令行中指定的标准选项配置，然后在 Configuration 实例上进行配置。运行以下代码，我们可以看到在 conf/hadoop-localhost.xml 指定的属性：

```
% hadoop ConfigurationPrinter -conf conf/hadoop-localhost.xml \
  | grep mapred.job.tracker=
mapred.job.tracker=localhost:8021
```

可以设置哪些属性

ConfigurationPrinter 是一个实用的工具，可以说明如何在环境中设置属性。

可以在 Hadoop 安装路径的 docs 目录中查看 core-default.html，hdfs-default.html 和 mapred-default.html 这几个 HTML 文件，借此看到 Hadoop 中所有公共属性的默认设置。每个属性都有说明，用以解释属性的作用和可能设置的值。

请注意，在客户端配置文件中设置时，有些属性并不起作用。比如，如果在作业提交中，设置 mapred.tasktracker.map.task.maximum 以希望改变运行作业的 tasktracker 的任务槽数，你会失望，因为此属性只能在 tasktracker 的 mapred-site.html 文件中设置。一般来说，可以通过属性名称来告诉组件该属性应该在哪里设置，mapred.tasktracker.map.tasks.maximum 以 mapred.tasktracker 开头的事实告诉我们，它只能为 tasktracker 守护进程设置。但这并不是一个硬性的规则，有些时候需要尝试或者犯一些错误，甚至读一读源代码。

本书讨论了 Hadoop 的许多非常重要的配置属性。可以在本书配套网站找到配置属性的参考资料，网址为 *http://www.hadoopbook.com*。

GenericOptionsParser 也允许用户设置个人自定义的属性，例如：

```
% hadoop ConfigurationPrinter -D color=yellow | grep color
color=yellow
```

-D 选项用来将键“color”设置成“yellow”。-D 选项的优先级比配置文件中的属性优先级高。这是十分有用的：可以把默认属性放入配置文件，需要时用-D 选项来覆盖它们。一个常见例子是，通过 -D mapred.reduce.tasks=n 来设置 MapReduce 作业中 reducer 的数量。这将覆盖集群上 reducer 的数量，或者在任何客户端配置文件中设置这个值。

GenericOptionsParser 和 ToolRunner 支持的其他选项在表 5-1 中列出。本章的稍后将进一步介绍 Hadoop 的配置 API。

注意：GenericOptionsParser(和 ToolRunner)使用 -D *property=value* 选项设置 Hadoop 属性，不要与使用的 Java 命令 -D *property=value* 设置 JVM 系统属性相混淆。JVM 系统属性语法规则不允许 D 和属性名称之间出现任何空格，而 GenericOptionsParser 要求用空格来分隔。JVM 系统属性来自 java.lang.System 这

个类，而 Hadoop 属性只能从 Configuration 对象进行访问。所以下面的命令行将不输出任何信息，因为 ConfigurationPrinter 没有使用 System 类：

```
% hadoop -Dcolor=yellow ConfigurationPrinter | grep color
```

如果希望通过系统属性进行配置，则需要将所需的系统属性镜像到配置文件中的“各种”小节。

表 5-1：GenericOptionsParser 和 ToolRunner 选项

选项	说明
-D *property=value*	将指定值赋给指定的 Hadoop 配置属性。可覆盖在配置中的任何默认属性或站点属性以及用-conf 选项设置的属性
-conf *filename* ...	将指定文件加入配置的资源列表。这是一种设置站点属性或同时配置多组属性的简便方法。
-fs *uri*	用指定的 URI 来设置默认的文件系统。这是 -D fs.default.name=uri 的快捷方式
-jt host:post	设置 jobtracker 的 host 和 port。这是 -Dmapred.job.tracker=host: port 的快捷方式
-files *file1,file2,...*	从本地的文件系统(或任何符合模式的文件系统)中复制指定的文件到 jobtracker 使用的共享文件系统中(通常为 HDFS)，让 MapReduce 程序能够在任务工作目录中得到上述文件(请参阅第 8 章，进一步了解用于复制文件到 tasktracker 机器上的分布式缓存机制)
-archives	从本地文件系统(或任何具有模式的文件系统)复制指定的归档到 jobtracker 使用的共享文件系统中(通常是 HDFS)，读取它们并让 MapReduce 程序能在任务工作目录中获得上述文件
-libjars *jar1,jar2,...*	从本地文件系统(或任何符合模式的文件系统)复制指定的 JAR 文件到 jobtracker 使用的共享文件系统中(通常是 HDFS)，并将它们加入到 MapReduce 任务的类路径中。这个选项适用于传送作业需要的 JAR 文件

5.3 编写单元测试

分别测试 MapReduce 中的 map 和 reduce 函数是十分容易的，这是由它们的函数式风格决定的。在已知输入的情况下，得到已知的输出。但是，如果输出是写入一个

OutputCollector，而不是简单地从调用方法中返回结果，就需要用一个模板代替 OutputCollector 以证实结果是正确的。有许多种 Java 模拟对象框架可以帮助我们建模。这里我们使用 Mockito，虽然其他还有许多模板也很好用，但它由于简洁的语法而著称。[①]

这里描述的所有测试都能在 IDE 中运行。

5.3.1 Mapper

例 5-4 是一个 mapper 的单元测试。

例 5-4：MaxTemperatureMapper 的单元测试

```
import static org.mockito.Matchers.anyObject;
import static org.mockito.Mockito.*;

import java.io.IOException;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.OutputCollector;
import org.junit.*;

public class MaxTemperatureMapperTest {

  @Test
  public void processesValidRecord() throws IOException {
    MaxTemperatureMapper mapper = new MaxTemperatureMapper();

    Text value = new Text("0043011990999991950051518004+68750+
                          023550FM-12+0382" +// Year ^^^^
      "99999V0203201N00261220001CN9999999N9-00111+99999999999");
                         // Temperature ^^^^^
    OutputCollector<Text, IntWritable> output =
      mock(OutputCollector.class);

    mapper.map(null, value, output, null);

    verify(output).collect(new Text("1950"), new IntWritable(-11));
  }
}
```

① 读者也可以查阅有关 MRUnit contrib 模型，这个模型可以简化 MapReduce 程序的单元测试。

测试十分简单：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。mapper 忽略输入的键和 reporter，所以我们可以传送任何值，包括传入一个空值。我们调用 Mockito 的 mock()方法，传入我们想要建模类型的类(一个静态导入)来创建 OutputCollector 的模板，然后引用 mapper 的 map()方法，执行测试的代码。最后我们使用 Mockito 的 verify()方法(同样是静态导入)来证实模板对象是以正确的方法和参数被调用的。这里我们证实了 OutputCollector 的 collect()方法是由代表年(1950)的 Text 对象和代表气温(-1.1°C)的 IntWritable 对象为参数来调用的。

在测试驱动模式下，创建一个 mapper 的实现来通过测试(请见例 5-5)。由于本章要进行类的扩展，所以为了方便起见，每个类被放入标识着版本号的不同的包中。例如，v1.Max TemperatureMapper 是 MaxTemperatureMapper 的第一个版本。实际上，我们当然会改进类而不重新打包。

例 5-5：通过 MaxTempatureMapperTest 的 mapper 的第一个版本

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

  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 = Integer.parseInt(line.substring(87, 92));
    output.collect(new Text(year), new IntWritable(airTemperature));
  }
}
```

这是一个十分简单的实现，将年份和气温值从行中读取出来并传到 OutputCollector。让我们加入一个丢失值的测试，该值在原始数据中表示气温+9999：

```
  @Test
  public void ignoresMissingTemperatureRecord() throws
  IOException {
    MaxTemperatureMapper mapper = new MaxTemperatureMapper();

    Text value = new Text("0043011990999991950051518004+68750+
```

```
                          023550FM-12+0382" +// Year ^^^^
    "99999V0203201N00261220001CN9999999N9+99991+99999999999");
                            // Temperature ^^^^^
  OutputCollector<Text, IntWritable> output =
    mock(OutputCollector.class);

  mapper.map(null, value, output, null);

  Text outputKey = anyObject();
  IntWritable outputValue = anyObject();
  verify(output, never()).collect(outputKey, outputValue);
}
```

由于丢失气温的记录已经被过滤，所以这个测试使用 Mockito 来证实 `OutputCollector` 中的 `collect` 方法未调用任何 `Text` 键或 `IntWritable` 值。

测试以 `NumberFormatException` 错误失败，因为 `parseInt()`不能解析以加号开头的整数，所以我们要修改这个实现(版本 2)来处理丢失值：

```
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);
  String temp = line.substring(87, 92);
  if (!missing(temp)) {
      int airTemperature = Integer.parseInt(temp);
      output.collect(new Text(year), new
        IntWritable(airTemperature));
  }
}

private boolean missing(String temp) {
  return temp.equals("+9999");
}
```

这个测试通过后，接下来写 reducer。

5.3.2 reducer

reducer 要找到指定键的最大值。下面是一个简单测试：

```java
@Test
public void returnsMaximumIntegerInValues() throws IOException{
  MaxTemperatureReducer reducer = new MaxTemperatureReducer();

  Text key = new Text("1950");
  Iterator<IntWritable> values = Arrays.asList(
    new IntWritable(10), new IntWritable(5)).iterator();
  OutputCollector<Text, IntWritable> output =
    mock(OutputCollector.class);

  reducer.reduce(key, values, output, null);

  verify(output).collect(key, new IntWritable(10));
}
```

我们建立一个迭代来遍历 `IntWritable` 值，以确认 `MaxTemperatureReducer` 的结果值是最大值。例 5-6 中的代码是已经通过测试的一个 `MaxTemperatureReducer` 的实现。注意，我们没有测试空值的迭代，理论上不需要这样做，因为 mapper 产生的每个键都有一个值，所以 MapReduce 不会在这种情况下调用 reducer。

例 5-6：最高气温 reducer 示例代码

```java
public class MaxTemperatureReducer extends MapReduceBase
  implements Reducer<Text, IntWritable, Text, IntWritable> {

  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));
  }
}
```

5.4 本地运行测试数据

现在我们已经能够让 mapper 和 reducer 在某确定的输入上工作了，下一步我们将编写一个作业驱动文件，并在一个开发机器的测试数据上运行。

5.4.1 在本地作业运行器上运行作业

使用前面介绍的 Tool 接口，将有效简化我们编写一个寻找每年最高气温的 MapReduce 作业驱动程序。(请参阅例 5-7 中的 MaxTemperatureDriver)

例 5-7：寻找最高气温的应用

```
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;
    }

    JobConf conf = new JobConf(getConf(), getClass());
    conf.setJobName("Max temperature");

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

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

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

    JobClient.runJob(conf);
    return 0;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new MaxTemperatureDriver(), args);
    System.exit(exitCode);
  }
}
```

MaxTemperatureDriver 实现了 Tool 的接口，所以我们可以通过

GenericOptionsParser 的支持设置选项。在 JobConf 描述的作业被执行之前，Run()方法就构建和配置了 JobConf 对象。在可设置的作业配置参数中，设置输入和输出文件的路径，mapper、reducer 和 combiner 类以及输出类型(输出类型由输出格式决定，输出格式一般是由 TextInputFormat 中规定的默认值，包括 LongWritable 键和 Text 值)。为作业设置一个名字是很好的做法，以便我们在执行过程中以及完成后能将其挑选出来。默认情况下，名字是 JAR 文件的名字，通常没有特殊的描述。

现在我们可以在一些本地文件上运行这个应用了。Hadoop 有一个本地的作业运行器，是在单一 JVM 上运行 Map-Reduce 作业的 MapReduce 执行引擎的精简版本。它设计用于测试，在 IDE 中用起来非常方便，我们可以在调试器中运行它，一步一步地检查 mapper 和 reducer 中的代码。

注意： 本地作业运行器只用于进行简单的 MapReduce 程序的测试，注定与完整的 MapReduce 实现有区别。最大的不同在于它不能运行多个 reducer(它同样支持 0 个 reducer)。通常情况下这是没问题的，因为尽管在一台机器上可以选择大量 reducer 以充分利用并行的优势，但大多数应用只使用一个 reducer。值得注意的是，即便将 reducer 的数量设置成大于 1，本地 runner 也会忽略这种设置而使用一个 reducer。

本地作业运行器也不支持 DistributedCache 特性(“分布式缓存”的介绍详见第 8 章)。

这些限制并不是本地作业运行器固有的，在 Hadoop 今后的版本中将放宽这些限制。

本地作业运行器可以通过一个配置设置来激活。一般而言，mapred.job.tracker 是一个 host:port 对，定义 jobtracker 的地址，当它是特殊值 local 的时候，作业将在没有外部 jobtracker 控制的情况下执行。我们可以输入如下命令来运行驱动程序：

```
% hadoop v2.MaxTemperatureDriver -conf conf/hadoop-local.xml \
input/ncdc/micro max-temp
```

同样，也可以用 GenericOptionsParser 提供的-fs 和-jt 选项：

```
% hadoop v2.MaxTemperatureDriver -fs file:/// -jt local
input/ncdc/micro max-temp
```

这个指令使用来自本地 input/ncdc/micro 目录的输入执行 MaxTemperatureDriver，并在本地的 max-temp 目录产生输出。注意，虽然设置了-fs，使用本地文件系统(file://)，但实际上本地作业运行器可以在任何文件系统上正常运行，包括 HDFS(如果 HDFS 上有新文件，可以立即试验一下。)

修正 mapper

这个异常表明 map 方法仍旧不能分析正的气温值。(如果栈轨迹没有足够的信息来诊断这个错误，则由于它在单一的 JVM 中运行，所以我们可以在本地调试器中进行测试)。前面，我们用它来处理特殊例子，即丢失的气温值+9999，但不是一般的任意正的气温值。当 mapper 包含更多的逻辑时，提出一个解析类来封装解析的逻辑是很有价值的，详见例 5-8(现在是版本 3)。

例 5-8：分析以 NCDC 格式的气温记录的类

```
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
    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;
  }
}
```

结果 mapper 很简单(见例 5-9)。它调用 parser 中的 parse()方法，从输入的行中分析感兴趣的字段，使用 isValidTemperature()方法来检查是否是合法的气温值，或者使用 parser 的 getter 方法来检查是否是检索出的年份和气温。注意，我们同样也要在 isValidTemperature()方法中检查质量状态字段和丢失的气温值，来过滤气温读取错误。

创建 parser 类的另一个好处是不用写重复的代码编写相似作业的 mapper。同样也让我们有机会直接编写一个有更多用途的 parser 单元测试。

例 5-9：将 utility 类应用于 parse 记录的 mapper 程序

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

  private NcdcRecordParser parser = new NcdcRecordParser();

  public void map(LongWritable key, Text value,
      OutputCollector<Text, IntWritable> output, Reporter reporter)
      throws IOException {

    parser.parse(value);
    if (parser.isValidTemperature()) {
      output.collect(new Text(parser.getYear()),
        new IntWritable(parser.getAirTemperature()));
    }
  }
}
```

经过这些修改后，测试得以通过。

5.4.2 测试驱动程序

让应用程序实现 Tool 可以提供灵活的配置选项，另外还可以加入强制性的 Configuration，使其更加适合测试。你可以自己编写测试，使用本地作业运行器在已知输入数据上运行一个作业，以检查是否得到预期输出。

有两种方法能够测试驱动。第一种方法是使用本地的作业运行器，在本地文件系统的一个测试文件上运行一个作业。例 5-10 中的代码给我们提供了一个提示。

例 5-10：使用本地运行作业运行器的 MaxTemperatureDriver 测试

```
@Test
public void test() throws Exception {
  JobConf conf = new JobConf();
  conf.set("fs.default.name", "file:///");
  conf.set("mapred.job.tracker", "local");

  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);
}
```

测试明确设置 fs.default.name 和 mapred.job.tracker，所以它使用本地文件系统和本地作业运行器。接着通过 Tool 的接口在少量已知数据上运行 MaxTemperatureDriver。测试最后，调用 checkoutput()方法来逐行比较实际的输出和预期的输出。

测试驱动程序的第二种方法是使用一个“迷你”集群来运行它。Hadoop 有一对测试类，即 MiniDFSCluster 和 MiniMRCluster，它们提供程序化的方法来创建运行中的集群。与本地作业运行器不同，它们允许在整个 HDFS 和 MapReduce 机器上运行测试。注意，在迷你集群中，不同的 tasktracker 启动各自的 JVM 来执行任务，这会使调试变得更加困难。

迷你集群广泛应用于 Hadoop 自带的自动化测试包，但也可用于测试用户代码。Hadoop 的 ClusterMapReduceTestCase 抽象类提供了一个编写此类测试的良好基础，它使用 setUp()和 tearDown()方法开始和终止运行进程中的 HDFS 和 MapReduce 集群，同时产生一个合适的 JobConf 对象(已被配置为与这两个方法一

起工作)。子类只需要得到 HDFS 中的数据(也许从本地文件复制得到)，运行 MapReduce 作业，然后确认输出是否与预期相同。参见本书示例代码中的 MaxTemperature DriverMiniTest 类。

前述测试充当的是回归测试，是一个输入边界用例及其预期结果的资源库。随着测试用例的增多，只需将它们加入输入文件，然后相应地更新预期的输出文件。

5.5 在集群上运行

在少量测试数据集上成功运行程序之后，下面准备在一个有完整数据集的 Hadoop 集群上运行它。第 9 章将全面介绍如何建立一个完整的分布式集群，当然你也可以用该方法在一个伪分布式集群上工作。

5.5.1 打包

单机上运行的代码不需要修改就能在集群上运行，但需要把程序打包成 JAR 文件，然后发送到集群。使用 Ant 能让这个过程变得更加简单，使用下面这样一个任务(可以在示例代码中找到完整的 build 文件)：

```
<jar destfile="job.jar" basedir="${classes.dir}"/>
```

如果每个 JAR 都有一个作业，则可以指定 main 类来运行 JAR 文件的 manifest。如果 main 类不在 manifest 中，就必须在命令行指定(见下文)。同样，任何非独立的 JAR 文件都必须打包到 JAR 文件的 lib 子目录中。(这与 Java 的 web application archive 或 WAR 文件相似，不同的是，后者的 JAR 文件放在 WEB-INF/lib 子目录下的 WAR 文件中。)

5.5.2 启动作业

我们需要运行驱动程序来启动作业，使用-conf 选项来指定将要运行作业的集群(使用-fs 和-it 选项可以达到同样的效果)：

```
% hadoop jar job.jar v3.MaxTemperatureDriver -conf conf/hadoop-
cluster.xml \input/ncdc/all max-temp
```

在 JobClient 上的 runJob()方法启动作业并跟踪进度，当 map 或 reduce 中任何一个的进度发生变化时，就输出一行。下面就是它的输出(为了看得更清楚，已删除了部分行)：

```
09/04/11 08:15:52 INFO mapred.FileInputFormat: Total input
paths to process : 101
09/04/11 08:15:53 INFO mapred.JobClient: Running job:
job_200904110811_0002
09/04/11 08:15:54 INFO mapred.JobClient: map 0% reduce 0%
09/04/11 08:16:06 INFO mapred.JobClient: map 28% reduce 0%
09/04/11 08:16:07 INFO mapred.JobClient: map 30% reduce 0%
...
09/04/11 08:21:36 INFO mapred.JobClient: map 100% reduce 100%
09/04/11 08:21:38 INFO mapred.JobClient: Job complete:
job_200904110811_0002
09/04/11 08:21:38 INFO mapred.JobClient: Counters: 19
09/04/11 08:21:38 INFO mapred.JobClient: Job Counters
09/04/11 08:21:38 INFO mapred.JobClient: Launched reduce tasks=32
09/04/11 08:21:38 INFO mapred.JobClient: Rack-local map tasks=82
09/04/11 08:21:38 INFO mapred.JobClient: Launched map tasks=127
09/04/11 08:21:38 INFO mapred.JobClient: Data-local map tasks=45
09/04/11 08:21:38 INFO mapred.JobClient: FileSystemCounters
09/04/11 08:21:38 INFO mapred.JobClient: FILE_BYTES_READ=12667214
09/04/11 08:21:38 INFO mapred.JobClient: HDFS_BYTES_READ=33485841275
09/04/11 08:21:38 INFO mapred.JobClient: FILE_BYTES_WRITTEN=989397
09/04/11 08:21:38 INFO mapred.JobClient: HDFS_BYTES_WRITTEN=904
09/04/11 08:21:38 INFO mapred.JobClient: Map-Reduce Framework
09/04/11 08:21:38 INFO mapred.JobClient: Reduce input groups=100
09/04/11 08:21:38 INFO mapred.JobClient: Combine output records=4489
09/04/11 08:21:38 INFO mapred.JobClient: Map input records=1209901509
09/04/11 08:21:38 INFO mapred.JobClient: Reduce shuffle bytes=19140
09/04/11 08:21:38 INFO mapred.JobClient: Reduce output records=100
09/04/11 08:21:38 INFO mapred.JobClient: Spilled Records=9481
09/04/11 08:21:38 INFO mapred.JobClient: Map output bytes=10282306995
09/04/11 08:21:38 INFO mapred.JobClient: Map input bytes=274600205558
09/04/11 08:21:38 INFO mapred.JobClient: Combine input
records=1142482941
09/04/11 08:21:38 INFO mapred.JobClient: Map output
records=1142478555
09/04/11 08:21:38 INFO mapred.JobClient: Reduce input records=103
```

输出包含很多有用的信息。在作业运行之前，ID 被打印出来：只要想找到该作业，都需要它，比如在日志文件中，或者在使用 hadoop job 命令来查询作业时。当作业完成后，它的统计数据(即计数器)将被打印出来。这对于确认作业是否预期完成是很有用的。例如，对于这个作业，从 HDFS(“HDFS_BYTES_READ”)上大约 34 GB 的压缩文件读入大约 275 GB 的输入数据进行分析(“Map 输入字节”)。输入

被分成 101 个合适大小的 gzipped 文件，因此即使不划分也不会出现任何问题。

作业、任务和任务尝试(Task Attempt) ID

作业 ID 的格式由 jobtracker(不是作业)执行的时间和一个由 jobtracker 维护的单独标识作业 jobtracker 实例递增的计数器组成。所以下面这个作业：

```
job_200904110811_0002
```

它是 jobtracker 运行的第二个作业(0002，作业编号以 1 起始)，开始于 2009 年 4 月 11 日 8：11。计数器首位为 0，以使作业 ID 排列更美观(在目录列表中)。

然而，当计数器达到 10000 时，如果没有重置，会使作业的 ID 变得更长(这个时候，排列就不怎么美观了)。

任务属于某个作业，它们的 ID 即把作业前缀替换成任务前缀，再加一个后缀来标识作业中的任务。例如：

```
task_200904110811_0002_m_000003
```

这是 ID 号为 `job_200904110911_0002` 作业中第 4 个 map 任务(`000003`，任务 ID 以 0 为起始)。在被初始化之后为一个作业创建任务 ID，所以不必指明任务执行的顺序。

任务由于失败(参阅第 6 章)或者推测式执行(参阅第 6 章)，可能会被执行多次，所以需要识别不同的任务执行的实例，任务尝试在 jobtracker 上被赋予一个唯一的 ID。例如：

```
attempt_200904110811_0002_m_000003_0
```

这是运行在 `task_200904110811_0002_m_000003` 任务上的第一个实例(0，任务尝试 ID 以 0 为起始)。任务尝试在作业执行期间根据需要进行分配，所以它们的顺序代表 tasktracker 的运行顺序。

如果作业在 jobtracker 重启之后重启，并恢复运行中的作业，那么最后的任务尝试 ID 从 1000 开始计数。

5.5.3 MapReduce 网络用户界面

Hadoop 的网络用户界面用来浏览作业的信息，跟踪作业在执行过程中的进度和在

作业完成后查看作业的统计数据和日志。可以在 *http://jobtracker-host:50030/*找到更多有关用户界面的信息。

jobtracker 页

图 5-1 是主页的全屏截图。第一部分给出一些有关 Hadoop 安装的细节，如版本号和编译时间，jobtracker 的当前状态(在这个示例中是运行状态)和它启动的时间。

ip-10-250-110-47 Hadoop Map/Reduce Administration

Quick Links

State: RUNNING
Started: Sat Apr 11 08:11:53 EDT 2009
Version: 0.20.0, r763504
Compiled: Thu Apr 9 05:18:40 UTC 2009 by ndaley
Identifier: 200904110811

Cluster Summary (Heap Size is 53.75 MB/888.94 MB)

Maps	Reduces	Total Submissions	Nodes	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes
53	30	2	11	88	88	16.00	0

Scheduling Information

Queue Name	Scheduling Information
default	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0002	NORMAL	root	Max temperature	47.52%	101	48	15.25%	30	0	NA

Completed Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0001	NORMAL	gonzo	word count	100.00%	14	14	100.00%	30	30	NA

Failed Jobs

none

Local Logs

Log directory, Job Tracker History

Hadoop, 2009.

图 5-1：jobtracker 页的屏幕截图

接下来是关于集群的一个概要的介绍，包括集群的负载能力和使用情况。它们分别显示了当前正在集群上运行的 map 和 reduce 的数量、提交的作业总数、当前可用的 tasktracker 节点数和集群的能力——在整个集群中可用的 map 和 reduce 的槽数（“Map Task Capacity”和“Reduce Task Capacity”)和每个节点平均可用槽数。被

jobtracker 列入黑名单的 tasktracker 节点数也被列出(更多有关黑名单的信息，请参阅第 6 章)。

在集群概要的下面，是正在运行的作业调度器(这里是默认调度器)。可以点击查看作业队列。

然后我们可以看到正在运行的、(成功)完成的和失败的作业。每部分都有一个作业表，其中横行显示作业的 ID、所属者、名字(使用 JobConf 的 setJobName()方法来设置，它设置的是 mapred.job.name 属性)和进度信息。

最后，在页面的底部，是一些链接，指向 jobtracker 的日志和历史记录：记录 jobtracker 运行过的所有作业的信息。在存储到历史页之前，主页上只显示 100 个作业(通过 mapred.jobtracker.completeuserjobs.maximum 属性来配置)。还要注意，作业历史记录是持久化的，所以可以从以前运行过的 jobtracker 中找到作业。

作业的历史记录

作业历史记录涉及已完成作业的事件及其配置。它还记录了作业是否成功完成。作业记录既可在 jobtracker 重启后恢复作业(查看 mapred.jobtracker.restart.recover 属性)，又可为用户提供他感兴趣的作业运行信息。

作业历史记录文件储存在 jobtracker 本地文件系统中日志目录下的 history 子目录中。通过更改 hadoop.job.history.location 属性，可以将历史文件放到自定义的位置。jobtracker 的历史记录文件在被系统删除前会保留 30 天。

在作业输出文件夹的 *_logs/history* 子目录中为用户存储另一个备份。这个路径可以通过设置 hadoop.job.history.user.Location 来覆盖。如果将其设置为特殊值 none，那么尽管作业历史记录被集中保存，也不会有作业历史被保存下来。用户的作业历史记录文件不会被系统删除。

历史日志包括作业、任务和尝试事件，这些都被存储在纯文本文件中。特殊作业的历史记录可以通过网络用户界面或者通过命令行 hadoop job -history(指定的作业输出文件夹) 来查看。

作业页

单击作业 ID 将进入作业页，如图 5-2 所示。在页面的最上方是作业的摘要，包括基本的信息如作业所有者、名字和作业运行时间。作业文件是整理过的配置文件，

包括在作业运行过程中的所有属性和有效值。如果对特定的属性设置没有把握，可以单击查看文件内部内容。

Hadoop job_200904110811_0002 on ip-10-250-110-47

User: root
Job Name: Max temperature
Job File: hdfs://ip-10-250-110-47.ec2.internal/mnt/hadoop/mapred/system/job_200904110811_0002/job.xml
Job Setup: Successful
Status: Running
Started at: Sat Apr 11 08:15:53 EDT 2009
Running for: 5mins, 38sec
Job Cleanup: Pending

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	101	0	0	101	0	0 / 26
reduce	70.74%	30	0	13	17	0	0 / 0

	Counter	Map	Reduce	Total
Job Counters	Launched reduce tasks	0	0	32
	Rack-local map tasks	0	0	82
	Launched map tasks	0	0	127
	Data-local map tasks	0	0	45
FileSystemCounters	FILE_BYTES_READ	12,665,901	564	12,666,465
	HDFS_BYTES_READ	33,485,841,275	0	33,485,841,275
	FILE_BYTES_WRITTEN	988,084	564	988,648
	HDFS_BYTES_WRITTEN	0	360	360
Map-Reduce Framework	Reduce input groups	0	40	40
	Combine output records	4,489	0	4,489
	Map input records	1,209,901,509	0	1,209,901,509
	Reduce shuffle bytes	0	18,397	18,397
	Reduce output records	0	40	40
	Spilled Records	9,378	42	9,420
	Map output bytes	10,282,306,995	0	10,282,306,995
	Map input bytes	274,600,205,558	0	274,600,205,558
	Map output records	1,142,478,555	0	1,142,478,555
	Combine input records	1,142,482,941	0	1,142,482,941
	Reduce input records	0	42	42

Map Completion Graph - close

Reduce Completion Graph - close

Go back to JobTracker

Hadoop, 2009.

图 5-2：作业页的屏幕截图

作业运行期间，可以在本页监视作业进度，本页也会定期更新这些数据。摘要下面是一个有关 map 和 reduce 进度的表格。“Num Task”显示的是这个作业的 map 和 reduce 任务的总数。其他列显示的是任务的状态：“Pending”(等待运行)，“Running”，“Complete”(成功完成)，“Killed”(任务运行失败-这列用“Failed”来标记将更准确一些)，最后一列显示的是在所有 map 和 reduce 任务中失败的任务尝试总数(若任务尝试是一个推测执行的副本，如果它所在的 tasktracker 运行中断，又或者被某用户取消，那么该任务尝试就会被标记为 killed)。

可以在本页的下方找到每个任务进度的完成图。reduce 完成图可以分成三部分：copy(用户 map 输出传输到 reduce 的 tasktracker 上时)、sort(用于输入合并 reduce 时)以及 reduce(用于正在运行的 reduce 函数产生最后输出时)。详情参阅第 6 章)。

在本页中间是作业计数器表。它们在作业运行期间动态更新，为作业进度和整体健康程度提供另一个观察点。在第 8 章将详细介绍计数器。

5.5.4 获取结果

一旦作业完成，有许多方法可以检索结果。每个 reducer 产生一个输出文件，所以在 max-temp 目录中将有 30 份文件，part-00000 到 part-00029。

提示： 理解这些“part”文件，一个简单方法是将其视为 max-temp 文件的部分文件。如果输出文件很庞大(在这个示例中不是如此)，那么将有多个 part，以便多个 reducer 并行工作。通常，文件如果采用这种分区形式，用起来仍然很方便：如作为另一个 MapReduce 作业的输入。在一些情况下，可以设定多分区结构，比如执行一个 map 端连接(详情参见第 8 章)或者一个 MapFile 查找(详情请参见第 8 章)。

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。若需要得到源模式目录中的所有文件，并在本地文件系统上把它们合并成一个单独的文件，可使用 `Hadoop fs` 命令中的-getmerge 选项：

```
% 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
```

由于 reduce 的输出分区是无序的(使用 hashpartitioner 的缘故)，我们可以进行输出排序。对 MapReduce 数据做些后期处理是很常见的，把这些数据放入分析工具，比如 R、电子数据表甚至关系数据库中。

如果检索的输出文件很小，可以使用另一种方法，用-cat 选项将输出文件打印到控制台：

```
% hadoop fs -cat max-temp/*
```

更进一步查看，我们可以看到某些结果有些问题。比如，1951 年(没有在这里显示出来)的最高气温是 590°C！我们怎样找到产生这个结果的原因呢？这是输入数据溢出还是程序中的 bug 呢？

5.5.5 调试作业

调试程序最古老的方法就是通过打印语句，这在 Hadoop 中同样可以实现。然而，新的难题出现了：当程序运行在十台，上百台甚至上千台节点上时，我们如何能找到并检测调试语句的输出呢，况且它可能分散于各节点之间。为了解决这个棘手的问题，我们将一个调试语句记录到一个错误日志中，它将发送一个信息来更新任务的状态信息以提示我们查看错误日志。正如后文所述，网络用户界面使得以上方法更容易实现，后面你将看到。

我们接着创建一个用户计数器来计算在整个数据集中不合理的气温记录总数。这给我们处理以下情况提供了很有价值的信息——如果是一个普通事件，我们可以从中得到此事件发生的条件以及如何提取出气温值而不是简单地丢掉记录。事实上，调试一个作业的时候，应当总在想是否能够使用计数器来获得发生的事件信息。即使需要使用 logging 或者状态信息，使用一个计数器来估计问题的程度也是有帮助的。

如果调试期间产生的日志数据过于庞大，那么你会面临多个选择。第一，可以将这些信息输入到 map 的输出流而不是输入到标准错误流，供 reduce 分析和汇总。这种方法通常需要改变程序结构，所以需要使用其他技术。第二种方法，可以编写一个程序(当然在 MapReduce 中)来分析作业产生的日志。有很多工具可简化此过程，如 Chukwa 工具(这是一个 Hadoop 子项目)。

我们把调试加入 mapper(版本 4)，而不是 reducer，因为我们希望能找到导致这些异常输出的数据源：

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

  enum Temperature {
    OVER_100
  }

  private NcdcRecordParser parser = new NcdcRecordParser();

  public void map(LongWritable key, Text value,
      OutputCollector<Text, IntWritable> output, Reporter reporter)
      throws IOException {

    parser.parse(value);
    if (parser.isValidTemperature()) {
      int airTemperature = parser.getAirTemperature();
      if (airTemperature > 1000) {
        System.err.println("Temperature over 100 degrees for
                           input: " + value);
        reporter.setStatus("Detected possibly corrupt record:
                           see logs.");
        reporter.incrCounter(Temperature.OVER_100, 1);
        }
        output.collect(new Text(parser.getYear()), new
        IntWritable(airTemperature));
    }
  }
}
```

如果气温超过了 100°C(以 1000 表示，因为气温读数是真实气温的十倍)，我们输出一行到标准错误流以代表有问题的行，同时使用 `Reporter` 的 `setStatus()` 方法来更新 map 中的状态信息，提示我们查看日志。我们还增加了一个计数器，它以 java 中的 `enum` 类型来表示。在这个程序中，定义一个 `OVER_100` 字段，统计气温超过 100°C 的记录的数量。

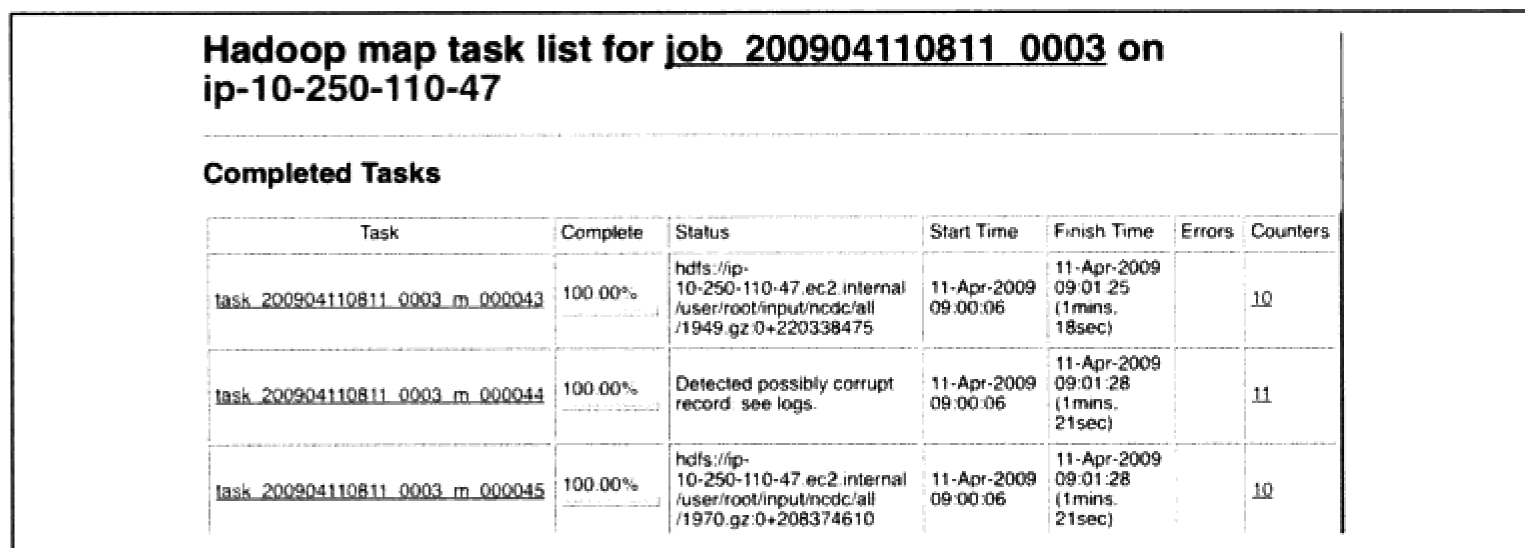
完成这些修改后，我们重新编译代码，重新创建 JAR 文件，然后重新运行该作业，并在运行时，转到任务页查看运行情况。

任务页

任务页包括许多链接，可通过它们详细查看作业中的任务。比如，点击“Map”链

接，你将看到一个页面，所有 map 任务的信息都列在这一页上。还可以只查看已完成的任务。

图 5-3 中的屏幕截图显示了该页中的一部分，即带有调试语句的作业。表中的每行代表一个任务，它提供的信息有：每个任务的开始时间和结束时间；tasktracker 提交的失败报告；查看计算每个任务数的计数器的链接。



Hadoop map task list for job_200904110811_0003 on ip-10-250-110-47

Completed Tasks

Task	Complete	Status	Start Time	Finish Time	Errors	Counters
task_200904110811_0003_m_000043	100.00%	hdfs://ip-10-250-110-47.ec2.internal/user/root/input/ncdc/all/1949.gz:0+220338475	11-Apr-2009 09:00:06	11-Apr-2009 09:01:25 (1mins, 18sec)		10
task_200904110811_0003_m_000044	100.00%	Detected possibly corrupt record: see logs.	11-Apr-2009 09:00:06	11-Apr-2009 09:01:28 (1mins, 21sec)		11
task_200904110811_0003_m_000045	100.00%	hdfs://ip-10-250-110-47.ec2.internal/user/root/input/ncdc/all/1970.gz:0+208374610	11-Apr-2009 09:00:06	11-Apr-2009 09:01:28 (1mins, 21sec)		10

图 5-3：任务页的屏幕截图

与调试相关的是 Status 列，它显示任务的最新状态信息。在任务开始前，它的状态为“initializing”，开始读取记录时，它以字节偏移量和长度作为文件名，显示它正在读取的文件的划分信息。你可以看到我们为任务 `task_200904110811_0003_m_000044` 进行调试时的状态显示，所以点击日志页可得到相关调试信息。(注意，这个任务有一个附加计数器，因为用户计数器有一个非零的计数。)

任务详细信息页

在任务页中，可以点击任何任务获得更多信息。图 5-4 的详细任务信息页显示了每个任务尝试。在这个示例中，只有一个尝试成功完成了任务。此图表提供了十分有用的数据信息，如运行任务尝试的节点及到任务日志文件和计数器的链接。

Action 列包括终止任务尝试链接。默认情况下，这项功能是关闭的，网络用户界面是只读接口。将 `webinterface.private.actions` 设置成 true，即可启用 action 链接。

注意：将 `webinterface.private.actions` 设置为 `true`，意味着允许任何一个访问 HDFS 网络接口的人删除文件。dfs.web.ugi 属性决定 HDFS 网络用户界面的用户，控制哪些文件能被查看或被删除。

Job job_200904110811_0003

All Task Attempts

Task Attempts	Machine	Status	Progress	Start Time	Finish Time	Errors	Task Logs	Counters	Actions
attempt_200904110811_0003_m_000044_0	/default-rack/ip-10-250-163-143.ec2.internal	SUCCEEDED	100.00%	11-Apr-2009 09:00:06	11-Apr-2009 09:01:25 (1mins, 19sec)		Last 4KB Last 8KB All	11	

Input Split Locations

/default-rack/10.250.202.127
/default-rack/10.250.123.223
/default-rack/10.250.115.79

Go back to the job
Go back to JobTracker

Hadoop, 2009

图 5-4：任务详细信息页的屏幕截图

对于 map 任务，页面中还有一部分显示了输入的分块(split)分布在哪些节点上。

通过跟踪其中某成功完成任务的任务尝试的日志文件链接(可以看到每个日志文件的最后 4 KB 或者 8 KB 或者整个文件)，发现其中不合理的输入记录(为使之适于在页面显示，行已进行分行和截断)。

```
Temperature over 100 degrees for input:
0335999999433181957042302005+37950+139117SAO +0004RJSN
V020113590031500703569999994
33201957010100005+35317+139650SAO
+000899999V02002359002650076249N004000599+0067...
```

这个记录看上去与其他记录格式上不同。可能是因为行中有空格，这是定义中没有指定的。

作业完成后，可以查看我们定义的计数器的值，以查看在整个数据集中有多少个超过 100°C 的记录。通过网络用户界面或者命令行，我们可以看到计数器：

```
% hadoop job -counter job_200904110811_0003
'v4.MaxTemperatureMapper$Temperature' \OVER_100
3
```

`-counter` 选项选取了作业的 ID、计数器的组名(这里一般是类名)和计数器名称(enum 名)。在超过十亿记录的整个数据集中、只有三个异常记录。对于许多大数据问题，挑选出不符合的记录是一个标准，我们需要谨慎处理这种情况，因为寻找的是一个极限值-最高气温值，而不是一个总的度量。抛弃三个记录也许不会改变结果。

Hadoop 用户日志

Hadoop 为不同的用户在不同的地方产生日志。如表 5-2 所示。可以使用 Chukwa (Hadoop 的一个子项目)将这些文件中的大多数文件汇总在一起，然后进行分析。

表 5-2：Hadoop 日志

日志	用户群	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 和另一个合并标准输出和错误流的文件。它们分别写入 HADOOP_LOG_DIR 变量定义的目录	第 9 章
HDFS 审计日志	管理员	记录所有 HDFS 请求的日志，默认是关闭的。虽然这是可以配置的，但是一般写入节点的日志中	第 10 章
MapReduce 作业历史日志	用户	在运行作业期间发生的事件记录日志(如任务完成时间)，保存在 jobtracker 中_logs/history 子目录中的作业输出目录中	本章“工作历史”小节
MapReduce 任务日志	用户	每个 tasktracker 子进程都使用 log4j 产生一个日志文件(syslog)，它是数据用来输出到标准输出流(stdout)和标准错误流(stderr)的文件。写入到 HADOOP_LOG_DIR 环境变量定义的 userlogs 的子目录中	参见下一小节

如本小节所述，MapReduce 任务日志可通过网络用户界面访问，这也是查看日志最简单的方法。也可以在运行任务尝试的 tasktracker 的本地文件系统的任务尝试目录中找到日志文件。如果开启任务 JVM 重用 (见“任务 JVM 重用”小节)，一个日志文件汇总整个 JVM 运行的日志信息，因此每个日志文件可包含多个任务尝试。网络用户界面隐藏了这些信息，只显示与被查看任务尝试相关的部分。

这些日志文件的内容很容易理解。任何写到标准输出流或者标准错误流中的数据都将直接写到相关的日志文件中。(当然，在 Streaming 中，标准输出流被用

来作为 map 或者 reduce 的输出，所以它没有显示在标准输出流日志中)

在 Java 中，可以使用 Apache Commons Logging API 编写任务的 syslog 文件。在这个示例中，由 log4j 实际记录日志：相关附加 log4j 文件叫 TLA(Task Log Appender)，它位于 Hadoop 配置目录下的 log4j.properties 文件中。

有一些控制用于管理任务日志的大小和记录保留时间。默认情况下，日志将在最短 24 小时后被删除(可用 mapred userlog.retain.hours 属性来设置)。还可以用 mapred.userlog.limit.kb 属性为每个日志文件的最大值设置一个阈值，默认为 0，这意味着没有阈值限制。

处理不合理的数据

能捕获到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。

```
@Test
public void parsesMalformedTemperature() throws IOException {
  MaxTemperatureMapper mapper = new MaxTemperatureMapper();
  Text value = new Text("0335999999433181957042302005+37950+
    139117SAO +0004" +          // Year ^^^^
    "RJSN V02011359003150070356999999433201957010100005+353");
                               // Temperature ^^^^^
  OutputCollector<Text, IntWritable> output =
    mock(OutputCollector.class);
  Reporter reporter = mock(Reporter.class);

  mapper.map(null, value, output, reporter);

  Text outputKey = anyObject();
  IntWritable outputValue = anyObject();
  verify(output, never()).collect(outputKey, outputValue);
  verify(reporter).incrCounter(MaxTemperatureMapper.
    Temperature.MALFORMED, 1);
}
```

引发问题的原因是这条记录与其他行的格式不同。例 5-11 显示了修改过的程序(版本 5)，它使用解析器忽略了那些没有首符号(+或者-)的气温字段。我们还引入一个计数器来测量因为这种原因而被忽略的记录的数量。

例 5-11：找到最高气温的 mapper

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

  enum Temperature {
    MALFORMED
  }

  private NcdcRecordParser parser = new NcdcRecordParser();

  public void map(LongWritable key, Text value,
      OutputCollector<Text, IntWritable> output, Reporter reporter)
      throws IOException {

    parser.parse(value);
    if (parser.isValidTemperature()) {
      int airTemperature = parser.getAirTemperature();
      output.collect(new Text(parser.getYear()), new
        IntWritable(airTemperature));
    } else if (parser.isMalformedTemperature()) {
      System.err.println("Ignoring possibly corrupt input: " + value);
      reporter.incrCounter(Temperature.MALFORMED, 1);
    }
  }
}
```

5.5.6 使用远程调试器

当一个任务失败并且没有记录足够信息来诊断错误时，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，因为你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。反之，可以设置运行作业的属性来指导 Hadoop 保留作业运行期间产生的所有中间值。这些数据可以单独在调试器上重新运行那些出错的任务。注意，如果任务在原处运行，且仍然在发生错误的节点上，这会增加错误重现的几率。

首先将 keep.failed.task.files 中的配置属性设置为 true，以便在任务失败时，tasktracker 能保留足够的信息让任务能在相同的输入数据上重新运行。然后再次运行作业，并使用 Web 用户界面看任务是在哪个节点失败的，看一看任务尝试 ID(从字符串 attempt_开始)。

接着需要运行一个特殊的任务运行器即 IsolationRunner，用前面保留的文件作为输入。登录到任务失败的节点，寻找任务尝试目录。它可能是在本地 MapReduce 目录下的其中一个，由 mapred.local.dir 属性来设置的(详情参见第 9 章)。如果这个属性是一个以逗号分隔的目录表(将负载分布到机器的物理磁盘上)，你也许需要查看所有的目录来找到那个任务尝试的目录。特定任务尝试目录的路径如下所示：

mapred.local.dir/taskTracker/jobcache/job-ID/task-attempt-ID

这个目录包含多个文件和子目录，包括 job.xml，包含任务尝试期间生效的所有作业配置属性，这些配置属性将被 IsolationRunner 用来创建一个 JobConf 实例。对于 map 任务，这个目录还包含一个含有输入划分文件序列化表示的文件，所以相同的输入数据还可供任务使用，以便任务取得相同的输入数据。对于 reduce 任务，则有一个 map 输出备份(形成 reducr 的输入)则储存在 output 目录中。

还有一个目录叫 work，它是任务尝试的工作目录。我们将其改为这个目录以运行 IsolationRunner。需要设置一些选项来连接远程调试器：[①]

```
% export HADOOP_OPTS="-agentlib:jdwp=transport=dt_socket,
server=y,suspend=y,\address=8000"
```

suspend=y 选项让 JVM 在运行代码前先等待调试器连接。用以下命令启动 IsolationRunner：

```
% hadoop org.apache.hadoop.mapred.IsolationRunner ../job.xml
```

下一步，设置断点，连接远程调试器(所有主流的 JAVA IDE 都支持远程调试，可参阅说明文档)，然后任务会在你的控制下运行。可以这样重新运行任务任意多次。幸运的话，可以找到并修改错误。

在这个过程中，可以使用其他标准的 Java 调试技术，如 kill-QUIT *pid* 或 jstack 来进行线程转储。

广而言之，有必要知道这种技术并不只适用于失败的任务。还可以保留成功完成任务的中间值，这样一来，即可方便地检查任务有没有失败。这时，将属性 keep.task.files.pattern 设置为一个正则表达式(与要保留的任务 ID 匹配)即可。

---

① 可以在 Java Platform Debugger Architecture 网页(网址为 *http://java.sun.com/javase/6/docs/technotes/guides/jpda/*)找到调试选项的更多信息。

5.6 作业调优

作业运行后，许多开发人员可能提出疑问："能够让它运行得更快一些吗？"有一些 Hadoop 专门的"疑点"值得检查一下，看看是不是它们引起的性能问题。开始在任务级别分析或优化之前，应该先看看表 5-3 所示的检查表。

表 5-3：作业调优检查表

优化内容	最佳实践	更多信息
mapper 的数量	运行 mapper 需要多长时间？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	第 7 章的"小文件和 CombineFile-InputFormat"小节
reducer 的数量	为了达到最高性能，reducer 的数目应该比 reducer 槽的数目稍微少一点。这将使 reducer 能够同一波中完成任务，并在 reducer 阶段充分使用集群	第 7 章的"选择 reducer 的数量"小节
combiner	作业能否充分利用 combiner 来减少通过 shuffle 传输的数据	第 2 章的"combiner"小节
中间值的产生	对 map 输出进行压缩能使作业执行更快	第 4 章的"压缩 map 输出"小节
自定义序列	如果正在使用自己定义的 Writable 对象或自定义的 comparator，则必须确保已实现 RawComparator	第 4 章的"迅速实现 RawComparator"小节
shuffle 运行	MapReduce shuffle 可以对一些内存管理的参数进行调整，将弥补性能的不足	第 6 章的"配置调优"小节

分析任务

正如调试一样，对分布式系统上运行的作业进行分析就像在 MapReduce 上遇到的挑战一样。Hadoop 允许你分析作业中的任务，并且在每个任务完成时，把分析好的信息放到机器中以便日后使用标准分析工具进行分析。

当然，对本地作业运行器上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差异很大的环境，并且数据流形式是不同的。如果 MapReduce 作业是 I/O 密集型的(很多作业都是)，那么更改代码来优化 CPU 的性能是没有意义的。为了保证所有调整都是有效的，应该在实际集群上对比新的执行时间和旧的执行时间。这说起来容易，做起来难，因为作业执行时间的改变取决于与其他作业的资源争夺以及 scheduler(调度器)决定的任务顺序。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(改变代码或者不改变代码)，并检查是否有明显的改进。

很不幸，有些问题(如内存溢出)只发生在集群上运行时，并在这些情况下，必然需要对发生问题的地方进行分析。

HPROF 分析工具

有许多配置属性可以控制分析过程，这些属性也可以通过 JobConf 方法简单实现。下面对 MaxTemperatureDrive(版本 6)的修改进行远程 HPROF 分析。HPROF 是一个 JDK 自带的分析工具，虽然只有基本功能，但是同样能提供程序 CPU 运行和堆使用情况等相关有用信息。[①]

```
conf.setProfileEnabled(true);
conf.setProfileParams("-agentlib:hprof=cpu=samples,heap=sites,
    depth=6," +"force=n,thread=y,verbose=n,file=%s");
conf.setProfileTaskRange(true, "0-2");
```

第一行启用了分析工具，默认为关闭(这相当于把 mapred.task.profile 配置属性设置为 true)。

接下来设置分析参数，即传到任务 JVM 的额外的命令行参数。(一旦启用分析，即使启用 JVM 重用，也会给每个任务分配一个新的 JVM。详见第 6 章的“任务 JVM 重用”小节)默认的参数定义了 HPROF 分析器，这里我们设置一个额外的 HPROF 选项，depth=6，以便能达到更深的栈跟踪深度(相较于 HPROF 默认模式)。JobConf 的 setProfileParams()方法相当于设置 mapred.task.profile.params。

最后，指定希望分析的任务。我们一般只需要来自某些任务的分析信息，所以使用 setProfileTaskRange()方法来定义想要分析的任务 ID 的范围。我们将其设置为 0-2(默认情况下)，这意味着 ID 为 0、1、2 的任务将被分析。第一个传进 setProfileTaskRange()方法的参数指明这是 map 任务的范围还是 reduce 任务的

① HPROF 使用字节码插入来分析代码，所以不必用特殊选项来重新编译应用程序。要想进一步了解 HPROF，请参见 kelly o'Hair 的“HPROF:A Heap/CPU profiling Tool in JESE 5.0，网址为 *http://java.sun.com/developer/technical/Articals/Programming/HPROF:html*”。

范围：true 代表 map 的，false 代表 reduce 的。范围是可以设置的，使用一个标注可以表示开放范围。例如，0-1、4、6-将指定除了 ID 为 2、3、5 之外的所有任务。对于要分析的 map 任务，可以使用 mapred.task.profile.maps 属性来控制，reduce 任务则由 mapred.task.profile.reduces 控制。

使用修改过的驱动程序来运行作业时，分析结果将出现在启动作业的文件夹下该作业的末尾。因为我们只分析了一小部分任务，所以可以在数据集的子集上运行该作业。

下面是一个 mapper 分析文件的一小段，它显示了 CPU 的抽样信息。

```
CPU SAMPLES BEGIN (total = 1002) Sat Apr 11 11:17:52 2009
rank self accum count trace method
1 3.49% 3.49% 35 307969 java.lang.Object.<init>
2 3.39% 6.89% 34 307954 java.lang.Object.<init>
3 3.19% 10.08% 32 307945 java.util.regex.Matcher.<init>
4 3.19% 13.27% 32 307963 java.lang.Object.<init>
5 3.19% 16.47% 32 307973 java.lang.Object.<init>
```

交叉引用跟踪号 307973 显示了同一文件的栈追踪轨迹。

```
TRACE 307973: (thread=200001)
    java.lang.Object.<init>(Object.java:20)
    org.apache.hadoop.io.IntWritable.<init>(IntWritable.java:29)
    v5.MaxTemperatureMapper.map(MaxTemperatureMapper.java:30)
    v5.MaxTemperatureMapper.map(MaxTemperatureMapper.java:14)
    org.apache.hadoop.mapred.MapRunner.run(MapRunner.java:50)
    org.apache.hadoop.mapred.MapTask.runOldMapper(MapTask.java:356)
```

因此可以看出，mapper 花了 3%的时间来构建 IntWritable 对象。这个发现表明重用输出的 Writable(版本 7)实例是有价值的：

例 5-12：重用文本和 IntWritable 输出对象

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

  enum Temperature {
    MALFORMED
  }

  private NcdcRecordParser parser = new NcdcRecordParser();
  private Text year = new Text();
  private IntWritable temp = new IntWritable();
```

```
  public void map(LongWritable key, Text value,
      OutputCollector<Text, IntWritable> output, Reporter reporter)
      throws IOException {

    parser.parse(value);
    if (parser.isValidTemperature()) {
      year.set(parser.getYear());
      temp.set(parser.getAirTemperature());
      output.collect(year, temp);
    } else if (parser.isMalformedTemperature()) {
      System.err.println("Ignoring possibly corrupt input: " + value);
      reporter.incrCounter(Temperature.MALFORMED, 1);
    }
  }
}
```

然而，我们知道只有在整个数据集上运行的运行作业有明显提升，这才是有意义的。在其他空闲的 11 个节点的集群上运行每个修改过的版本五次，统计结果显示作业执行时间并没有明显不同。当然，这只是针对具体的代码、数据和硬件的综合结果，应该在具体环境中以相同的基准来运行，然后看这样的更改在具体配置下是否效果显著。

其他分析工具

本书写作时，获取分析输出的机制是 HPROF 专有的。在此改进之前，我们可以使用 Hadoop 的分析设置触发其他分析器进行分析 (详见具体的分析工具的文档)，这需要我们手动来检索 tasktracker 的分析输出来进行分析。

如果没有在所有 tasktracker 机器上安装分析工具，可以考虑使用分布式缓存(详见第 8 章的“分布式缓存”小节)便能在需要的机器使用分析工具的二进制包。

5.7 MapReduce 的工作流

至此，你已经知道如何使用 MapReduce 来编写程序。我们目前还未考虑过如何将一个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)。处理过程更加复杂时，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 MapReduce 之上的高级语言，如 Pig、HIVE 或 Cascading。一个直接的好处是它让你免于处理到 MapReduce 作业的转换，而是将重心集中于正在进行的分析上。

5.7.1 将问题分解成 MapReduce 作业

让我们看一个更复杂的问题，我们想把它将其转换成 MapReduce 工作流。

假设我们想找到每个气象台年度每天平均气温最高的记录。例如，要计算 029070-99999 气象台的 1 月 1 日的平均每日最高气温的记录，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平均气温最大的值。

我们如何使用 MapReduce 来计算它呢？计算自然分成下面两部分。

1. 计算每对 station-date 的每日最高气温。
   在本例中的 MapReduce 程序是最高气温程序，不同在于本例中的键是一个综合的 station-date 对，而不仅是年份。
2. 计算每个 station-day-month 键的平均每日最高气温。
   mapper 从上一个作业得到输出记录(station-date，最高气温值)，并丢掉年份将其值赋予到记录(station-day-month，最高气温值)中。然后 reduce 为每个 station-day-month 键得到平均最高气温值。

第一部分的输出看上去就是我们想要的气象台的结果。(示例的 mean_max_daily_temp.sh 脚本提供了 Hadoop Streaming 中的一个实现)

```
029070-99999 19010101 0
029070-99999 19020101 -94
...
```

前两个字段来自键，最后一列是指定的所有气象台和日期读入数据中的最高气温。第二阶段，计算这些年份最高气温的平均值：

```
029070-99999 0101 -68
```

以上是气象台 029070-99999，整个世纪 1 月 1 日平均每日最高气温-6.8°C。

只用一个 MapReduce 过程就能完成这个计算，但它可能会让部分程序员花更多

精力。[1]

设置更多 MapReduce 过程阶段将导致更多可分解的、可维护的 mapper 和 reducer。第 14 章中的案例学习包括大量使用 Map-Reduce 来解决的实际问题，在每个例子中，数据处理任务都是使用两个或者更多的 MapReduce 作业来实现的。第 14 章的详细介绍，对于理解如何将问题分解成一个 MapReduce 工作流没有多大意义。

相较于我们已经做的，map 和 reduce 函数完全可以进一步分解。mapper 一般执行输入格式分析、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。使用 `ChainReducer`，你可以运行一系列的 mapper，再运行一个 reducer 和该 MapReduce 作业中的其他 mapper 链。

5.7.2 运行独立的作业

MapReduce 作业流中的作业不止一个时，问题会随之而来：如何管理这些作业，让它们按顺序执行？有几种方法，其中主要要考虑的是否有一个线性的工作链或一个更复杂的作业的有向无环图(directed acyclic graph，DAG)。

对于一个线性的链，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()`方法就会抛出一个 `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，并将前一个作业输出的中间数据清除掉。

对于比线性链更复杂的问题，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，甚至是一次性的作业)。最简单的是 `org.apache.hadoop.mapred.jobcontrol` 包中的：`JobControl` 类。`JobControl` 的实例有一个作业的运行图，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

---

① 尝试这样的处理是十分有趣的。提示：可参阅第 8 章的“二次排序”小节。

作业失败，JobControl 将不执行与之有依赖关系的作业。

不同于在客户端运行并提交作业的 JobControl，Hadoop Workflow Scheduler (Hadoop 工作流调度器，HWS)[①]是作为服务器运行的，客户端将工作流提交给调度器。工作流完成后，调度器将发送一个 HTTP 回复到客户端，通知其作业状态。HWS 可以在同一工作流中运行不同类型的作业，比如，先运行一个 Pig 作业，然后再运行一个 java MapReduce 作业。

① 本书写作期间，Hadoop 工作流调度器(Hadoop Workflow Scheduler)依旧在发展中。详见 *https://issues.apache.org/jira/browse/HADOOP-5303*。

第 6 章

MapReduce 的工作原理

在本章中，我们将深入了解 Hadoop 中 MapReduce 的工作原理。这些知识将为我们在第 7 章和第 8 章学习编写更高级的 MapReduce 程序奠定良好的基础。

6.1 运行 MapReduce 作业

你可以用一行代码来运行一个 MapReduce 作业：JobClient.runJob(conf)。它很简短，但其幕后隐藏着大量的处理细节。本小节将揭示 Hadoop 运行作业时所采取的步骤。

整个过程如图 6-1 所示。在最上层，有 4 个独立的实体。

- 客户端，提交 MapReduce 作业。
- jobtracker，协调作业的运行。jobtracker 是一个 Java 应用程序，它的主类是 `JobTracker`。
- tasktracker，运行作业划分后的任务。tasktracker 是一个 Java 应用程序，它的主类是 `TaskTracker`。
- 分布式文件系统(一般为 HDFS，请参见第 3 章)，用来在其他实体间共享作业文件。

6.1.1 提交作业

`JobClient` 的 `runJob()` 方法是用于新建 `JobClient` 实例和调用其 `submitJob()` 方

法的简便方法(参见图 6-1 的步骤 1)。提交作业后，runJob()将每秒轮询作业的进度，如果发现与上一个记录不同，便把报告显示到控制台。作业完成后，如果成功，就显示作业计数器。否则，导致作业失败的错误会被记录到控制台。

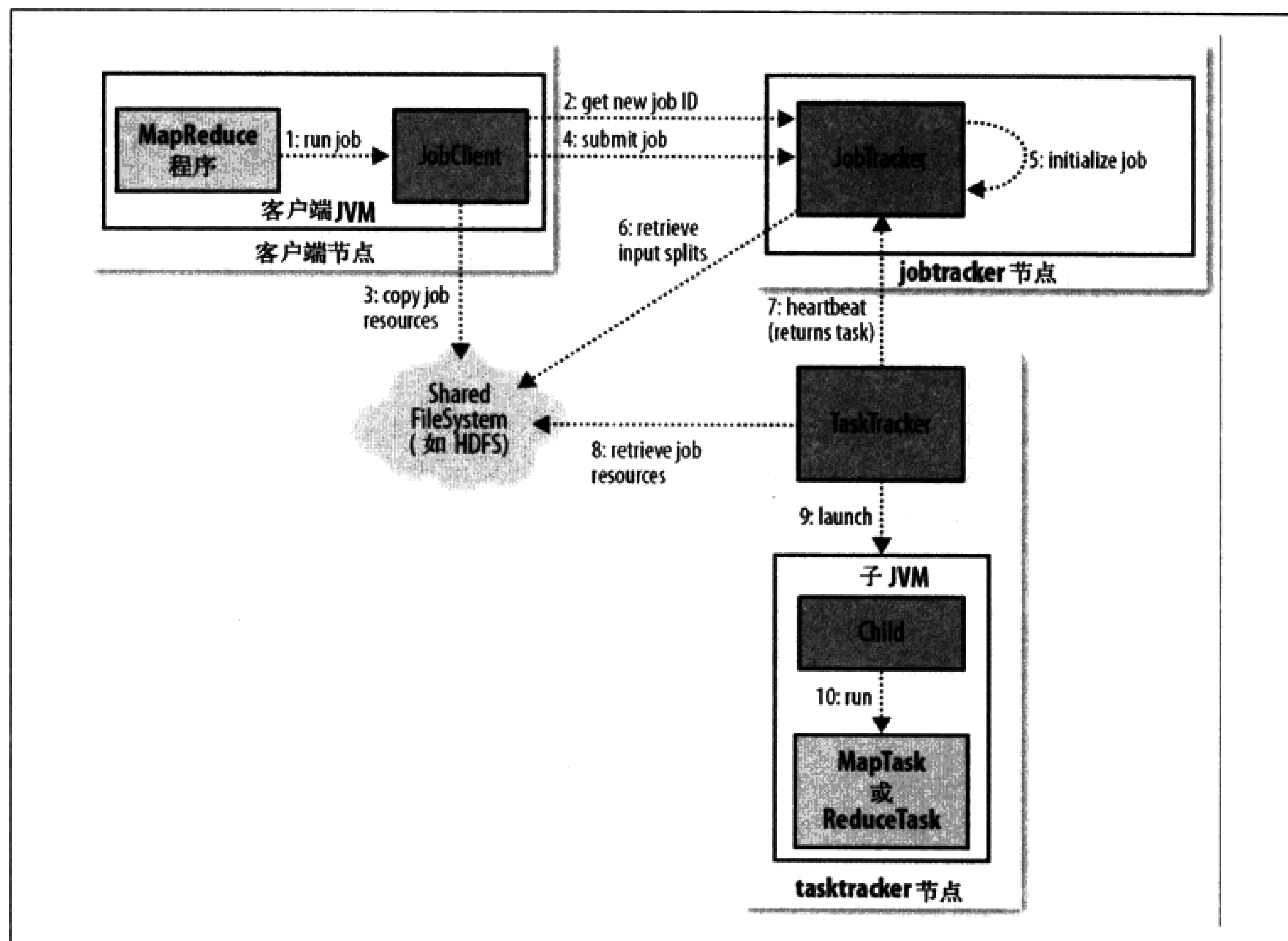


图 6-1：Hadoop 运行 MapReduce 作业的工作原理

JobClient 的 submitJob()方法所实现的作业提交过程如下。

1. 向 jobtracker 请求一个新的作业 ID(通过调用 JobTracker 的 getNewJobId())(步骤 2)。
2. 检查作业的输出说明。比如，如果没有指定输出目录或者它已经存在，作业就不会被提交，并有错误返回给 MapReduce 程序。
3. 计算作业的输入划分。如果划分无法计算，比如因为输入路径不存在，作业就不会被提交，并有错误返回给 MapReduce 程序。
4. 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划分——复制到一个以作业 ID 号命名的目录中 jobtracker 的文件系统。作业 JAR 的副本较多(由 mapred.submit.replication 属性控制，默认为

10)，如此一来，在 tasktracker 运行作业任务时，集群能为它们提供许多副本进行访问。(步骤 3)。

5. 告诉 jobtracker 作业准备执行 (通过调用 `JobTracker` 的 `submitJob()` 方法)(步骤 4)。

6.1.2 作业的初始化

`JobTracker` 接收到对其 `submitJob()` 方法的调用后，会把此调用放入一个内部的队列中，交由作业调度器进行调度，并对其进行初始化。初始化包括创建一个代表该正在运行的作业的对象，它封装任务和记录信息，以便跟踪任务的状态和进程(步骤 5)。

要创建运行任务列表，作业调度器首先从共享文件系统中获取 `JobClient` 已计算好的输入划分信息(步骤 6)。然后为每个划分创建一个 map 任务。创建的 reduce 任务的数量由 `JobConf` 的 `mapred.reduce.tasks` 属性决定，它是用 `setNumReduceTasks()` 方法来设置的，然后调度器便创建这么多 reduce 任务来运行。任务在此时指定 ID 号。

6.1.3 任务的分配

TaskTracker 执行一个简单的循环，定期发送心跳(heartbeat)方法调用 Jobtracker。心跳方法告诉 jobtracker，tasktracker 是否还存活，同时也充当两者之间的消息通道。作为心跳方法调用的一部分，tasktracker 会指明它是否已经准备运行新的任务，如果是，jobtracker 会为它分配一个任务，并使用心跳方法的返回值与 tasktracker 进行通信(步骤 7)。

在 jobtracker 为 tasktracker 选择任务之前，jobtracker 必须先选定任务所在的作业。本章后面将解释多个调度算法(详见第 7 章的“作业调度”小节)，但是默认的方法是简单维护一个作业优先级列表。选择好作业后，jobtracker 就可以为该作业选定一个任务。

针对 map 任务和 reduce 任务，tasktracker 有固定数量的槽。例如，一个 tasktracker 可能可以同时运行两个 map 任务和两个 reduce 任务。(准确数量由 tasktracker 核的数量和内存大小来决定，详见第 9 章的“内存”小节)。默认调度器在处理 reduce 任务槽之前，会填满空闲的 map 任务槽，因此，如果 tasktracker 至少有一个空闲的 map 任务槽，jobtracker 会为它选择一个 map 任务；否则选择一个 reduce 任务。

要选择一个 reduce 任务，jobtracker 只是简单地从尚未运行的 reduce 任务列表中选取下一个来执行，并没有考虑数据的本地化。然而，对于一个 map 任务，它考虑的是 tasktracker 的网络位置和选取一个距离其输入划分文件最近的 tasktracker。在最理想的情况下，任务是 data-local(数据本地化)的，与分割文件所在节点运行在相同的节点上。同样，任务也可能是 rack-local(机架本地化)的：和分割文件在同一个机架，但不在同一节点。一些任务既不是数据本地化的，也不是机架本地化的，从与它们自身运行的不同机架上检索数据。可以通过查看作业的计数器得知每种类型任务的比例(详见第 8 章的“内置计数器”小节)。

6.1.4 任务的执行

现在，tasktracker 已经被分配了任务，下一步是运行任务。首先，它本地化作业的 JAR 文件，将它从共享文件系统复制到 tasktracker 所在的文件系统。同时，将应用程序所需要的全部文件从分布式缓存复制到到本地磁盘(详见第 8 章的“分布式缓存”小节)(步骤 8)。然后，为任务新建一个本地工作目录，并把 JAR 文件中的内容解压到这个文件夹下。第三步，新建一个 `TaskRunner` 实例来运行任务。

`TaskRunner` 启动一个新的 Java 虚拟机(步骤 9)来运行每个任务(步骤 10)，使得用户定义的 map 和 reduce 函数的任何缺陷都不会影响 tasktracker(比如导致它崩溃或者挂起)。但在不同的任务之间重用 JVM 还是可能的(详见本章的“任务 JVM 重用”小节)。

子进程通过 umbilical 接口与父进程进行通信。它每隔几秒便告知父进程它的进度，直到任务完成。

流和管道

流和管道都运行特殊的 map 和 reduce 任务，目的是运行用户提供的可执行程序，并与之通信。

应用流时，流任务使用标准输入和输出流，与进程(可以用任何语言编写)进行通信。另一方面，管道任务则监听套接字，发送其环境中的一个端口号给 C++进程，如此一来，在开始时，C++进程即可建立一个与其父 Java 管道任务的持续连接套接字。

在这两种情况下，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 或者 reduce 函数来执行它并把输出的键/值对传回 Java 进程。从 tasktracker 的角度看，就像 tasktracker 的子进程在运行自己的 map 或者 reduce 代码。

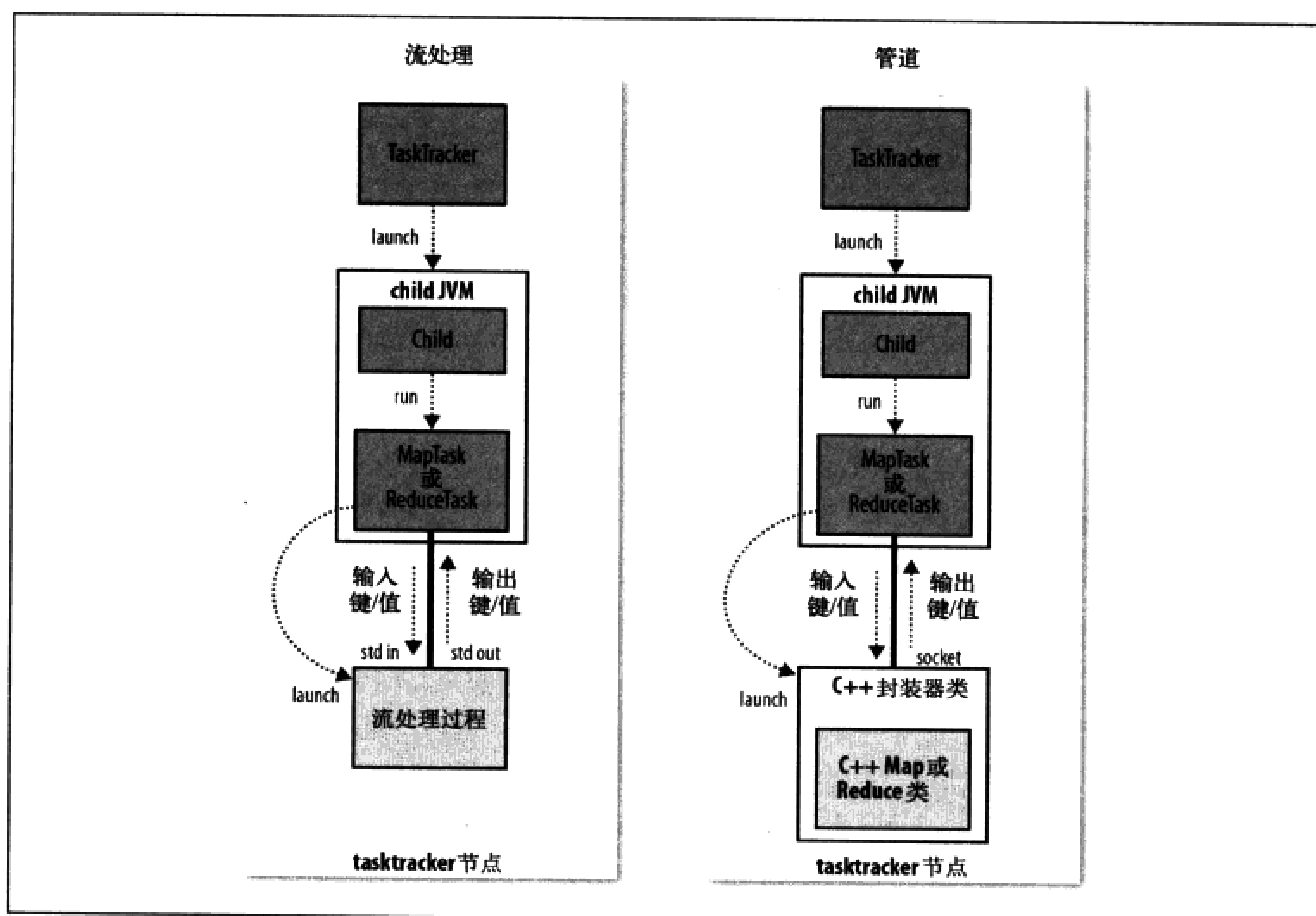


图 6-2：tasktracker 中执行的流和管道及其子进程的关系

6.1.5 进度和状态的更新

MapReduce 作业是一个长时间运行的批量作业，可以运行数秒甚至数小时。由于这是一个很长的时间段，所以对于用户而言，能够得知作业进展是很重要的。一个作业和它的每个任务都有一个状态，包括：作业或者任务的状态(比如，运行，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，那么它们是如何与客户端通信的呢？

任务正在运行时，对任务进度(即任务完成率)保持追踪。对于 map 任务，这是已处理完输入的百分比。对于 reduce 任务，则稍微有点复杂，但程序仍然会估计 reduce 输入已处理的百分比。整个过程分成三部分，与 shuffle 的三个阶段相对应(详见“shuffle 和排序”)。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。因为已经完成拷贝和排序阶段(每个占 1/3)，并且已经完成 reduce 阶段的一半(1/6)。

MapReduce 的进度如何构成？

进度并不总是可测量的，然而它能告诉 Hadoop 有任务正在运行。比如，写输出记录的这个任务也用进度来表示，尽管它不能用百分比这样的数字来表示已经写了的记录，因为即使通过任务来产生输出，也无法知道后面的情况。

进度报告很重要，因为它意味着 Hadoop 并没有让正在运行的任务停下来。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)。
- 在一个报告中设置状态描述(使用 Reporter 的 setStatus()方法)。
- 增加一个计数 (使用 Reporter 的 incrCounter()方法)。
- 调用 Reporter 的 progess()方法。

任务也有一组计数器可以对任务运行过程中各个事件进行计数(详见第 1 章的“运行测试”示例)，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。

如果任务报告了进度，便会设置一个标志以表明状态变化将被发送到 tasktracker。在另一个线程中，每隔三秒检查此标志一次，如果已设置，则告知 tasktracker 当前任务状态。同时，tasktracker 每隔五秒发送心跳到 jobtracker(5 秒这个间隔是最小值，因为心跳间隔是由集群的大小来决定的：对于一个更大的集群，间隔会更长一些)，并且在此调用(指心跳调用)中，所有由 tasktracker 运行的任务，它们的状态都会被发送至 jobtracker。计数器的发送间隔通常大于 5 秒，因为计数器占的带宽相对较高。

jobtracker 将这些更新合并起来，产生一个全局视图，表明正在运行的所有作业及其所含任务的状态。最后，正如前面提到的，`JobClient` 通过每秒查看 jobtracker 来接收最新状态。客户端也可以使用 `JobClient` 的 `getJob()` 方法来得到一个 `RunningJob` 的实例，后者包含作业的所有状态信息。

6.1.6 作业的完成

jobtracker 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 `JobClient` 查询状态时，它将得知任务已成功完成，所以便显示一条消息告知用户，然后从 `runJob()` 方法返回。

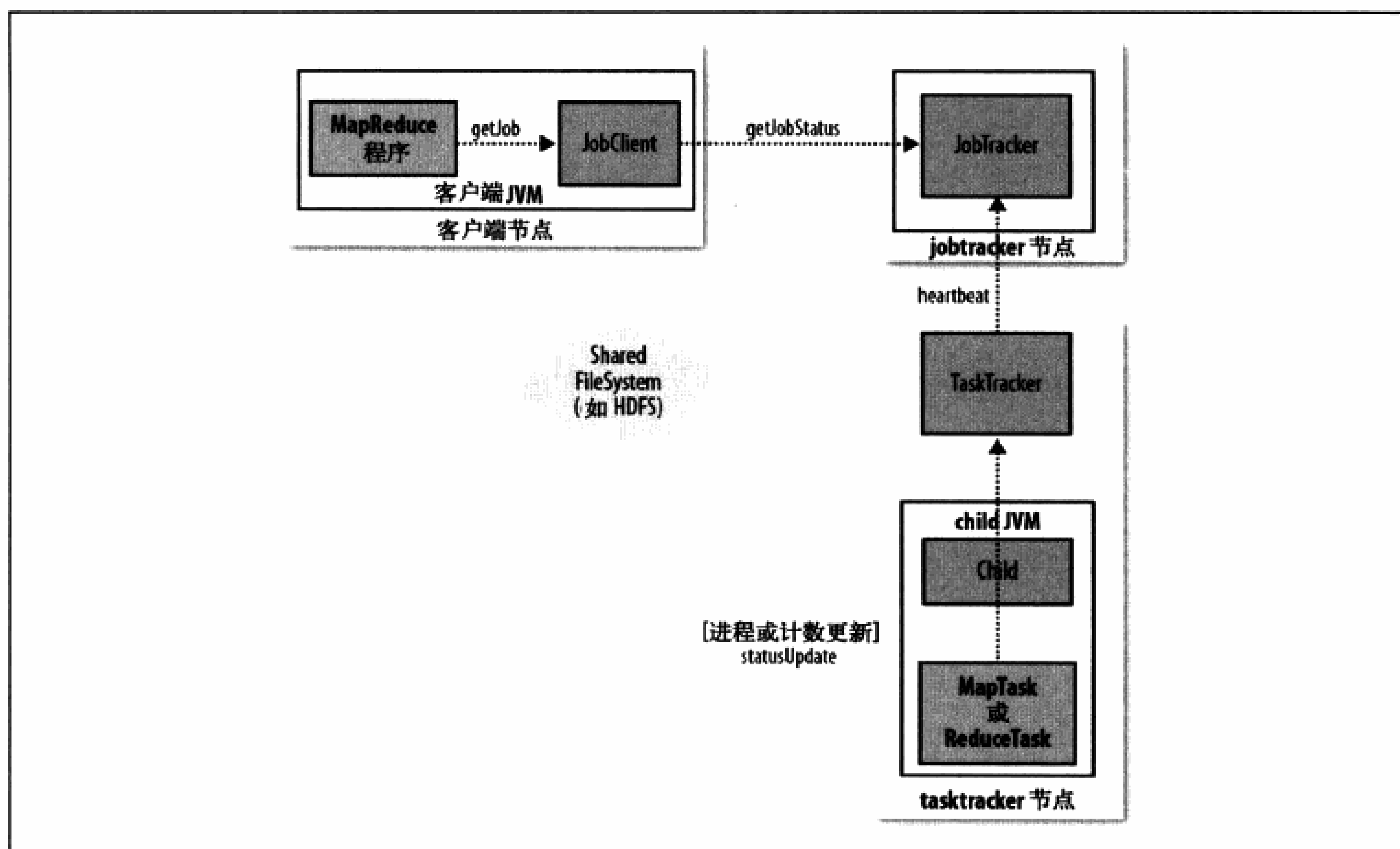


图 6-3：状态更新在 MapReduce 系统中的传播方式

如果 jobtracker 有相应的设置，也会发送一个 HTTP 作业通知。希望收到回调(指 HTTP 作业通知的回调)的客户端可以通过 `job.end.notification.url` 属性来设置。

最后，jobtracker 清空作业的工作状态，指示 tasktracker 也清空作业的工作状态(比如删除中间输出)。

6.2 失败

在真实环境中，用户代码存在缺陷，进程会崩溃，机器会产生故障。使用 Hadoop 最主要的好处之一是它能处理此类故障而使作业得以顺利完成。

6.2.1 任务失败

首先考虑子任务失败的情况。最常见的情况是 map 或 reduce 任务中的用户代码抛出运行时异常。如果发生这种情况，子任务 JVM 进程会在退出之前向其 tasktracker 父进程发送错误报告。错误报告最后被记入用户日志。tasktracker 会将此次任务尝试(task attempt)标记为 failed(失败)，释放一个槽以便运行另外一个任务。

对于流任务，如果流进程以非零退出代码退出运行，则会被标记为 failed(失败)。这种行为是由 stram.non.zero.exit.is.failure 属性(默认值为 true)决定的。

另一种错误情况是子 JVM 突然退出——可能有 JVM 错误，由 MapReduce 用户代码某些特殊原因而造成 JVM 退出。在这种情况下，tasktracker 会注意到进程已经退出，并将此次尝试标记为 failed(失败)。

对于任务的挂起，处理方式则有不同。tasktracker 注意到自己已经有一段时间没有收到进度更新，因此进而将任务标记为 failed(失败)。在此之后，子 JVM 进程将被自动杀死。[①]任务失败的超时间隔通常为 10 分钟，这可按照每个作业的方式进行设置(或按照每个集群的方式进行设置)，具体做法是把 mapred.task.timeout 属性设置为一个毫秒为单位的值。

将超时间隔设置为 0 将关闭超时判定，因而造成长时间运行的任务永远不会被标记为 failed。在这种情况下，被挂起的任务永远不会释放它的槽，最终降低整个集群的效率。因此，尽量避免采用这种设置，同时确保任务能够定期汇报其进度。(见本章补充知识“MapReduce 的进度如何构成”)

jobtracker 被通知一个任务尝试失败时(通过 tasktracker 的心跳调用实现)，它将重新调度该任务的执行。jobtracker 会尝试避免重新调度之前失败过的 tasktracker 上的任务。此外，如果一个任务的失败次数超过 4 次，它将不会再被重试。这个值是可以设置的：尝试运行任务的最多次数，对于 map 任务，是由 mapred.map.max.attempts 属性控制，而对于 reduce 任务，则由 mapred.reduce.max.attempts 属性控制。在默认情况下，如果有任何任务失败次数大于 4 (或被配置的某个值)，整个作业都会失败。

对于一些应用程序，我们不希望一旦有任务失败就中止运行整道作业，因为即使任务失败，其结果可能还是可用的。在这种情况下，可以通过 mapred.max.map.failures.percent 和 mapred.max.reduce.failures.percent 属性来设置在不触发任务失败的情况下允许任务失败的最多次数。

任务尝试也是可以杀死的，这不同于在它失败时被杀死。任务尝试被杀死可能是由于它是一个推测副本(相关详情可参见本章的“推测式执行”)，或因为它所处的 tasktracker 失败了，这样一来，jobtracker 便将在它上面运行的所有任务尝试标记为 killed。被杀死的任务尝试不会被计入任务运行尝试次数(由 mapred.map.max.attempts 和

① 如果一个流进程被挂起，tasktracker 并不会尝试终止它(即使运行它的 JVM 将会被终止)，因此应该为这种情况做好预防措施，使用其他的方法来终止孤儿进程。

mapred.reduce.max.attempts 设置)，因为这不是由于任务失败而导致的尝试禁止。

用户也可以使用网络用户界面或命令行(输入 hadoop job 来查看相应的选项)来杀死或取消任务执行尝试。也可以采用前面描述的相同的机制来杀死作业。

6.2.2 tasktracker 失败

tasktracker 失败是另一种失败形式。如果某 tasktracker 由于崩溃或运行过于缓慢而失败，它将停止向 jobtracker 发送心跳(或者很少发送心跳)。jobtracker 会注意到此 tasktracker 已经停止发送心跳(如果它有 10 分钟仍没有接收到心跳，这由 mapred.tasktracker.expiry.interval 属性来设置，以毫秒为单位)，会将它从等待任务调度的 tasktracker 池中移除。jobtracker 会安排此 tasktracker 上已经运行并成功完成的 map 任务返回，如果它们属于未完成的作业的话，因为它们的中间输出都存放在故障 tasktracker 的本地文件系统上，这是 reduce 任务无法访问的。任何进行中的任务也都会被重新调度。

tasktracker 还可以被 jobtracker 放入黑名单，即使它并没有失败。如果在它上面的任务的失败次数远远高于集群平均任务失败次数，它就会被放入黑名单。被放入黑名单中的 tasktracker 可以通过重启从 jobtracker 的黑名单中被移出。

6.2.3 jobtracker 失败

jobtracker 失败是在所有失败中最严重的一种。目前，Hadoop 没有用于处理 jobtracker 失败的机制——它是一个单点故障——因此在这种情况工作下，作业注定会失败。然而，这种失败形式发生的概率很小，因为具体某台机器失败的几率很小。未来版本的 Hadoop 可能会通过运行多个 jobtracker 的方法来解决这个问题，任何时候，都只有其中一个是主 jobtracker(使用 ZooKeeper 作为 jobtracker 的一种协调机制来决定哪一个是主 jobtracker，详见第 13 章)。

6.3 作业的调度

早期版本的 Hadoop 使用一种非常简单的方法来调度用户的作业：按照作业提交的顺序来运行，使用的是 FIFO(先进先出)调度算法。一般情况下，每个作业会使用整个集群，因此作业必须等待直到轮到自己运行。虽然共享集群具有很强的为大量用户提供大量资源的潜力，但问题在于如何公平地在用户之间分配资源，这需要一

个更好的调度器。生产型的作业需要及时结束，使正在进行小型 ad hoc 查询的用户能够在短时间内得到返回结果。

随后，设置作业优先级的功能被加入进来，通过设置 `mapred.job.priority` 属性或 `JobClient` 的 `setJobPriority()` 方法来设置优先级(这两种方法将选择 `VERY_HIGH`, `HIGH`, `NORMAL`, `LOW`, `VERY_LOW` 中的一个值来作为优先级)。当作业调度器选择下一个作业开始运行时，它选择的是优先级最高的。然而，在 FIFO 调度中，优先级并不支持抢占，所以高优先级的作业仍然会被长时间运行的在高优先级作业运行之前开始的低优先级的作业所阻塞。

现在，Hadoop 的 MapReduce 有多种调度器可选。默认的是原始的基于队列的 FIFO 调度器，还有一种叫公平调度器(Fair Scheduler)的多用户调度器。

Fair Scheduler

Fair Scheduler(公平调度器)的目标是让每个用户公平地共享集群。如果只有一个作业在运行，它会得到整个集群的所有资源。随着提交的作业越来越多，空闲的任务槽会被以这种方式分配给它们，使每个用户都能公平地共享集群。属于某个用户的一道短的作业将在合理的时间内完成，即便另一个用户的长作业正在运行且长作业还将继续运行。

作业被放在池中，而且在默认情况下，每个用户都有自己的池。提交作业超过其他用户的用户，不会因此而比其他用户获得超过平均值的集群资源。可以用 map 和 reduce 的槽数来定义用户池的最小容量，也可以设置每个池的权重。

Fair Scheduler 支持抢占，如果一个池在特定的一段时间内未能得到公平的资源分配，调度器就会终止运行池中得到过多资源的任务，以便把槽让给资源不足的池。

Fair Scheduler 是一个贡献模块。要使用它，将它的 JAR 文件从 Hadoop 的 contrib/fairscheduler 目录复制到 lib 目录中，即可将其放入 Hadoop 的类路径。随后如下设置 `mapred.jobtracker.taskScheduler` 属性：

```
org.apache.hadoop.mapred.FairScheduler
```

无需更多配置，Fair Scheduler 即可运行。但要想充分发挥它特有的优势，了解如何配置它(包括它的网络接口)，请参阅 *src/contrib/fairscheduler* 目录下的 README 文件。

6.4 shuffle 和排序

MapReduce 保证每个 reducer 的输入都已按键排序。系统执行排序的过程——map

输出传到 reducer 作为后者的输入——即称为 shuffle(混洗或称洗牌)[1]。在本小节中，我们将学习 shuffle 是如何工作的，有一个基本的了解，对于需要优化 MapReduce 程序是大有帮助的。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许多细节(以及随时间而产生的变化)。从许多方面来看，shuffle 是 MapReduce 的心脏，是奇迹发生的地方。

6.4.1 map 端

map 函数开始产生输出结果时，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，并处于效率的原因预先进行排序。如图 6-4 所示。

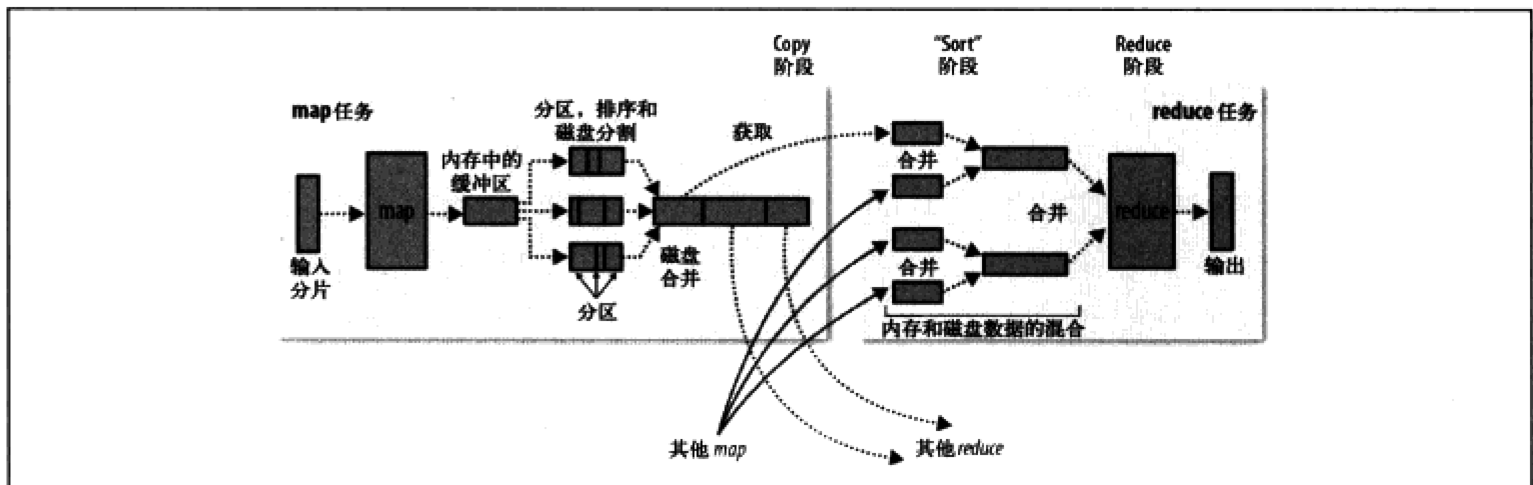


图 6-4：MapReduce 的 shuffle 和排序

每个 map 任务都有一个环形内存缓冲区，任务会把输出写到此。默认情况下，缓冲区的大小为 100 MB，此值可以通过 `io.sort.mb` 属性来修改。当缓冲内容达到指定大小时(`io.sort.spill.percent`，默认为 `0.80`，80%)，一个后台线程便开始把内容溢写(spill)到磁盘中。在线程工作的同时，map 输出继续被写到缓冲区，但如果在此期间缓冲区被填满，map 会阻塞直到溢写过程结束。

溢写将按轮询方式写到 `mapred.local.dir` 属性指定的目录，在一个作业相关子目录中。在写到磁盘之前，线程首先根据数据最终被传送到的 reducer，将数据划分成相应的分区。在每个分区中，后台线程按键进行内排序(in-memory sort)。此时如果有一个 combiner，它将基于排序后输出运行。

一旦内存缓冲区达到溢写阈值，就会新建一个溢写文件，因此在 map 任务写入其

① 事实上，此术语 shuffle 并不准确。因为在某些语境中，它只是代表 reduce 任务获取 map 输出的部分过程。在这节，我们将其理解为从 map 产生输出到 reduce 的消耗输入的整道过程。

最后一个输出记录之后，会有若干个溢写文件。在任务完成之前，溢写文件被合并成一个已分区且已排序的输出文件。配置属性 io.sort.factor 控制着一次最多能合并多少流，默认值是 10。

如果已经指定 combiner，并且溢写次数至少为 3(min.num.spills.for.combine 属性的值)时，combiner 就在输出文件被写之前执行。前面曾讲过，combiner 会针对输入反复运行，但不会影响最终结果。运行 combiner 的意义在于使 map 输出更紧凑，从而只有较少数据被写到本地磁盘然后传给 reducer。

map 输出被写到磁盘时，对它进行压缩往往是个很好的主意，因为这样会让写入磁盘的速度更快，节约磁盘空间和减少传给 reducer 的数据量。默认情况下，输出是不压缩的，但是只要将 mapred.compress.map.output 设置为 true，就可以启用此功能。使用的压缩库由 mapred.map.output.compression.codec 定义，要想进一步了解压缩格式，请参见第 4 章的“压缩”小节。

reducer 通过 HTTP 得到输出文件的分区。用于服务于文件分区的工作线程，其数量由任务的 tracker.http.threads 属性来控制。此设置针对的是每个 tasktracker，而不是针对每个 map 任务槽。默认是 40，在运行大规模作业的大型集群上，此值可以根据需要而增加。

6.4.2 reduce 端

现在转到处理过程的 reduce 这一端。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)，不过在现在，tasktracker 需要它为分区文件运行 reduce 任务。而且，reduce 任务需要为其特定分区文件从集群上若干个 map 任务的 map 输出。map 任务可以在不同时间完成，因此只要有一个任务结束，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，因此能够并行地取得 map 输出。默认是 5 个线程，但这个默认值可以通过设置 mapred.reduce.parallel.copies 属性来改变。

注意：reducer 如何知道要从哪个 tasktracker 取得 map 输出呢？

map 任务成功完成后，它们会通知其父 tasktracker 状态已更新，然后 tasktracker 进而通知 jobtracker。这些通知在前面介绍的心跳交流机制中传输。因此，对于指定作业，jobtracker 知道 map 输出和 tasktracker 之间的映射关系。reducer 中的一个线程定期向 jobtracker 获取 map 输出位置，直到得到所有输出位置。

tasktracker 并没有在第一个 reducer 检索之后就立即从磁盘上删除 map 输出，因为

reducer 可能失败。反之，它们会等待，直到被 jobtracker 告知可以删除，这是作业完成后才执行的。

如果 map 输出相当小，则会被复制到 reduce tasktracker 的内存中(缓冲区大小由 mapred.job.shuffle.input.buffer.percent 属性控制，它说明用作此用途的堆空间的百分比)；否则，被复制到磁盘。内存缓冲区达到阈值大小(由 mapred.job.shuffle.merge.percent 决定)或达到 map 输出阈值(由 mapred.inmem.merge.threshold 控制)时，会被合并，进而被溢写到磁盘中。

随着磁盘上积累的副本增多，后台线程会将它们合并为一个更大的、排好序的文件。这会为后面的合并节省一些时间。注意，任何压缩的 map 输出(通过 map 任务)都必须在内存中被解压缩，以便于合并。

所有 map 输出被复制期间，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，维持其按顺序排序。这将循环进行。比如，如果有 50 个 map 输出，而合并系数是 10(10 为默认设置，由 io.sort.facotr 属性设置，与 map 的合并类似)，合并将进行 5 轮。每轮将 10 个文件合并成一个文件，因此最后有 5 个中间文件。

最后阶段，即 reduce 阶段，合并直接把数据输入 reduce 函数，而不是最后还有一轮将这 5 个文件合并成一个已排序的文件。此举省略了一次磁盘往返行程。最后的合并既可来自内存中，也可来自磁盘。

注意： 每轮合并的文件数实际上比这个例提出的更微妙。目标是合并最小数量的文件，最后一轮得到合并系数。因此如果有 40 个文件，我们不会以 10 个文件为一轮，进行 4 轮合并而得到 4 个文件。相反地，第一轮只合并 4 个文件，随后的三轮合并所有 10 个文件。最后，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。注意，这并没有改变往返行程的次数，它只是一个用来减少写入磁盘数据量的优化方法，因为最后一轮总是直接合并到 reduce。

在 reduce 阶段，对已排序输出中的每个键依次调用 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 tasktracker 节点也运行数据节点，所以第一个块副本会被写到本地磁盘。

6.4.3 配置的调整

现在应该理解如何调整 shuffle 以提高 MapReduce 性能了。以下相关设置都以作业为基本单位(打上标记的除外)。如表 6-1 和表 6-2 所示，适用于常规作业的默认值

也被包含在这两个表中。

表 6-1：map 端可调整的属性

属性名称	类型	默认值	说明
io.sort.mb	int	100	对 map 输出进行排序时所使用的内存缓冲区的大小，以兆字节为单位
io.sort.record.percent	float	0.05	保留的 io.sort.mb 比例，用来储存 map 输出的记录边界。剩余的空间用来储存 map 输出记录本身
io.sort.spill.percent	float	0.80	map 输出内存缓冲和记录边界索引，两者使用比例的阈值。达到此值，开始至磁盘的溢写过程。
io.sort.factor	int	10	排序文件时一次合并的最大流数。这个属性也在 reduce 中使用。将此默认值增加到 100 是非常常见的
min.num.spills.for.combine	int	3	运行 combiner 所需要的溢写文件的最小数量(如果已定义 combiner 的话)
mapred.compress.map.output	boolean	false	压缩 map 输出
mapred.map.output.compression.codec	Class name	org.apache.hadoop.io.compress.DefaultCodec	压缩 map 输出的编解码器
task tracker.http.threads	int	40	每个 tasktracker 用于将 map 输出传给 reducer 的工作线程的数量。这是集群范围的设置，不能由某个单独的作业进行设置

表 6-2：reduce 端可调整的属性

属性名称	类型	默认值	描述
mapred.reduce.parallel.copies	int	5	用于将 map 输出复制到 reducer 的线程的数量

续表

属性名称	类型	默认值	描述
mapred.reduce.copy.backoff	int	300	在声明失败之前，reducer 获取一个 map 输出所花费的最大时间，以秒为单位。如果失败，reducer 可以根据此时间尝试重传。(利用指数后退 (exponential backoff)[①])
io.sort.factor	int	10	排序文件时一次合并的流的最大数量。这个属性也在 map 端使用
mapred.job.shuffle.input.buffer.percent	float	0.70	整个堆空间的百分比，用于 shuffle 的复制阶段，分配给 map 输出缓存
mapred.job.shuffle.merge.percent	float	0.66	map 输出缓存(由 mapred.job.shuffle.input.buffer.percent 定义)，其使用比例的阈值，用于启动合并输出和磁盘溢写的过程
mapred.inmem.merge.threshold	int	1000	启动合并输出和磁盘溢写过程的最大 map 输出数量。0 或者更小的数意味着没有阈值限制，并且溢写行为将由 mapred.job.shuffle.merge.percent 单独控制
mapred.job.reduce.input.buffer.percent	float	0.0	在 reduce 过程中，用来在内存中保存 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 开始之前，所有的 map 输出都合并到磁盘中，以便为 reduce 提供尽可能多的内存。然而，如果 reducer 需要的内存较少，可以增加此值来减少写入磁盘的次数

① 译注：指数后退是一种帮助决定发送速度，用于避免网络拥挤的算法。此处用于决定超时后重试次数及间隔时间，它的计算公式 $backoff(t) = init \times base^{t-1}$，满足 $backoff(1) + ... + backoff(\max_fetch_retries) \le \max$，所以重试次数 max_fetch_retries 的计算公式是：$\max_fetch_retries = \frac{\log_{\max \times (base-1))/init+1}}{\log_{base}}$。例如，max 即 mapred.reduce.copy.backoff 的默认值为 300(5 分钟)，Hadoop 实现中 init 为 4，base 为 2，我们通过公式可以计算出重试次数为 6，重次的时间分别为 4，8，16，32，64，128，加起来共 252 秒(4.2 分钟)。

优化的一般原则是为 shuffle 指定尽量多的内存空间。然而，有一个平衡问题，你要确保 map 和 reduce 函数能得到足够的内存来运行。这就是为什么编写 map 和 reduce 函数时尽量少用内存的原因——它们不应使用不限量的内存(例如，应避免在 map 中堆积一系列的值)。

为 map 和 reduce 任务运行的 JVM 指定的内存大小由 mapred.child.java.opts 属性来设置。应该让任务节点上这个内存大小尽量大，在第 9 章的“内存”小节，我们将讨论分配内存中所有需要考虑的约束条件。

在 map 这一端，可以通过避免多次磁盘溢写来获得最佳性能。如果你能估计 map 输出大小，就可以合理地设置 ip.sort.*属性来减少溢写的次数。具体说来，如果可以，应该增加 io.sort.mb 的值。MapReduce 计数器将计算在作业运行过程中溢写到磁盘中的记录总数，这对于调优很有帮助。注意，计数包括 map 和 reduce 两端的溢写。

在 reduce 这一端，当中间值能够全部存放在内存中时，就能获得最佳性能。默认情况下，这是不可能发生的，因为一般情况下所有内存都预留给 reduce 函数。但是如果 reduce 函数的内存需求很小，那么将 mapred.inmem.merge.threshold 设置为 0，将 mapred.job.reduce.input.buffer.percent 设置为 1.0(或者一个更低的值，详见表 6-2)会带来性能的提升。

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，详见第 9 章的“其他 Hadoop 属性”小节)。

在 2008 年 4 月，Hadoop 在通用 TB 字节排序基准测试中获胜(详见附录 A“在 Apache Hadoop 中的 TB 字节排序”)，它使用的一个优化方法就是将中间值保存在 reduce 这一端的内存中。

6.5 任务的执行

在 6.1 节，我们一开始就了解了 MapReduce 系统如何在整个作业的上下文执行任务。在本小节，我们将了解 MapReduce 用户对任务执行的更多控制。

6.5.1 推测式执行

MapReduce 模型将作业分割成任务，然后并行运行任务，使作业的整体执行时间

少于顺序执行的时间。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不执行该任务的作业。在一个作业由成百上千或成千上万任务组成时，可能出现少数“拖后腿”的任务是很常见的。

任务可能由于多种原因而执行缓慢，包括硬件老化或软件配置错误，但是，尽管比预计执行时间长，任务总归能够成功完成，所以检测起来非常困难。Hadoop 不会尝试诊断或者修复执行缓慢的任务；相反，在一个任务运行比预期要慢的时候，它会进行检测，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式执行。

必须认识到一点，如果同时启动两个备份任务，它们会互相竞争，导致推测式执行无法正确执行。这对于集群资源是一种浪费。相反地，推测式执行的任务只有在一个作业所有任务都启动之后才启动，并且推测式执行任务只针对已运行一段时间(至少一分钟)且比作业中其他任务平均进度慢的任务。一个任务成功完成后，任何正在运行的副本任务都将被终止，因为它们已经不再需要了。因此，如果原任务在推测任务前完成，推测任务就会被终止；同样地，如果推测任务先完成，那么原任务就会被终止。

推测式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 bug，有时会造成任务被挂起或运行速度减慢，这种情况下依赖推测式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 bug 也许也会影响推测式任务。应该修复 bug，使任务不会被挂起或者运行速度减慢。

默认情况下，推测式执行是启用的。它可以基于集群或者基于某个单独的作业，独立为 map 任务和 reduce 任务启用或禁用。相关的属性如表 6-3 所示。

表 6-3：推测式执行的属性

属性名称	类型	默认值	说明
mapred.map.tasks.speculative.execution	boolean	true	在一个 map 任务运行缓慢时确定是否运行额外的 map 任务实例
mapred.reduce.tasks.speculative.execution	boolean	true	在一个 reduce 任务运行缓慢时确定是否运行额外的 reduce 任务实例

为什么会想到关闭推测式执行？推测式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，推测式执行会减少整体的吞吐量，因为冗余的任务在执行时会减慢作业的执行时间。鉴于此，一些集群管理员在集群上会关闭推测式执行，让用户根据个人作业需要而开启。Hadoop 老版本尤其如此，因为它在调度推测任务时会过度使用推测式执行。

6.5.2 任务 JVM 重用

Hadoop 在它们自己的 Java 虚拟机上运行任务，以区分其他正在运行的任务。为每个任务启动一个新的 JVM 将耗时大约 1 秒，对运行 1 分钟左右的作业而言，这是微不足道的。但是，作业有大量非常短的任务(通常是 map 任务)或初始化时间长的作业，此时对后续任务重用 JVM，就可以获得性能的提升。

启用任务 JVM 重用后，任务并没有同时运行在一个 JVM 中。JVM 顺序运行任务。尽管 tasktracker 可以一次运行多个任务，但都运行在独立的 JVM 中。控制 tasktracker map 任务槽数量和 reduce 任务槽数量的属性将在第 9 章阐述。

控制任务 JVM 重用的属性是 `mapred.job.reuse.jvm.num.tasks:`。它定义指定作业每个 JVM 运行的 task 任务的最大数量，默认为 1(详见表 6-4)。来自不同作业的任务总是运行在不同的 JVM 上。如果属性设置为-1，则意味着同一作业中共享一个 JVM 的任务的数量将不受限制。JobConf 中的 `setNumTasksToExecutePerJvm()` 方法也可以用于设置这个属性。

表 6-4：任务 JVM 重用的属性

属性名称	类型	默认值	说明
mapred.job.reuse.jvm.num.tasks	int	1	在一个 tasktracker 上一给定的作业的每个 JVM 可以运行的任务最大数。值-1 表示无限制；同一个 JVM 可以被该作业的所有任务使用

通过充分利用 HotSpot JVM 所用的运行时优化，计算密集型的任务也可以受益于任务 JVM 重用机制。在运行一段时间后，HotSpot JVM 构建足够多的信息来检测代码中的性能关键部分，动态地将热点部分的 Java 字节代码转换成本地机器代码。这对运行时间长的处理十分合适，但那些只运行几秒钟或者几分钟的进程，不能充分利用 HotSpot 的优势。在这些情况下，启用任务 JVM 重用功能是很有必要的。

共享 JVM 的另一个用处是作业各个任务间的状态共享。将相关数据存储到一个静态的字段后，任务就可以较快速访问共享数据。

6.5.3 跳过坏记录

大型数据集十分庞杂，经常会有损坏的记录，也经常有不同格式的记录和缺失的字段。理想情况下，代码可以顺利应付这些状态。但实际情况中，忽略这些错误的记录只是权宜之计。依赖于分析的方法，如果只有一小部分记录被影响，那么忽略它们将不会显著影响结果。然而，如果一个任务由于遇到一个损坏的记录而发生问题——抛出一个运行时异常——然后任务失败。失败的任务将被重新运行(因为错误可能是硬件问题或任务控制之外的一些原因导致的)，但是如果一个任务失败 4 次，那么整个作业会被标记为失败(详见第 6 章的“任务失败”小节)。如果由于数据而导致任务抛出一个异常，那么重新运行任务将无济于事，因为它每次都会因相同的原因而失败。

注意： 如果正在使用 `TextInputFormat`(详见第 7 章的“TextInputFormat”小节)，则可以设置一个预期的行最大长度来防止错误文件。在文件中的错误将呈现成非常长的行，这将引起内存溢出错误并导致任务失败。通过将 `mapred.linerecordreader.Maxlength` 设置为一个适合内存的值(一般比输入数据行长度更大一些)，记录读取将跳过(长)损坏的行而不会导致任务失败。

处理错误记录的最佳位置是 mapper 和 reducer 代码。可以检测出错误记录并忽略它，或通过抛出一个异常来取消这个作业。还可以使用计数器来计算作业中错误记录的总数，从而查看问题所影响的范围。

极少数情况下，因为 bug 存在于第三方的库中，且无法在 mapper 或者 reducer 中修改它，所以你不能处理这个问题。在这些情况下，可以使用 Hadoop 的 skipping 模式(忽略模式)选项来自动跳过错误记录。skipping 模式启用后，任务将正在处理的记录报告给 tasktracker。任务失败时，tasktracker 重新运行该任务，跳过导致任务失败的记录。由于增加的网络流量和错误记录的维护，只有在任务失败两次后才会启用 skipping 模式。

因此对于一个一直由某个错误记录而导致失败的任务，tasktracker 将运行以下任务尝试得到相应结果。

1. 任务失败。
2. 任务失败。
3. 开启 skipping 模式。任务失败但是错误记录仍然储存在 tasktracker 中。

4. skipping 模式仍在启用。任务忽略上一次任务尝试中失败的错误记录继续运行。

在默认情况下，skipping 模式是关闭的，可以使用 SkipBadRedcord 类单独为 map 和 reduce 任务启动它。注意，每次进行任务尝试，skipping 模式都只能检测出一个错误记录，因此这种机制仅适用于检测个别错误记录(每个任务有一些)。需要增加任务尝试最大值(通过 mapred.map.max.attemps 和 mapred.reduce.max.attemps)，给与 skipping 模式足够多的尝试次数来检测并跳过一个输入划分中的所有错误记录。

Hadoop 检测出来的错误记录以序列文件的形式保存在作业输出目录的_logs/skip 子目录下。在作业完成后，可查看这些错误记录进行诊断(使用 hadoop fs -text 选项)。

6.5.4 任务执行环境

Hadoop 为 map 和 reduce 任务提供运行环境的信息。例如，map 任务可以知道它处理的文件的名称(详见第 7 章的“mapper 的文件信息”小节)，map 和 reduce 任务可以得知任务尝试次数。表 6-5 中的属性可以从作业配置中获得，通过为 mapper 或 reducer 提供一个 configure()方法的实现(配置信息作为参数被传回此方法)。

表 6-5：任务执行环境的属性

属性名称	类型	说明	示例
mapred.job.id	String	作业 ID。(详见 5.5 节，了解格式的描述)	job_200811201130_0004
mapred.tip.id	String	任务 ID	task_200811201130_0004_m_000003
mapred.task.id	String	Task attempt ID (而非任务 ID)	attempt_200811201130_0004_m_000003_0
mapred.teask.partition	int	作业中的任务的 ID	3
mapred.task.is.map	boolean	此任务是否是 map 任务	true

流环境变量

Hadoop 设置作业配置参数作为流程序的环境变量。但它用下划线来代替非字母数字符号，以确保名称的合法性。下面的 Python 表达式表明如何用 Python 流脚本来检索 mapred.job.id 属性的值。

```
os.environ["mapred_job_id"]
```

也可以用流启动程序的-cmdenv 选项，为 MapReduce 启动的流进程设置变量。(一次设置一个希望设置的变量)。比如，下面便设置了 MAGIC_PARAMETER 环境变量：

```
-cmdenv MAGIC_PARAMETER=abracadabra
```

任务的副作用文件

从 map 和 reduce 任务写输出，通常的方式是使用 OutputCollector 来收集键值对。有一些应用需要比键值对更灵活的方式，因此这些应用程序直接将 map 或 reduce 任务的输出文件写到分布式文件系统，如 HDFS(还有其他方法可产生多个输出，详见第 7 章的“多个输出”小节)。

我们必须注意，确保同一个任务的多个实例不会尝试向同一个文件进行写操作。需要避免两个问题。第一个问题是，如果任务失败并被重试，那么在第二个任务运行时原来的部分输出依旧是存在的，所以应先删除第一个任务的旧文件。第二个问题是，在启用推测式执行的情况下，同一任务的两个实例会同时向同一个文件进行写操作。

Hadoop 通过将输出写到任务尝试指定的临时文件夹，解决了任务的常规输出问题。这个目录是{mapred.out put.dir}/_temporary/${mapred.task.id}。若任务执行成功，目录的内容就被复制到此作业的输出目录(${mapred.output.dir})。因此，如果一个任务失败并重试，第一个任务尝试的部分输出就会被清除。一个任务和该任务的推测实例位于不同的工作目录，并且只有先完成的任务才会把其工作目录中的内容传到输出目录——其他的都被丢弃。

注意： 任务完成时提交输出的方法由一个 OutputCommitter 来实现，它与 OutputFormate 相关联。FileOutputFormate 的 OutputCommitter 是一个 FileOutputCommitter，后者实现了前面描述的提交规则。OutputFormat 的 getOutputCommitter() 方法也许会被覆盖以返回一个自定义的 OutputCommitter，以免你用不同的方式来实现提交过程。

Hadoop 也为应用程序开发人员提供了使用这个特征的机制。一个任务可以通过检索其配置文件中 mapred.work.output.dir 属性的值来找到它的工作目录。或者，MapReduce 程序使用 Java API 来调用 FileOutputFormate 的 getWorkOutputPath()静态方法以得到代表工作目录的 Path 对象。框架会在执行任务之前创建工作目录，所以我们没有必要自己新建。

举一个简单的例子，假设有一个将图像文件从一种格式转换到另一种格式的程序。一种方法是使用只有 map 的作业，其中每个 map 被赋予一组要转换的图像(可以使用 NlineInputFormat，详情可参见第 7 章)。如果 map 任务把转换后的图像写到它的工作目录，那么在任务成功完成之后，这些图像会被传到输出目录。

第 7 章

MapReduce 的类型与格式

MapReduce 的数据模型非常简单：它的 Map 和 Reduce 函数使用键/值对(key/value pair)进行输入/输出。本章将深入讨论 MapReduce 的数据模型，尤其是诸如文本或者二进制类型的数据如何在 MapReduce 中使用。

7.1 MapReduce 类型

Hadoop MapReduce 中的 map 和 reduce 函数遵循以下的形式：

```
map: (K1, V1) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)
```

通常来说，map 函数输入的键/值(k1 和 v1)类型与用于输出的(k2 和 v2)不同。然而，reduce 函数的输入类型必须与 map 函数的输出类型相一致。同时，reduce 函数的输入类型与输出类型(k3 和 v3)又可能与上述两者都不同。以下的 Java 代码体现了这个规定：

```
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;
}
```

注意，OutputCollector 纯粹是为了收集键/值对(所以需要键与值的类型信息)，Reporter 是用来更新计数和状态信息的。(在 MapReduce API 0.20.0 或更新版本中，这两个函数被放入同一个上下文对象中。)

如果需要使用 combiner，那么 combiner 的形式与 reduce 函数一样(并且需要实现 Reducer 接口)，区别在于它的输出键/值对类型应该是中间的键/值对类型(也就是 K2 和 V2)，从而可以被 reduce 函数处理。

```
map: (K1, V1) → list(K2, V2)
combine: (K2, list(V2)) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)
```

通常 combiner 与 Reduce 函数是一样的，这样的话，K3 与 K2 类型相同，V3 与 V2 类型相同。

partitioner 接受中间的键/值对(K2 和 V2)并且返回一个分区索引。但实际上，partition 的值只是由键决定的，值将被忽略。

```
partition: (K2, V2) → integer
```

或者用 Java 写：

```
public interface Partitioner<K2, V2> extends JobConfigurable {

  int getPartition(K2 key, V2 value, int numPartitions);
}
```

好了，理论课到此结束，对配置 MapReduce 作业有帮助么？表 7-1 总结了配置选项。它显示了可以配置的属性的类型以及与类型属性兼容的属性类型。

输入数据的类型是通过输入格式进行设定的。例如，对于 TextInputFormat，它的键类型就是 LongWritable，而值类型就是 Text。其他的类型可以通过调用 JobConf 中的方法来进行显式地设置。如果没有显式地设置，中间的类型将被默认设置为(最终的)输出类型，也就是 LongWritable 和 Text。综上所述，如果 K2 与 K3 是相同类型，就不需要手工调用 setMapOutputKeyClass()，因为它将被自动

设置为 `setOutputKeyClass()`指定的类型；同样，如果 V2 与 V3 一样的话，只需要设定 `setOutputValueClass()`即可。

表 7-1：MapReduce 类型配置

属性	JobConf 方法　set 方法	输入类型 K1	输入类型 V1	中间类型 K2	中间类型 V2	输出类型 K3	输出类型 V3
用于配置类型的属性							
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()						•
必须与类型一致的属性							
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()					•	•

手动设置每一个步骤的输入和输出类型一定很奇怪，为什么不能从最初输入的类型推导出每个步骤的输入/输出类型呢？原来 Java 的泛型机制具有很多限制，类型擦除导致了运行时类型并不一直可见，所以需要 Hadoop 时不时地“提醒”一下。这也导致了可能在某些 MapReduce 任务中出现不兼容的输入和输出类型，因为这些配置在编译时无法检查出来。与 MapReduce 任务兼容的类型已经在表 7-1 列出。所有的类型不兼容将在任务真正执行的时候被发现，所以一个比较聪明的做法是在执行任务前先用少量的数据跑一次测试任务，以发现所有的类型不兼容问题。

默认的 MapReduce 任务

如果在跑一个 MapReduce 任务的时候没有指定 Mapper 或者 Reducer 会怎么样？我们用一个最简单的 MapReduce 任务试试看：

```
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;
    }

    JobConf conf = new JobConf(getConf(), getClass());
    FileInputFormat.addInputPath(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));
    JobClient.runJob(conf);
    return 0;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new MinimalMapReduce(), args);
    System.exit(exitCode);
  }
}
```

只设置了输入和输出文件的路径，并且使用以下的命令在我们的气象数据上进行试验：

```
% hadoop MinimalMapReduce "input/ncdc/all/190{1,2}.gz" output
```

我们得到了一个叫 part-00000 的输出文件，下面是这个文件的前几行(为了适应页大小而做了截断处理)：

```
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...
```

每一行都是一个整数，然后是一段原始气象数据，使用制表位隔开。老实说，这并不怎么有用，但能够帮助我们了解 Hadoop 是如何设置默认类型的。例 7-1 的示例与上面的代码段完成了一模一样的事情，但是它显式地设置了类型参数：

例 7-1：最简单的 MapReduce 任务，但所有的参数已显式设置

```
public class MinimalMapReduceWithDefaults extends Configured
implements Tool {

  @Override
  public int run(String[] args) throws IOException {
    JobConf conf = JobBuilder.parseInputAndOutput(this,
      getConf(), args);
    if (conf == null) {
      return -1;
    }

    conf.setInputFormat(TextInputFormat.class);

    conf.setNumMapTasks(1);
    conf.setMapperClass(IdentityMapper.class);
    conf.setMapRunnerClass(MapRunner.class);

    conf.setMapOutputKeyClass(LongWritable.class);
    conf.setMapOutputValueClass(Text.class);

    conf.setPartitionerClass(HashPartitioner.class);

    conf.setNumReduceTasks(1);
```

```
    conf.setReducerClass(IdentityReducer.class);

    conf.setOutputKeyClass(LongWritable.class);
    conf.setOutputValueClass(Text.class);

    conf.setOutputFormat(TextOutputFormat.class);

    JobClient.runJob(conf);
    return 0;
  }
  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new
      MinimalMapReduceWithDefaults(), args);
      System.exit(exitCode);
  }
}
```

run()方法的开始几行被简化了，我们把设定输入/输出以及打印使用说明的代码放入一个帮助方法。几乎所有的 MapReduce 程序都会接收两个参数(输入与输出)，所以以这样的方法写是个不错的选择。以下列出与 JobBuilder 有关的方法作为参考：

```
public static JobConf parseInputAndOutput(Tool tool,
    Configuration conf, String[] args) {

  if (args.length != 2) {
    printUsage(tool, "<input> <output>");
    return null;
  }
  JobConf jobConf = new JobConf(conf, tool.getClass());
  FileInputFormat.addInputPath(jobConf, new Path(args[0]));
  FileOutputFormat.setOutputPath(jobConf, new Path(args[1]));
  return jobConf;
}

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);
}
```

回到例 7-1 中的 MinimalMapReduceWithDefaults，虽然还有很多默认的任务配置，但是被高亮显示的部分是执行任务最关键的部分，接下来我们逐个讨论一下。

默认的输入格式是 TextInputFormat，它产生的键类型是 LongWritable(也就是文件中某一行的偏移量)，值类型是 Text(一行文本)。这也解释了最后输出的整数的意义：它们是行偏移量。

虽然看上去很像，但是 setNumMapTasks 函数调用并不一定将 map 任务的数量设定成 1。事实上它只是一个暗示。真正 map 任务的数量将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 当中的话)。

默认的 mapper 是 IdentityMapper，它将输入的键和值原封不动地写入输出：

```
public class IdentityMapper<K, V> extends MapReduceBase
    implements Mapper<K, V, K, V> {

  public void map(K key, V val, OutputCollector<K, V> output,
                  Reporter reporter) throws IOException {
  output.collect(key, val);
  }
}
```

IdentityMapper 是个泛型类型，它可以接受任何类型的参数，只要输入和输出的键类型相同，并且输入和输出的值类型相同就可以。在这里，map 的输出类型是 LongWritable 和 Text。

map 任务是由 MapRunner 负责运行的，它是默认的 MapRunnable 的实现，并且顺序地为每一条记录执行一次 mapper 的 map()任务。

默认的 partitioner 是 HashPartitioner，它将每条记录的键进行哈希处理来决定某条记录应该放到哪个分区上。每个分区上会对应一个 reducer，所以分区的个数和 reducer 的个数是相同的。

```
public class HashPartitioner<K2, V2> implements Partitioner<K2,
    V2> {

  public void configure(JobConf job) {}

  public int getPartition(K2 key, V2 value, int numPartitions) {
    return (key.hashCode() & Integer.MAX_VALUE) % numPartitions;
  }

}
```

每条记录的键的哈希值将与最大的整型值做一次按位与操作从而变成一个非负整

数，并且用取模操作来决定它将被分到哪个分区。

默认情况下，只有一个 reducer，也就是一个分区。这样的话 partitioner 的操作将由于所有的数据都被放到同一个分区而变得无关。但是，如果有很多 reducer 的话，了解 HashPartitioner 的作用就非常重要了。如果对于键的散列函数足够的好，那么记录将会被平均地分到不同的分区上，并且所有具有相同键的记录都由一个 reducer 来完成。

如何选择 reducer 的数量？

默认 1 个的 reducer 配置是为了让 Hadoop 新手容易上手。在任何真实的应用中都会将它设置成一个更加大的数目，否则所有的 reduce 任务都会放到单一的 reducer 去跑，效率非常低。(注意，在本地的任务环境中，只支持 0 个或 1 个 reducer。)

最优的 reducer 个数取决于集群中可用的 reducer 任务槽的数目。reducer 槽的数目是集群中节点数与 mapred.tasktracker.reduce.tasks.maximum 的乘积。

一个通常的做法是设置比 reducer 任务槽数目稍微小一些的 reducer 个数，这会给 reducer 任务留有余地(容许一些错误发生而不需要延长作业运行时间)。

默认的 reducer 是 IdentityReducer，同样，它也是一个泛型类型。它将所有的输入写入到输出。

```
public class IdentityReducer<K, V> extends MapReduceBase
    implements Reducer<K, V, K, V> {

  public void reduce(K key, Iterator<V> values,
            OutputCollector<K, V> output, Reporter reporter)
    throws IOException {
    while (values.hasNext()) {
      output.collect(key, values.next());
    }
  }
}
```

对于这个任务来说，输出的键类型是 LongWritable，而值类型是 Text。事实上，此 MapReduce 程序所有键的类型都是 LongWritable，所有值的类型都是 Text，因为它们是输入键/值的类型，并且 map 和 reduce 函数都是恒等函数，所以会保留输入的类型到输出。但是对于大多数 MapReduce 程序来说，输入和输出的

类型在中间过程中会改变，所以就像上一节中描述的那样，必须手动地配置每一步的输入和输出的类型。

在发送给 Reducer 处理以前，所有的记录都会(分组)进行排序。在这个例子中，键是按照数值的方式进行排序，所以输入文件中的行会被交叉地放入一个输出文件。

默认的输出格式是 TextOutputFormat，它将键和值转换成字符串并用 tab 分隔写入输出文件，每条记录是一行。这就是为什么示例中的输出文件是用 tab 分隔的：这是 TextOutputFormat 的一个特点。

默认的流操作

在进行流操作的时候，默认的配置与上述的很相似，但也有差别。最简单的形式如下：

```
% hadoop jar $HADOOP_INSTALL/contrib/streaming/hadoop-*-
streaming.jar \
  -input input/ncdc/sample.txt \
  -output output \
  -mapper /bin/cat
```

注意，必须手动指定一个 mapper，因为默认的 IdentityMapper 将无法使用。无法使用的原因是 TextInputFormat 的类型是 LongWritable 和 Text，而流操作的输入类型都是 Text。IndentityMapper 无法将 LongWritable 转换为 Text，导致无法使用。

如果我们使用一个非 Java 的 mapper，并且输入的格式是 TextInputFormat，那么流操作会做一些特殊的处理。它并不会将键传递给 mapper，而只是传递值。事实上这样做是非常有用的，因为键只是文件中的行偏移量，而值就是行中的数据，也就是几乎所有程序关心的部分。这个任务的效果就是将输入的值进行排序。

如果将更多的默认设置写出来，那么命令行看起来应该如下：

```
% hadoop jar $HADOOP_INSTALL/contrib/streaming/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
```

mapper 和 reducer 的参数可以是一条命令或者一个 Java 类。可以通过 `-combiner` 参数按需设置 combiner。

流操作的键与值

一个流操作的程序可以修改输入的分隔符(用于将键与值从输入文件中分开并且传入 mapper)。默认情况下是 Tab，但是如果输入的键或值中本身有 Tab 分隔符的话，最好将分隔符修改成其他符号。

类似地，当 map 和 reduce 将结果输出的时候，也需要一个可以配置的分隔符选项。更进一步，键可以不仅仅是每一条记录的第 1 个字段，它可以是一条记录的前 n 个字段(可以在 `stream.num.map.output.key.fields` 和 `stream.num.reduce.output.key.fields` 中进行设置)，而剩下的字段就是值。比如有一条记录是 `a，b，c`，且用逗号分隔，如果 n 设为 2，那么键就是 `a`、`b`，而值就是 `c`。

mapper 和 reducer 的分隔符是相互独立配置的。这些属性可参见表 7-2，显示见图 7-1。

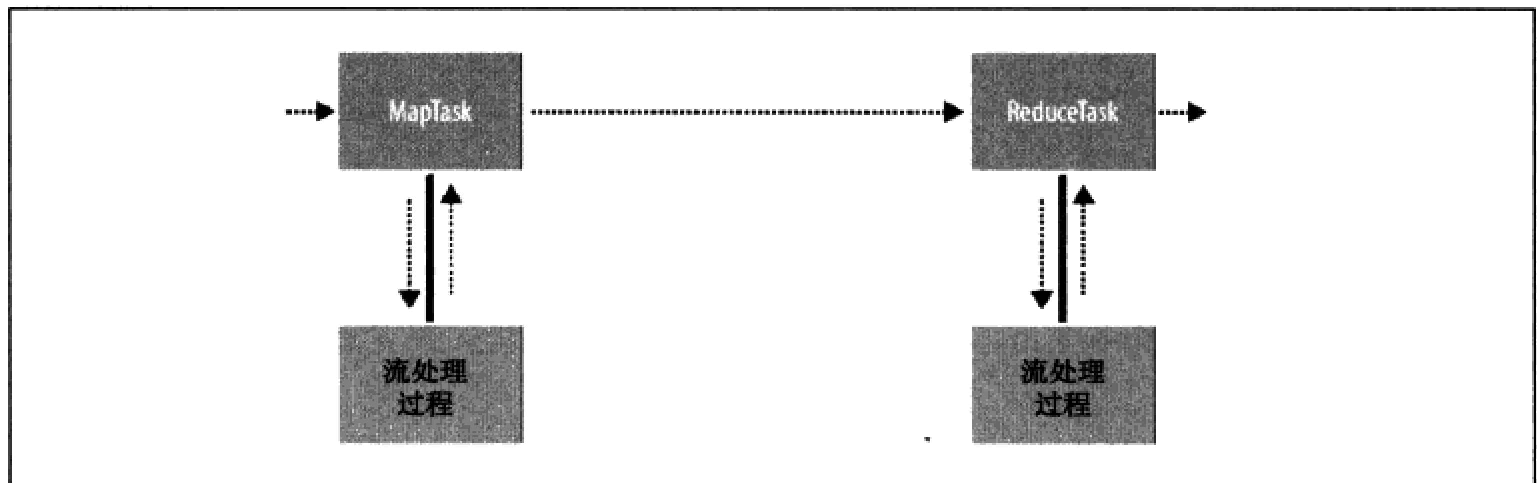


图 7-1：如何在流操作时使用分隔符

同时，这些属性和输入与输出的格式无关。比如，如果 `stream.reduce.output.field.separator` 被设置成冒号，并且 reduce 流将记录 a:b 写入了标准输出，那么处理流的 reducer 一定会知道键是 a，值是 b。如果使用默认的 `TextOutputFormat`，那么这条记录会使用 Tab 将键和值分隔写入输出文件。`TextOutputFormat` 的分隔符可以在 `mapred.textoutputformat.separator` 中设置。

表 7-2：流操作分隔符属性

属性名称	类型	默认值	描述
stream.map.input.field.separator	String	\t	将输入键/值字符串作为字节流传递到流 map 处理时使用的分隔符
stream.map.output.field.separator	String	\t	将来自流 map 处理的输出分割成键/值字符串时使用的 separator，这些键/值将给 map 输出使用
stream.num.map.output.key.fields	int	1	由 stream.map.output.field.separator 分割的字段数量，这些字段可看成时 map 输出键
stream.reduce.input.field.separator	String	\t	将输入键/值字符串作为字节流传递到流 reduce 处理时使用的分隔符
stream.reduce.output.field.separator	String	\t	将来自流 reduce 处理的输出分割成键/值字符串时使用的分隔符，这些键/值将给最终 reduce 输出使用
stream.num.reduce.output.key.fields	int	1	由 stream.reduce.output.field.separator 分割的字段数量，这些字段可看成 reduce 输出键

7.2 输入格式

Hadoop 可以处理很多格式的数据，从一般的文本文件到一整个数据库。本节将探讨这些数据格式。

7.2.1 输入分片与记录

就像在第 2 章中看到的一样，一个输入分片(input split)就是能够被单个 map 操作处理的输入块。每一个 map 操作只处理一个输入分片，并且一个一个地处理每条记录，也就是一个键/值对。输入分片和记录都是逻辑上的，并不必要将它们对应到文件(虽然一般情况下都是这样的)。在数据库中，一个输入分片可以是一个表上的若干行，而一条记录就是这若干行中的一行(事实上 DBInputFormat 就是这么做的，它是一种可以从关系数据库获取数据的一种格式)。

输入分片在 Java 中用 InputSplit 表示(和所有提到过的类一样，它在 org.apache.hadoop.mapred 包中)。

```
public interface InputSplit extends Writable {

  long getLength() throws IOException;

  String[] getLocations() throws IOException;

}
```

一个 InputSplit 有一个以字节为单位的长度以及一组存储位置(即一组主机名)。注意，一个分片并不是数据本身，而是对数据的引用。存储位置是为了让 MapReduce 系统将 map 操作放在离存储位置最近的机器上，而长度是为了将单元排序以使得最大的单元能够最先得到处理，以提高效率(这也是一种贪心近似算法)。

MapReduce 应用程序员并不需要直接控制 InputSplit，因为它是 InputFormat 生成的。InputFormat 需要生成 InputSplit 并将它们分割成记录。在我们探讨 InputFormat 实例之前，先来看一下它在 MapReduce 中是如何工作的：

```
public interface InputFormat<K, V> {

  InputSplit[] getSplits(JobConf job, int numSplits) throws
    IOException;

  RecordReader<K, V> getRecordReader(InputSplit split, JobConf
                                      job, Reporter reporter)
                                      throws IOException;
}
```

JobClient 调用 getSplits()方法，并以 numSplits 为参数传入期望的 map 任务数，这个参数将作为一个参考值，InputFormat 可以返回一个不同于这个值个数的单元。在计算好(实际的)分布的个数后，客户端将它们发送到 jobtracker 上，jobtracker 会使用它们的存储位置信息将它们调度到相应的 tasktracker 上执行。

在 tasktracker 上，map 任务会将输入分片传递到 InputFormat 的 getRecordReader()方法中从而获得相应的 RecordReader。RecordReader 基本就是记录上的迭代器，map 任务会使用 RecordReader 来读取记录并且生成键/值对，然后再传递给 map 函数。以下代码演示了这个过程：

```
K key = reader.createKey();
```

```
    V value = reader.createValue();
    while (reader.next(key, value)) {
      mapper.map(key, value, output, reporter);
    }
```

RecordReader 的 next()方法会不断地读取记录并生成键/值对。当 RecordReader 达到输入流的尾部时，next()会返回 false，从而结束整个循环。

注意： 这段代码说明了每次传入 map 函数的都是同一个 key 和 value 的对象，只是它们的内容被(next()方法)改变了。默认 key 和 value 对象是不可改变的用户对此可能有些不解。如果在 map()函数之外有对于 key 和 value 的引用，那么这可能会造成一些问题，因为它们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，那么请保存它的一个副本，比如，对于 Text 对象可以使用它的拷贝构造函数：new Text(value)。

这样的情况在 reducer 中也会发生。reducer 中的 value 对象也会被复用，所以如果需要的话，也必须保存它的一个副本(参考例 8-14)。

最后，MapRunner 只是运行 mapper 的一种方式。MultithreadedMapRunner 是另一个 MapRunnable 的实现，它可以使用指定个数的线程并行地运行 map 函数(使用 mapred.map.multithreadedrunner.threads 来设置)。对于大多数数据处理的情况，它对于 MapRunner 没有优势。但是，对于那些由于需要连接外部服务器的、处理单个记录时间比较长的 mapper 来说，它可以使多个 mapper 在同一个 JVM 下以尽量避免竞争方式执行。(可以参考第 14 章。它有一个使用 MultithreadedMapRunner 的应用。)

FileInputFormat

FileInputFormat 是所有使用文件作为数据源的 InputFormat 实现的基类(见图 7-2)。它提供了两样东西：一个配置定义哪些文件被包含在一个任务里；一个生成输入分片的实现。将输入分片分割成记录是其子类的“职责”。

FileInputFormat 的输入路径

一个作业的输入是以一个路径的集合来表示的，它在如何约束输入方面提供了很大的灵活性。FileInputFormat 提供了 4 个静态方法来设定一个 JobConf 的输入路径：

```
public static void addInputPath(JobConf conf, Path path)
public static void addInputPaths(JobConf conf, String
```

```
commaSeparatedPaths)
public static void setInputPaths(JobConf conf, Path...
inputPaths)
public static void setInputPaths(JobConf conf, String
commaSeparatedPaths)
```

其中，addInputPath()和 addInputPaths()方法可以将一个或多个路径加入一个 JobConf 的路径列表；setInputPaths()方法一次设定所有的路径，并且会覆盖之前对路径列表做的所有操作。

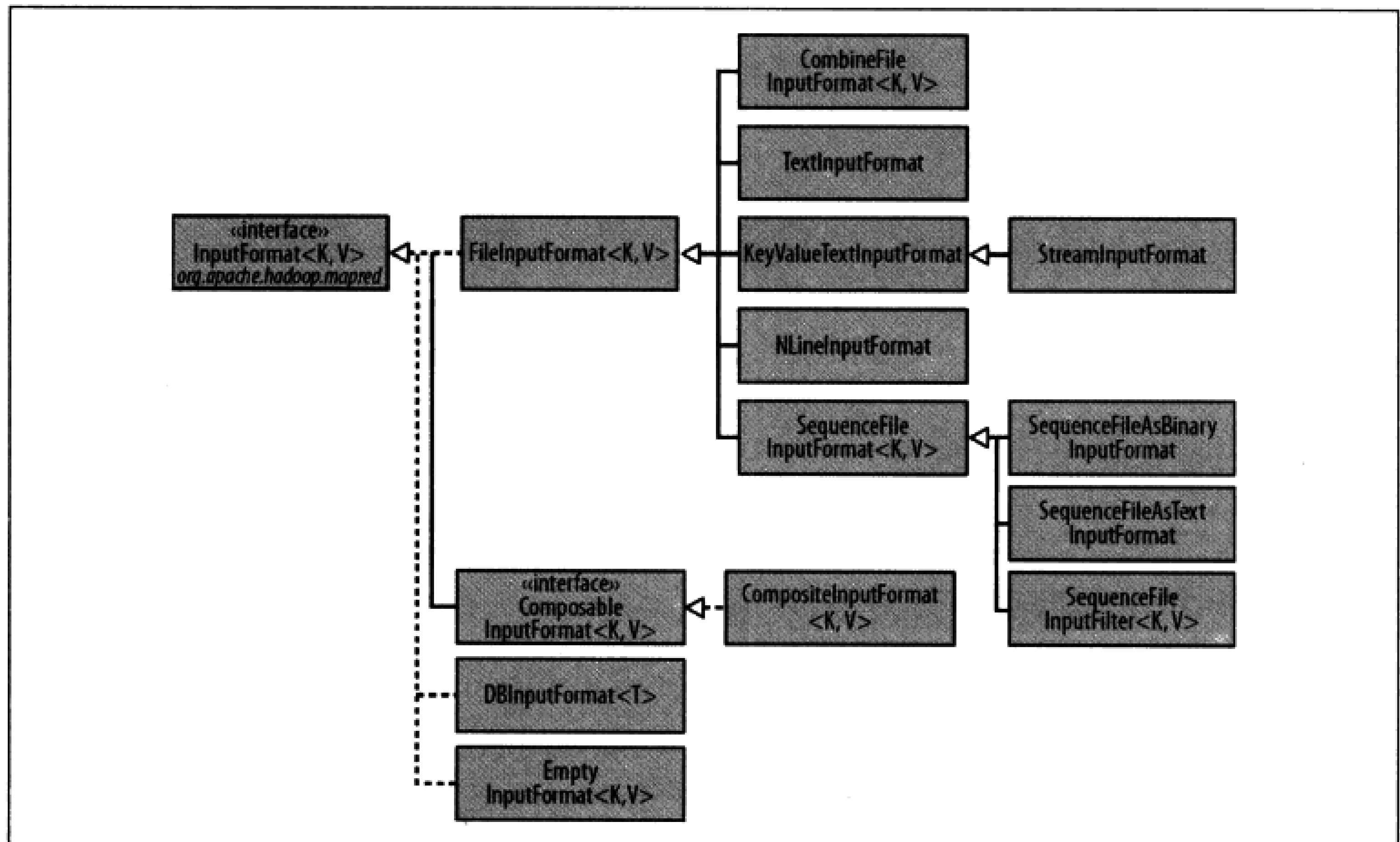


图 7-2：InputFormat 类的层次结构

一个路径可以是一个文件一个目录或者是一个 glob，即一个文件和目录的集合。目录路径包含这个目录下的所有文件。

对目录路径的处理并不是递归的。事实上，这些目录只能包含文件。如果包含文件夹，那么它会被当作(普通)文件处理，从而产生错误。为此，可以使用一个文件 glob 或者一个过滤器来选择需要处理的文件。

add 和 set 方法可以指定需要包含的文件。如果需要指定一些需要被排除的文件，可以使用 setInputPathFilter()在 FileInputFormat 中设置一个过滤器：

```
public static void setInputPathFilter(JobConf conf, Class<?
extends PathFilter> filter)
```

过滤器的详细介绍可参见第 3 章的“PathFilter”。

即使不手动设置过滤器，`FileInputFormat` 也会自动设置一个默认的过滤器来排除隐藏文件(即以“.”和“_”开头的文件)。如果调用 `setInputPathFilter()`，那么它会在原有默认过滤器的基础上增加一个过滤器。也就是说，自定义的过滤器只能看到非隐藏文件。

路径和过滤器在配置属性中也能设置(见表 7-3)。这在流操作和管道操作的时候会非常有用。在流和管道操作的时候一般使用`-input` 选项来指定，不需要手动设置。

表 7-3：输入路径和过滤器

属性名称	类型	默认值	描述
mapred.input.dir	逗号分隔的路径	无	作业的输入文件。包含逗号的路径都可由转义字符避免，例如：{a,b} 可以变成 {a\,b}
mapred.input.path Filter.class	PathFilter 类名	无	应用于作业的输入文件的 filter

FileInputFormat 的输入分片

`FileInputFormat` 是如何将文件转换为输入分片的呢？`FileInputFormat` 只会分割大文件。这里的“大”指的是大于 HDFS 块的大小，这在大多应用程序中是合理的；然而，这个值也可以通过设置不同的 Hadoop 变量而改变(详见表 7-4)。

表 7-4：控制分片大小的属性

属性名称	类型	默认值	描述
mapred.min.split.size	int	1	文件分片最小字节大小
mapred.max.split.size	long	Long.MAX_VALUE 9223372036854775807	文件分片最大字节大小
dfs.block.size	long	64 MB，即 67108864	HDFS 中的块大小(按字节)

最小的分片大小是 1 个字节，不过某些设置可以使它低于这个值。

应用程序可以强制设置输入分片的大小——将它设得比 HDFS 块的大小大一些，以强制使分片比文件块大。如果数据存储在 HDFS 上，那么这样做是没有好处的，

因为这样做会增加很多 map 操作不是本地的文件块。

最大的分片大小默认是 Java 中 long 类型的最大值。当它的值被设置成小于块大小时，那么将强制分片的大小小于块。

分片的大小是由下述的公式得到的(参见 `FileInputFormat` 的 `computeSplitSize()` 方法)：

```
max(minimumSize, min(maximumSize, blockSize))
```

默认情况下：

```
minimumSize < blockSize < maximumSize
```

所以分片的大小就是块的大小。不同的设置以及得出的不同的分片大小见表 7-5。

表 7-5：控制分片的大小

分片大小的最小值	分片大小的最大值	块的大小	分片的大小	说明
1(默认值)	Long.MAX_VALUE (默认值)	64 MB (默认值)	64 MB	默认分片大小与默认块大小一致
1(默认值)	Long.MAX_VALUE (默认值)	128 MB	128 MB	增加分片大小最自然的方法是有更大的 HDFS 块，可以通过设置 dfs.block.size 或者在构造文件时基于单个文件
128 MB	Long.MAX_VALUE (默认值)	64 MB (默认值)	128 MB	使得分片大小的最小值大于块大小，通过这种方式增大分片大小，但是这是以本地为代价的
1(默认值)	32 MB	64 MB (默认值)	32 MB	使得分片大小的最大值小于块大小，通过这种方式减少分片大小

小文件与 CombineFileInputFormat

相对处理少量的大文件而言，Hadoop 在处理大量小文件时的性能稍微逊色一些。一个原因是 FileInputFormat 生成的 InputSplit 总是一整个或一部分的输入文件。如果文件比较小(“小”的定义是指文件的大小远远小于 HDFS 的块大小)，并且数量很多，那么每次 map 操作的时候只会处理很少的输入数据，但是会有很多 map 任务，每次新的 map 操作都会造成一定的性能损失。试将一个 1 GB 的文件分割成 16 个 64 MB 的文件，和将它分割成 10 000 个 100 KB 的文件，对这两者进行处理。这 10 000 来个文件每个都需要进行一次 map 操作，这将比只需要 16 次 map 操作每次处理 64 MB 数据慢几十甚至上百倍。

CombineFileInputFormat 可以缓解这个问题。它对这种情况做了一定的优化。FileInputFormat 将每个文件分割成 1 个或多个单元，而 CombineFileInputFormat 可以将多个文件打包到一个输入单元中，这样每次 map 操作就会有更多的数据来处理。更重要的是，CombineFileInputFormat 会考虑到节点和集群(相对于数据)的位置信息以决定哪些文件应该被打包到一个单元中，所以原本 MapReduce 的效率并不会下降。

当然，如果可能，应该尽量避免许多小文件。因为 MapReduce 处理数据的最佳速度最好与数据在集群中传输的速度相同，而处理小文件将增加作业的 seek 操作数。还有，在 HDFS 集群中存储大量的小文件将会加重名称节点的负担。一个可以减少大量小文件的方法是使用 SequenceFile 将这些小文件合并成一个或多个大文件：可以将文件名作为键(如果不需要键，可以用 NullWritable 代替)，值可以是文件的内容。详情可见例 7-4。但如果在 HDFS 中已经有大量的小文件，最好使用 CombineFileInputFormat。

注意： CombineFileInputFormat 不仅仅可以处理大量小文件，它在处理大文件的时候也有好处。CombineFileInputFormat 使 map 操作中处理的数据量与 HDFS 的块大小之间的耦合度降低了。

如果 mapper 可以在几秒钟之内处理一个块的数据，则可以将 CombineFileInputFormat 的最大单元大小设成块大小的较小的整数倍(通过 mapred.max.split.size 设定)，以使每个 map 操作处理多个块。这样的话，整个 map 的操作时间减少，因为 mapper 减少，意味着启动一个比较小的 map 操作所浪费的时间更少。

由于 CombineFileInputFormat 是一个抽象类，并且没有提供继承它的实体类，

所以使用的时候需要一些额外的工作(希望日后会有一个默认的实现)。比如，如果要使 CombineFileInputFormat 与 TextInputFormat 相同，需要创建一个 CombineFileInputFormat 的实体类，并且实现 getRecordReader()方法。

避免分割

有些应用程序可能并不希望它们的输入文件被分割成若干个单元，这样一来，每个 mapper 可以处理一整个文件。比如，检查文件中的记录是否有序的一个简单的方法是顺序扫描每一条记录并且比较前一条记录是否比后一条要小(或相等)。如果要将它实现为一个 map 操作，那么这个 map 操作必须能够访问整个文件[1]。

有很多方法可以保证输入文件不被分割。第一种(最简单但不怎么漂亮的)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，当然也可以将它设置成 long.MAX_VALUE，也就是输入分片大小的上限。另外一种方法就是继承 FileInputFormat 实体类，并且重载 isSplitable()方法[2]，将它的返回值设置为 false。例如，以下就是一个不可分割的 FileInputFormat：

```
import org.apache.hadoop.fs.*;
import org.apache.hadoop.mapred.TextInputFormat;

public class NonSplittableTextInputFormat extends
TextInputFormat {
  @Override
  protected boolean isSplitable(FileSystem fs, Path file) {
    return false;
  }
}
```

mapper 中的文件信息

处理输入分片的 mapper 可以从作业配置对象的某些特定属性值中获得输入分片的有关信息。可以通过在 mapper 中实现 configure()方法来获取 JobConf 对象。表 7-6 列举了其属性值。这些是对前面所有 mapper 和 reducer 属性的扩展。

① SortValidator.RecordStatsChecker 的 mapper 就是这样实现的。

② 这个方法的名字是 isSplitable，但是(在英语中)一般的拼写方法是 splittable(双写 t)。

表 7-6：文件输入分片的属性值

属性名称	类型	说明
map.input.file	String	正在处理的输入文件路径
map.input.start	long	分片开始处的字节偏移量
map.input.length	long	分片的长度(按字节)

下一节中将讨论如何使用这些属性访问输入分片的文件名。

将整个文件作为一条记录处理

有些情况下，需要 mapper 能够访问整个文件中的内容。对此，一部分是不对文件进行分割，另一部分则是需要一个 RecordReader 来读取文件内容作为 map 操作的值，见例 7-2 的 WholeFileInputFormat。

例 7-2：将整个文件作为一条记录的 InputFormat

```
public class WholeFileInputFormat
    extends FileInputFormat<NullWritable, BytesWritable> {

  @Override
  protected boolean isSplitable(FileSystem fs, Path filename) {
    return false;
  }

  @Override
  public RecordReader<NullWritable, BytesWritable> getRecordReader(
      InputSplit split, JobConf job, Reporter reporter) throws
      IOException {

    return new WholeFileRecordReader((FileSplit) split, job);
  }
}
```

在 WholeFileInputFormat 中，键是被忽略的，它被表示成 NullWritable，而值被表示成 BytesWritable 对象。它定义了两个方法：一个是将 isSplitable()方法重载成返回 false 值，以表示不分割文件；另一个是重载 getRecordReader()方法返回一个定制的 RecordReader 来读取整个文件。

例 7-3：WholeFileInputFormat 的 RecordReader 将整个文件作为一条记录

```
class WholeFileRecordReader implements RecordReader<NullWritable,
```

```
BytesWritable> {

  private FileSplit fileSplit;
  private Configuration conf;
  private boolean processed = false;

  public WholeFileRecordReader(FileSplit fileSplit, Configuration
      conf) throws IOException {
    this.fileSplit = fileSplit;
    this.conf = conf;
  }

  @Override
  public NullWritable createKey() {
    return NullWritable.get();
  }

  @Override
  public BytesWritable createValue() {
    return new BytesWritable();
  }

  @Override
  public long getPos() throws IOException {
    return processed ? fileSplit.getLength() : 0;
  }

  @Override
  public float getProgress() throws IOException {
    return processed ? 1.0f : 0.0f;
  }

  @Override
    public boolean next(NullWritable key, BytesWritable value)
      throws IOException {
      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 void close() throws IOException {
    // do nothing
  }
}
```

WholeFileInputFormat 负责将 FileSplit 转换成一整条记录，记录的键是 null 而值是这个文件的内容。因为只有一条记录，所以 WholeFileInputFormat 维护一个 processed 变量来表示记录是否已经被处理过。如果 next()方法被调用，而 processed 是 false，就将目标文件打开，并用 Hadoop 的 IOUtils 把文件的内容放入 byte 数组，数组的长度就是文件的长度。然后将这个数组传递给 BytesWritable 对象，最后返回 true 以表示成功读取记录。

其他一些方法都是一些简单的用来生成正确键/值对、获取 RecordReader 位置和状态的方法。另外，还有一个 close()方法，在 MapReduce 结束时使用。

接下来看一下如何使用 WholeFileInputFormat。设想有一个 MapReduce 任务将小文件打包成一个 SequenceFile，键是原来的文件名，值是对应文件的内容。如例 7-4 所示。

例 7-4：将若干个小文件打包成 SequenceFile 的 MapReduce 任务

```
public class SmallFilesToSequenceFileConverter extends Configured
  implements Tool {

  static class SequenceFileMapper extends MapReduceBase
      implements Mapper<NullWritable, BytesWritable, Text,
        BytesWritable> {

    private JobConf conf;

    @Override
    public void configure(JobConf conf) {
```

```
    this.conf = conf;
  }

  @Override
  public void map(NullWritable key, BytesWritable value,
      OutputCollector<Text, BytesWritable> output, Reporter
      reporter) throws IOException {

    String filename = conf.get("map.input.file");
    output.collect(new Text(filename), value);
  }

}

@Override
public int run(String[] args) throws IOException {
  JobConf conf = JobBuilder.parseInputAndOutput(this, getConf(),
    args);
  if (conf == null) {
    return -1;
  }

  conf.setInputFormat(WholeFileInputFormat.class);
  conf.setOutputFormat(SequenceFileOutputFormat.class);

  conf.setOutputKeyClass(Text.class);
  conf.setOutputValueClass(BytesWritable.class);

  conf.setMapperClass(SequenceFileMapper.class);
  conf.setReducerClass(IdentityReducer.class);

  JobClient.runJob(conf);
  return 0;
}

public static void main(String[] args) throws Exception {
  int exitCode = ToolRunner.run(new
    SmallFilesToSequenceFileConverter(), args);
  System.exit(exitCode);
}
}
```

由于输入的格式是 WholeFileInputFormat，所以 mapper 只需要找到输入文件分

片的文件名，它是通过访问 `JobConf` 对象的 `map.input.file` 属性，而此属性被 MapReduce 框架设置成输入分片的文件名，但只针对 `FileSplit` 的实例(包括大多数 `FileInputFormat` 的子类)。Reducer 的类型是 `IdentityReducer`，而输出格式是 `SequenceFileOutputFormat`。

以下是在一些小文件上运行的结果，我们使用了两个 reducer，所以会生成两个 SequenceFile：

```
% hadoop jar job.jar SmallFilesToSequenceFileConverter \
-conf conf/hadoop-localhost.xml -D mapred.reduce.tasks=2
input/smallfiles output
```

结果中有两个 SequenceFile，可以通过 HDFS 中的`-text` 选项来进行检查：

```
% hadoop fs -conf conf/hadoop-localhost.xml -text output/part-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-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，除了 e 文件以外，每个文件分别包含 10 个相对应的字母(比如，a 文件中有 10 个“a”字母)，e 文件的内容为空。我们可以看到这些文件的文本表示，即它们文件名后跟着它们的十六进制的表示。

至少有一种方法可以改进我们的程序。就像先前曾经提到过的，让每个 mapper 处理一个文件是不合算的，所以最理想的是继承 `CombineFileInputFormat` 而不是 `FileInputFormat`(在相关的示例代码中有实现)。同样，如果想将文件打包成 Hadoop Archive 格式而不是 SequenceFile，详情可参考第 3 章的 Hadoop Archive 一节。

7.2.2 文本输入

Hadoop 非常擅长处理非结构化文本数据。本节将讨论处理文本数据的输入格式。

TextInputFormat

TextInputFormat 是默认的 InputFormat，每一行数据都是一条记录。它的键是 LongWritable 类型的，存储该行在整个文件中的偏移量(以字节为单位)。值是行中的数据内容，不包括任何行终止符(换行符和回车符等)，它是 Text 类型的。所以，下面的文件：

```
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

将被分割成 4 条记录，每条记录被表示成下面的键/值对形式：

```
(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。在这种情况下很难取得行号，因为文件会被分割成分片，并且每个分片是单独处理的。行号是一个顺序的标记：也就是说每次读取一行的时候需要对行号进行计数。所以在分片内维护行号是可能的，但在文件中是不可能的。

然而，每一行的在文件中的偏移量是可以在分片内单独确定的，而不需要其他分片，因为每个分片都知道自己的上一个分片，那么只需要将先前的分片的大小附加到这个偏移量上就可以知道它在整个文件中的偏移量了。通常，对于每行需要一个唯一标识的应用来说，这样的方法已经足够了。如果再加上文件名，那么这条记录的标识符在全局的文件系统内也是唯一的。当然，如果每一行都是定长的，那么计算行号就仅仅是将这个偏移量除去每一行的长度而已。

输入分片与 HDFS 块之间的关系

FileInputFormat 生成的每一条逻辑记录有时候并不能很好地放入一个 HDFS 的文件块中。比如，TextInputFormat 的每一条记录是一行，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并没有什么影响，如行不会被损或出错，但是这个现象应该引起注意，因为这意味着那些“本地的”map 操作(也就是处理数据的和存储数据的是同一个节点的 map 操作)而言，可能会跨节点地读一部分数据，由此而来的额外的一点开销一般不是特别明显。

图 7-3 说明了这一点：一个文件分成几行，而行的边界与 HDFS 块的边界没有对齐。每个分片的边界与逻辑记录的边界对齐，在这里也就是行边界，所以我们看到第一个分片包含前 5 行，即使第 5 行跨了两个块，而第二个分片从第 6 行开始。

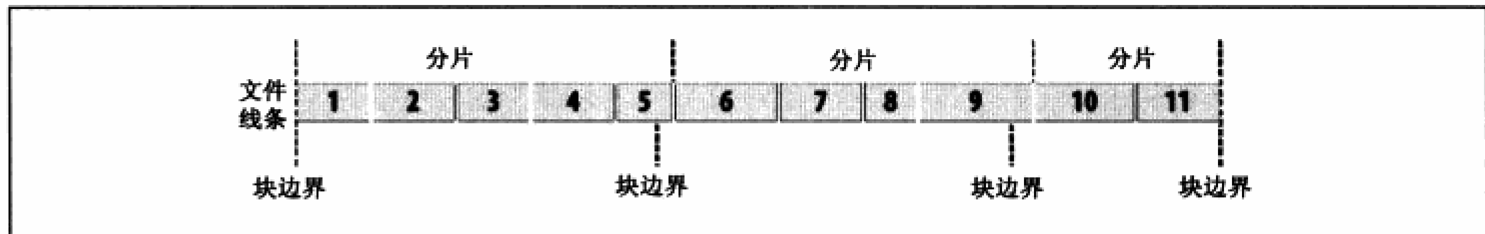


图 7-3：TextInputFormat 的逻辑记录与 HDFS 块

KeyValueTextInputFormat

TextInputFormat 的键，也就是每一行在文件中的字节偏移量，在大多数情况下用处并不大。事实上，一般的做法会让每一行包含一个键/值对，并且键与值之间使用某个分隔符进行区别，比如 Tab 符。比如，以下是由 Hadoop 的 TextOutputFormat 输出的结果，TextOutputFormat 是 Hadoop 默认的 OutputFormat。如果要正确处理这种文件，就需要 KeyValueTextInputFormat。

在 key.value.separator.in.input 中可以设置键与值的分隔符，它的默认值是一个 Tab。以下是一个示例，其中→代表一个(水平方向的)Tab 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.
```

就像前面使用 TextInputFormat 的例子，输入是只有一个分片，这个分片包含四条记录的文件，只不过这次的键是排在 Tab 之前的 Text 的序列。

```
(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.)
```

NLineInputFormat

使用 TextInputFormat 和 KeyValueTextInputFormat 的时候，每个 mapper 会收到数行输入。行的数量与输入分片的大小和行的长度有关。如果想让每个 mapper 收到指定数量的行，就需要使用 NLineInputFormat。就像 TextInputFormat 一

样，键是行的偏移量而值就是行本身。

N 就是每个 mapper 收到的行的数量。当 N 被设置为 1(也就是默认值)时，每个 mapper 会正好收到一行的输入。`mapred.line.input.format.linespermap` 属性能够控制 N 的值。仍以刚才的四行输入为例：

```
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

如果在这个例子里，N 是 2，那么每个输入单元包含两行。一个 mapper 会收到前两行键/值对：

```
(0, On the top of the Crumpetty Tree)
(33, The Quangle Wangle sat,)
```

另一个 mapper 会收到后两行：

```
(57, But his face you could not see,)
(89, On account of his Beaver Hat.)
```

键与值与 `TextInputFormat` 是完全一样的。不一样的是输入分片的生成方法。

通常来说，对少量行进行 map 任务是比较低效的(由于任务初始化的开销)，但是有些应用程序会对少量的数据做一些分散的(也就是 CPU 密集型的)计算任务，然后产生输出。模拟是一个不错的例子。通过创建一个输入文件(指定输入参数)，每行一个，便可以执行一个参数清理：并行运行一系列模拟找出模型是如何随参数而变化的。

注意：在一些时间很长的模拟任务中，可能会出现任务超时的情况。当一个任务在 10 分钟内没有报告它的状态，那么 tasktracker 将视其为任务失败，并且将整个进程终止，(详情参见第 6 章)

解决这个问题最好的方法就是定时地向 tasktracker 报告自己的状态，如发送一段状态信息，或者自增计数器。参见第 6 章。

另一个例子是让 Hadoop 去引导从多个数据源加载数据，如数据库。可以创建一个“种子”文件，记录所有的数据源，一行一个。然后每个 mapper 被分到一个数据源，并从这些数据源中加载数据到 HDFS 中来。这个程序并不需要 reduce 阶段，所以 reducer 的数量应该被设成 0(通过调用 Job 的 `setNumReduceTasks()`)。另外，也可以让 MapReduce 作业处理加载到 HDFS 的数据。相关示例参见附录 C。

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 文档要被分割成多个分片，对每个分片分别处理就会有一定的难度。当然，也可以将整个 XML 文档给一个 mapper 处理(如果工作量不是太大的话)，使用“将整个文件作为一个记录处理”的技术。

由一条条记录(即 XML 文档碎片)组成的 XML 文档可以使用字符串匹配或者正则表达式将它们分割，并且找出开始和结束的标记。这可以解决由 MapReduce 框架进行分割的问题，因为下一个开始标记可以通过从分片的开始处进行扫描而确定，就像 `TextInputFormat` 确定新一行的边界一样。

对此，Hadoop 提供了 `StreamXmlRecordReader` 类(在 `org.apache.hadoop.streaming` 包中，当然，它也可以脱离流处理来使用)。需要使用它的时候，可以将输入格式设定成 `StreamInputFormat`，然后将 `stream.recordreader.class` 设定成 `org.apache.hadoop.streaming.StreamXmlRecordReader`。这个 reader 可以通过设置作业配置属性以获得开始和结束标记的模式。详情见此类的帮助文档[①]。

下面有一个例子：维基百科使用 XML 格式来提供大量的数据，它们都可以使用这种方法在 MapReduce 中进行处理。数据存放在一个大的 XML 文档中，它存储了很多元素，如 `page` 元素表示每一页的内容及其他元数据。使用 `StreamXmlRecordReader`，这些 `page` 元素会被当成记录并且由每个 mapper 进行处理。

7.2.3 二进制输入

Hadoop 的 MapReduce 不仅仅可以处理文本信息，还可以处理二进制数据。

SequenceFileInputFormat

Hadoop 的 `SequenceFileInputFormat` 存储二进制键/值对的序列。由于它们是可分割的，所以它们可以很好地适应 MapReduce 的环境(它们有同步点，这样一来，reader 可以从文件中的任意一点[比如分片的开始点]，找到某条记录的起点)，并且它们还可以支持压缩，以及它们可以使用一些序列化技术来存储任意一种格式。

---

① 访问 *https://issues.apache.org/jira/browse/HADOOP-2439*，了解改进过的 XML 输入格式。

如果需要在 MapReduce 中使用顺序文件，就必须使用 SequenceFileInputFormat。键和值是由顺序文件决定，所以只需要使输入的格式适应这些顺序文件即可以。比如，如果输入文件中键的格式是 IntWritable，而值是 Text，就像我们在第 4 章看到的那样，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。

注意： 虽然无法从名称上看出，但 SequenceFileInputFormat 事实上也可以处理 MapFile。如果在处理顺序文件时遇到了一个目录，SequenceFileInputFormat 会将它当作一个 MapFile 进行处理，从而读取它的数据文件。这样一来，就很清楚为何没有 MapFileInputFormat 了。

SequenceFileAsTextInputFormat

SequenceFileAsTextInputFormat 是 SequenceFileInputFormat 的变体，它将键和值转换为 Text 对象。转换的时候会调用键和值的 toString()方法。这个格式可以使顺序文件作为流操作的输入。

SequenceFileAsBinaryInputFormat

SequenceFileAsBinaryInputFormat 也是 SequenceFileInputFormat 一种变体，它将顺序文件的键和值作为二进制对象。它们被封装为 BytesWlitable 对象，因而应用程序可以任意地将这些字节数组解释为它们想要的类型。如果再结合使用 SequenceFile.Reader 中的 appendRaw()方法，它还能使 MapReduce 可以处理任意二进制数据类型，当然，使用 MapReduce 本身的序列化机制可能是一个更好的方法。详情参见第 4 章。

7.2.4 多种输入

虽然一个 MapReduce 程序可能有多个输入文件(由文件模块(glob)、过滤器和路径组成)，但所有文件都由同一个 InputFormat 类和同一个 Mapper 类来解释。但通常情况下，数据格式会随时间演变，必须使当前的 mapper 能够适应和处理遗留的数据格式。或者，有些数据源会提供相同的数据，但是是不同格式的。这种情况下可能需要对不同的数据集进行联接操作。详情参见第 8 章。例如，有些数据可能是使用 Tab 分隔的文本文件，另一些可能是二进制的顺序文件。哪怕它们有共同的格式，有时候它们的表示是不一样的，这就需要分别进行解析。

这些问题可以用 MultipleInputs 类来解决，它可以在每一个输入路径上规定 InputFormat 和 Mapper 的类型。比如，我们可能想同时使用英国 Met Office[①]的气象数据和 NCDC 的气象数据来估计最高气温，则可以像下面这样设置输入路径：

```
MultipleInputs.addInputPath(conf, ncdcInputPath,
    TextInputFormat.class, MaxTemperatureMapper.class)
MultipleInputs.addInputPath(conf, metOfficeInputPath,
    TextInputFormat.class, MetOfficeMaxTemperatureMapper.class);
```

这段代码取代了通常的对 FileInputFormat.addInputPath() 和 conf.setMapperClass()的调用。Met Office 和 NCDC 的数据都是文本文件，所以可以使用 TextInputFormat。但是这两个文件中每一行的表示是不同的，所以需要两个不一样的 mapper。MaxTemperatureMapper 处理 NCDC 的数据并将它们转换成年份和气温的键/值对，同时 MetOfficeMaxTemperatureMapper 将 Met Office 的数据转换成年份和气温。重要的一点是两个 mapper 的输出类型是一致的，这样就可以使用同一个 reducer 来进行操作。

MultipleInputs 类有一个重载版本的 addInputPath()方法，它没有 mapper 参数：

```
public static void addInputPath(JobConf conf, Path path, class<?
extends InputFormat> inputFormatClass)
```

只有一个 mapper(通过 JobConf 的 setMapper()方法设定)但有多种输入格式的时候会非常有用。

7.2.5 数据库格式的输入/输出

DBInputFormat 是一个使用 JDBC 并且从关系数据库中读取数据的一种输入格式。由于它没有任何碎片技术，所以在访问数据库的时候必须非常小心，太多的 mapper 可能会使数据库受不了。由于这个原因，DBInputFormat 最好是在加载小量的数据集的时候用，可能在通过 MultipleInputs 与来自 HDFs 的大数据集连接时使用。与之相对应的输出格式是 DBOutputFormat，在将数据存入数据库时比较有用[②]。

HBase 中的 TableInputFormat 可以让 MapReduce 程序访问 HBase 中表里的数据。同样，TableOutputFormat 是可以让 MapReduce 的输出写入 HBase。

---

① Met Office 的数据通常只有研究机构和高校才能访问。不过有一些气象站的月数据可以通过 *http://www.metoffice.gov.uk/climate/uk/stationdata/*获得。

② 访问 *http://www.cloudera.com/blog/2009/03/06/database-access-with-hadoop/*，可深入了解详细介绍如何在 Hadoop 中使用数据库格式。

7.3 输出格式

对于前一节中的每一种输入格式，Hadoop 都有一种输出格式与之对应。OutputFormat 类的层次结构如图 7-4 所示。

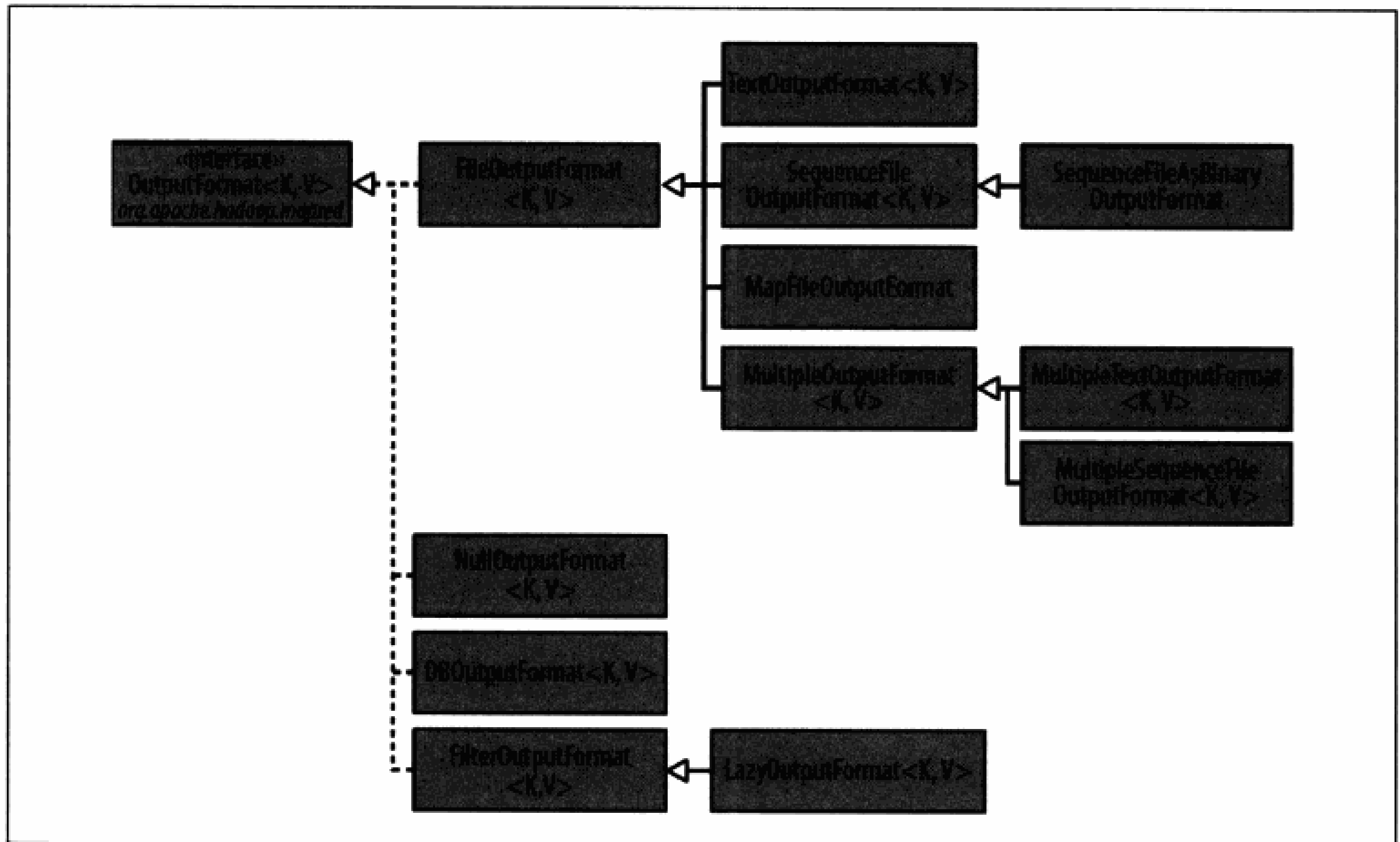


图 7-4：OutputFormat 类的层次结构

7.3.1 文本输出

默认的输出格式 `TextOutputFormat`，将每条记录以一行的形式存入文本文件。它的键和值可以是任意类型，因为通过 `toString()`方法可以将它们转换为字符串类型再进行输出。每个键/值对内部都默认用 Tab 符进行分隔，当然也可以在 `mapred.textoutputformat.separator` 属性中设置分隔符。与它对应的输入格式是 `KeyValueTextInputFormat`，因为它将键/值对通过可配置的分隔符从文本中分隔开来。

如果需要，也可以省略键或值(或者都省略，即使用 `NullOutputFormat`，它什么也不输出)，通过将输出类型设置成 `NullWritable`。同时，也不会有分隔符被输出，这样可以使输出更适合 `TextInputFormat` 处理。

7.3.2 二进制输出

SequenceFileOutputFormat

正如其名，SequenceFileOutputFormat 将它的输出写入一个顺序文件。如果输出需要作为后续 MapReduce 任务的输入，那么这是一个较好的选择，因为它的格式非常紧凑，并且还可以被压缩。对它的压缩可以由 SequenceFileOutputFormat 中的静态方法进行实现，。在第 8 章，介绍了如何使用 SequenceFileOutputFormat。

SequenceFileAsBinaryOutputFormat

SequenceFileAsBinaryOutputFormat 与 SequenceFileAsBinaryInputFormat 相对应，它将键/值对当作二进制数据写入一个顺序文件。

MapFileOutputFormat

MapFileOutputFormat 将结果写入一个 MapFile 中。MapFile 中的键必须是排序的，所以在 reducer 中必须保证输出的键有序。

注意：Reduce 操作的输入键一定是有序的，但是对于它输出的键，由于是 reduce 函数控制的，MapReduce 框架中没有硬性规定它必须有序。所以要使用 MapFileOutputFormat，就必须手动保证输出的键有序。

7.3.3 多个输出

FileOutputFormat 及其一些子类会在某个目录下生成若干个文件。每个 reducer 会输出一个文件并且文件的名字与分区相对应：part-00000，part-00001，等等。有时候可能需要对输出的文件名进行控制，或者让每个 reducer 输出多个文件。这时候需要用到 MultipleOutputFormat 和 MultipleOutputs 类。

实例：数据分区

如果想将所有的气象数据按照气象站分割开，就需要写一个程序，它以每个气象站输出一个文件，并且此文件包含了所有关于该气象站的数据。

一种方法是让每个 reducer 处理一个气象站的数据。这样的话我们必须做两件事：

第一，我们必须实现一个 partitioner 将使同一个气象站的数据分配到同一个分区；第二，我们需要将 reducer 的个数设成气象站的总数。以下是 partitioner 的实现如下：

```
public class StationPartitioner implements
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) {
    ...
  }

  @Override
  public void configure(JobConf conf) { }
}
```

其中，`getPartition(String)`方法并没有给出实现，它的作用是将 stationId 转换成 partitionId。为此，返回这个气象站在所有列表中的索引即可。

这样做有两个弊端。首先，必须在运行这个程序前事先知道气象站的数目和分区的个数。虽然 NCDC 的元数据中包含那些气象站，但是无法保证数据中的气象站 Id 能够与这些元数据完全匹配。如果元数据中有此气象站但是真正的数据中却没有它的数据，那么会浪费一个 reducer slot。更糟糕的是，如果元数据中没有这个气象站，但是真正的数据中出现了它的数据，那么它将不会被 reducer 处理，而是直接被扔掉。一种解决这个问题的方法就是去将所有的气象站从数据中抽取出来，但很遗憾，这需要额外的工作。

另一个弊端更不易察觉。一般来说，对一个应用程序固定分区的个数是一个坏主意，因为可能会导致分区不均匀。让很多 reducer 去做一些比较轻的作业并不是一个很好的组织作业的方法。更好的办法是使用少量的 reducer，而使每个 reducer 做更加多的活，这样的话，启动和维护 reducer 的附加开销就会少很多。不同气象站之间的数据量差异很大，有些气象站是一年前刚投入使用的，而另一些可能已经工作了一个世纪。如果其中一些 reducer 的工作量远远超过另一些，它们就会拖慢整个任务的执行时间，并且这是不必要的。

注意：在以下两种情况下是允许手动设定分区的个数的(或者说是 reducer 的个数)：

1. 没有 reducer。也就是程序只执行 map 任务的时候。
2. 一个 reducer。运行一组小作业，每个作业将前一个作业的输出合并成单个文件时，比较有用。但是前提是数据量足够小，一个 reducer 很快就能完成。

最好能根据集群的容量来决定分区的个数，因为 reducer 的任务槽越多，任务就会越快完成。这就是 `HashPartitioner` 表现如此出色的原因：它可以适应任意数量的 partitioner，并且能够较好的保证 partitioner 之间是比较均匀的。

如果我们使用 `HashPartitioner`，那么每个分区上将包含多个气象站的数据。如果需要实现每个气象站一个输出文件，我们就需要让每个 reducer 写多个文件使用 `MultipleFileOutputFormat`。

MultipleFileOutputFormat 类

`MultipleFileOutputFormat` 类可以输出到多个文件，这些文件的名字是根据输出的键/值对进行区分的。`MultipleFileOutputFormat` 类是个抽象类，它有两个实体类：`MultipleTextOutputFormat` 和 `MultipleSequenceFileOutputFormat` 类。它们分别对应 `TextOutputFormat` 和 `SequenceOutputFormat` 的多文件版本。`MultipleFileOutputFormat` 类提供了一些 `Protected` 方法使得输出时能够控制文件名命名方式。在例 7-5 中，我们将创建 `MultipleTextOutputFormat` 类的一个子类，并且重载 `generateFileNameForKeyValue()` 方法，使它返回一个 `stationId`，这个 `stationId` 是我们从值中抽取出来的。[①]

例 7-5：用 MultipleOutputFormat 类将整个数据集分割到 stationId 文件

```
public class PartitionByStationUsingMultipleOutputFormat extends
  Configured implements Tool {

  static class StationMapper extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, Text> {

    private NcdcRecordParser parser = new NcdcRecordParser();

    public void map(LongWritable key, Text value,
        OutputCollector<Text, Text> output, Reporter reporter)
```

① 此方法仍然不能解决数据分布不均匀的问题(数据倾斜)。例如 stationID 为 0 的气象站有一个月的数据，而 stationID 为 1 的气象站有一整年的数据，则处理后者的 reducer 会花更多时间，从而影响整个作业的运行时间。

```
      throws IOException {
    parser.parse(value);
    output.collect(new Text(parser.getStationId()), value);
    }
  }

  static class StationReducer extends MapReduceBase
    implements Reducer<Text, Text, NullWritable, Text> {

@Override
public void reduce(Text key, Iterator<Text> values,
    OutputCollector<NullWritable, Text> output, Reporter reporter)
    throws IOException {
  while (values.hasNext()) {
    output.collect(NullWritable.get(), values.next());
  }
  }
}

static class StationNameMultipleTextOutputFormat
  extends MultipleTextOutputFormat<NullWritable, Text> {

  private NcdcRecordParser parser = new NcdcRecordParser();

  protected String generateFileNameForKeyValue(NullWritable key,
      Text value,String name) {
    parser.parse(value);
    return parser.getStationId();
  }
}

@Override
public int run(String[] args) throws IOException {
  JobConf conf = JobBuilder.parseInputAndOutput(this, getConf(), args);
  if (conf == null) {
    return -1;
  }

  conf.setMapperClass(StationMapper.class);
  conf.setMapOutputKeyClass(Text.class);
  conf.setReducerClass(StationReducer.class);
  conf.setOutputKeyClass(NullWritable.class);
  conf.setOutputFormat(StationNameMultipleTextOutputFormat.class);
```

```
    JobClient.runJob(conf);
    return 0;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(
      new PartitionByStationUsingMultipleOutputFormat(), args);
    System.exit(exitCode);
  }
}
```

StationMapper 将 stationId 从记录中抽取出来并将它作为键。这样可以让同一个气象站的数据被切分到同一个分区。StationReducer 将这个键变成 NullWritable，这样的话当最终产生输出的时候，每个文件中会只包含气象记录(TextOutputFormat 会忽略 NullWritable 类型)。

最终的结果就是将每个气象站的数据放入属于它的文件里，以下就是在部分数据集上运行的结果：

```
-rw-r--r-- 3 root supergroup 2887145 2009-04-17 10:34 /output/010010-99999
-rw-r--r-- 3 root supergroup 1395129 2009-04-17 10:33 /output/010050-99999
-rw-r--r-- 3 root supergroup 2054455 2009-04-17 10:33 /output/010100-99999
-rw-r--r-- 3 root supergroup 1422448 2009-04-17 10:34 /output/010280-99999
-rw-r--r-- 3 root supergroup 1419378 2009-04-17 10:34 /output/010550-99999
-rw-r--r-- 3 root supergroup 1384421 2009-04-17 10:33 /output/010980-99999
-rw-r--r-- 3 root supergroup 1480077 2009-04-17 10:33 /output/011060-99999
-rw-r--r-- 3 root supergroup 1400448 2009-04-17 10:33 /output/012030-99999
-rw-r--r-- 3 root supergroup 307141 2009-04-17 10:34 /output/012350-99999
-rw-r--r-- 3 root supergroup 1433994 2009-04-17 10:33 /output/012620-99999
```

generateFileNameForKeyValue()方法的返回值实际上是一个相对路径，当然也创建任意深度的子目录。比如，以下方法能够将每个气象站每年的数据切分出来：

```
protected String generateFileNameForKeyValue(NullWritable key,
    Text value, String name) {
  parser.parse(value);
  return parser.getStationId() + "/" + parser.getYear();
}
```

MultipleOutputFormat 还有很多特性没有被讨论，比如它可以复制输入路径的结构，或者它也可以对于只有 map 任务的输出进行控制。更多信息可以参考它的 Javadoc。

MultipleOutputs 类

在 Hadoop 中还有一种方法可以产生多种输出，即 `MultipleOutputs` 类。和 `MultipleOutputFormat` 类不一样的是，它可以产生不同类型的输出。当然，这样就无法控制对输出的命名。例 7-6 演示了如何使用 `MultipleOutputs` 类对数据进行切分。

例 7-6：使用 MultipleOutputs 类将整个数据集切分到以 stationId 命名的文件中

```
public class PartitionByStationUsingMultipleOutputs extends
  Configured implements Tool {

  static class StationMapper extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, Text> {

    private NcdcRecordParser parser = new NcdcRecordParser();

    public void map(LongWritable key, Text value,
        OutputCollector<Text, Text> output, Reporter reporter)
        throws IOException {

      parser.parse(value);
      output.collect(new Text(parser.getStationId()), value);
    }
  }

  static class MultipleOutputsReducer extends MapReduceBase
    implements Reducer<Text, Text, NullWritable, Text> {

    private MultipleOutputs multipleOutputs;

    @Override
    public void configure(JobConf conf) {
      multipleOutputs = new MultipleOutputs(conf);
    }

    public void reduce(Text key, Iterator<Text> values,
        OutputCollector<NullWritable, Text> output, Reporter reporter)
        throws IOException {

      OutputCollector collector = multipleOutputs.getCollector("station",
        key.toString().replace("-", ""), reporter);
      while (values.hasNext()) {
        collector.collect(NullWritable.get(), values.next());
```

```java
      }
    }

    @Override
    public void close() throws IOException {
      multipleOutputs.close();
    }
  }

  @Override
  public int run(String[] args) throws IOException {
    JobConf conf = JobBuilder.parseInputAndOutput(this,
        getConf(), args);
    if (conf == null) {
      return -1;
    }

    conf.setMapperClass(StationMapper.class);
    conf.setMapOutputKeyClass(Text.class);
    conf.setReducerClass(MultipleOutputsReducer.class);
    conf.setOutputKeyClass(NullWritable.class);
    conf.setOutputFormat(NullOutputFormat.class);
                              // suppress empty part file
    MultipleOutputs.addMultiNamedOutput(conf, "station",
      TextOutputFormat.class,NullWritable.class, Text.class);

    JobClient.runJob(conf);
      return 0;
    }
    public static void main(String[] args) throws Exception {
      int exitCode = ToolRunner.run(new
        PartitionByStationUsingMultipleOutputs(), args);
      System.exit(exitCode);
  }
}
```

MultipleOutputs 类是用来在原有输出的基础上增加输出路径的。输出是命名的，并且可能被输出到一个(单命名输出)或多个(多命名输出)文件中。在我们这个例子中，我们需要多个文件，每个文件代表一个气象站。我们调用 MultipleOutputs 中的 addMultiNamedOutput()方法来声明输出的名字(这里是“station”)、输出格式及输出类型。另外，MultipleOutputs 的默认输出类型是 NullOutputFormat。

在 reducer 生成输出的过程中，我们在 configure()方法中构造一个 MultipleOutputs 的实例，并将它赋给一个变量。我们在 reduce()方法中为多命名输出获取一个 OutputCollector。getCollector()方法接收一个名字(也就是“station”)参数和一个字符串标识符参数。在这里我们的标识符是气象站，其中的“-”符号被忽略了，因为 MultipleOutputs 的命名中只能包含字母和数字。

最终的结果输出到多个文件，其中文件的命名格式是 station-<station 标识符>-r-<分区索引>。文件名中有一个 r 是因为这个文件是 reducer 产生的。最后跟一个分区索引是保证不同的 reducer 对同一个气象站进行输出的时候不会有冲突。由于我们都是按站点进行分割的，所以在这里不会有冲突(但在一般情况不会)。

运行一次后，部分输出文件如下所示(目录列表中的其他列已被删除)：

```
/output/station_01001099999-r-00027
/output/station_01005099999-r-00013
/output/station_01010099999-r-00015
/output/station_01028099999-r-00014
/output/station_01055099999-r-00000
/output/station_01098099999-r-00011
/output/station_01106099999-r-00025
/output/station_01203099999-r-00029
/output/station_01235099999-r-00018
/output/station_01262099999-r-00004
```

MultipleOutputFormat 和 MultipleOutputs 的区别

它们几乎做了同样的事情。区别如下表所示。

特性	MultipleOutputFormat	MultipleOutputs
完全控制文件名和目录名	是	否
不同输出有不同键/值类型	否	是
使用同一作业的 map 和 reduce	否	是
每个记录多种输出	否	是
使用任意 OutputFormat	否，需要子类	是

总而言之，MultipleOutputs 的功能更加完善，而 MultipleOutputFormat 对输出的目录结构和文件命名有着更多控制。

7.3.4 延迟输出

FileOutputFormat 的子类会产生形如“part-nnnnn”之类的输出文件，哪怕它们是空的。有些应用程序会要求不创建空文件。这时可以使用 LazyOutputFormat。它是一个 wrapper，并可以保证在第一条记录输出的时候才真正创建文件。使用它的时候，调用 JobConf 的 setOutputFormatClass()方法，并且将它和真正的输出格式一起传入。

流处理的时候可以增加-lazyOutput 参数来开启 LazyOutputFormat。

7.3.5 数据库输出

在 7.2.5 节已经讲过如何向关系数据库和 HBase 写入数据。

第 8 章

MapReduce 特性

本章主要关注 MapReduce 一些更高级的特性，包括计数器、排序和数据集的连接。

8.1 计数器

对于正在进行分析的数据，往往有很多东西是你希望知道，但对于分析并不重要。例如，在统计无效记录时，发现无效记录占整个数据集的比例很高，你就可能想了解为什么有这么多记录被标记为无效——也许是检查无效记录的程序中有错误？或者数据质量差，的确有许多无效记录。在发现这些之后，你可能决定增加数据集的大小以便能有更多好记录满足有意义分析的需要。

计数器是一个非常有用的途径，用于收集有关作业的统计数据，无论是对于质量控制，还是应用层面的统计数据。它还有助于对问题的诊断。如果想在 map 或 reduce 任务中添加一条日志记录信息，那么通常更好的方法看是否可以用一个计数器来记录特定情况的发生。对于大型分布式作业来说，计数器的值比日志记录输出更容易获得，同时还可以得到这种条件发生次数的记录，而这比从日志文件集得到该记录容易得多。

8.1.1 内置计数器

Hadoop 对于每个作业都维护着一些内置计数器(见表 8-1)，它们为作业记录了不同

方面的内容。例如：有字节和记录处理数量的，可由此确认是否消耗了符合预期数量的输入并且产生了符合预期数量的输出。

表 8-1：内置计数器

组别	计数器	描述
Map-Reduce 框架	Map input records	在作业中被所有 map 消费的输入记录数量。在一个记录被 RecordReader 读入并传入 map()方法时增加
	Map skipped records	在作业中被所有 map 跳过的输入记录数量
	Map input bytes	在作业中被所有 map 消费的未压缩的输入字节数量。一个记录被 RecordReader 读入并传入 map()方法时增加
	Map output records	在作业中被所有 map 生产的输出记录数量。在 map 的 OutputCollector 上 collect()方法被调用时增加
	Map output bytes	在作业中被所有 map 生产的未压缩的输出字节数量。在 map 的 OutputCollector 上 collect()方法被调用时增加
	Combine input records	在作业中被所有 combiner 消费的输入记录数量。每当一个值被 combiner 的值迭代器读入时增加。注意，该计数器是被 combiner 消耗的值的数量，而不是不同键组的数量(因为不一定一组的每个键对应一个 combiner，因此它这个指标并不是很有用
	Combine output records	在作业中被所有 combiner 生产的输出记录数量。在 combiner 的 OutputCollector 上 collect()方法被调用时增加
	Reduce input groups	在作业中被所有 reducer 消费的不同键组数量。该框架调用 reducer 的 reduce()函数时增加

续表

组别	计数器	描述
	Reduce input records	在作业中被所有 reducer 消费的输入数量。一个值被 reducer 的迭代器读入时增加。如果 reducer 消费了所有的输入那么该计数器应该和 map 输出记录数量相同
	Reduce output records	在作业中被所有 reduce 生产的 reduce 输入记录数量。在 reducer 的 OutputCollector 上 collect()方法被调用时增加
	Reduce skipped groups	在作业中被所有 reducer 跳过的不同键组数量
	Reduce skipped Records	在作业中被所有 reducer 跳过的输入记录数量
	Spilled records	作业中在所有 map 和 reduce 任务中溢出磁盘的记录数量
文件系统	Filesystem bytes read	map 和 reduce 任务通过每个文件系统读取的字节数。每个文件系统一个计数器：文件系统可能是本地，HDFS，S3，KFS 等
	Filesystem bytes written	map 和 reduce 任务通过每个文件系统写入的字节数
作业计数器	Launched map tasks	已启动的 map 任务数。包含推测式启动的任务
	Launched reduce tasks	已启动的 reduce 任务数。包含推测式启动的任务
	Failed map tasks	失败的 map 任务数
	Failed reduce tasks	失败的 reduce 任务数
	Data-local map tasks	运行于和输入数据同一节点上的 map 任务数
	Rack-local map tasks	运行于和输入数据在同一机架的结点上的 map 任务数
	Other local map tasks	运行于和输入数据在不同机架的节点上的 map 任务数。由于内部机架带宽稀缺，并且 Hadoop 尽力将 map 任务放在靠近它们输入数据的地方，所以这个计数器的值应该是很低的

计数器被与它们相关的任务所维护，并定期发送到 tasktracker，然后发送到 jobtracker，这样就能在全局范围内汇总这些计数器(第 6 章“进展和状态更新”有详细描述)。这些内置的作业计数器实际上是 jobtracker 维护的，因此它们不必通过网络发送，这点不同于其他计数器，包括用户自定义计数器等。

任务的计数器的内容每次都被全部发送，而不是从上次传送的内容之后发送，此举保证了不会因为丢失消息而出错。此外，作业运行时，如果任务失败计数器可能会减少。作业成功完成之后，计数器的值才是确定的。

8.1.2 用户自定义 Java 计数器

MapReduce 允许用户代码定义一组计数器。它们在 mapper 或 reducer 中可根据需要递增。计数器被定义为 Java 枚举型，这可以使相互关联的计数器成组。一个作业可以定义任意数量的枚举类型，每个枚举型可以有任意数量的字段。枚举的名称就是该组的名称，枚举的字段就是计数器的名称。计数器是全局的：MapReduce 框架在作业的最后汇总所有 map 和 reduce 中的计数器信息来统计出总数。

我们已在第 5 章中创建了一些计数器用于计算天气数据集中的错误记录数，例 8-1 中的程序扩展了这个例子来计算丢失记录的数量和气温质量代码的分布。

例 8-1：运行最高气温作业的应用程序，包括计算丢失及畸形记录和质量代码

```
public class MaxTemperatureWithCounters extends Configured implements Tool {

  enum Temperature {
    MISSING,
    MALFORMED
  }

  static class MaxTemperatureMapperWithCounters extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

    private NcdcRecordParser parser = new NcdcRecordParser();

    public void map(LongWritable key, Text value,
        OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {

      parser.parse(value);
      if (parser.isValidTemperature()) {
          int airTemperature = parser.getAirTemperature();
          output.collect(new Text(parser.getYear()),
            new IntWritable(airTemperature));
       } else if (parser.isMalformedTemperature()) {
        System.err.println("Ignoring possibly corrupt input: " + value);
        reporter.incrCounter(Temperature.MALFORMED, 1);
```

```
      } else if (parser.isMissingTemperature()) {
        reporter.incrCounter(Temperature.MISSING, 1);
      }

      // dynamic counter
      reporter.incrCounter("TemperatureQuality", parser.getQuality(), 1);

    }
  }

  @Override
  public int run(String[] args) throws IOException {
    JobConf conf = JobBuilder.parseInputAndOutput(this, getConf(),
      args);
    if (conf == null) {
        return -1;
    }

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

    conf.setMapperClass(MaxTemperatureMapperWithCounters.class);
    conf.setCombinerClass(MaxTemperatureReducer.class);
    conf.setReducerClass(MaxTemperatureReducer.class);

    JobClient.runJob(conf);
    return 0;
  }
  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new MaxTemperatureWithCounters(),
      args);
    System.exit(exitCode);
  }
}
```

了解这段程序用途的最好方法是在完整的数据集上运行它：

```
% hadoop jar job.jar MaxTemperatureWithCounters input/ncdc/all
output-counters
```

作业成功完成后，它在最后打印出计数器(由 JobClient 的 runJob()方法完成)。下面是我们感兴趣的部分：

```
09/04/20 06:33:36 INFO mapred.JobClient: TemperatureQuality
```

```
09/04/20 06:33:36 INFO mapred.JobClient:  2=1246032
09/04/20 06:33:36 INFO mapred.JobClient:  1=973422173
09/04/20 06:33:36 INFO mapred.JobClient:  0=1
09/04/20 06:33:36 INFO mapred.JobClient:  6=40066
09/04/20 06:33:36 INFO mapred.JobClient:  5=158291879
09/04/20 06:33:36 INFO mapred.JobClient:  4=10764500
09/04/20 06:33:36 INFO mapred.JobClient:  9=66136858
09/04/20 06:33:36 INFO mapred.JobClient: Air Temperature Records
09/04/20 06:33:36 INFO mapred.JobClient:  Malformed=3
09/04/20 06:33:36 INFO mapred.JobClient:  Missing=66136856
```

动态计数器

该代码利用了一个动态计数器，而这不是由一个 Java 枚举型定义的。因为一个 Java 枚举型的字段是在编译时定义的，不能在正使用的枚举型中创建新的计数器。这里我们想计算气温质量代码的分布，虽然格式规范定义了它可以选取的值，但用一个动态计数器来输出它实际取得的值更方便。我们在 Reporter 对象里使用的方法需要一个用字符串命名的组和计数器名称：

```
public void incrCounter(String group, String counter, long amount)
```

使用枚举型和使用字符串这两种创建和访问计数器的方法实际上是相同的，因为 Hadoop 是将枚举型转成字符串通过远程过程调用(RPC)发送计数器的。枚举使用起来略微方便一些，提供类型安全，适合大部分作业。在少数情况下，需要动态创建计数器时，可以使用字符串接口。

易读的计数器名

默认情况下，计数器的名称是严格限定的 Java 枚举类名。当这些名称出现在 Web 界面或是控制台时，它们就不是非常容易读。Hadoop 提供了一种用资源绑定的方法来改变显示名称。我们这里已经做了这一步，因此我们看见了“Air Temperature Records” 而不是“Temperature$MISSING”。对于动态计数器，组或计数器名称就用来作为显示名称，因此这通常不是问题。

提供易读名称的方法如下：在枚举后面创建一个属性文件，使用下划线作为内嵌类的分隔符。属性文件应该与包含该枚举类型的最顶层类在相同目录。在例 8-1 中，该文件就是命名为 *MaxTemperatureWithCounters_Temperature.Properties* 的计数器属性文件。

属性文件应该包含一个名为 `CounterGroupName` 的属性，它的值就是整组的显示名称。在枚举中的每个字段应有一个相对应的属性，其名称为字段名称加上.name

的后缀名，其值就是计数器显示名称。下面是 *MaxTemperatureWithCounters_Temperature.Properties* 的内容：

```
CounterGroupName=Air Temperature Records
MISSING.name=Missing
MALFORMED.name=Malformed
```

Hadoop 使用标准 Java 本地化机制来加载符合当前运行环境的属性。例如：可以在名为 MaxTemperatureWithCounters_Tempera ture_zh_CN.Properties 的文件中创建中文版的属性，在中文环境中运行时它们就会被用到。参考有关 java.util.Property-ResourceBundle 文档可获取更多信息。

获取计数器

除了通过 Web 界面和命令行(使用 `hadoop job-counter`)，还可以通过 Java API 获取计数器。你可以一边运行作业一边获取，不过更通常的做法是在作业运行结束后获取计数器，这时它们的值是固定的。例 8-2 的程序计算了缺少气温字段的记录的比例。

例 8-2：此应用程序计算缺少气温字段的记录的比例

```
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.mapred.*;
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;
    }
    JobClient jobClient = new JobClient(new JobConf(getConf()));
    String jobID = args[0];
    RunningJob job = jobClient.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.getCounter(
      MaxTemperatureWithCounters.Temperature.MISSING);

    long total = counters.findCounter("org.apache.hadoop.mapred.
        Task$Counter","MAP_INPUT_RECORDS").getCounter();

    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，我们通过 JobID 并调用 getJob()方法从 JobClient 中获取 RunningIob 对象，然后检查指定 ID 是否有对应的作业。它们可能没有，这有可能是由于 ID 号错误或是因为 jobtracker 不再有该作业记录(只有最新的 100 个作业被保存在内存中，并且如果 jobtracker 重启则都被清空)。

在确认作业已经完成后，我们调用 RunningJob 的 getCounters()方法，该方法返回一个计数器对象，封装了一个作业的所有计数器。Counters 类提供了查找计数器名称和值的各种方法。我们用 getCounter()方法得到一个枚举来找到缺少气温字段的记录的数量。

还有 findCounter()方法返回 Counter 对象。我们用这种方法得到内置 map 输入记录计数器。为此，我们通过它的组名(严格限定的枚举类名)和计数器名(也可是字符串)[1]来引用计数器。

最后，打印缺少气温域记录的比例。下面是从整个气象数据集得出的：

```
% hadoop jar job.jar MissingTemperatureFields job_200904200610_0003
```

① 内置计数器的枚举目前还不是公共 API 的一部分，所以只能通过这一方法来获得它们。可通过 *https://issues.apache.org/jira/browse/HADOOP-4043* 弥补这一缺陷。

```
Records with missing temperature fields: 5.47%
```

8.1.3 用户自定义流计数器

MapReduce 流程序可以通过发送一条特殊的格式化行信息到标准错误流来增加计数器，这是在该情况下的一种控制方法。该行信息必须具有如下格式：

```
reporter:counter:group,counter,amount
```

这个 Python 片段显示了如何让"Temperature"组中的"Missing"计数器增加 1：

```
sys.stderr.write("reporter:counter:Temperature,Missing,1\n")
```

类似地，状态消息可能会以如下格式发送：

```
reporter:status:message
```

8.2 排序

对数据进行排序是 MapReduce 的核心。即使应用程序并不关心排序本身，也可以使用 MapReduce 提供的排序机制来组织数据。在本节中，我们将研究数据集排序的不同方式，看看如何控制 MapReduce 的排序顺序。

8.2.1 准备

我们将对气象数据集根据气温进行排序。存储为 `Text` 对象的气温记录不适合排序，因为有符号整型不是根据字典顺序来排序。相反，我们将用顺序文件来存储数据，这些顺序文件的 `IntWritable` 键代表气温(排序正确)，它的 `Text` 的值是数据行。[①]

在例 8-3 中的 MapReduce 作业是一个只有 map 的作业，同时过滤掉那些无效气温读数记录的输入。每个 map 创建一个压缩过的顺序文件作为输出。这是调用了下面的命令：

```
% hadoop jar job.jar SortDataPreprocessor input/ncdc/all
input/ncdc/all-seq
```

① 一个针对此问题的常用方法(特别是是基于文本的流应用)是增加一个消除所有负数的偏移量，并在左侧补 0，这样所有的数都是相同的字符数了。另一种方法可以参见本章的"流"小节。

例 8-3：将气象数据转换成 SequenceFile 格式的 MapReduce 程序

```
public class SortDataPreprocessor extends Configured implements Tool {

  static class CleanerMapper extends MapReduceBase

    implements Mapper<LongWritable, Text, IntWritable, Text> {

    private NcdcRecordParser parser = new NcdcRecordParser();

    public void map(LongWritable key, Text value,
        OutputCollector<IntWritable, Text> output, Reporter reporter)
        throws IOException {

      parser.parse(value);
      if (parser.isValidTemperature()) {
        output.collect(new IntWritable(parser.getAirTemperature()),
          value);
      }
    }
  }

  @Override
  public int run(String[] args) throws IOException {
    JobConf conf = JobBuilder.parseInputAndOutput(this, getConf(),
      args);
    if (conf == null) {
      return -1;
    }

    conf.setMapperClass(CleanerMapper.class);
    conf.setOutputKeyClass(IntWritable.class);
    conf.setOutputValueClass(Text.class);
    conf.setNumReduceTasks(0);
    conf.setOutputFormat(SequenceFileOutputFormat.class);
    SequenceFileOutputFormat.setCompressOutput(conf, true);
    SequenceFileOutputFormat.setOutputCompressorClass(conf,
      GzipCodec.class);
    SequenceFileOutputFormat.setOutputCompressionType(conf,
      CompressionType.BLOCK);

    JobClient.runJob(conf);
    return 0;
```

```
  }
  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new SortDataPreprocessor(), args);
    System.exit(exitCode);
  }
}
```

8.2.2 部分排序

在第 7 章中，我们看到，MapReduce 根据输入记录的键来排序。例 8-4 是对含有 IntWritable 键的顺序文件排序的另一种方式。

例 8-4：使用默认 HashPartitioner 对含有 IntWritable 键的 SequenceFile 进行排序

```
public class SortByTemperatureUsingHashPartitioner extends Configured
  implements Tool {

  @Override
  public int run(String[] args) throws IOException {
    JobConf conf = JobBuilder.parseInputAndOutput(this, getConf(),
      args);
    if (conf == null) {
      return -1;
    }

    conf.setInputFormat(SequenceFileInputFormat.class);
    conf.setOutputKeyClass(IntWritable.class);
    conf.setOutputFormat(SequenceFileOutputFormat.class);
    SequenceFileOutputFormat.setCompressOutput(conf, true);
    SequenceFileOutputFormat.setOutputCompressorClass(conf,
      GzipCodec.class);
    SequenceFileOutputFormat.setOutputCompressionType(conf,
      CompressionType.BLOCK);

    JobClient.runJob(conf);
    return 0;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new
       SortByTemperatureUsingHashPartitioner(),args);
```

```
    System.exit(exitCode);
  }
}
```

控制排序顺序

对于键的排序顺序是由 RawComparator 控制的，具体如下。

1. 如果属性 mapred.output.key.comparator.Class 被设置，则使用这个类的实例。(使用 JobConf 中的 setOutputKeyComparatorClass()方法是设置此属性的一种方便的方式。)
2. 否则，键必须是 WritableComparable 的子类并且使用已注册的该键所属的类的 comparator。
3. 如果没有已注册的 comparator，那么使用 RawComparator 对字节流进行反序列化使其变为对象比较，并且委托代理给 WritableComparable 的 compareTo()方法。

这些规则强调了为什么对你自己的定制的 Writable 类注册 RawComparators 优化版本是非常重要的，并且用你自己的 comparator 来覆写排序顺序更简单(我们将在本章的“二次排序”中涉及)。

假设我们使用 30 个 reducer 来运行这个程序：[①]

```
% hadoop jar job.jar SortByTemperatureUsingHashPartitioner \
-D mapred.reduce.tasks=30 input/ncdc/all-seq output-hashsort
```

该命令生成 30 个输出文件，每个都已排序。但是，没有简单的方法合并文件(如：在纯文本的情况下串联合并)，以产生一个全局排序文件。对于大多数应用来说，这并不重要。例如：如果要做查找，那么有部分排序的文件是不错的选择。

应用程序：已划分的 MapFile 文件查找

例如，在多文件的情况下，通过键来执行查找效果不错。如果将输出格式改为 MapFileOutputFormat，如例 8-5 所示，则输出为 30 个 map 文件，我们可以针对它们执行查找。

① 参考第 4 章，了解怎样使用 Hadoop 的排序例程完成相同的事情。

例 8-5：排序 SequenceFile 和产生 MapFiles 作为输出的 MapReduce 程序

```
public class SortByTemperatureToMapFile extends Configured
implements Tool {

  @Override
  public int run(String[] args) throws IOException {
    JobConf conf = JobBuilder.parseInputAndOutput(this, getConf(), args);
    if (conf == null) {
      return -1;
    }

    conf.setInputFormat(SequenceFileInputFormat.class);
    conf.setOutputKeyClass(IntWritable.class);
    conf.setOutputFormat(MapFileOutputFormat.class);
    SequenceFileOutputFormat.setCompressOutput(conf, true);
    SequenceFileOutputFormat.setOutputCompressorClass(conf,
      GzipCodec.class);
    SequenceFileOutputFormat.setOutputCompressionType(conf,
      CompressionType.BLOCK);

    JobClient.runJob(conf);
    return 0;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new SortByTemperatureToMapFile(),
      args);
    System.exit(exitCode);
  }
}
```

MapFileOutputFormat 提供了一对方便的静态方法来执行针对 MapReudce 输出的查找，其用法如例 8-6 所示。

例 8-6：从 MapFiles 集合中获得与指定键第一个匹配的条目

```
public class LookupRecordByTemperature extends Configured implements Tool {

  @Override
  public int run(String[] args) throws Exception {
    if (args.length != 2) {
      JobBuilder.printUsage(this, "<path> <key>");
      return -1;
```

```
    }
    Path path = new Path(args[0]);
    IntWritable key = new IntWritable(Integer.parseInt(args[1]));
    FileSystem fs = path.getFileSystem(getConf());
    Reader[] readers = MapFileOutputFormat.getReaders(fs, path, getConf());
    Partitioner<IntWritable, Text> partitioner =
      new HashPartitioner<IntWritable, Text>();
    Text val = new Text();
    Writable entry =
      MapFileOutputFormat.getEntry(readers, partitioner, key, val);
    if (entry == null) {
      System.err.println("Key not found: " + key);
      return -1;
    }
    NcdcRecordParser parser = new NcdcRecordParser();
    parser.parse(val.toString());
    System.out.printf("%s\t%s\n", parser.getStationId(),
      parser.getYear());
    return 0;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new LookupRecordByTemperature(),
      args);
    System.exit(exitCode);
  }
}
```

getReaders()方法为每个由该 MapReduce 作业创建的输出文件打开一个 MapFile.Reader 对象。然后 getEntry()方法用 partitioner 来选择一个和键匹配的 reader，并调用 reader 的 get()方法得到键对应的值。如果 getEntry()返回 null，意味着没有找到匹配的键。否则，它返回值，并且我们将该值转换为一个 station ID 和年份。

要查看代码运行情况，先找到气温在-10℃的第一个条目(注意，气温是以整数代表度数的十分之一来表示的，所以输入的气温为-100)：

```
% hadoop jar job.jar LookupRecordByTemperature output-
hashmapsort -100
357460-99999 1956
```

为了获得指定键的所有记录，我们也可以直接使用 readers。由于返回的 readers 数组是根据分片来排列的，因此可以用同样的 partitioner 查找与给定键匹配的 reader。

```
Reader reader = readers[partitioner.getPartition(key, val,
readers.length)];
```

之后一旦我们有了 **reader**，我们可用 `MapFile` 的 `get()` 方法得到第一个键，之后循环调用 `next()` 来获取下一个键和值，直到键改变为止。例 8-7 显示了这一过程。

例 8-7：从 MapFiles 集合中获得指定键的所有条目

```
public class LookupRecordsByTemperature extends Configured
implements Tool {

  @Override
  public int run(String[] args) throws Exception {
    if (args.length != 2) {
      JobBuilder.printUsage(this, "<path> <key>");
      return -1;
    }
    Path path = new Path(args[0]);
    IntWritable key = new IntWritable(Integer.parseInt(args[1]));
    FileSystem fs = path.getFileSystem(getConf());

    Reader[] readers = MapFileOutputFormat.getReaders(fs, path, getConf());
    Partitioner<IntWritable, Text> partitioner =
      new HashPartitioner<IntWritable, Text>();
    Text val = new Text();

    Reader reader = readers[partitioner.getPartition(key, val,
      readers.length)];
    Writable entry = reader.get(key, val);
    if (entry == null) {
      System.err.println("Key not found: " + key);
      return -1;
    }
    NcdcRecordParser parser = new NcdcRecordParser();
    IntWritable nextKey = new IntWritable();
    do {
      parser.parse(val.toString());
      System.out.printf("%s\t%s\n", parser.getStationId(), parser.getYear());
    } while(reader.next(nextKey, val) && key.equals(nextKey));
    return 0;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new LookupRecordsByTemperature(),
```

```
      args);
    System.exit(exitCode);
  }
}
```

获取所有-10℃读数并对它们进行计算的示例如下：

```
% hadoop jar job.jar LookupRecordsByTemperature output-
hashmapsort -100 \2> /dev/null | wc -l
1489272
```

8.2.3 全局排序

怎样用 Hadoop 产生全局排序文件？最简单的方法是使用单个分区。[①]但是这对于大型文件来说效率十分低下，因为一台机器不得不处理所有的输出，这样等于丧失了 MapReduce 提供的并行体系结构所带来的好处。

相反，可以产生一系列排序过的文件，如果将其连接在一起就可以形成一个全局排序文件。方法就是使用一个 partitioner 负责输出的总体顺序。例如：如果我们有 4 个分区，我们把在气温 10℃以下的键放在第一个分区， 10℃～0℃的在第二个，0℃～10℃的在第三个，高于 10℃的在第四个。

虽然这种方法确实有效，但必须谨慎地选择分区大小来确保它们比较均匀，使作业不会以一个 reducer 处理为主。对于刚才描述的分区方案，在分区中的相对大小如下：

```
Temperature range <  -10° C [  -10° C, 0° C) [0° C, 10° C) >= 10° C
Proportion of records 11% 13% 17% 59%
```

这些分布不是非常均匀。为创建更均匀的分区，我们需要对整个数据集的气温分布有更好的了解。很容易写一个 MapReduce 作业来统计落入气温集合桶的记录数量。例如：图 8-1 显示了大小为 1℃的桶的分布，图上每个点对应一个桶。

虽然我们可以用这些信息来构建一个非常均匀的划分设置，但实际上由于我们运行这样一个作业需要使用整个数据集来构建，因此这种方法并不理想。一种可能的做法是通过对键进行抽样来得到比较均匀的分区。抽样背后的理念是，通过观察键的一个小子集来估计键的分布，该子集之后可用来构建分区。幸运的是，我们不必为此自己编写代码，因为 Hadoop 自带一组采样器。

---

① 最好使用 Pig，它可以用一个排序命令。详情参见第 11 章。

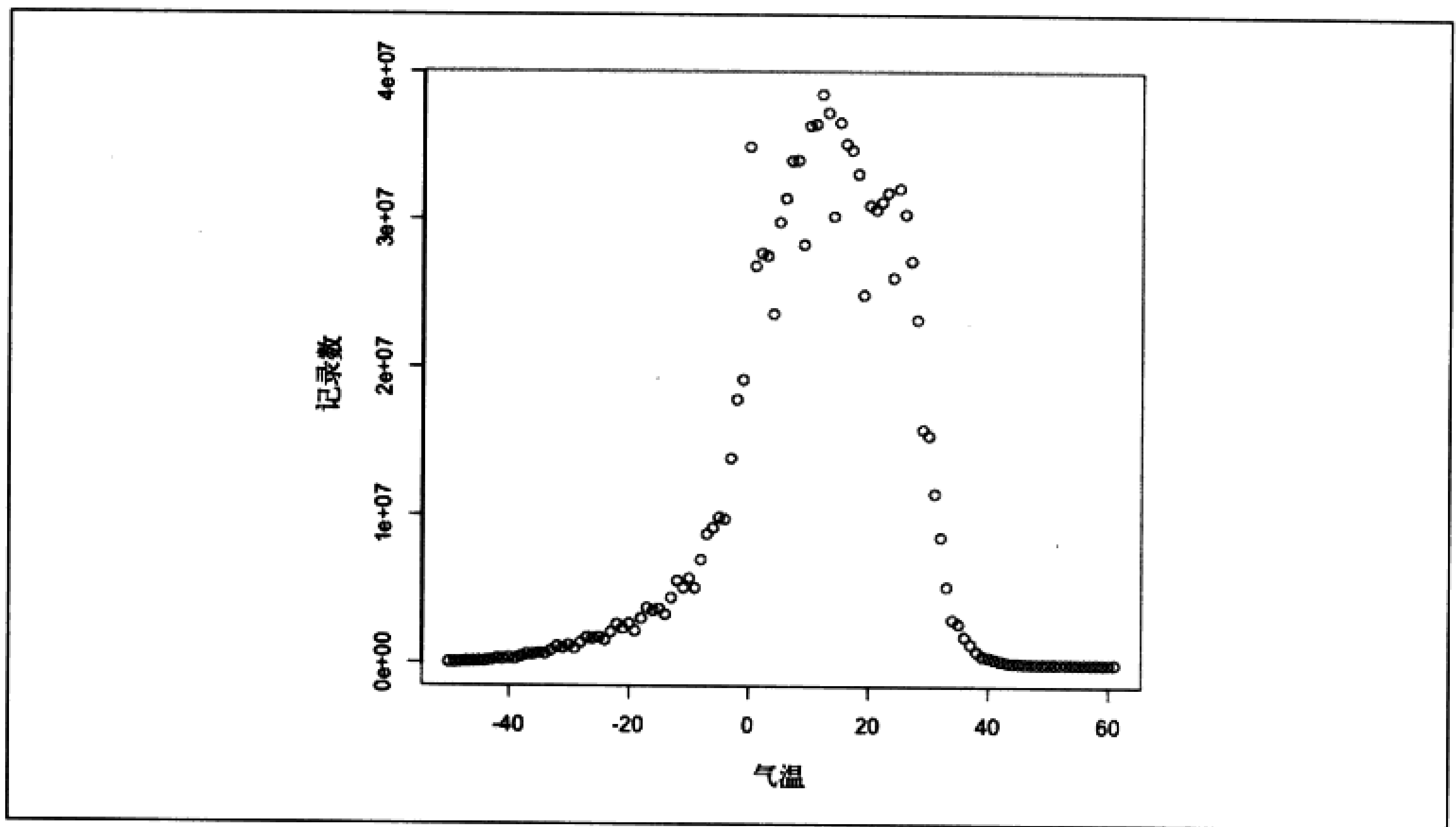


图 8-1：气象数据集的气温分布

在 InputSampler 类中定义了一个嵌套的 Sampler 接口，实现指定 InputFormat 和 JobConf 返回一组键样本。

```
public interface Sampler<K,V> {
  K[] getSample(InputFormat<K,V> inf, JobConf job) throws IOException;
}
```

此接口通常不直接由客户端调用，而是使用 InputSampler 的 writePartiotionFile()静态方法，它创建一个顺序文件来存储定义分区的键：

```
public static <K,V> void writePartitionFile(JobConf job,
    Sampler<K,V> sampler) throws IOException
```

TotalOrderPartitioner 使用该顺序文件来为排序该作业创建分区。例 8-8 把这些放在了一起。

例 8-8：用 TotalOrderPartitioner 排序含有 IntWritable 键的 SequenceFile 来实现对全局数据排序的 MapReduce 程序

```
public class SortByTemperatureUsingTotalOrderPartitioner extends
  Configured implements Tool {

  @Override
  public int run(String[] args) throws Exception {
```

```
    JobConf conf = JobBuilder.parseInputAndOutput(this, getConf(),
      args);
    if (conf == null) {
      return -1;
    }

    conf.setInputFormat(SequenceFileInputFormat.class);
    conf.setOutputKeyClass(IntWritable.class);
    conf.setOutputFormat(SequenceFileOutputFormat.class);
    SequenceFileOutputFormat.setCompressOutput(conf, true);
    SequenceFileOutputFormat.setOutputCompressorClass(conf,
      GzipCodec.class);
    SequenceFileOutputFormat.setOutputCompressionType(conf,
      CompressionType.BLOCK);

    conf.setPartitionerClass(TotalOrderPartitioner.class);

    InputSampler.Sampler<IntWritable, Text> sampler =
      new InputSampler.RandomSampler<IntWritable, Text>(0.1, 10000, 10);

    Path input = FileInputFormat.getInputPaths(conf)[0];
    input = input.makeQualified(input.getFileSystem(conf));

    Path partitionFile = new Path(input, "_partitions");
    TotalOrderPartitioner.setPartitionFile(conf, partitionFile);
    InputSampler.writePartitionFile(conf, sampler);

    // Add to DistributedCache
    URI partitionUri = new URI(partitionFile.toString() + "#_partitions");
    DistributedCache.addCacheFile(partitionUri, conf);
    DistributedCache.createSymlink(conf);
    JobClient.runJob(conf);
    return 0;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(
      new SortByTemperatureUsingTotalOrderPartitioner(), args);
    System.exit(exitCode);
  }
}
```

我们用 RandomSampler 选择同一概率的键(这里为 0.1)。还有要获取的样本的最大

数量和分区样本的最大数量的参数(这里分别为 10 000 和 10)；这里的设置是 InputSampler 作为应用程序运行时的默认值，当第一次遇到这些限制时，采样器便停止。采样器运行在客户端上，因此限制下载分区的数量很重要，这样采样器运行得很快。实际上，花费在运行采样器上的时间只是整个作业时间的一小部分。[①]

InputSampler 写入的分区文件称为_partitions，我们把其设置到输入目录中(它不会被当作输入文件因为它是以下划线开头)。为了共享在集群上带有分区文件的任务，我们把它添加到分布式缓存(详见第 8 章)。

在运行时，采样器选择了-5.6℃，13.9℃和 22.0℃作为分区边界(分区为四个)，它将把原有分区选择转化为更为均匀的分区大小：

```
Temperature range <   -5.6° C [  -5.6° C, 13.9° C) [13.9° C, 22.0
° C) >= 22.0° C
Proportion of records 29% 24% 23% 24%
```

输入数据决定着最适合的采样器。例如：SplitSampler，只对在一个分区内的前 n 个记录采样，这就不是很适合排序过的数据因为它不是从整个分区内选择键。

另一方面，IntervalSampler 在整个分区中定期选择键，这对已排序的数据是更好的选择。RandomSampler 是一个很好的通用采样器。如果都不符合自己的应用(记住，采样是产生大小近似相同的分区)，可以自己写一个 Sampler 接口的实现。

InputSampler 和 TotalOrderPartitioner 的优点之一是你可以自由选择分区数。这一选择往往由集群里 reducer 的数量决定(选择比总数稍多一点的数量以允许失败的出现)。然而，TotalOrderPartitioner 只会在分区边界不同时起作用：选择一个大数量可能会产生一个问题是，只有一个很小的键空间时，可能会发生碰撞。

以下便是运行方式：

```
% hadoop jar job.jar
SortByTemperatureUsingTotalOrderPartitioner \
-D mapred.reduce.tasks=30 input/ncdc/all-seq output-totalsort
```

该程序输出 30 个分区，其中每个内部排序，此外，对这些分区，在分区 i 的所有键比在分区 i+1 的键小。

① 在一些应用中，有些输入已经排序过或至少部分排序过，这很常见。例如，气象数据集是以时间排序的，这可能会引入一定的倾向，使 RandomSampler 成为一个比较保险的选择。

8.2.4 二次排序

MapReduce 框架在记录到达 reducer 之前对这些记录按键排序。然而对于任何特定的键，其对应的值没有排序。每次运行时值的顺序甚至都不固定，因为它们来自不同的 map 任务，每次运行时完成时间都不同。一般而言，MapReduce 的程序都编写为不依赖于值到达 reduce 函数的顺序。然而，在特定的方式下排序和分组键来排序值是可行的。

为了说明这一想法，考虑一个计算每年的最高气温的 MapReduce 的程序。如果我们按照降序排列这些值(气温)，我们就不必遍历它们以找到最大——我们只需获取每年的第一个值而忽略其他。(这种方法不是最有效的方法来解决这个问题，但它大致说明了二次排序的作用。)

为了实现它，我们将键变成复合的：结合年份与气温。我们希望先按年份(升序)排序，再按气温(降序)排序：

```
1900 35° C
1900 34° C
1900 34° C
...
1901 36° C
1901 35° C
```

如果我们所做的只是改变键，那么这起不了什么作用，因为现在同年记录并不能(一般情况下)去同一个 reducer，因为它们的键不同。例如，(1900 年，35℃)和(1900 年，34℃)可能到不同的 reducer。通过设置一个 partitioner 使其按照键的年份部分进行分区，我们可以保证同年记录能到同一个 reducer。然而，这仍然不够实现我们的目标。一个 partitioner 确保一个 reducer 接收一年的所有记录，它没有改变 reducer 通过分区按键成组的事实：

```
        分区  组
1900 35°C |   |
1900 34°C |   |
1900 34°C |   |
...
1901 36°C |   |
1901 35°C |   |
```

最后一步是控制分组的设置。如果我们在 reducer 中以键的年份部分来分组值，那么我们会看到所有同一年的记录在同一个 reduce 组。同时，因为它们以气温降序排列，所以第一个就是最高气温：

```
      分区  组
1900 35°C |   |
1900 34°C |   |
1900 34°C |   |
  ...
1901 36°C |   |
1901 35°C |   |
```

总之，这里有一个得到有效值排序的方法：

- 将键设为综合原始键和值的复合键。
- 键 comparator 应该以复合键排序，即原本的键和原本的值。
- partitioner 和复合键 comparator 的分组应只考虑对本来键的分区和分组。

Java 代码

例 8-9 代码结合以上所有的部分。该程序再次使用纯文本输入。

例 8-9：通过排序在键中的气温找出最高气温

```
public class MaxTemperatureUsingSecondarySort
  extends Configured implements Tool {

  static class MaxTemperatureMapper extends MapReduceBase
    implements Mapper<LongWritable, Text, IntPair, NullWritable> {

    private NcdcRecordParser parser = new NcdcRecordParser();

    public void map(LongWritable key, Text value,
        OutputCollector<IntPair, NullWritable> output, Reporter
        reporter)throws IOException {

      parser.parse(value);
      if (parser.isValidTemperature()) {
      output.collect(new IntPair(parser.getYearInt(),
          + parser.getAirTemperature()), NullWritable.get());
      }
    }
  }

  static class MaxTemperatureReducer extends MapReduceBase
    implements Reducer<IntPair, NullWritable, IntPair, NullWritable> {

    public void reduce(IntPair key, Iterator<NullWritable> values,
      OutputCollector<IntPair, NullWritable> output, Reporter reporter)
```

```
      throws IOException {

    output.collect(key, NullWritable.get());
    }
  }

  public static class FirstPartitioner
    implements Partitioner<IntPair, NullWritable> {

    @Override
    public void configure(JobConf job) {}

    @Override
    public int getPartition(IntPair key, NullWritable value, int
      numPartitions) {
      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 IOException {
    JobConf conf = JobBuilder.parseInputAndOutput(this, getConf(),
      args);
    if (conf == null) {
      return -1;
    }

    conf.setMapperClass(MaxTemperatureMapper.class);
    conf.setPartitionerClass(FirstPartitioner.class);
    conf.setOutputKeyComparatorClass(KeyComparator.class);
    conf.setOutputValueGroupingComparator(GroupComparator.class);
    conf.setReducerClass(MaxTemperatureReducer.class);
    conf.setOutputKeyClass(IntPair.class);
    conf.setOutputValueClass(NullWritable.class);

    JobClient.runJob(conf);
    return 0;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new
      MaxTemperatureUsingSecondarySort(), args);
    System.exit(exitCode);
  }
}
```

在 mapper 中，我们创建了一个键代表年份和气温，使用 `IntPairWritable` 实现。(`IntPair` 就像我们在第 4 章"实现一个用户的 Writable"中开发的 `TextPair` 类。)不需要在值中携带任何信息，因为我们可以从键上得到在 reducer 中的最高气温，因此我们使用 `NullWritable`。reducer 通过二次排序得到的第一个键是一个 `IntPair`，它由年份和该年的最高气温组成。`IntPair` 的 `toString()` 方法创建一个制表符分隔的字符串，因此输出是一系列由制表符分隔的年份/气温对。

注意： 许多应用程序需要访问的所有排序的值，而不是像我们这里提供的只是第一个值。为此，需要填充值字段，因为在 reducer 中只能得到第一个键。这就需要一些键和值之间信息的不可避免的重复。

我们用自定义 partitioner，设置为通过键的第一个字段(年份)来进行分区。为了按年份升序和按气温降序，我们使用自定义键 comparator，从中提取的关键字段和执行相应的比较。同样，为了通过年份分组键，我们设置一个自定义 comparator，使用 `setOutputValueGroupingComparator()`来提取键的第一个字段进行比较。[①]

运行这个程序，它提供了每年的最高气温：

```
% hadoop jar job.jar MaxTemperatureUsingSecondarySort
input/ncdc/all output-secondarysort
% hadoop fs -cat output-secondarysort/part-* | sort | head
1901 317
1902 244
1903 289
1904 256
1905 283
1906 294
1907 283
1908 289
1909 278
1910 294
```

流

要在 Streaming 流中做二次排序，我们可以利用 Hadoop 提供的类库。下面是用来做二次排序的驱动程序：

```
hadoop jar $HADOOP_INSTALL/contrib/streaming/hadoop-*-
  streaming.jar \
  -D stream.num.map.output.key.fields=2 \
  -D mapred.text.key.partitioner.options=-k1,1 \
  -D mapred.output.key.comparator.class=\
org.apache.hadoop.mapred.lib.KeyFieldBasedComparator \
  -D mapred.text.key.comparator.options="-k1n -k2nr" \
  -input input/ncdc/all \
  -output output_secondarysort_streaming \
  -mapper src/main/ch08/python/secondary_sort_map.py \
  -partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner\
  -reducer src/main/ch08/python/secondary_sort_reduce.py \
```

① 为简单起见，这些自定义 comparator 没有优化，参考第 4 章“实现一个快速 RawComparator”采取的步骤使它们更快。

```
    -file src/main/ch08/python/secondary_sort_map.py \
    -file src/main/ch08/python/secondary_sort_reduce.Py
```

我们的 map 函数(例 8-10)发出带有年份和气温字段的记录。我们要将这两个字段的组合视为一个键，所以我们设置 `stream.num.map.output.key.fields` 为 2。这意味着值将是空的，就像 Java。

例 8-10：在 Python 中二次排序的 map 函数

```
#!/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)
```

但是，我们不希望按所有键进行分割，所以使用 `KeyFieldBasedPartitioner` 进行分割，它允许我们按部分键进行分区。`mapred.text.key.partitioner.options` 中配置了 partitioner。值 K1，1 指明 partitioner 只使用键的第一个字段，其中假定字段被 `map.output.key.field.Separator` 属性(默认情况下制表符)中定义的字符串分开。

下一步，我们需要一个按年份字段升序和按气温字段降序排序的 comparator，这样，reduce 函数可以简单地返回每组的第一个记录。Hadoop 为此提供了一个理想选择 `KeyFieldBasedComparator`。比较顺序是由一个类似用于 GNU 排序的规范定义的。它是通过 `mapred.text.key.comparator.options` 属性设置的。值 `-k1n -k2nr` 在本例中表示“按数值顺序排序第一个字段，然后按数值逆序排序第二个字段。”就像 partitioner`KeyFieldBasedPartitioner`，它用 `map.Output.key.field.separator` 定义的分隔符将一个键分为字段。

在 Java 版本中，必须设置 comparator 分组。然而，在流中，组不以任何方式划分，所以在 reduce 函数中我们必须自己通过查看年份的变动检测分组界限(例 8-11)。

例 8-11：在 Python 中二次排序的 reducer 函数

```
#!/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
```

运行该流程序时，我们得到的输出与 Java 版本相同。最后，请注意，KeyFieldBasedPartitioner 和 KeyFieldBasedComparator 不仅可用于流程序，也适用于 Java MapReduce 的程序。

8.3 联接

MapReduce 能够在大型数据集之间进行联接(join)，但是相当程度上需要从零开始编写代码做联接。相对于编写 MapReduce 程序，可以考虑使用一个较高层次的框架，如 Pig，Hive 或 Cascading，其中的联接操作是其实现的核心部分。

让我们简单考虑一下我们正努力解决的问题，例如，我们有两个数据集，气象站数据库和气温记录，我们希望联接两者。例如，我们希望看到每个气象站的历史数据，并在每个输出行内含有气象站的元数据。如图 8-2 所示。

如何实现联接这取决于数据集的大小和它们的划分情况。如果一个数据集很大(气温记录)，但另一部分则是小到可以分发到集群中的每个节点(如气象站元数据)，那么连接可由一个 MapReduce 作业实现，该作业将每个气象站的记录放在一起(如按 station ID 部分排序)。mapper 或 reducer 使用这个较小的数据集来查找一个气象站的元数据，这样每个记录就能一并输出。见本章“二次数据分布”上的这种做法，在那里我们重点讨论了将数据分发到 tasktracker 的机制。

如果两个数据集都过于大而不能复制到集群中的每个节点，则可以使用 MapReduce，使用一个 map 端联接或 reduce 端联接来连接它们，一个简单的例子就是一个用户数据库和一些用户活动的日志记录(如访问日志记录)。对于一般的应用，将用户数据库(或日志记录)分发到 MapReduce 的所有节点是不可行的。

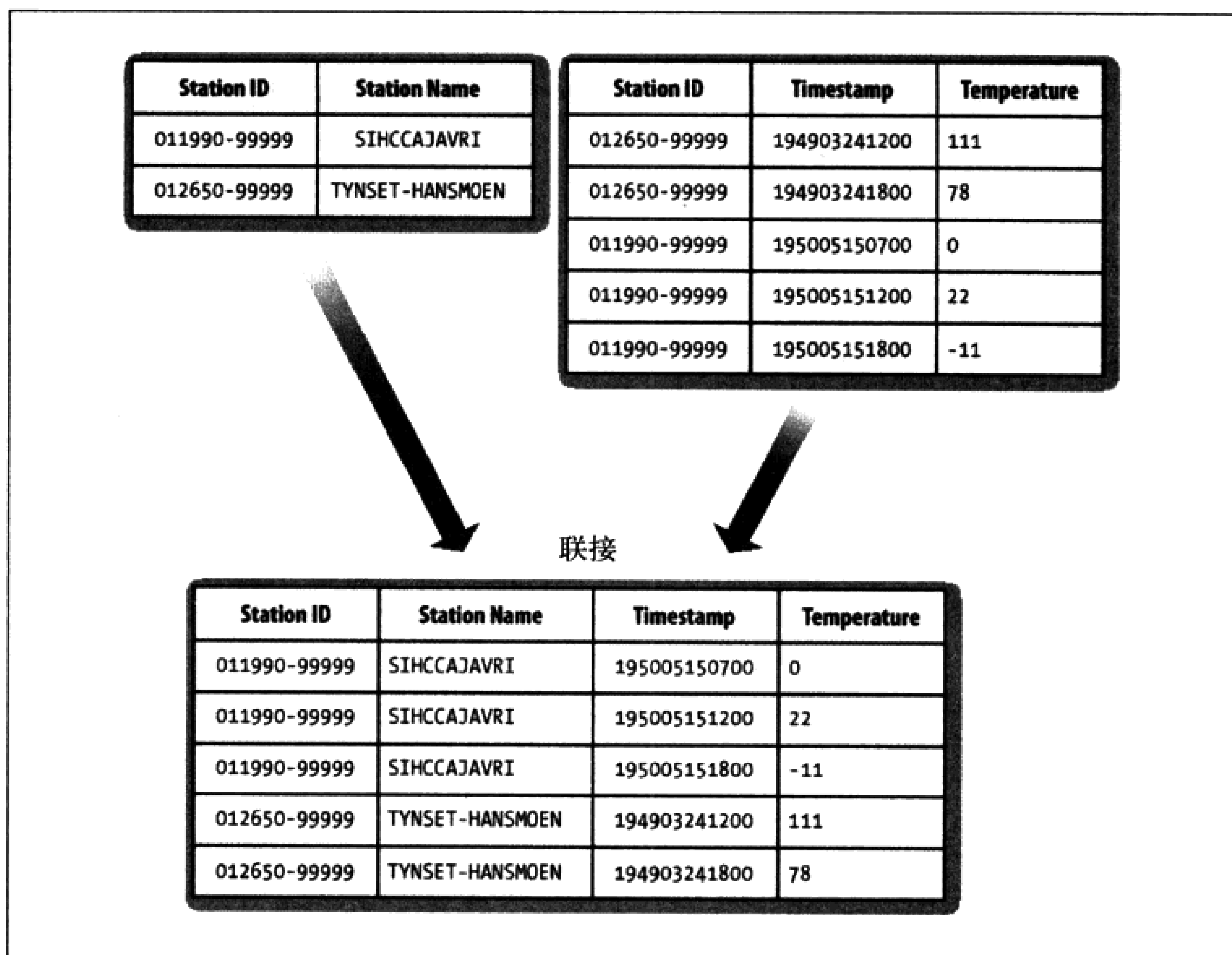


图 8-2：两个数据集内联接

8.3.1 map 端联接

map 端联接是在数据到达 map 函数之前就执行合并的，虽然对于每个 map 的输入必须以某种特定方式划分并排序。每个输入数据集必须被分为相同大小的分区并且每部分按相同的键排序(合并键)。所有特定键的记录必须驻留在同一个分区。这听起来像一个严格的要求(并且它确实是)，但实际上符合 MapReduce 作业输出的说明。

一个 map 端联接可以用来联接几个作业的输出，这些作业含有相同数量的 reducer，相同的键，并且输出文件不可分割(例如比一个 HDFS 块小，或者被压缩了)。在天气实例中，如果按 station ID 对气象站文件部分排序，也按 station ID 对记录同样排序，并且作业含有相同 reducer 数量，那么两个输出将满足运行 map 端联接的条件。

使用从 org.apache.hadoop.mapred.join 包中的 CompositeInputFormat 运行 map 端联接。CompositeInputFormat 的输入源和联接类型(内部或外部)是通过一个根据简单语法写出来的联接表达式配置的。详情和相关例子可参见该软件包的文档。

该 org.apache.hadoop.examples.Join 例子是一个运行 map 端联接的通用的命令行程序，因为它允许你运行任何指明 mapper 和 reducer 的 MapReduce 作业，将不同的输入通过给定的联接操作进行联接。

8.3.2 reduce 端联接

reduce 端联接比 map 端联接更普遍，因为输入数据集不需要有特定的结构，但是效率低，因为所有数据集必须经过 MapReduce 洗牌。其基本思路是 mapper 给每个记录打上它的源标签，并且用联接键作为 map 的输出键，使有相同键的记录可以聚集在同一个 reducer。我们用以下多种方法使其在实践中有效工作。

多样输入

一般来说，该数据集的输入源有不同的格式，因此可以很方便地使用 MultipleInputs 类(参见第 7 章)来分隔的用于解析和标记每个源的逻辑。

二次排序

如前所述，reducer 会看到来自含有相同键的源的记录，但并不保证它们有任何特定的顺序。然而，为了执行联接，有顺序地接收数据是很重要的。对于气象数据联接，对于每个键气象站记录必须是第一个值，因此 reducer 可以在气象记录中填充气象站名称并直接发出它们。当然，如果我们将收到的数据在内存中缓冲，那么无序地接受数据是可行的，但应该避免这样，因为任何组的记录数量都可能很大并且超过了 reducer 可用的内存量。[①]

8.4 节介绍了如何对每个键对应的值排序，所以这里使用了这种技术。

为标记每个记录，我们对键使用第 4 章的 TextPair 来存储的 station ID 和标签。对标签值的唯一要求是它们以某种方式排序使得气象站记录在气象记录之前到达。这可以通过标记气象站记录为 0，气象记录为 1 来实现。mapper 类具体实现如例 8-12 和例 8-13 所示。

---

① contrib 目录下的 data_join 包通过在内存中缓冲记录实现 reduce 端联接，所以受到这个限制。

例 8-12：标记气象站记录实现 reduce 端联接的 mapper

```
public class JoinStationMapper extends MapReduceBase
    implements Mapper<LongWritable, Text, TextPair, Text> {
  private NcdcStationMetadataParser parser = new
    NcdcStationMetadataParser();

  public void map(LongWritable key, Text value,
      OutputCollector<TextPair, Text> output, Reporter reporter)
      throws IOException {

    if (parser.parse(value)) {
      output.collect(new TextPair(parser.getStationId(), "0"),
        new Text(parser.getStationName()));
    }
  }
}
```

例 8-13：标记气象记录实现 reduce 端联接的 mapper

```
public class JoinRecordMapper extends MapReduceBase
    implements Mapper<LongWritable, Text, TextPair, Text> {
  private NcdcRecordParser parser = new NcdcRecordParser();

  public void map(LongWritable key, Text value,
      OutputCollector<TextPair, Text> output, Reporter reporter)
      throws IOException {

    parser.parse(value);
        output.collect(new TextPair(parser.getStationId(), "1"), value);
    }
}
```

reducer 知道它会先收到气象站记录，因此它从值中摘录气象站名称并作为每个输出记录的一部分输出(例 8-14)。

例 8-14：连接已标记的气象站记录和已标记的气象记录的 reducer

```
public class JoinReducer extends MapReduceBase implements
    Reducer<TextPair, Text, Text, Text> {

  public void reduce(TextPair key, Iterator<Text> values,
      OutputCollector<Text, Text> output, Reporter reporter)
      throws IOException {
```

```
    Text stationName = new Text(values.next());
    while (values.hasNext()) {
      Text record = values.next();
      Text outValue = new Text(stationName.toString() + "\t" +
        record.toString());
      output.collect(key.getFirst(), outValue);
    }
  }
}
```

该代码假定在气象记录中的每个 station ID 在气象站数据集中有一个明确的匹配记录。如果不是这样，我们需要扩展该代码，使用另一个 TextPair 将标签放入值对象中。然后 reduce()方法能够知道哪个条目是气象站名称，并在处理气象数据集前检测(处理)丢失或重复的条目。

注意： 因为在 reducer 的值迭代器的对象是重用的(为了提高效率的目的)，代码复制值迭代器中第一个 Text 对象是非常重要的：

```
Text stationName=new Text(values.next())
```

如果不进行复制，那么 stationName 引用将指向一个值，该值是它转成字符串时读入的，这是一个错误。

将这些作业放在一起的是驱动类，如例 8-15 所示。最重要的一点是，我们对键的第一部分，即 station ID，分区和分组，我们用自定义 partitioner(KeyPartitioner) 和自定义 comparatorFirstComparator(从 TextPair 来)来实现。

例 8-15：联接气象记录和气象站名称的应用程序

```
public class JoinRecordWithStationName extends Configured implements Tool {

  public static class KeyPartitioner implements Partitioner
      <TextPair, Text> {
    @Override
    public void configure(JobConf job) {}

    @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;
    }

    JobConf conf = new JobConf(getConf(), getClass());
    conf.setJobName("Join record with station name");

    Path ncdcInputPath = new Path(args[0]);
    Path stationInputPath = new Path(args[1]);
    Path outputPath = new Path(args[2]);

    MultipleInputs.addInputPath(conf, ncdcInputPath,
      TextInputFormat.class, JoinRecordMapper.class);
    MultipleInputs.addInputPath(conf, stationInputPath,
      TextInputFormat.class, JoinStationMapper.class);
    FileOutputFormat.setOutputPath(conf, outputPath);

    conf.setPartitionerClass(KeyPartitioner.class);
    conf.setOutputValueGroupingComparator(TextPair.
      FirstComparator.class);

    conf.setMapOutputKeyClass(TextPair.class);

    conf.setReducerClass(JoinReducer.class);
    conf.setOutputKeyClass(Text.class);

    JobClient.runJob(conf);
    return 0;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new JoinRecordWithStationName(),
      args);
    System.exit(exitCode);
  }
}
```

在样本数据上运行该程序产生的输出如下：

```
011990-99999 SIHCCAJAVRI 0067011990999991950051507004+68750...
011990-99999 SIHCCAJAVRI 0043011990999991950051512004+68750...
```

```
011990-99999 SIHCCAJAVRI 0043011990999991950051518004+68750...
012650-99999 TYNSET-HANSMOEN 0043012650999991949032412004+62300...
012650-99999 TYNSET-HANSMOEN 0043012650999991949032418004+62300...
```

8.4 次要数据的分布

次要数据(side data)可以被定义为一个作业为处理主要数据集所需要的额外的只读数据。目前的挑战是为所有 map 或 reduce 任务(遍布集群)提供方便快捷的方式来访问次要数据。

除了本节所述的分布机制，还可以在内存中的静态区域缓存次要数据，因此，在同一作业中连续运行在同一 tasktracker 的任务可以共享数据。第 6 章曾介绍了如何启用此功能。如果使用这种方法，请注意所使用的内存量，因为它可能影响 shuffle 所需的内存(详见第 6 章)。

8.4.1 使用作业配置

可以使用 `JobConf`(继承于 `Configuration`)的各种 setter 方法在作业配置中设置任意的键/值对。如果需要传递小块元数据到任务，这是非常有用的。为得到在任务中的值，可以覆盖在 `mapper` 或 `reducer` 的 `configure()`方法并且使用传入的 `JobConf` 对象中的 getter 方法。

通常一个基本类型足够对元数据进行编码，但对于任意对象你既可以自己处理序列化(如果有一个将对象和字符串互相转换的现有机制)，也可以使用 Hadoop 的 `Stringifier` 类。`DefaultStringifier` 使用 Hadoop 的序列化框架来序列化对象(参见第 4 章)。

不要用该机制来传输比几千字节更多的数据量，因为它会给在 Hadoop 守护程序中的内存使用的增加压力，特别是在运行着数百个作业的系统。该作业配置由 jobtracker，tasktracker 和子 JVM 读取，并且每次配置读取时，所有条目都将被读入内存，即使它们不被使用。用户属性不在 jobtracker 或 tasktracker 中读取，所以，它们只会徒然浪费时间和内存。

8.4.2 分布式缓存

相对于在作业配置中对次要数据进行序列化，更好的方法是使用 Hadoop 的分布式

缓存机制来分布数据集。它提供了为该任务及时复制文件和存档文件到任务节点的服务以便在运行时使用它们。为了节省网络带宽，每个作业文件通常复制到任何特定的节点一次。

用法

对于使用 `GenericOptionsParser` 的工具(本书中很多程序里用列过它，参见第 5 章)，可以指定文件分布方式为一个以逗号分隔的 URI 列表，该列表作为`-files`选项的参数。文件可以在本地文件系统，在 HDFS，或在另一 Hadoop 可读文件系统(如 S3)。如果没有提供，则视文件为本地文件。(即使默认的文件系统不是本地文件系统，也如此。)

也可以使用`-archives`选项复制存档文件(JAR 文件，ZIP 文件，tar 文件和 gzipped tar 文件)到任务，这些在任务节点没有存档。`-libjars` 选项将增加 JAR 文件到 mapper 和 reducer 的 classpath 类路径中。如果还没有绑定库 JAR 文件到作业 JAR 文件，这是很有用的。

注意：流不使用分布式缓存复制整个集群的流脚本。使用`-file` 选项(注意单数)指定要复制的文件，并对每个要复制的文件重复该操作。此外，用`-file` 选项指定的文件只能是文件路径，而不是 URI，因此必须从运行流作业的客户端的本地文件系统访问。

流也接受`-files`和`-archives`选项来复制文件到分布式缓存供的流脚本使用。

让我们来看看如何使用分布式缓存共享气象站名称的元数据文件。运行的命令如下：

```
% hadoop jar job.jar
MaxTemperatureByStationNameUsingDistributedCacheFile \
-files input/ncdc/metadata/stations-fixed-width.txt
input/ncdc/all output
```

此命令将复制本地文件 stations-fixed-width.txt (没有方案提供，因此路径会自动解释为本地文件)到任务节点，所以我们可以用它来查找气象站名称。`MaxTemperatureByStationNameUsingDistributedCacheFile`的清单如例 8-16 所示。

例 8-16：该应用程序按气象站名称找到最高气温，气象站以站名显示，站名得自作为分布式缓存文件传入的查找表

```
public class MaxTemperatureByStationNameUsingDistributedCacheFile
  extends Configured implements Tool {
```

```
static class StationTemperatureMapper extends MapReduceBase
  implements Mapper<LongWritable, Text, Text, IntWritable> {

  private NcdcRecordParser parser = new NcdcRecordParser();

  public void map(LongWritable key, Text value,
      OutputCollector<Text, IntWritable> output, Reporter reporter)
      throws IOException {

    parser.parse(value);
    if (parser.isValidTemperature()) {
        output.collect(new Text(parser.getStationId()),
          new IntWritable(parser.getAirTemperature()));
    }
  }
}

static class MaxTemperatureReducerWithStationLookup extends
  MapReduceBase
  implements Reducer<Text, IntWritable, Text, IntWritable> {

  private NcdcStationMetadata metadata;

  @Override
  public void configure(JobConf conf) {
    metadata = new NcdcStationMetadata();
    try {
      metadata.initialize(new File("stations-fixed-width.txt"));
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }

  public void reduce(Text key, Iterator<IntWritable> values,
      OutputCollector<Text, IntWritable> output, Reporter reporter)
      throws IOException {

    String stationName = metadata.getStationName(key.toString());

    int maxValue = Integer.MIN_VALUE;
    while (values.hasNext()) {
      maxValue = Math.max(maxValue, values.next().get());
```

```
      }
      output.collect(new Text(stationName), new IntWritable(maxValue));
    }
  }

  @Override
  public int run(String[] args) throws IOException {
    JobConf conf = JobBuilder.parseInputAndOutput(this, getConf(),args);
    if (conf == null) {
      return -1;
    }

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

    conf.setMapperClass(StationTemperatureMapper.class);
    conf.setCombinerClass(MaxTemperatureReducer.class);
    conf.setReducerClass(MaxTemperatureReducerWithStationLookup. class);

    JobClient.runJob(conf);
    return 0;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(
      new MaxTemperatureByStationNameUsingDistributedCacheFile(),
        args);
      System.exit(exitCode);
  }
}
```

该程序按气象站名称查找最高气温，所以 mapper(`StationTemperatureMapper`)只是发出(station ID，Temperature)对。对于 combiner，我们重用 `MaxTemperatureReducer`(在第 2 章和第 5 章)来找出对于任何指定的 map 输出组的最高气温。该 reducer(`MaxTemperatureReducerWithStationLookup`)不同于 combiner，因为除了寻找最高气温，它还使用缓存文件来查找站名。

我们使用 reducer 的 `configure()`方法和文件原来的名称来获得缓存文件，该名称与任务的工作目录相关。

注意： 可以使用分布式缓存来复制不适合内存的文件。MapFiles 在这方面很擅长，因为它们是直接磁盘查找格式(参见第 4 章)。由于 MapFiles 是一个文件和已定义的目录结

构的集合，应该把它们归为存档文件格式(JAR, ZIP, tar 或 gzipped tar)，并使用-archives 选项把它们添加到缓存。

下面是一个输出的片段，显示一些气象站的一些最高气温：

```
PEATS RIDGE WARATAH     372
STRATHALBYN RACECOU     410
SHEOAKS AWS             399
WANGARATTA AERO         409
MOOGARA                 334
MACKAY AERO             331
```

工作原理

启动作业后，Hadoop 将-files 和-archive 选项指定的文件复制到 jobtracker 的文件系统(通常是 HDFS)。然后，在任务运行前，tasktracker 从 jobtracker 文件系统复制文件到本地磁盘作为缓存，这样任务可以访问这些文件。从任务的角度来看，该文件就存放在那儿(而并不关心它们从 HDFS 来)。

该 tasktracker 还负责维护一个引用计数，这个引用计数记录了使用缓存中各文件的任务数。任务运行后，该文件的引用计数减少 1，当它达到零时，就能被删除。当缓存超过一定规模——默认为 10 GB 时，文件就被删除给新文件腾出空间。缓存的大小可以通过设置配置属性 local.cache.size 更改，大小以字节计算。

虽然这种设计并不保证来自同一作业(运行于同一个 tasktracker 上)的后续作业会找到缓存中的文件，但因为一个作业中的任务通常被安排在同样的时间运行，因此其他作业很可能没有机会运行从而导致原有任务的文件从缓存中被删除。

文件是在 tasktrackers 上的${mapred.local.dir}/taskTracker/archive 目录下本地化的。然而，应用程序不需要知道这一点，因为这些文件是从任务的工作目录通过符号来链接的。

分布式缓存 API

大多数应用程序不需要使用 DistributedCache API，因为它们可以通过 GenericOptionsParser 间接使用分布式缓存。GenericOptionsParser 使得使用分布式缓存更方便：例如，它复制本地文件到 HDFS，然后 JobClient 使用 addCacheFile() 和 addCacheArchive() 方法通知 DistributedCache 它们在 HDFS 的位置。当文件本地化时，该 JobClient 还使 DistributedCache 通过添加片段标识符到文件 URI 来创建符号链接。例如，通过 URI *hdfs://namenode/foo/*

bar#myfile 指定的文件在任务的工作目录中被符号链接为 myfile。

在任务节点上，直接访问本地文件是最方便的，但有时可能需要获得所有可用的缓存文件的列表。`JobConf` 有两个方法可以达到该目的：`getLocalCacheFiles()`和 `getLocalCacheArchives()`，它们都返回一个指向本地文件的 `Path`(路径)对象数组。

8.5 MapReduce 的类库

Hadoop 附带一个 mapper 和 reducer 常用功能的库。表 8-2 列出了它们及其简要说明。如需进一步了解它们的用法，请查阅它们的 Java 文档。

表 8-2：MapReduce 的类库

类	描述
ChainMapper, ChainReducer	在单个 mapper 上运行一系列 mapper，并且在单个 reducer 上一个 reducer 之后跟着一系列的 mapper。(符号表示：M+RM*，其中 M 是一个 mapper，R 是一个 reducer)。这与运行多种 MapReduce 作业相比，可以大大减少磁盘 I/O 的数量
FieldSelectionMapReduce	mapper 和 reducer 可以从输入键和值中选择字段(类似 Unix 的 cut 命令)并输出键和值
IntSumReducer, LongSumReducer	为每个键统计整型值并计算总和的 reducer
InverseMapper	一个将键和值互换的 mapper
TokenCounterMapper	一个将输入值拆成单词(使用 java 的 StringTokenizer)并且连同各单词的计数 1 输出 mapper
RegexMapper	从输入值中找到匹配正则表达式的匹配项，并连同各匹配项的计数 1 输出 mapper

第 9 章

Hadoop 集群的安装

本章解释如何创建 Hadoop 使其能在机器集群上运行。HDFS 和 MapReduce 运行在一个单一机器上对于学习这些系统是有很多帮助的，但是要使它们有效地工作还是必须运行在多节点上。

创建 Hadoop 集群时有很多选择，如创建自己的集群运行在租借的硬盘上，或者使用在云计算中作为服务而提供的 Hadoop。本章和下一章将完整介绍如何创建和操作自己的集群，即使你已在使用许多维护已配置好的 Hadoop 服务，仍可以从中了解到 Hadoop 从操作角度是如何工作的。

9.1 集群说明

Hadoop 被设计运行在商业硬件上，这就意味着你不再受限于卖方昂贵的硬件；甚至可以从任意大范围的卖家中选择标准的且普通可用的硬件来创建自己的集群。

“一般”不意味着“低档”。低档机器经常使用便宜的元件，错误率比更贵的(但仍然是一般等级的)机器更高。在操作十台或成百上千台机器时，更高的错误率会导致更高的维护费用，从而证明买便宜的元件来省钱是错误的。另一方面，大型数据库等级的机器同样不推荐，因为它们在价格或性能曲线中不占优势。而且，即使少量的这种大型机器可以实现同等的多台普通机器的集群，一个高等级机器出现故障时，对整个集群的影响会更大，因为这台机器在集群数量中占有更大的比例。

硬件说明的文字即将结束，但对读者来说，尚缺乏一个直观的认识(或描述)。以下

说明是 2008 年后期运行 Hadoop 数据节点和 tasktracker 机器的典型选择方案：

中央处理器

两个四核英特尔 Xeon 2.0 GHz CPU

内存

8 GB ECC RAM[①]

存储器

4×TB SATA 磁盘

网络

千兆位以太网

此硬件说明与具体的集群必然不同，但 Hadoop 被设计成能使用多核和多磁盘，因而可以充分利用更多、更强的硬件。

为什么不使用 RAID？

使用 RAID(Redundant Array of Independent Disks，磁盘阵列)对数据节点存储器(尽管 RAID 被作为名称节点磁盘使用，为了防止它的元数据损坏)没啥好处。RAID 提供的冗余是不必要的，因为 HDFS 会通过节点间的复制来实现这一点。

此外，RAID 普遍用于增加性能的条带化(RAID 0)，结果证明比 HDFS 使用的 JBOD(Just a Bunch Of Disks，只是一组磁盘)配置更慢，JBOD 是在所有磁盘中轮询调度 HDFS 块。这种情况的原因是 RAID 0 的读写操作被 RAID 阵列中最慢的磁盘的速度限制了。在 JBOD 中，磁盘操作是独立的，所以操作的平均速度比最慢的磁盘要更快。磁盘性能在实践中会有相当大的变动，即使是相同模式的磁盘。在一些实现在 Yahoo!集群(*http://markmail.org/message/xmzc45zi25htr7ry*)的基准上，JBOD 在一个测试(Gridmix)中显示比 RAID 0 快 10%，而在另一个测试(HDFS 写吞吐量)中强 30%。

最后，如果一个磁盘在 JBOD 配置中失败了，HDFS 能忽略失败的磁盘继续操作，但是对于 RAID，单块磁盘的失败会导致整个阵列(和从此之后的节点)不可用。

Hadoop 的主体是用 Java 写的，因此它能运行在任意一个有 JVM 的平台上，不过在非 Unix 平台的机器上运行并非明智之举，因为有足够的空间来存放 Unix 控件

① 强烈推荐 ECC 主存，很多 Hadoop 用户已经报告称 Hadoop 集群使用非 ECC 主存时会出现许多校验和错误。

(比如控制脚本)。事实上，Windows 操作系统是不支持生产平台的(即使它们可以在 Cygwin 作为开发平台上使用，详见附录 A)。

集群要有多大呢？这个问题不会有确定的答案，但是 Hadoop 的好处在于你可以从一个小的集群(像 10 个节点)开始，并且随着存储器和计算机需求的扩大而扩大。在很多方面，一个更好的问题是：集群会成长多快？考虑存储容量会得到较好的答案。

例如，如果数据每周增大 1 TB，并且有三个 HDFS 副本，然后每周需要一个额外的 3 TB 作为原始数据存储。要允许一些中间文件和日志(假定 30%左右)的空间，依此算出平均每周大约需要一台新机器(2008 年)。实际上，不会每周买一台新的机器添加到集群。做类似的一个封底计算就会让你直观认识到集群有多大：在这个例子中，一个存储 2 年数据的集群就需要 100 台机器。

对一个小的集群(大约 10 个节点)，名称节点和 jobtracker 运行在单个主节点上，通常是能接受的(只要对于名称节点的元数据至少有一个副本存储在远程的文件系统中)。随着集群和存储在 HDFS 中的文件数量的增加，名称节点需要更多的主存，所以名称节点和 jobtracker 会被移到不同的机器。

第二名称节点会同名称节点运行在同样的机器上，但是因为主存使用的原因(第二名称节点同第一名称节点有相同的主存需求)，它最好运行在一个不同的硬件片上，特别是对于更大的集群。(这个问题在本章的“Master node scenarios”有更多详细的讨论。)运行名称节点的机器一般是 64 位硬件来避免 32 位结构中 Java 3 GB 堆空间的限制。[①]

网络拓扑

普通的 Hadoop 集群结构由一个两阶网络拓扑组成，如图 9-1 所示。一般每个机架有 30～40 个服务器，带有一个 1 GB 的交换器(图中只展示了 3 个)，并向上传输到一个核心交换器或者路由器(正常是 1 GB 或更多)。很明显，在相同机架中的节点间的宽带总和比不同机架中的节点间的要更大。

① 传统建议称集群中的其他机器(jobtracker、数据节点/tasktracker)应该是 32 位的，来避免指针的内存开销。Sun 的 Java 6 更新包 14 带来的新特性指第 14 个更新包“压缩的普通对象指针”清除了许多这样的开销，所以现在在 64 位硬盘上运行没有什么实际的缺点。

机架感知

为了得到 Hadoop 最大的性能，配置 Hadoop 很重要，它包含网络拓扑。如果集群在单一机架上运行，就没有什么要做的了，因为这是默认的。然而，对多机架的集群，我们需要映射节点到机架上。通过映射，放置 MapReduce 任务在节点中时，Hadoop 将优先做机架内传输而不是机架外传输(有更多的带宽可用)。HDFS 能更智能地放置副本，在性能和适应力上权衡。

网络位置(如节点和机架)可以表示成一棵树，它反映了网络中位置之间的“距离”。名称节点在决定哪里存放块的副本时，会用到网络位置；当一个 map 任务被分配到一个 tasktracker 上运行时，jobtracker 节点会使用网络位置来确定作为 map 任务输入最近副本的位置。

对于图 9-1 的网络，机架拓扑是由两个网络位置来描述的，即/交换机/机架 1 和/交换机 1/机架 2。因为在集群中只有一个最高级交换器，位置可以简化成/交换机 1 和/机架 2。

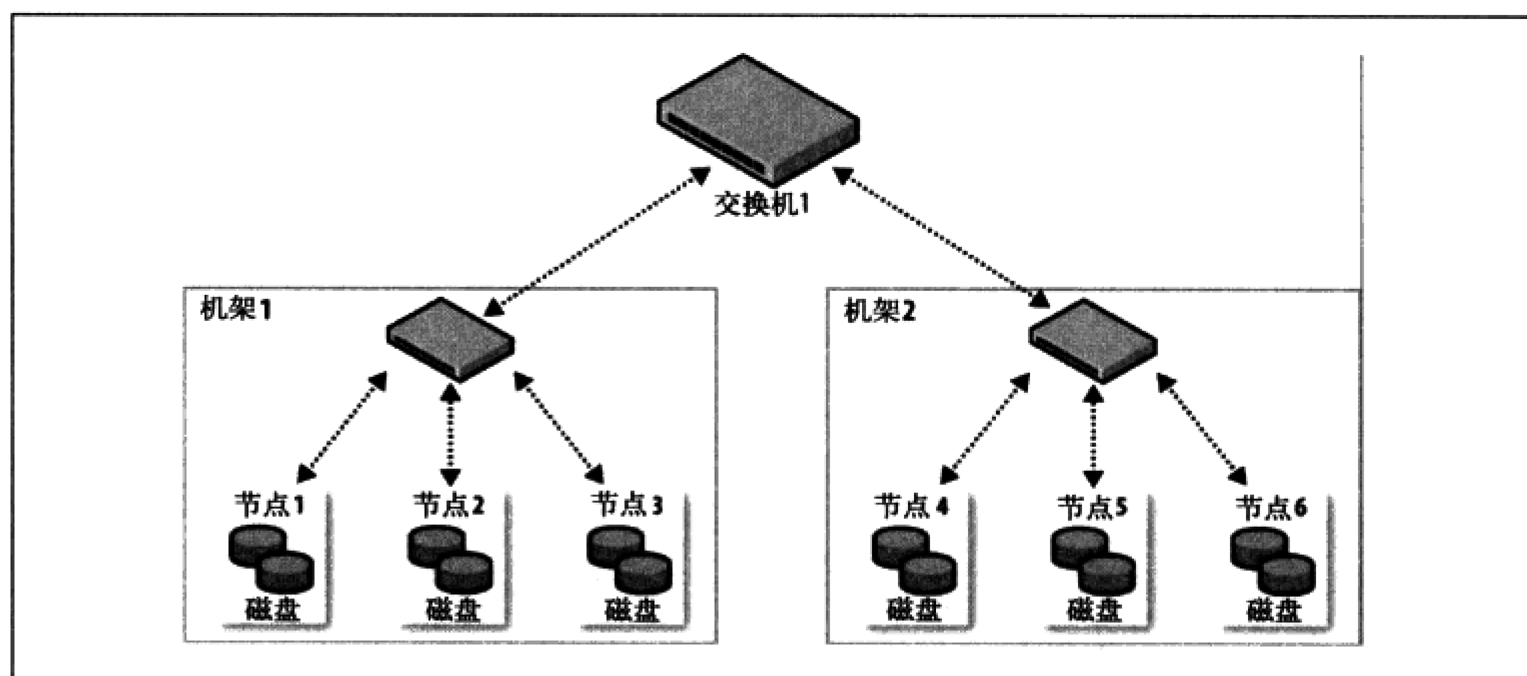


图 9-1：Hadoop 集群典型的两阶网络结构

Hadoop 配置必须指定一个在节点地址和网络位置间的映射。这个映射由一个 Java 接口描述。DNSToSwitchMapping 的接口名如下：

```
public interface DNSToSwitchMapping {
  public List<String> resolve(List<String> names);
}
```

参数 names 是 IP 地址的列表，返回值是一个对应网络位置的字符串列表。

`topology.node.switch.mapping.impl` 配置属性定义一个 `DNSToSwitchMapping` 接口的实现，名称节点和 jobtracker 常使用它处理工作者节点网络位置。

对于例子中的网络，我们将映射 node1、node2 和 node3 到/rack1，映射 node4、node5 和 node6 到/rack2。

大多数设置不需要自己实现接口，因为默认实现是 `ScriptBasedMapping`，它运行一个用户定义脚本去决定映射。这个脚本的位置由 `topology.script.file.name` 的属性控制。此脚本必须传入一个可变数量的参数即被映射的主机名或 IP 地址，并且它必须生成相应的网络位置给标准输出，用空格分开。示例代码包含此脚本。

如果没有指定脚本的位置，默认操作就是映射所有节点到一个单一网络位置 */default-rack*。

9.2 集群的建立和安装

硬件到位后，接下来把它装入机架并安装运行 Hadoop 所需的软件。

安装和配置 Hadoop 有许多方法。本章描述如何通过使用 Apache Hadoop 分布从头开始安装配置，并且涵盖设置 Hadoop 时需要了解的问题。当然，如果想使用 RPMs 或者 Debian 包管理 Hadoop 安装，并且想通过 Cloudera 的分布来启动，可参见附录 B 的相应描述。

为了缓解安装和维护每一个节点上相同的软件的负担，使用一个自动化安装方法(比如 Red Hat Linux' Kickstart 或者 Debian)完全自动化安装。这些工具通过记录在安装过程(例如磁盘分区安排)中对于选项的回答来使其自动化操作系统安装，就像安装包一样。尤其重要的是，它们也提供在此过程最后运行脚本的钩子，这对执行那些不包含在标准安装中的最终系统调整和自定义是很宝贵的。

下面将描述运行 Hadoop 所需要的自定义。这些应当全部被添加到安装脚本中。

9.2.1 安装 Java

运行 Hadoop 需要 Java 6 或者更新的版本。最新稳定的 Sun JDK 是不错的选择，不过其他商家的 Java 分布式也一样可以使用。以下指令可以确定 Java 是否正确安装：

```
% java -version
```

```
java version "1.6.0_12"
Java(TM) SE Runtime Environment (build 1.6.0_12-b04)
Java HotSpot(TM) 64-Bit Server VM (build 11.2-b01, mixed mode)
```

9.2.2 创建 Hadoop 用户

首先，创建一个专门的 Hadoop 用户，将 Hadoop 安装与运行在相同机器上的其他服务分离，这是不错的尝试。

一些集群管理员选择将用户的主目录放在一个 NFS 挂载驱动上，以便帮助 SSH 密钥的分布(参见后文描述)。NFS 一般在 Hadoop 集群之外。如果使用 NFS，则可以考虑 Autofs，系统存取文件时，可按要求挂载 NFS 文件系统。Autofs 为 NFS 服务的失效提供保护，并且可通过使用复制的文件系统来解决。还有些其他的 NFS 陷阱，例如同步 UIDs 和 GIDs 也要提防。为了帮助在 Linux 中建立 NFS，可参见 *http://nfs.sourceforge.net/nfs-howto/index.html* 中的“HowTo”。

9.2.3 安装 Hadoop

从 Apache Hadoop 发行页 (*http://hadoop.apache.org/core/releases.html*) 下载 Hadoop，并在一个合理的位置上解压，例如/usr/local(/opt 是另一个标准的选择)。注意，Hadoop 不应在 Hadoop 用户的主目录下安装，因为那可能是一个 NFS 挂载目录。

```
% cd /usr/local
% sudo tar xzf hadoop-x.y.z.tar.gz
```

还需要改变 Hadoop 文件的拥有者成为 hadoop 用户和用户组：

```
% sudo chown -R hadoop:hadoop hadoop-x.y.z
```

注意：一些管理员想在同一系统的不同位置中安装 HDFS 和 MapReduce。在本书写作期间，只有从同一 Hadoop 发布中解压的 HDFS 和 MapReduce 是相容的；然而，在将来的发布中，兼容性的要求将会放宽。如果是那样，独立安装就是有意义的，因为它将给予我们更多升级选择(详情参见第 10 章)。例如升级 MapReduce 时很方便的——可能给一个 bug 打补丁——同时 HDFS 仍在运行。

注意，分离安装 HDFS 和 MapReduce 仍能共享配置，通过使用`--config`的选项(启动后台程序时)指向一个普通的配置目录。它们也可被记录到相同的目录，因为它们生成的记录文件会以避免冲突的方式命名。

9.2.4 测试安装

创建了安装文件，就可以通过将其安装到集群的机器上来测试它了。这个过程可能要反复几次，安装过程中可能会发现一些集群的问题。等它开始工作，可以尝试配置 Hadoop 并运行测试。这个过程会在下面的部分详述。

9.3 SSH 配置

Hadoop 控制脚本依靠 SSH 来执行集群范围内的操作，例如停止和开始所有集群中后台程序的脚本。注意，控制脚本是可选的——集群范围内的操作也可由其他机制执行(例如一个分布式的 shell)。

为了操作流畅，SSH 需要设置为允许集群中机器上的 hadoop 用户不需密码登录。最简单的方法是生成一个公用/私有密钥对，并且使用 NFS 在集群中共享它。

首先，在 hadoop 用户账户中通过键入如下命令来生成一个 RSA 密钥：

```
% ssh-keygen -t rsa -f ~/.ssh/id_rsa
```

即使我们想无需密码登录，没有密码的密钥仍是不妥当的(运行一个本地的伪分布式集群时，用空密码也是可以的，详情参见附录 A)，因此在提示需要一个时，我们可以定义一个密码。使用 ssh-agent 可以免去为每一个连接输入口令。

私有密钥放在-f 选项指定的文件中，即~/.ssh/rsa，公共的密钥是以相同的名称后附加.pud 存储在文件中的，即~/.ssh/id_rsa.pud。

接下来我们需要确认公共密钥在我们需要连接到的集群所有机器的~/.ssh/authorized_keys 文件中。如果 hadoop 用户的主目录是一个 NFS 文件系统，像前面描述的那样，通过键入如下命令可使密钥在整个集群中共享：

```
% cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

如果使用 NFS 时主目录没有被共享，公用密钥将需要通过其他一些手段来实现共享。

最后确定 ssh 代理是否运行来检测你是否可以通过从主控机(master)到工作机(worker)的 ssh 安全验证[1]，然后运行 ssh-add 来保存密码。这样一来，以后就不再需要输入密码通过工作机的 ssh 安全验证了。

① 参见其 man page，了解怎样启动 ssh-agent。

9.4 Hadoop 配置

有少数几个文件用来控制一个 Hadoop 安装的配置，其中最重要的一些文件如表 9-1 所示。

表 9-1：Hadoop 配置文件

文件名	格式	描述
hadoop-env.sh	bash 脚本	在运行 Hadoop 的脚本中使用的环境变量
core-site.xml	Hadoop 配置 XML	Hadoop 核心[①]的配置，例如 HDFS 和 MapReduce 中很普遍的 I/O 设置
hdfs-site.xml	Hadoop 配置 XML	HDFS 后台程序设置的配置：名称节点，第二名称节点和数据节点
mapred-site.xml	Hadoop 配置 XML	MapReduce 后台程序设置的配置：jobtracker 和 tasktracker
masters	纯文本	记录运行第二名称节点的机器(一行一个)的列表
slaves	纯文本	记录运行数据节点和 tasktracker 的机器(一行一个)的列表
hadoop-metrics.properties	Java 属性	控制 Hadoop 怎么发布 metrics(参见第 10 章)的属性
log4j.properties	Java 属性	系统日志文件的属性、名称节点审计日记和 tasktracker 子进程(参见第 5 章)的日志的属性

这些文件全部可在 Hadoop 分布的 conf 目录下找到。只要后台程序通过`-config`选项定义本地文件系统中目录的位置，配置目录可以移动到文件系统的其他位置(Hadoop 安装位置以外，会使升级更轻松一点)。

9.4.1 配置管理

Hadoop 没有为配置信息提供单一的、全局的位置。相反，集群中每一个 Hadoop 节点都有自己的配置文件集，并由管理者确保在整个系统中配置同步。Hadoop 通

① Hadoop 核心(Hadoop Core)是 Hadoop 的一个项目名，现已分成 Common，HDFS。

过使用 rsync(参见接下来的讨论)为同步配置提供一个基本的设备，当然并行 shell 工具也能帮助完成这些任务，比如 dsh 或者 pdsh。

Hadoop 允许一个配置文件集供所有主控机和工作机所使用。这样做最大的优点就是简洁，统一(只是一个配置)和可操作性(Hadoop 脚本足以管理任意单一配置的创建)。

但对于某些集群，一体适用的配置模式就不行了。例如，如果使用与现有机器不同硬盘规格的新的机器扩展集群，就需要为新机器进行不同的配置来使用它们额外的资源。

在这些情况下，我们需要机器类型的概念，并为每种类型维护一个单独的配置。Hadoop 没有提供工具来做这些事情，但是有一些非常好的工具可以精确地做这种类型的配置管理，例如 Puppet，cfengine 和 bcfg2。

对任意大小的集群来说，保持所有的机器同步是一个挑战：在应用一个更新时，有机器失效该怎么办？——等它变得有用时，谁又能确保它能得到更新？这是个严峻的问题，又会引发安装的不统一，所以即使使用 Hadoop 控制脚本管理 Hadoop，为维护集群使用配置管理工具仍不失为一个好的想法。这些工具做一些常规维护也很有效的，例如安全漏洞补丁和更新系统包。

控制脚本

Hadoop 包含为运行命令和控制整个集群中后台程序的开始停止的脚本。使用这些脚本(在 bin 目录下可以找到)，需要先告诉 Hadoop 哪些机器在集群中。为此，有两个文件 masters 和 slaves，每一个都包含机器的主机名或者 IP 地址的列表(每条线上一个)。masters 实际上是一个误导的名称，它其实决定哪些机器将运行第二名称节点。slaves 文件列出了运行数据节点和 tasktracker 的机器。masters 和 slaves 文件都在配置目录下，通过改变在 hadoop-env.sh 中的 HADOOP_SLAVES 的设置，slaves 文件可以放在别处(并且指定另一个名称)。同样，这些文件不需要分布给工作节点，因为只有运行在名称节点或者 jobtracker 上的控制脚本使用它们。

我们不需要在 masters 文件中指定名称节点和 jobtracker 运行在哪台(或哪些)机器，因为这由运行脚本的机器决定。(其实，在 masters 文件中指定这些将会导致第二名称节点的运行位置不总是按我们所想要的那样)例如，start-dfs.sh 脚本，启动集群中的所有的 HDFS 后台程序，在运行该脚本的机器上运行名称节点。详见后文描述。

1. 在本地机器上启动名称节点(运行脚本的机器)。
2. 在 slaves 文件中记录的每一台机器上启动数据节点。
3. 在 masters 文件中记录每一台机器上启动第二名称节点。

有一个简单的脚本名为 start-mapred.sh，它可以启动集群中的所有 MapReduce 后台程序。细节如下。

1. 在本地机器上的启动 jobtracker。
2. 在 slaves 文件中记录的每一台机器上启动 tasktrcker。

注意，masters 没有被 MapReduce 控制脚本使用。

此外还有 stop-dfs.sh 和 stop-mapred.sh 脚本，用来停止对应的(相应的)启动脚本所启动的后台程序。

通过使用 hadoop-daemon.sh 脚本来启动和停止 Hadoop 后台程序。如果使用前面提到的脚本，就不需要直接调用 hadoop-daemon.sh。但如果需要从另一个系统中或者从自己的脚本中控制 Hadoop 后台程序，那么 hadoop-daemon.sh 脚本是一个很好的接入点。同样地，使用 hadoop-daemons.sh(带有一个“s”)在多台主机上启动的相同的后台程序是很方便的。

主节点情景

根据集群的大小，主机后台程序的运行会有许多的配置：名称节点、第二名称节点和 jobtracker。在一个小的集群中(数十个节点)，把它们放在单个机器中是很方便的，然而随着集群的增大，最好还是将它们分离。

名称节点有高内存要求，因为它在内存中保存整个命名空间的文件和块的元数据。第二名称节点虽然绝大多数时间空闲着，但在它创建一个检查点时会与第一名称节点占用相同的内存量。(这在第 10 章的“文件系统映像和编辑日志”小节有更详细解释)。对有很大数量的文件的文件系统来说，可能一台机器上没有足够的物理内存同时运行第一和第二名称节点。

第二名称节点会为文件系统元数据最近创建的检查点保留一个副本。万一失效(或崩溃)所有的名称节点的元数据文件时，在另一个不同的节点上保留副本(失去时效的)就可以使名称节点恢复(详情参见第 10 章)。

在一个运行着很多 MapReduce 作业的集群中，jobtracker 会使用相当大的内存和 CPU 资源，因此它也应该在一个专用的节点上运行。

无论主机后台程序在一个或多个节点上运行，接下来的说明都适用。

- 从名称节点机器上运行 HDFS 控制脚本。masters 文件应包含第二名称节点的地址。
- 从 jobtracker 机器上运行 MapReduce 控制脚本。

当名称节点和 jobtracker 在分离的节点上，它们的 slaves 文件需要保持同步，因为集群中的每一个节点应该运行一个数据节点和 tasktracker。

9.4.2 环境设置

在本小节中，我们考虑如何设置 hadoop-env.sh 的变量。

内存

Hadoop 默认分配 1000 MB(1 GB)的内存给它运行的每一个后台程序。这是由 hadoop-env.sh 中的环境变量 HADOOP_HEAPSIZE 控制的。另外，tasktracker 启动独立的子 JAVA 虚拟机来运行 map 和 reduce 任务，所以我们在考虑工作机总体内存占用情况时有必要把这些也考虑在内。

在 tasktracker 上同一时刻能运行的 map 任务的最大量是由 mapred.tasktracker.map.tasks.maximum 属性控制的，该属性默认为两个任务。对 reduce 任务也有一个相应的属性 mapred.tasktracker.reduce.tasks. maximum，它也是默认为两个任务。通过改变 mapred.child.java.opts 属性可以改变分配给每个子 JAVA 虚拟机的内存空间。默认设置时-Xmx200m，分给每一个任务 200 MB 的内存。(顺便说一句，我们也可以在这里提供其他的 JAVA 虚拟机选项。例如，启用 Verbose GC 调试 GC)默认的配置为一台工作机使用 2800 MB 的内存(见表 9-2)。

表 9-2：工作机节点内存计算

JVM	默认使用的内存(MB)	被 8 个处理器使用的内存,每个子处理器 400 MB(MB)
数据节点	1000	1000
tasktracker	1000	1000
tasktracker 子 map 任务	2 × 200	7 × 400
tasktracker 子 reduce 任务	2 × 200	7 × 400
合计	2800	7600

能同时在一个 tasktracker 上运行的任务的数量是由机器上可用处理器数量控制的。因为 MapReduce 作业通常是 I/O 密集型的，任务数处理器更多会提高利用率。超额申请的数量与任务对 CPU 的利用率相关，但是有个较好的规则，就是分配数量为处理器一至两位数量的任务(同时包括 map 和 reduce 任务)。

例如，如果有 8 个处理器并且想在每个处理器上运行两个程序，则可以设置 `mapred.tasktracker.map.tasks.maximum` 和 `mapred.tasktracker.reduce.tasks.maximum` 为 7(而不是 8，因为数据节点和 tasktracker 各自占据一个槽)。如果你也将每一个子任务可用的内存增加到 400 MB，那么全部使用的内存将是 7600 MB(见表 9-2)。

Java 内存分配是否够 8 GB 的物理内存，依赖于机器上运行的其他进程。如果正在运行 Streaming 或者 Pipes 程序，这个分配可能不合适(并且分配给子程序的内存可能会减少)，因为没有足够的内存运行用户(Streaming 或者 Pipes)的程序。当它导致严重的性能下降时，要将程序换出来避免这种情况。基于集群的准确的内存设是很必要的，并且能根据集群中监控内存使用情况得出的经验使其最优化。像 Ganglia(参见第 10 章的)一类的工具对于收集信息是很有帮助的。

Hadoop 也提供控制 MapReduce 操作使用内存大小的设置。它可以控制每个任务，详见第 10 章。

对于主节点，名称节点，第二名称节点和 jobtracker 后台程序中的每一个都默认使用 1000 MB，总计为 3000 MB。

注意：名称节点会耗尽内存，因为涉及每一个文件的每一个块都在内存之中。例如，1000 MB 对几百万个的文件是足够的。我们可以通过设置 hadoop-env.sh 中的 `HADOOP_NAMENODE_OPTS`，包含一个设置内存大小的 JVM 选项，不改变其他 Hadoop 后台程序的内存分配，而增加名称节点的内存。`HADOOP_NAMENODE_OPTS` 允许你向名称节点的 JVM 传入额高的选项。例如，如果使用 Sun JVM，-Xmx2000m 将指定 2000 MB 的内存分配给名称节点。

如果改变了名称节点的内存分配，不要忘记对第二名称节点做相同的改变(使用 `HADOOP_SECONDARYNAMENODE_OPTS` 变量)，因为它的内存需求同第一名称节点是相同的。因此最好还是在一台不同的机器上运行第二名称节点。

对其他 Hadoop 后台程序也有相应的环境变量，都可以定制其内存分配。详见 hadoop-env.sh。

Java

打算使用的 Java 实现的位置由 hadoop-env.sh 中 JAVA_HOME 设置决定，如果 hadoop-env.sh 文件中没有设置，就由 JAVA_HOME shell 环境变量决定。最好是在 hadoop-env.sh 中设置值会，这样一来，在一个地方定义清楚，就能保证整个集群使用的是同一个版本的 Java。

系统日志文件

Hadoop 生成的系统日志文件默认存储在$HADOOP_INSTALL/logs 中。通过 hadoop-env.sh 中的 HADOOP_LOG_DIR 设置可以改变此位置。改变会有一定好处，比如将日志文件存在 Hadoop 安装的目录之外，这样即使之后升级后安装目录改变，日志还将存在一个地方。一个常见的选择是/var/log/hadoop，在 hadoop-env.sh 中设置如下：

```
export HADOOP_LOG_DIR=/var/log/hadoop
```

如果这个日志目录不存在，就自动新建(这种情况下，要确认 Hadoop 用户有权限新建它)。运行在机器上的每一个 Hadoop 后台程序会新建两个日志文件。第一个日志文件是通过 log4j 写的日志输出。这个以.log 结尾的文件，在诊断问题时应该是第一个调用端口，因为大多数应用日志信息都会被写到这里。标准的 Hadoop log4j 配置使用 Daily Rolling File Appender 每日轮转文件添加规则来轮转日志文件。旧的日志文件不会被删除，所以应该定期删除或存档，以节省本地节点的磁盘空间。

第二个日志文件合并了标准输出和标准错误日志。这个以.out 结尾日志文件可能经常很小甚至为空，因为 Hadoop 使用 log4j 来记录日志。只有当后台程序重启时，它才会被轮转，并且只有最后 5 个日志被保留。旧的日志文件以 1～5 之间的数字为后缀，5 就是最老的文件。

日志文件名称(包括两种类型)包括运行着后台程序的用户名称、后台程序名称和机器主机名。例如，*hadoop-tomdatanode-sturges.local.log.2008-07-04* 是轮转后的日志文件的名称。这个命名结构使得在集群中所有机器可根据需要在一个目录下存储日志，因为文件名是唯一的。

日志文件名称中的用户名实际上是 *hadoop-env.sh* 的 HADOOP_IDENT_ STRING 设置的默认选项。如果想给 Hadoop 实例一个不同的标识来命名日志文件，将 HADOOP_IDENT_STRING 改为自己想要的标识符即可。

SSH 设置

控制脚本允许使用 SSH 从主节点到(远程的)工作节点上运行指令。从不同方面来看，定制 SSH 设置都很有用。例如，想缩短连接超时(使用 ConnectTimeout 选项)，控制脚本不会闲等一个失效的节点是否会响应。显然，过犹不及，如果超时设置太短，忙碌的节点会被略过，这也不好。

另一个有用的 SSH 设置是 StrictHostKeyChecking，把它设置成 no，可以自动给已知的主机文件添加新的主机密钥。默认情况下，设置成 ask，会让用户确认它们密钥有效，但在大的集群环境下这种设置并不合适。[①]

要传入额外选项给 SSH，可在 hadoop-env.sh 中定义 HADOOP_SSH_OPTS 环境变量。更多 SSH 的设置，可参见 ssh 和 ssh_config 手册。

Hadoop 控制脚本可以使用 rsync 给集群的所有节点发布配置文件。这个功能默认是关闭的，但是通过 hadoop-env.sh 中的 HADOOP_MASTER 设置，无论后台程序何时启动，工作机后台程序都将通过 HADOOP_MASTER 同步到本地节点的 Hadoop_INSTALL。

如果有两台主机——名称节点和 jobtracker——运行在不同的机器上该怎么办呢？我们可以挑选其中一个作为源，另一个从它这里远程同步，所有的工作机都可以这样操作。其实我们可以使用任意一台机器，甚至在 Hadoop 集群外的机器来进行同步。

因为默认的 HADOOP_MASTER 是未设置的，这就有一个问题：我们怎样才能确定当前在工作节点上 hadoop-env.sh 是有 HADOOP_MASTER 设置的？对于小的集群来说，写一个小的脚本从服务器复制 hadoop-env.sh 到所有的工作节点是很容易的。对于较大的集群，利用 dsh 这样的工具可以进行并行复制。或者，一个合适的 hadoop-env.sh 可以被作为自动安装脚本的一部分来创建(例如 Kickstart)。

启动一个大型大集群并同步时，由于工作节点大约同时启动，很多工作节点的远程同步请求会造成主节点失效。为了避免这种情况，可将 HADOOP_SLAVE_SLEEP 设置为很短的时间，例如 0.1，表示十分之一秒。在集群的所有节点上运行指令时，主机会在每台轮流发送指令给每台工作节点的间隔时间休眠。

---

① 有关 SSH 主机密钥安全提示的更多讨论，可访问 *http://www.securityfocus.com/infocus/1806* 参考 Brian Hantch 文章“SSH 主机密钥保护”。

9.4.3 重要的 Hadoop 后台程序属性

Hadoop 有一些难以理解的配置属性。在这一小节中，我们将关注所有真实工作着的集群中，那些必须定义的属性(或者至少明白为什么默认值是合适的)。这些属性是在 Hadoop 位置文件中设定的：*core-site.xml*，*hdfs-site.xml*，和 *mapred-site.xml*。例 9-1 展示了一个典型的文件集合的范例。注意，大多数变量定义为 final，是为了防止它们被作业配置覆盖。第 5 章的“配置 API”小节进一步介绍了如何写 Hadoop 配置文件。

例 9-1：典型的配置文件集

```
<?xml version="1.0"?>
<!-- core-site.xml -->
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://namenode/</value>
    <final>true</final>
  </property>
</configuration>

<?xml version="1.0"?>
<!-- hdfs-site.xml -->
<configuration>
  <property>
    <name>dfs.name.dir</name>
    <value>/disk1/hdfs/name,/remote/hdfs/name</value>
    <final>true</final>
  </property>

  <property>
    <name>dfs.data.dir</name>
    <value>/disk1/hdfs/data,/disk2/hdfs/data</value>
    <final>true</final>
  </property>

  <property>
    <name>fs.checkpoint.dir</name>
    <value>/disk1/hdfs/namesecondary,/disk2/hdfs/namesecondary</value>
    <final>true</final>
  </property>
```

```
</configuration>

<?xml version="1.0"?>
<!-- mapred-site.xml -->
<configuration>
  <property>
    <name>mapred.job.tracker</name>
    <value>jobtracker:8021</value>
    <final>true</final>
  </property>

  <property>
    <name>mapred.local.dir</name>
    <value>/disk1/mapred/local,/disk2/mapred/local</value>
    <final>true</final>
  </property>

  <property>
    <name>mapred.system.dir</name>
    <value>/tmp/hadoop/mapred/system</value>
    <final>true</final>
  </property>

  <property>
    <name>mapred.tasktracker.map.tasks.maximum</name>
    <value>7</value>
    <final>true</final>
  </property>

  <property>
    <name>mapred.tasktracker.reduce.tasks.maximum</name>
    <value>7</value>
    <final>true</final>
  </property>

  <property>
    <name>mapred.child.java.opts</name>
    <value>-Xmx400m</value>
    <!-- Not marked as final so jobs can include JVM debugging options -->
  </property>
</configuration>
```

HDFS

运行 HDFS，需要指明一台机器作为名称节点。此时属性 `fs.default.name` 是一个 HDFS 文件系统 URI，它的主机是名称节点的主机名或 IP 地址，并且端口是名称节点将要为 RPC 监听的端口。如果没有指定端口，默认使用 8020。

注意：控制脚本使用的 masters 文档不会被 HDFS(或 MapReduce)后台程序用来决定主机名称。其实，masters 文档只是被脚本使用，如果不使用，则可以忽视。

---

fs.default.name 属性也同时指定了默认的文件系统。默认的文件系统是一般解析相对路径，因为相对路径用起来方便输入、简洁(并且避免特定名称节点的地址被写死)。例如，例 9-1 中定义的默认的文件系统，相对的 URI /a/b 解析得到的是 *hdfs://namenode/a/b*。

注意：使用 HDFS 时，使用 fs.default.name 来指定 HDFS 名称节点和默认文件系统，意味着 HDFS 是服务器配置中的默认文件系统。不过，为了方便，也可以指定一个不同的文件系统作为客户端配置的默认文件系统。

例如，如果使用 HDFS 和 S3 文件系统，可以指定这两个其中的一个来作为客户端配置中的默认文件系统，系统在涉及默认文件系统有一个相对的 URI 的，而另一个文件系统也有一个绝对的 URI。

HDFS 还需要设置一些其他的配置属性：为名称节点和数据节点设置存储目录。属性 `dfs.name.dir` 指定一个目录列表，让名称节点不断地存储文件系统元数据(编辑日志和文件系统镜像)。每一个元数据文件的副本都存储在一个目录下。配置 `dfs.name.dir` 使名称节点元数据写在一个或两个本地磁盘和一个远程的磁盘上，如 NFS 安装目录，这是很常见的。这样的设置是为避免本地磁盘的失效和整个名称节点的失效，在这两种情况下文件都能恢复，启动一个新的名称节点。(第二名称节点进行名称节点的周期性的检查点备份，它不提供名称节点的最新备份。)

还应设置 `dfs.data.dir` 属性，它为数据节点指定了一个目录列表区存储它的块。与使用多重目录的名称节点不同，数据节点会轮询写入它的存储目录，所以为了提高性能应该为每一个本地磁盘指定一个存储目录。读取性能也会因存储在多个磁盘而提升，因为块分布在其间，与此相应可以在这些磁盘对不同的块进行并发的读。

注意：为达到最高性能，挂载存储磁盘时应加上 noatime 选项。此设置意味着读文件时，最后访问时间信息不会被写入，通过这样来获得更佳的性能。

最后，我们要配置第二名称节点存储它的文件系统检查点的位置。fs.checkpoint.dir 属性指定一个目录的列表来保存检查点。就像名称节点的存储目录保留着名称节点元数据的副本，检验点的文件系统镜像也应复制存储在每一个检验点目录中的。

表 9-3 总结了 HDFS 的重要的配置属性。

表 9-3：重要的 HDFS 后台程序属性

属性名称	类型	默认值	描述
fs.default.name	URI	file:///	默认文件系统。URI 定义了主机名和 jobtracker 的 RPC 服务器运行的端口。默认端口为 8020。此属性可在 core-site.xml 中设置
dfs.name.dir	用逗号隔开的目录名	${hadoop.tmp.dir}/dfs/name	存储名称节点永久元数据的目录的列表。名称节点在列表中的每一个目录下存储着元数据的副本
dfs.data.dir	用逗号隔开的目录名	${hadoop.tmp.dir}/dfs/name	数据节点存储块的目录的列表
fs.checkpoint.dir	用逗号隔开的目录名	${hadoop.tmp.dir}/dfs/namesecondary	第二名称节点用来存储检查点的目录的列表。它在列表的每一个目录下存储着检查点的副本

注意：注意，HDFS 的存储目录默认在 Hadoop 的临时目录下(hadoop.tmp.dir 属性，其默认目录是/tmp/hadoop-${user.name})。所以这些属性的设置很重要，可保证数据在系统清空临时目录时不会丢失。

MapReduce

运行 MapReduce 需要指明一台机器作为 jobtracker，在小的集群中可以与名称节点为同一台机器。为此，要设置 mapred.job.tracker 属性的主机名或 IP 地址和端口使 jobtracker 可以监听。注意，此属性不是一个 URI，而是一对主机端口，是通过一个冒号分开的。一般端口号为 8021。

在 MapReduce 作业中，中间数据和工作文件是写入临时本地文件的。由于这个数据包含可能非常大的 map 任务的输出，就需要配置控制本地临时存储位置的 mapred.local.dir 属性确保使用的磁盘分区足够大。mapred.local.dir 属性用一个逗号分隔列表的目录名称表示，应该尽量使用所有可用的本地磁盘来分布进行磁盘 I/O。一般，我们都使用存储数据块的磁盘和分区(但是在不同的目录)存储 MapReduce 临时数据，之前已提到由 dfs.data.dir 属性控制。

MapReduce 使用分布式文件系统与运行 MapReduce 任务的 tasktracker 共享文件(例如工作 JAR 文件)。mapred.system.dir 属性会指定共享文件能存储的目录。这个目录与默认的文件系统(在 fs.default.name 中配置)相关，通常是 HDFS。

最后，设置 mapred.tasktracker.map.tasks.maximum 和 mapred.tasktracker.reduce.tasks.maximum 属性来确定 tasktracker 机器上可用核的数量，mapred.child.java.opts 属性确定 tasktracker 的子 JVM 可用的内存大小。详见第 9 章。

表 9-4 总结了 HDFS 的重要的配置属性。

表 9-4：重要的 MapReduce 后台程序属性

属性名称	类型	默认值	描述
mapred.job.tracker	主机名和端口	local	jobtracker 的 RPC 服务器运行的主机名和端口。如果设置为默认值 local，则当运行一道作业时，jobtracker 也在同一进程内(此时，不必启动 MapReduce 守护进程)
mapred.local.dir	用逗号分隔的目录名称	${hadoop.tmp.dir}/mapred/local	MapReduce 存储作业中间数据的目录列表。当作业结束时数据会被清空
mapred.system.dir	URI	${hadoop.tmp.dir}/mapred/system	当一道作业运行时，与存储共享文件的 fs.default.name 相关的目录
mapred.tasktracker.map.tasks.maximum	int	2	任一时刻 tasktracker 上运行 map 任务的数量
mapred.tasktracker.reduce.tasks.maximum	int	2	任一时刻在 tasktracker 上运行的 reduce 任务的数量

续表

属性名称	类型	默认值	描述
Mapred.child.java.opts	String	-Xmx200m	用来启动 tasktracker 子进程，运行 map 和 reduce 任务的 JVM 选项。此属性可以在每道作业上设置，这对为 debug 设置 JVM 属性比较有用

9.4.4 Hadoop 后台程序地址和端口

Hadoop 后台程序一般都会运行一个为后台程序间通信的 RPC 服务器(表 9-5)和一个提供给人们使用的网页 HTTP 服务器(表 9-6)。每一个服务器都可以通过设置网络地址和端口号来配置。定义网络地址为 `0.0.0.0` 后，Hadoop 将绑定所有的地址到当前机器上。当然，可以指定一个单一的地址去绑定。端口号 0 的指示服务器在一个空闲的端口上启动：这一般不推荐，因为与设置此集群内的防火墙策略不协调。

表 9-5：RPC 服务器属性

属性名称	默认值	描述
fs.default.name	file:///	设置 HDFS URI 时，这个属性决定了名称节点的 RPC 服务器地址和端口，默认端口为 8020
dfs.datanode.ipc.address	0.0.0.0:50020	数据节点的 RPC 服务器地址和端口设置
mapred.job.tracker	local	当设为主机名和端口时，这个属性指定 jobtracker 的 RPC 服务器地址和端口。一般使用的端口是 8021
mapred.task.tracker.report.address	127.0.0.1:0	tasktracker 的 RPC 服务器地址和端口。tasktracker 的子 JAVA 虚拟机和 tasktracker 通信时使用。这种情况下使用任意空闲的端口都是可以的，因为服务器只绑定回送地址。机器没有回送地址时才需改变这个设置

除了 RPC 服务器之外，数据节点还运行块传输的 TCP/IP 服务器。服务器地址和端口由 `dfs.datanode.address` 属性设置，默认值是 0.0.0.0:50010。

表 9-6：HTTP 服务器属性

属性名称	默认值	描述
mapred.job.tracker.http.address	0.0.0.0:50030	jobtrackerHTTP 服务器地址和端口
mapred.task.tracker.http.address	0.0.0.0:50060	tasktrackerHTTP 服务器地址和端口
dfs.http.address	0.0.0.0:50070	名称节点的 HTTP 服务器地址和端口
dfs.datanode.http.address	0.0.0.0:50075	数据节点的 HTTP 服务器地址和端口
dfs.secondary.http.address	0.0.0.0:50090	第二名称节点的 HTTP 服务器地址和端口

这里还有控制数据节点和 tasktracker 使用哪个网络接口作为它们的 IP 地址(对于 HTTP 和 RPC 服务器)的设置，其相关属性是 `dfs.datanode.dns.interface` 和 `mapred.tasktracker.dns.interface`，都使用默认的网络接口。我们可以设置任意特定接口的地址，例如 `eth0`。

9.4.5 其他 Hadoop 属性

本小节讨论将设置时要考虑的其他一些属性。

集群成员

为了在以后添加和移除节点，可以定义一个经验证的、可作为数据节点或 tasktracker 加入集群机器列表。这个列表通过使用 `dfs.hosts`(为数据节点)和 `mapred.hosts`(为 tasktracker)属性来指定类别，相应的 `dfs.hosts.exclude` 和 `mapred.hosts. exclude` 文件用于拒绝操作。详见第 10 章的“授权和关闭节点”小节。

服务级授权

我们可以定义 ACL 去控制哪些用户和群组有许可链接到各 Hadoop 服务。服务定义在协议级，所以有些为 MapReduce 工作提交，有些是名称节点通信，等等。默认授权设置为关闭(见 `hadoop.security.authorization` 属性)，所以所有用户都可以访问所有的服务。详情参见 hadoop-policy.xml 配置文件。

缓冲大小

Hadoop 为它的 I/O 操作提供了大小为 4 KB(4096 字节)的缓冲。这是一个保守的设置，对于日后新的硬件和操作系统，通过增加它的大小可以看见性能的提升。64 KB(65536 字节)或者 128 KB(131072 字节)是较普遍的选择。可以使用 `core-site.xml` 中的 `io.file.buffer.size` 属性改变缓冲大小。

HDFS 块大小

默认的 HDFS 块大小是 64 MB，但是许多集群使用 128 MB(134 217 728 字节)或者甚至是 256 MB(268 435 456 字节)给 map 操作更多的数据操作。hdfs-site.xml 文件中的 dfs.block.size 属性可以控制这个配置。

备用存储空间

默认情况下，数据节点将尝试使用它们存储目录下所有可用的空间。如果想保留一些空间给非 HDFS 使用，以字节为单位设置 `dfs.datanode.du.reserver` 属性就可保留一定空间。

垃圾

Hadoop 文件系统有一个垃圾设备，在该设备中删除的文件并没有真正被删除，而是被移到一个垃圾文件夹中，在那里它们会在被系统永久删除前保留一段极短的时间。这段极短的时间是通过 `core-site.xml` 的 `fs.trash.interval` 配置属性设置的，以分钟为单位。默认的垃圾间隔时间是 0，此时垃圾设备没有任何作用。

像在很多操作系统中一样，Hadoop 的垃圾设备是一个用户级的特征，意思是只有使用文件系统 shell 时，删除的文件才会放入垃圾箱。使用程序删除文件时，它们会立刻被删除，但可以建立一个 Trash 实例以程序方式使用垃圾箱，只要调用 `moveToTrash()`方法并传入想要删除的文件的路径。此方法将返回一个表示成功的值，若值为 false 则意味着垃圾站未启用或此文件已经在垃圾站中。

当垃圾已启用，每一个用户在它的主目录之下有一个自己的垃圾目录.Trash。文件恢复很简单：在.Trash 的子目录下查找文件并且将它从垃圾子树中移除。

HDFS 自动删除在垃圾文件夹中的文件，但其他文件系统不会这样，所以我们需要为定期执行这些工作做好准备。可以用 expunge 指令删除垃圾箱中存放超过时间的文件，使用文件系统命令如下：

```
% hadoop fs -expunge
```

Trash 类中的 expunge()函数效果相同。

任务内存限制

在一个共享的集群中，某一个用户的 MapReduce 程序错误不应影响集群中的节点。例如，map 或者 reduce 任务产生了一个内存漏洞，运行 tasktracker 的机器耗尽内存并影响到其他运行着的程序。为了防止这种情况发生，设置 `mapred.child.ulimit`，它会为由 tasktracker 启动虚拟内存中的子程序设置了一个最大容量限制。它以 KB(千字节)为单位，并且应该比 `mapred.child.java.opts` 设置的 JAVA 虚拟机内存要大；否则，子 JAVA 虚拟机不会启动。

同时，也可以使用 limits.confs 在操作系统级别设置进程限制。

作业调度器

特别地，在多用户的 MapReduce 设置中，可以考虑将默认的 FIFO 作业调度器换为功能更强大的其他调度器[①]。详见 6.3。

9.5 安装之后

一旦创建 Hadoop 集群并开始运行，我们需要给予用户权限使用它。这就需要为每一个用户创建一个主目录，并且对它设置权限许可：

```
% hadoop fs -mkdir /user/username
% hadoop fs -chown username:username /user/username
```

这时设置目录空间限制比较合适。下面给用户目录设置了一个 1 TB 的限制：

```
% hadoop dfsadmin -setSpaceQuota 1t /user/username
```

9.6 Hadoop 集群基准测试

集群建立得正确吗？回答这个问题最好的答案是基于经验：运行一些作业并确认能得到预期的结果。基准意味着好的测试，这样就可以与其他集群比较来检验新集群是否按预期工作，而且基准测试的结果能帮我们将集群调整到最佳性能。这些经常由

① Hadoop 作业调度器除了默认的 FIFO 调度器，常用的还有 Yahoo! 开发的能力调度器(Capacity Scheduler)和 Facebook 开发的公平调度器(Fair Scheduler)。

监控系统(参见 10.2 节)配合完成，我们可以观察资源是如何被整个集群使用的。

为了得到最佳的结果，应在一个没有其他使用者的集群上运行基准测试程序。事实上，在它投入服务前，用户还没有使用时是最好的。一旦用户在集群上有了周期性调度的作业。就很难找到何时集群没被使用(除非对用户安排停工期)，所以应该在这发生之前运行基准程序测试直至满意为止。

试验显示，大多数新系统硬件的故障都是硬盘故障。通过运行 I/O 密集型基准测试——也就是接下来将要描述的——我们可以在集群开始使用前先“热身”。

9.6.1 Hadoop 基准测试

Hadoop 带有一些基准测试程序，可以最少的准备成本轻松运行。基准测试程序被打包在测试程序 JAR 文件中，通过无参数地调用 JAR 文件可以得到其列表：

```
% hadoop jar $HADOOP_INSTALL/hadoop-*-test.jar
```

无参数地调用时，大多数基准测试程序都会显示使用说明。例如：

```
% hadoop jar $HADOOP_INSTALL/hadoop-*-test.jar TestDFSIO
TestFDSIO.0.0.4
Usage: TestFDSIO -read | -write | -clean [-nrFiles N] [-
fileSize MB] [-resFile
resultFileName] [-bufferSize Bytes]
```

用 TestDFSIO 基准测试 HDFS

TestDFSIO 用来测试 HDFS 的 I/O 性能。它通过使用 MapReduce 作业来完成测试作为并行读写文件的便捷方法。每个文件的读或写都在单独的 map 任务中进行，并且 map 的输出可以用来收集统计刚刚处理过的文件。这个统计数据在 reduce 中累加起来得出一个汇总。

以下命令写了 10 个文件，每个 1000 MB：

```
% hadoop jar $HADOOP_INSTALL/hadoop-*-test.jar TestDFSIO -write
-nrFiles 10
-fileSize 1000
```

在运行的最后，结果被写入控制台并同时记录在一个本地文件(以追加的形式，以便重新运行基准测试而不会丢失之前的结果)：

```
% cat TestDFSIO_results.log
```

```
----- TestDFSIO ----- : write
           Date & time: Sun Apr 12 07:14:09 EDT 2009
       Number of files: 10
Total MBytes processed: 10000
      Throughput mb/sec: 7.796340865378244
Average IO rate mb/sec: 7.8862199783325195
IO rate std deviation: 0.9101254683525547
    Test exec time sec: 163.387
```

文件默认写在*/benchmarks/TestDFSIO* 目录下(可以通过设置 `test.build.data` 系统属性来改变)，一个叫 `io_data` 的目录中。

使用`-read` 语句运行读取基准测试程序。注意，这些文件必须已经存在(由 `TestDFSIO -write` 写入)：

```
% hadoop jar $HADOOP_INSTALL/hadoop-*-test.jar TestDFSIO -read
-nrFiles 10
-fileSize 1000
```

实际运行结果如下：

```
----- TestDFSIO ----- : read
           Date & time: Sun Apr 12 07:24:28 EDT 2009
       Number of files: 10
Total MBytes processed: 10000
      Throughput mb/sec: 80.25553361904304
Average IO rate mb/sec: 98.6801528930664
IO rate std deviation: 36.63507598174921
    Test exec time sec: 47.624
```

完成基准测试后，可以通过参数`-clean` 从 HDFS 上删除所有生成的文件：

```
% hadoop jar $HADOOP_INSTALL/hadoop-*-test.jar TestDFSIO -clean
```

用排序测试 MapReduce

Hadoop 自带一个部分排序的程序。这对测试整个 MapReduce 系统很有用，因为整个输入数据集都会通过洗牌传输(至 reducer)。一共三个步骤：生成一些随机的数据，执行排序，然后验证结果。

首先我们通过使用 `RandomWriter` 生成一些随机的数据。它以每个节点 10 个 map 的方式运行一个 MapReduce 作业，并且每一个 map 生成(近似)10 GB 的随机二进制数据，带有不同长度的键和值。如果想要改变这些值，可以设置属性 `test.randomwriter.maps_per_host` 和 `test.randomwrite.bytes_per_map`。

还有一些设置可以限定键和值的长度范围，详情请参见 RandomWriter。

这里给出如何调用 RandomWriter(在 JAR 文件中，而非这个测试的文件)将它的输出写到 random-data 目录：

```
% hadoop jar $HADOOP_INSTALL/hadoop-*-examples.jar randomwriter
random-data
```

接下来运行 Sort 程序：

```
% hadoop jar $HADOOP_INSTALL/hadoop-*-examples.jar sort random-
data sorted-data
```

排序程序的整个执行时间是我们比较关注的度量标准，不过通过网页 UI(*http://jobtracker-host:50030/*)观测作业进度也很有帮助，这样可以了解作业的各阶段要花多少时间。调整 5.6 节提到的参数也很有用。

通过最后的检测，我们确定 sorted-data 中的数据是正确排序的：

```
% hadoop jar $HADOOP_INSTALL/hadoop-*-test.jar testmapredsort -
sortInput random-data \
-sortOutput sorted-data
```

这条命令运行了 SortValidator 程序，该程序对未排序和已排序的数据做一系列的检测来确认排序是否正确。它在运行的最后将结果报告给控制台：

```
SUCCESS! Validated the MapReduce framework's 'sort' successfully.
```

其他基准测试

其实还有很多更多的 Hadoop 基准测试，但下面这些是广泛使用的。

- MRBench(用 mrbench 调用)多次运行一道小作业。它是排序测试的好搭档，检测小的作业是否运行良好。[①]
- NNBench(用 nnbench 调用)对名称节点的压力测试很有用。
- Gridmix 是一套设计为模拟一个现实的集群作业负荷的基准测试，可以模拟许多在实践中可见的数据存取模式。详情请见分发包中的 src/benchmarks/gridmix 目录。[②]

---

① 访问 *https://issues.apache.org/jira/browse/HADOOP-307*，此测试作者的原意是评估作业启动时的代价。

② 类似地，PigMix 是针对 Pig 的一套基准测试，可从 *http://wiki.apache.org/pig/ PigMix* 获取。

9.6.2 用户作业

为了调优，最好能包含一些用户运行的少数具有代表性的作业，集群要为此进行调优而不只是标准的基准测试。如果这是你的第一个 Hadoop 集群并且还没有任何用户作业，那么 Gridmix 是一个很好的替代。

在将自己的任务作为基准测试进行运行时，我们应该为每次使用的用户任务选择一个数据集，以便于比较每次运行情况。建立一个新的集群或者升级集群时，应使用相同的数据集来比较各次运行的性能。

9.7 云计算中的 Hadoop

尽管许多组织选择在内部运行 Hadoop，但使用云计算，在一个租用的硬件上或者作为一个服务运行 Hadoop 也是很流行的。比如，Cloudera 为 Hadoop 在公有或者私有云提供了工具(见附录 B)。

在本小节中，我们会观察 Hadoop 在 Amazon EC2 上运行，在低投入试用原则下尝试 Hadoop 集群，这是一个很不错的方法。

Amazon EC2 中的 Hadoop

Amazon Elastic Compute Cloud(EC2)是一个计算服务，它允许客户租用计算机(实例)让他们运行他们自己的应用。客户可以按照需求运行和终止实例，按活动实例的运行时间支付费用。

> **EC2 网络拓扑**
>
> EC2 API 没有为客户端提供对整个网络拓扑的控制。在写作本书期间，还不能有任何线索可以提示实例的位置给出实例(也就是区分不同的机架)。我们也无法得知实例启动后会在哪一个机架上，这意味着 Hadoop 将整个集群视为一个机架，即使现实中实例很可能被分散到不同机架。Hadoop 仍会工作，但不会有它应有的高效。

Hadoop 带有一套脚本，用于在 EC2 上轻松运行 Hadoop。这些脚本可以执行很多

操作，例如启动或者终止运行集群，或添加实例到现有集群中。

在 EC2 上运行 Hadoop 尤其适用于某些工作流。例如，如果在 Amazon S3 中存储数据，然后在 EC2 上运行集群并运行读取这个 S3 数据的 MapReduce 作业，最后在关掉集群之前将输出写回 S3 中。如果长期使用集群，复制 S3 数据到运行在 EC2 上的 HDFS 中，则会使处理更高效，因为 HDFS 能充分利用数据所在位置，S3 则不能(因为 S3 与 EC2 存储不在同一个节点上)。

设置

在 EC2 上运行 Hadoop 之前，需要通过 Amazon 的 Getting Started Guide(从 EC2 的网址 *http://aws.amazon.com/ec2/*链接)建立一个账户，安装 EC2 命令行工具，并运行一个实例。

接下来，复制 Hadoop EC2 执行脚本。如果已经在工作区中安装 Apache Hadoop(附录 A)，就可以在安装的子目录 src/contrib/ec2 下找到它们。如果没有安装 Hadoop，现在开始做吧。(注意，这个脚本还可以用来运行 Apache Hadoop 集群，或者 Cloudera 的 Hadoop 分发包，详见附录 B。)

需要配置脚本来设置 Amazon Web Service 证书、安全密钥的细节、Hadoop 的版本和使用的 EC2 实例类型(最适合运行 Hadoop 的是加大型实例)。编辑文件 src/contrib/ec2/bin/hadoop-ec2-env.sh 来完成这些工作，文件中的注释说明了如何设置每一个变量。

运行集群

我们现在已经准备好创建一个集群了。运行一个有一个主节点(正在运行名称节点和 jobtracker)和五个工作节点(正在运行数据节点和 tasktraker)的名为 test-hadoop-cluster 的集群，键入以下命令：

```
% bin/hadoop-ec2 launch-cluster test-hadoop-cluster 5
```

如果不存在，以上命令将为集群创建 EC2 安全组，并且使主节点和工作节点之间可以自由访问。SSH 也可以从任意位置访问。一旦安全组建立起来，主机实例就会运行，一旦它启动了，5 个工作机实例也会运行。这是因为工作节点独立运行，主节点的主机名就能够传递给工作机实例，并允许数据节点和 tasktracker 启动时连接到主机。

运行 MapReduce 作业

作业需要在 EC2 中运行(EC2 主机名需被正确解析)，为此，我们需要传输作业的 JAR 文件给集群。脚本提供了一个方便的方法。下面的命令复制 JAR 文件到主节点：

```
% bin/hadoop-ec2 push test-hadoop-cluster /Users/tom/htdg-
examples/job.jar
```

这也是登录集群的捷径。这个命令可以登录主机(使用 SSH)，也可以通过 EC2 实例 ID 来登录集群中的任意节点。(还有一个 screen 指令的捷径，或许更好。)

```
% bin/hadoop-ec2 login test-hadoop-cluster
```

集群的文件系统是空的，所以在我们运行任务之前，需要向其中放入数据。通过使用 Hadoop 的 distcp 工具从 S3 中执行并行复制是传输数据到 HDFS 的有效方法：

```
# hadoop distcp s3n://hadoopbook/ncdc/all input/ncdc/all
```

数据复制之后，我们可以用常用方法运行一道作业：

```
# hadoop jar job.jar MaxTemperatureWithCombiner input/ncdc/all
output
```

当然，我们也可以指定输入为 S3，效果相同。若在相同的输入数据上运行多道作业，最好先复制数据到 HDFS 上以节省带宽：

```
# hadoop jar job.jar MaxTemperatureWithCombiner
s3n://hadoopbook/ncdc/all output
```

我们可以通过使用 jobtracker 的网页 UI 跟踪作业的进度，见 *http://master_host:50030/*。

终止集群

要关闭集群，首先从 EC2 节点中退出/注销并且从我们的工作区发出 terminate-cluster 命令：

```
# exit
% bin/hadoop-ec2 terminate-cluster test-hadoop-cluster
```

此时要求确认是否关闭集群中的所有实例。

第 10 章

Hadoop 的管理

第 9 章主要讲解如何安装 Hadoop 集群。在本章中，我们将学习如何维护集群使其正常运行。

10.1 HDFS

10.1.1 持久化的数据结构

作为管理员，理解 HDFS 的组成部分(即名称节点、第二名称节点和数据节点)如何组织磁盘上的持久性数据等是非常重要的。有必要知道哪些文件可以帮助自己诊断问题或发现错误。

名称节点的目录结构

新格式化的名称节点创建的目录结构如下：

```
${dfs.name.dir}/current/VERSION
                       /edits
                       /fsimage
                       /fstime
```

第 9 章曾讲到，`dfs.name.dir` 属性是一个目录列表，是每个目录的镜像。这种机制提供了一定的弹性，尤其是其中一个目录是推荐的 NFS 类型时。

VERSION(版本)文件是 Java 属性文件，它包含所用 HDFS 的版本信息。典型文件内容如下：

```
#Tue Mar 10 19:21:36 GMT 2009
namespaceID=134368441
cTime=0
storageType=NAME_NODE
layoutVersion=-18
```

`layoutVersion` 为负整数，定义的是 HDFS 的持久数据结构的版本。此版本号和 Hadoop 的发行版本号无关。每次布局改变，这个版本号就会递减(例如，版本-18之后是-19)。这种情况下，HDFS 需要升级，因为如果新名称节点的存储布局是旧版本，此名称节点(或数据节点)将无法运行。HDFS 升级请参见第 10 章。

`namespaceID` 是文件系统的唯一标识，是文件系统首次格式化时创建的。名称节点使用它来识别新的数据节点，因为它们在向名称节点注册后才能知道 `namespaceID`。

cTime 属性标记了名称节点存储的创建时间。对于新格式化的存储，此属性的值总是为零，但只要文件系统被更新，它也会被更新到一个新的时间戳。

`storageType` 指出此存储目录包含一个名称节点的数据结构。

在名称节点的存储目录中，还有 edits，fsimage 和 fstime 这三个文件。它们都是二进制文件，使用 Hadoop 的 `Writable` 对象作为其序列化格式(详见 4.3 节)。要想了解这些文件的用途，我们需要更深入地了解名称节点的工作原理。

文件系统映像和编辑日志

文件系统客户端执行写操作 (如新建文件或移动文件)的时候，首先会被记录在编辑日志中。该名称节点还有一个文件系统元数据的内存表示标识，后者会在编辑日志被更改后得以更新。在内存中的元数据用来处理读请求。

编辑日志会在每次写操作之后但尚未将成功代码返回给客户端时，被刷新和同步。对于写入到多个目录中的名称节点，在成功返回之前，写入流必须被刷新和同步到每个拷贝中。这将确保操作不会由于机器故障而丢失。

fsimage 文件是文件系统元数据的持久性检查点。但是，它不会更新文件系统的每个写操作，因为写出 fsimage 文件(可以增长至数兆字节的规模)的速度非常慢。但这并不影响系统的弹性，因为如果名称节点失败，其元数据的最新状态可以被重建，具体方式是从磁盘中将 fsimage 加载到内存，然后将此应用于编辑日志中的每

个操作。事实上，这正是名称节点在启动时做的事情(详见 10.1.2 小节)。

注意：fsimage 文件包含文件系统中所有目录和文件 inode 的序列化形式。每个 inode 是一个文件或目录的元数据的内部表示，并包含此类信息：文件的复制等级；修改和访问时间；访问权限；块的大小以及组成文件的块。对于目录，则存储修改时间、权限和配额元数据。

fsimage 文件没有记录块存储在哪个数据节点。取而代之的是，名称节点将这种映射保留在内存中，其做法是在这些数据节点加入集群时要求数据节点所包含的块列表，此后定期执行，以确保名称节点的块映射是最新的。

如前所述，edits 文件会无限增大。虽然在名称节点运行期间这不会对系统造成影响，但如果名称节点重新启动，会花很长时间来运行编辑日志中的每个操作。在此期间，文件系统会脱机，这往往是很不可取的。

解决方案是运行并创建一个第二名称节点，其目的是产生主节点的内存中文件系统元数据的检查点。[①]检查点处理过程如下(如图 10-1 所示)。

1. 第二名称节点让主节点滚动 edits 文件，因此新的编辑操作能够进入一个新的文件中。
2. 第二名称节点从主节点获取 fsimage 和 edits(使用 HTTP GET)。
3. 第二名称节点将 fsimage 加载到内存中，执行 edits 中的每个操作，然后创建一个新的统一的 fsimage 文件。
4. 第二名称节点将新的 fsimage 发送到主节点(通过 HTTP POST)。
5. 原节点用第二名称节点的新的 fsimage 替换旧的 fsimage，旧的 edits 文件用步骤 1 中的 edits 替换。同时更新 fstime 文件，记录检查点发生的时间。

在该过程结束时，主节点有了一个最新的 fsimage 文件和更短的 edits 文件(它不一定为空，因为在设置检查点时，它可能已经接收到一些 edits)。名称节点在安全模式下时，管理员可以使用 `hadoop dfsadmin -saveNamespace` 命令来手动运行此过程。

这个过程清楚表明第二名称节点为什么与原节点有类似的内存需要(因为它把 fsimage 加载到内存中)，因此，第二名称节点在大型集群上是需要专用机器的。

---

① 从 Hadoop 0.21.0 版本起(在本书出版后)，第二名称节点将被一个具有相同功能的检查点节点取代。同时会有一个新的叫备份节点并具有相同功能的名称节点推出，其目的是维护一份名称节点元数据的最新副本，将其作为在 NFS 上储存元数据副本的替代方案。

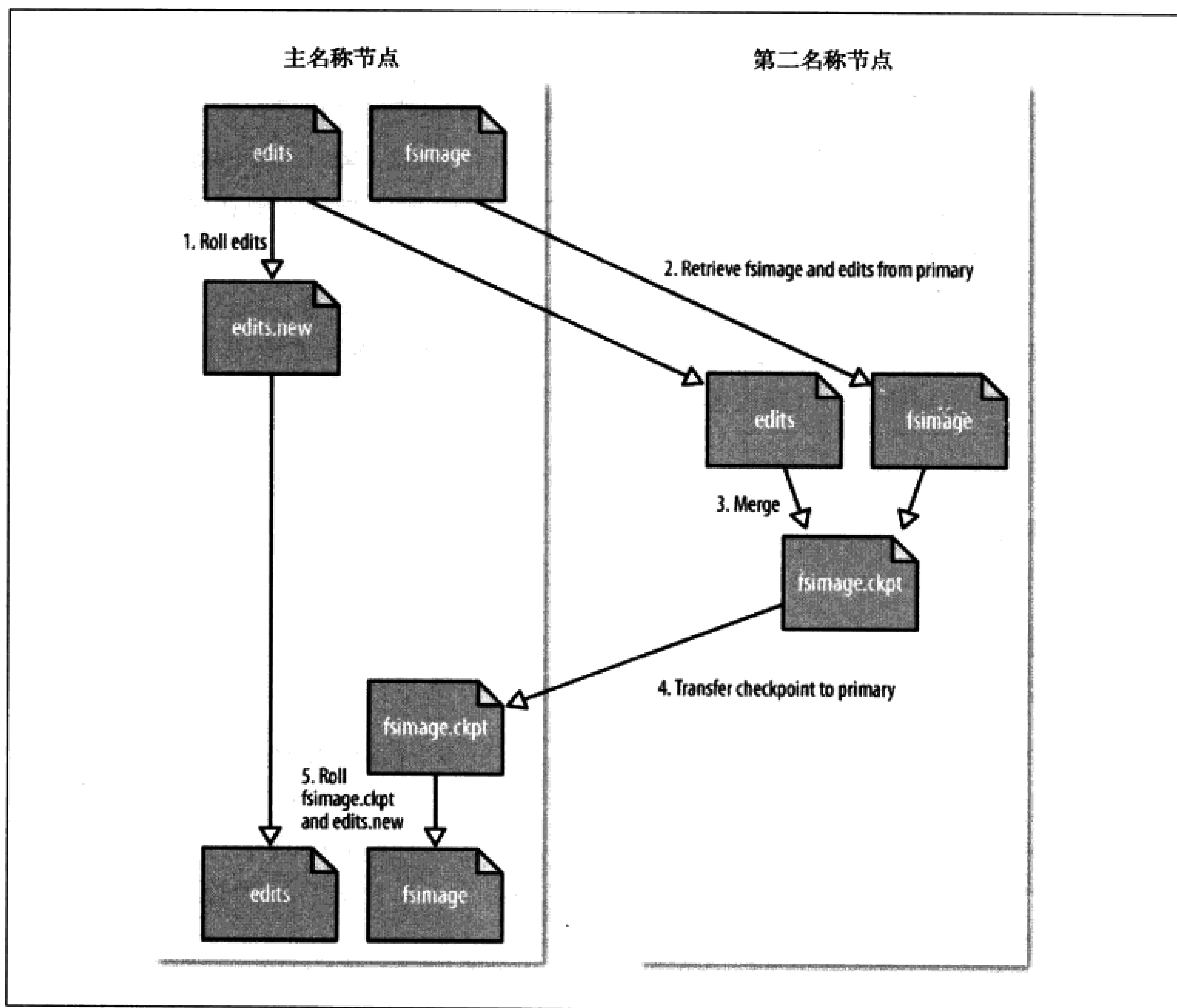


图 10-1：检查点过程图

检查点的时间表由两个配置参数决定。第二名称节点每小时插入检查点(fs.checkpoint.period 以秒为单位)，如果编辑日志已达到 64 MB (fs.checkpoint.size 以字节为单位)会更快，就每隔 5 分钟检查一次。

第二名称节点的目录结构

检查点过程的另一作用是，第二名称节点在处理过程结束时会有一个检查点，后者在子目录 *previous.checkpoint* 中。这可以作为名称节点元数据的(不是最新的)备份源。

```
${fs.checkpoint.dir}/current/VERSION
                    /edits
                    /fsimage
                    /fstime
                    /previous.checkpoint/VERSION
```

```
                                          /edits
                                          /fsimage
                                          /fstime
```

此目录和第二名称节点当前目录的布局与名称节点的完全相同。这是有意这样设计的，因为万一整个名称节点发生故障(在没有用于恢复的备份，甚至 NFS 中也没有)，就可以从第二名称节点恢复。具体方式有两种。第一，复制相关储存目录到一个新的名称节点。第二，在启动一个名称节点守护进程时，第二名称节点通过使用 -importCheckpoint 选项作为一个新的原节点继续运行任务。-importCheckpoint 选项将加载 fs.checkpoint.dir 属性定义的目录中的最新检查点的名称节点数据，但只有在 dfs.name.dir 目录中没有元数据时，才不会有重写之前元数据的风险。

数据节点的目录结构

与名称节点不同，数据节点不需要明确格式化，因为它们会自动在启动时创建自己的储存目录。关键的文件和目录如下：

```
${dfs.data.dir}/current/VERSION
                        /blk_<id_1>
                        /blk_<id_1>.meta
                        /blk_<id_2>
                        /blk_<id_2>.meta
                        /...
                        /blk_<id_64>
                        /blk_<id_64>.meta
                        /subdir0/
                        /subdir1/
                        /...
                        /subdir63/
```

数据节点的 VERSION 文件与名称节点的 VERSION 文件十分相似：

```
#Tue Mar 10 21:32:31 GMT 2009
namespaceID=134368441
storageID=DS-547717739-172.16.85.1-50010-1236720751627
cTime=0
storageType=DATA_NODE
layoutVersion=-18
```

namespaceID，cTime 和 layoutVersion 都与名称节点中的值一样(事实上，在数据节点第一次连接名称节点时，就会从名称节点获取 namespaceID)。storageID

对数据节点是唯一的(对所有存储目录都一样)，并被名称节点用于独一无二地标识数据节点。storageType 将此目录标识为数据节点储存目录。

数据节点的 current 储存目录下的其他文件名都有 blk_refix 前缀。有两种类型：HDFS 块本身(由文件的原始字节组成)和块的元数据(以.meta 为后缀)。块文件由被存储文件的原始字节组成；元数据文件由一个包含版本和类型信息的头文件和一系列块的区域校验和组成。

当目录中的块数增加到一定规模时，数据节点将新建一个子目录，用于保存新的块及其元数据。目录中的块数达到 64 时，就新建一个子目录(通过 dfs.datanode.numblocks 配置属性来设置)。新建一个足够宽泛的树，使得系统即便有大量块，目录不至于过深。通过采取这一措施，数据节点可确保每个目录中的文件数量是可控的，以避免某一个目录中有太多的文件(上万或数十万)时绝大多数操作系统都会碰到的问题。

如果配置属性 dfs.data.dir 指定了多个目录(在不同的驱动器上)，系统将以轮询的方式将数据写入块。注意，这些数据块并不是被复制到同一个数据节点上的每个块，块复制是在不同数据节点间进行的。

10.1.2 安全模式

名称节点启动时，它所做的第一件事情是加载其镜像文件(fsimage)到内存，并应用编辑日志(edits)中的编辑记录。一旦重新创建与文件系统元数据一致的内存映像，它就会创建一个新的 fsimage 文件(自己创建一个检查点而不是求助于第二名称节点，这样的效率更高)和一个空的编辑日志。只有在这个时候，名称节点才开始监听 RPC 和 HTTP 请求。如果名称节点以安全模式运行，则意味着它只向客户端提供文件系统的只读视图。

注意： 严格说来，在安全模式下，只能保证访问文件系统元数据的操作(如产生一个目录列表)能工作。只有当块在集群的当前数据节点时，才可以进行读取文件，而且对文件进行修改(写入、删除或重命名)都是不允许的。

回想一下，系统中的块的存储位置并不是由名称节点来保存的——此信息以块列表的形式储存在数据节点中。执行系统常规操作期间，名称节点在内存中储存块地址的分布。在安全模式下，需要给数据节点一些时间来登入名称节点及其块列表，使名称节点能有足够多的块地址来高效运行文件系统。如果名称节点没有等到足够多的数据节点，则会启动程序，开始复制块到新的数据节点，在大多数情况下，都不

需要如此(因为它只需要等更多数据节点登入)，并且这将造成集群资源非常的紧张。事实上，在安全模式下，名称节点并不为数据节点发出任何块复制或者删除的指令。

到达最小副本条件后，再过 30 秒，系统便退出安全模式。最小副本条件是指在整个文件系统中 99.9%的块达到最低复制水平(默认是 1，由 `dfs.replication.min` 设定)。

启动一个新格式化的 HDFS 集群时，因为系统中没有数据块，所以名称节点不会进入安全模式。

表 10-1：安全模式中的属性

属性名称	类型	默认值	描述
dfs.replication.min	int	1	写操作成功所需要的最小副本数
dfs.safemode.threshold.pct	float	0.999	名称节点退出安全模式之前系统中满足 dfs.replication.min 定义的最小副本数级别的块的百分比。将其设置成 0 或更少将强制名称节点不以安全模式启动。将这个值设置成大于 1，意味着名称节点永远不会退出安全模式
dfs.safemode.extension	int	30000	满足最小副本条件 dfs.safemode.threhold.pct 后持续到安全模式的时间，单位为毫秒。对于小型集群(十来个节点)，此属性的值可以设置为 0

进入安全模式和退出安全模式

要想查看名称节点是否处于安全模式，可以使用 `dfsadmin` 命令，如下所示：

```
% hadoop dfsadmin -safemode get
Safe mode is ON
```

也可以从 HDFS 网页 UI 的首页了解名称节点是否处于安全模式。

有时，如果想在运行命令前等待名称节点退出安全模式，可以使用 `wait` 选项，如下所示：

```
hadoop dfsadmin -safemode wait
# command to read or write a file
```

管理员可以让名称节点在任何时候进入和退出安全模式。有些情况下，在集群上进行维护或者在更新后确定数据是否可以读取，是有必要在进入和退出之间进行切换的。要进入安全模式，使用以下命令即可：

```
% hadoop dfsadmin -safemode enter
Safe mode is ON
```

名称节点在启动并仍处于安全模式时，可以使用此命令来保证它将永远不会离开安全模式。另一个确保名称节点一直在安全模式的方法是将 dfs.safemode.threshold.pct 的值设为大于 1。

可以通过以下命令让名称节点离开安全模式：

```
% hadoop dfsadmin -safemode leave
Safe mode is OFF
```

10.1.3 审计日志

HDFS 能够记录文件系统的所有访问请求，这是一些组织为审计目的而提出的功能要求。审计日志用 log4j 日志的 INFO 级来实现，但由于 log4j.properties 的日志下限被设置为 WARN 级，所以在默认配置中，审计日志是关闭的。

```
log4j.logger.org.apache.hadoop.fs.FSNamesystem.audit=WARN
```

用 INFO 来替换 WARN，即可启用审计日志，之后每个 HDFS 事件的结果都将作为一行日志被写入名称节点的日志。下面是一个例子，是/user/tom 列表状态请求：

```
2009-03-13 07:11:22,982 INFO
org.apache.hadoop.hdfs.server.namenode.FSNamesystem.audit:
ugi=tom,staff,admin ip=/127.0.0.1 cmd=listStatus src=/user/tom
dst=null
perm=null
```

配置 log4j 使审计日志写入一个单独的文件，不与名称节点的其他日志条目混在一起，这是一个很好的方法。在 Hadoop 维基 *http://wiki.apache.org/hadoop/HowToConfigure* 可以找到具体配置示例。

10.1.4 工具

dfsadmin

dfsadmin 工具是一个查询 HDFS 状态信息的多功能工具，能在 HDFS 上执行管理员的操作。可以使用 hadoop dfsadmin 调用其功能。要想执行更改 HDFS 状态的

命令，一般需要超级用户权限。

dfsadmin 的命令如表 10-2 所示。

表 10-2：dfsadmin 的命令

命令名称	描述
-help	显示指定命令的帮助说明，如果不指定哪个命令，就显示所有的命令
-report	显示文件系统的统计数据(和网页用户界面中显示的类似)和已连接的数据节点的信息
-metasave	将正在复制或删除的块的信息和已连接的数据节点的列表导入 Hadoop 日志目录下的一个文件中
-safemode	改变或者查询安全模式的状态。详见 10.1.2 节
-saveNamespace	将当前内存中的文件系统映像另存为一个新的 fsimage 文件，并重置 edits 文件。这个操作只能在安全模式中执行。
-refreshNodes	更新数据节点的设置，允许连接到名称节点。详情参见 10.3.2 节
-upgradeProgess	获得 HDFS 升级过程中的信息或强制执行更新。详情参见 10.3.3 节
-finalizeUpgrade	移除以前版本的数据节点和名称节点的储存目录。更新之后且集群在新版本上成功运行时可以使用这个命令。详情参见 10.3.3 节
-setQuota	设置目录配额。目录配额设置了目录树中的文件或者目录的数量限制。目录配额有助于防止用户创建大量的小文件，帮助名称节点保留内存空间(回忆一下在内存中储存的文件系统的文件、目录和块的计算信息)
-clrQuota	清空定义的目录配额
-setSpaceQuota	设置目录的空间配额。空间配额设置了可在目录树中储存的文件大小限制。给用户指定一个限定大小的储存空间清
-clrSpaceQuota	空指定的空间配额
-refreshServiceAcl	重置名称节点的服务级别授权策略文件

文件系统检查(fsck)

Hadoop 提供一个 fsck 工具来检查 HDFS 中文件的健康情况。该工具将查找所有数

据节点中丢失的块和副本不足或者副本过多的块。下面这个例子检查一个小型集群的整个文件系统

```
% hadoop 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
Missing replicas: 0 (0.0 %)
Number of data-nodes: 4
Number of racks: 1

The filesystem under path '/' is HEALTHY
```

fsck 递归走查文件系统的命名空间，从指定的路径开始(这里是文件系统的 root)，然后检查它找到的文件。它为每个检查的文件打上一个小圆点。为了检查文件，fsck 会检索文件块的元数据，并查找有问题或者不一致的地方。注意，fsck 检索名称节点中它的所有信息，它并没有与任何数据节点通信来实际检索任何数据块。

fsck 的大部分输出都很直观，但它查找的下面这些条件需要解释一下。

过量复制块

指数量超过其所属文件目标副本数。过量复制并不常见，HDFS 会自动删除多余的副本。

复制不足块

指数量不足其所属文件目标副本数。HDFS 会自动创建少量新的副本直到满足目标副本数。可以使用 `hadoop dfsadmin -metasave` 得到正在复制(或等待复制)的块的信息。

复制错误的块

指不满足块副本放置规则。例如，对于一个在多个机架的集群中的三级副本，如果三个块的副本全都在同一个机架上，那么这个块就是复制错误的块，因为副本应该分布在至少两个机架上以保证其可恢复。

复制错误的块不能由 HDFS 自动修复(本书写作时如此)。一个变通的做法是，增加块所属文件的副本数来手动修复这个问题(使用 hadoop fs -setrep)，等到块被复制之后，再将文件副本数恢复为原值。

损坏的块

指其副本被损坏的块。如果块中有一个未损坏的副本，也不会报告为损坏的块，名称节点会复制未损坏的副本直到达到目标副本数。

无副本的块

指在集群中没有副本的块。

损坏的块和无副本的块是我们最为关心的，因为它们意味着数据已经丢失了。默认情况下，fsck 会留下有损坏块或者无副本块的文件，但可以要求它执行下面的某个操作。

- 使用-move 选项把受影响的文件移到 HDFS 中的/lost+found 目录。 文件被分成连续的块以帮助你补救这些文件。
- 使用-delete 选项删除受影响的文件。文件在删除后不能被恢复。

找到文件中包含的块 fsck 工具提供一个简单方法用于找到任何一个特定文件中包含哪些块。例如：

```
% hadoop 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]
```

这说明文件/user/tom/part-00007 是由一个块组成的，并显示了块所在的数据节点。所用的 fsck 的选项如下所示。

- -files 选项显示文件名、文件大小、块的数量和它们的健康情况(是否有丢失的块)。
- -blocks 选项显示文件中每个块的信息，每行显示一个块。
- -racks 选项显示机架的位置和每个块的数据节点的地址。

不带参数执行 hadoop fsck，则显示所有可用的指令。

数据节点的块扫描器

每个数据节点运行一个块扫描器，后者将定期检查存储在数据节点中的所有块。这样一来，坏的块就可以在被客户端读取之前得以修复。DataBlockScanner 维护着一个待检验块列表，逐个扫描这些块，找出校验和错误。扫描器使用一个节流机制以限制数据节点上的磁盘带宽。

块每隔三周定期检查以避免磁盘错误(由 dfs.datanode.scan.period.housrs 属性控制，其默认值为 504 小时)。损坏的块会被报告给名称节点进行修复。

可以通过访问数据节点的网络接口 http://datanode:50075/blockScannerReport 获得块的检查报告。下面这个报告一目了然：

```
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*，报告开头便是数据节点上所有块的列表及其最新检查状态。下面是块列表的一部分(为适合页面大小，这里分成多行显示)：

```
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，随后是一些键/值对。这个状态根据最后一次块扫描是否找到校验

和错误来判定为 failed(错误)或 ok(正常)。如果扫描是后台线程执行的，那么扫描类型就是 local(本地的)。如果是由客户端或者一个远程的数据节点执行的，那么扫描类型就是 remote(远程的)。如果块扫描尚未执行，扫描类型就是 none。最后一段信息是扫描时间，包括从 1970 年 1 月 1 日午夜至今的毫秒数和一个更易读的值。

均衡器

随着时间的推移，数据节点上的块的分布会变得不均衡。一个负载不平衡的集群可能影响 MapReduce 的本地化优势，为负载高的数据节点带来更大的压力，因此最好能避免这种情况。

均衡器程序是一个 Hadoop 守护进程，用来重新分布块，具体做法是在遵循块副本放置策略(把块副本放在不同机架上以减少数据丢失)的同时，把块从使用过度的数据节点移到使用不足的数据节点(详见“副本放置”小节)。移动块直到集群任务趋于平衡，这意味着每个数据节点的使用率(在节点上使用的空间占节点整体负载能力的比率)与集群使用率(在集群上使用的空间占集群整体负载能力的比率)之差要在指定百分比以下。可以使用下面的命令启动均衡器程序：

```
% start-balancer.sh
```

-threshold 参数定义指定百分比来保证集群的负载均衡。标志是可以选择的，本例中指定的值是 10%。

在任一时间，每个集群上只能运行一个均衡器。

均衡器一直运行直到集群趋于均衡为止，不能移动块或者失去与名称节点的联系。它在标准日志目录中产生一个日志文件，记录它执行的每次重分配。下面是来自一个小型集群上短时间运行后的输出：

```
Time Stamp Iteration# Bytes Already Moved Bytes Left To Move
Bytes 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 6.0729333333333333 minutes
```

均衡器被设计成在后台运行，不会加重集群负担或者对使用集群的其他客户造成影响。它限定了从一个节点复制到另一个节点的带宽。默认是 1MB/s，但它可以通过 *hdfs-site.xml* 文件中的 dfs.balance.bandwidthPerSec 属性来设置，以字节为单位。

10.2 监控

监控是系统管理的重要组成部分。在本小节，我们将学习 Hadoop 的监控机制，了解如何加入外部监控系统。

监控的目的是检测集群是否提供预期的服务级别。主守护进程对监控最重要，它们是：(第一和第二)名称节点和 `jobtracker`。数据节点和 `tasktracker` 出现故障是正常的，特别是在大型集群上运行时，所以应该提供一个额外的能力，以便集群可以容忍随时可能发生的小部分节点死亡。

除了后面要讲的一些措施，一些管理员还会定期运行测试作业对集群的健康状况进行测试。

要加入更多监控能力到 Hadoop，后续还有很多工作要做，这部分内容本书将不予涉及。例如，Chukwa[①]是一个基于 HDFS 和 MapReduce 的数据收集和监控系统，它擅长挖掘日志数据来找出大规模的趋势。另一个例子是定义一个 Hadoop 守护进程的服务生命周期，包括加入一个“ping”方法来获得一个守护进程的健康状况概要说明。[②]

10.2.1 日志

所有 Hadoop 守护进程产生的日志文件，对了解系统运行中发生的事件都十分有用。第 9 章阐述了如何配置这些文件。

设置日志级别

在调试一个问题时，在系统中临时改变特定组件的日志级别是十分方便的。

Hadoop 守护进程有一个网页，用于改变任何 log4j 日志名日志级别，它位于 daemon 守护进程网页用户界面的*/logLevel*。按照惯例，Hadoop 中的日志名对应于记录日志的类的名称，当然也有例外，所以应查看源代码从中找到日志名。

例如，要调试 `JobTrcker` 类的日志，浏览 `jobtracker` 的网页用户界面 *http://jobtracker-host:50030/logLevel*，并将日志名 `org.apache.hadoop.mapred.JobTracker` 的级别设置为 DEBUG。

可以像下面这样通过命令行提示窗口来完成上述操作：

---

① *http://hadoop.apache.org/chukwa*。

② *https://issues.apache.org/jira/browse/HADOOP-3628*。

```
% hadoop daemonlog -setlevel jobtracker-host:50030 \
  org.apache.hadoop.mapred.JobTracker DEBUG
```

守护进程重新启动时，以这种方式来改变的日志级别将被重置，通常情况下，我们也希望如此。当然，如果要对日志级别实现一个持久的更改，通常需要更改配置目录中的 log4j.properties 文件。这种情况下，要增加下面这行设置：

```
log4j.logger.org.apache.hadoop.mapred.JobTracker=DEBUG
```

获得栈的轨迹

Hadoop 守护进程会提供一个网页(在网页用户界面中的/stacks)，为守护进程的 JVM 中正在运行的所有线程产生线程转储。例如，可以从 *http://jobtracker-host:50030/stacks* 得到一个 jobtracker 线程转储。

10.2.2 度量

HDFS 和 MapReduce 守护进程收集事件和度量相关数据(统称为度量)。例如，数据节点收集的度量(及其他)有：写入的字节数、被复制的块的数量和从客户端到来的读取请求数(本地和远程都有)。

度量隶属于上下文(context)，并且 Hadoop 目前使用"dfs"，"MapReduce"，"rpc"和"jvm"这些上下文。Hadoop 守护进程经常收集多个上下文下的度量。例如，数据节点为"dfs"，"rpc"和"jvm"上下文收集度量信息。

上下文定义着发布单位，你可以选择来发布"dfs"上下文，而非"jvm"上下文。可以在文件 *conf/hadoopmetrics.properties* 中设置度量，默认情况下，所有的上下文都已配置，因此它们不会发布自己的度量。默认配置文件的内容(去掉注释)如下：

```
dfs.class=org.apache.hadoop.metrics.spi.NullContext
mapred.class=org.apache.hadoop.metrics.spi.NullContext
jvm.class=org.apache.hadoop.metrics.spi.NullContext
rpc.class=org.apache.hadoop.metrics.spi.NullContext
```

这个文件中的每行都配置有一个不同的上下文，指定了该上下文环境下处理度量的类。这个类必须是 MetricContext 接口的一个实现，而且，正如其名，NullContext 类既不发布度量，也不更新度量。[①]

---

① 这里的上下文(可能不幸地)会有歧义，因为它既可以指一些度量(如"dfs"上下文)，也可以指发布度量的类(如 NullContext)。

MetricsContext 的其他实现将在下面几个小节介绍。

> **度量与计数器有何不同？**
>
> 主要的差异在于它们的范围：度量是由 Hadoop 守护进程收集的，而计数器(详见第 8 章)是为 MapReduce 任务收集的，汇总为对整个作业的计数。它们的读者也不同：一般而言，度量针对管理员，而计数器针对 MapReduce 用户。
>
> 它们收集和汇总的方法也不同。计数器是 MapReduce 的功能，并且 MapReduce 系统保证计数器的值由产生它们的 tasktracker 传播，然后再传给 jobtracker，最后返回运行 MapReduce 作业的客户端。(计数器是通过 RPC 心跳来传播的，详见第 6 章)。tasktracker 和 jobtracker 都会执行汇总。
>
> 度量的收集机制与接受更新的组件的关系是松耦合的，并且还有各种可插拔的输出，包括本地文件、Ganglia 和 JMX。守护进程收集这些度量并在输出之前进行汇总。

FileContext

`FileContext` 将度量写入一个本地文件。它有两个配置属性：`fileName`(定义要写入的文件的绝对名称)和 `period`(文件更新期间的时间间隔，以秒为单位)。这两个属性都是可选的，如果不设置这两个属性，metrics 将每隔五秒写到标准输出。

通过附加属性名到上下文名(用点分隔)，配置属性即可应用于上下文并被指定。例如，要将“jvm”转储到一个文件中，像下面这样修改其配置即可：

```
jvm.class=org.apache.hadoop.metrics.file.FileContext
jvm.fileName=/tmp/jvm_metrics.log
```

在第一行中，我们改变“jvm”的上下文以使用 `FileContext`。在第二行，将“jvm”上下文的 `fileName` 属性设置为一个临时文件。下面是来自日志文件中的两行输出(为适合页面大小，已经将其转成多行)：

```
jvm.metrics: hostName=ip-10-250-59-159, processName=NameNode,
sessionId=, gcCount=46, ↵
gcTimeMillis=394, logError=0, logFatal=0, logInfo=59, logWarn=1, ↵
memHeapCommittedM=4.9375, memHeapUsedM=2.5322647,
```

```
memNonHeapCommittedM=18.25, ↵
memNonHeapUsedM=11.330269, threadsBlocked=0, threadsNew=0,
threadsRunnable=6, ↵
threadsTerminated=0, threadsTimedWaiting=8, threadsWaiting=13
jvm.metrics: hostName=ip-10-250-59-159,
processName=SecondaryNameNode, sessionId=, ↵
gcCount=36, gcTimeMillis=261, logError=0, logFatal=0,
logInfo=18, logWarn=4, ↵
memHeapCommittedM=5.4414062, memHeapUsedM=4.46756,
memNonHeapCommittedM=18.25, ↵
memNonHeapUsedM=10.624519, threadsBlocked=0, threadsNew=0,
threadsRunnable=5, ↵
threadsTerminated=0, threadsTimedWaiting=4, threadsWaiting=2
```

`FileConext` 适用于在本地系统进行调试，但它不适用于大型集群，因为输出文件分布在整个集群上，分析起来十分困难。

GangliaContext

Ganglia(*http://ganglia.info/*)是适用于大型集群的一个开源分布式监视系统。它为集群上每个节点带来的资源负担极少。Ganglia 自己就能收集度量，例如 CPU 和内存使用情况，通过使用 `GangliaContext`，即可将 Hadoop 度量加入 Ganglia。

`GangliaContext` 有一个必需的属性，即 `servers`，它是一个以空格或以逗号分隔的 Ganglia 服务器主机-端口对。要想进一步了解如何配置这个上下文，可访问 Hadoop Wiki(*http://wiki.apache.org/hadoop/GangliaMetrics*)。

能够从 Ganglia 中得到的信息如图 10-2 所示，它表明 jobtracker 队列中的任务数随时间的变化情况。

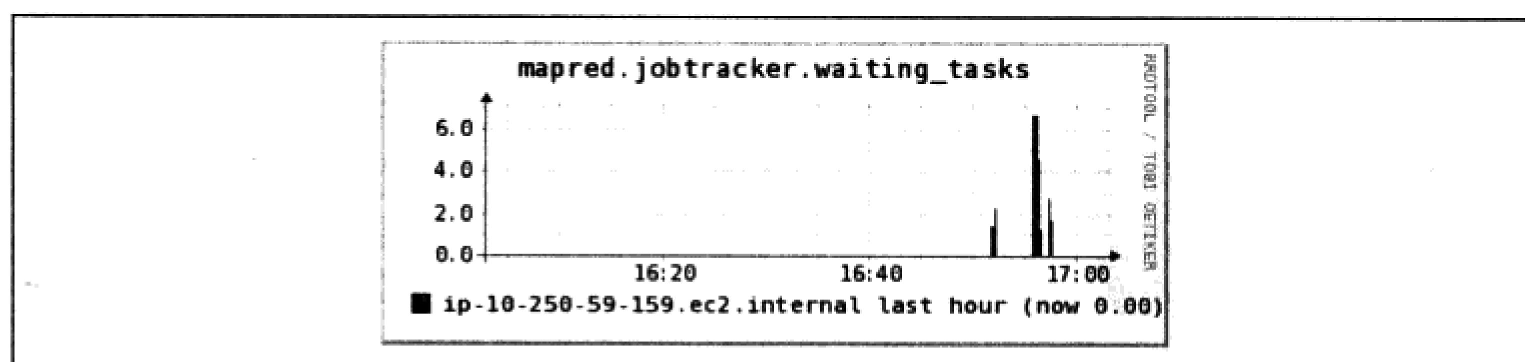


图 10-2：Ganglia 的显示片段：jobtracker 队列中的任务数

NullContextWithUpdateThread

FileContext 和 GangliaContext 都会将度量加入一个外部系统。然而，一些监控系

统——如著名的 JMX——需要从 Hadoop 中提取出度量。NullContextWith-UpdateThread 就是为此而设计的。和 NullContext 相同，它并不发布任何度量，但它会额外运行一个定时器。定时器定期更新储存在内存中的度量。这确保了另一个系统在取得度量时，这些度量是最新的。

除了 NullContext 以外的所有 MetricContext 实现都会执行这个更新功能(并且它们都提供一个 period 属性，默认值为五秒)，所以只有在未使用另一个输出来收集度量时才需要用 NullContextWithUpdateThread。如果正在使用 GangliaContext，那么它会确保度量被更新，因此可以额外使用一个不带更多度量系统配置的 JMX。JMX 将在后文详细介绍。

CompositeContext

CompositeContext 允许你输出相同的度量集到多个上下文，例如 FileContext 和 GangliaContext。配置有点复杂，最好能有例子加以说明，如下所示：

```
jvm.class=org.apache.hadoop.metrics.spi.CompositeContext
jvm.arity=2
jvm.sub1.class=org.apache.hadoop.metrics.file.FileContext
jvm.fileName=/tmp/jvm_metrics.log
jvm.sub2.class=org.apache.hadoop.metrics.ganglia.GangliaContext
jvm.servers=ip-10-250-59-159.ec2.internal:8649
```

arity 属性用来定义子上下文的数量。在本例中，有两个。每个子上下文的属性名都被修改为包含定义子上下文数，例如 jvm.sub1.class 和 jvm.sub2.class。

10.2.3 Java 管理扩展

Java 管理扩展(JMX)是一个标准的 Java API，用于监视和管理应用程序。如表 10-3 所示，Hadoop 包含多个管理 beans(MBeans)，它们向 JMX 应用提供 Hadoop 度量。Hadoop 有向外提供“dfs”和“rpc”上下文度量的 MBeans，但没有“mapred”上下文（在写本书的时候还没有)或者“jvm”上下文(因为 JVM 本身提供一个更丰富的 JVM 度量)的 MBeans。

表 10-3：Hadoop 的 MBeans

MBean 类	后台进程	度量
NameNode-ActivityMBean	名称节点	名称节点活动度量，例如创建文件操作的数量

续表

MBean 类	后台进程	度量
FSNamesystem-Mbean	名称节点	名称节点状态度量，例如已连接数据节点的数量
DataNodeActivity-Mbean	数据节点	数据节点活动度量，例如读入的字节数
FSDataset-Mbean	数据节点	数据节点储存度量，例如容量、空闲的储存空间
RpcActivity-Mbean	所有使用 RPC 的守护进程：名称节点\数据节点\jobtracker，tasktracker	RPC 统计数据，例如平均处理时间

JDK 自带一个工具 Jconsole，用于查看正在运行的 JVM 中的 MBeans。它很适合用于浏览 Hadoop 的 度量，如图 10-3 所示。

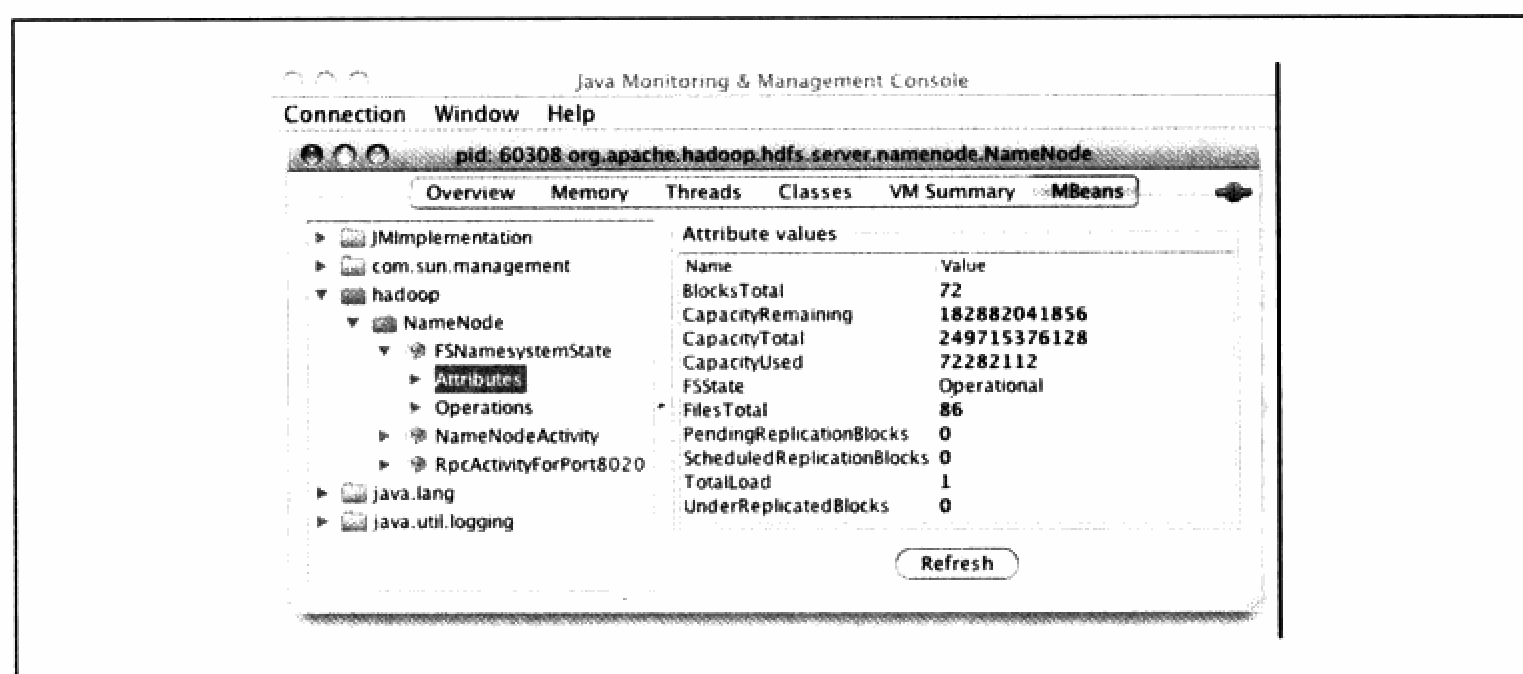


图 10-3：本地运行的名称节点的 JConsole 视图，显示其文件系统状态的度量

注意：虽然可以使用默认的度量配置通过 JMX 看到 Hadoop 度量，但除非把 `MetricsContext` 实现改为非 `NullContext`，否则它们不会被更新。例如，如果只使用 JMX 这一种方法来监视度量，那么 `NullContextWithUpdateThread` 是很合适的。

许多第三方的监视和警告系统(如 Nagios 或者 Hyperic)可以查询 MBeans，让 JMX 自然地从一个已有的监控系统中来监控 Hadoop 集群。这时需要启动对 JMX 的远程访问，并选择一个适合集群的安全级别。可选的方法包括密码验证、SSL 连接和

SSL 客户验证。要想进一步了解如何配置这些选项，请参见 Java 官方文档[①]。

所有启用 JMX 远程访问的选项都涉及设置 Java 系统属性，这是通过编辑 *conf/hadoop-env.sh* 文件实现的。下面的配置设置显示了，如何启用在名称节点上以密码验证方式的 JMX 远程访问(SSL 是关闭的)。其他 Hadoop 守护进程的处理也类似。

```
export HADOOP_NAMENODE_OPTS="-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.password.file=$HADOOP_CONF_
DIR/jmxremote.password
-Dcom.sun.management.jmxremote.port=8004 $HADOOP_NAMENODE_OPTS"
```

Jmxremote.password 文件以纯文本形式列出了用户名和对应的密码，有关此文件格式的详细信息，请参见 JMX 的文档。

有了这个配置，我们可以使用 JConsole 来浏览远程节点上的 MBeans。或者使用许多 JMX 工具之一来检索 MBean 属性值。下面的例子使用 jmxquery 命令行工具来检索副本不足的块的数量(Nagios 插件可从 *http://code.google.com/p/jmxquery/*下载)：

```
% ./check_jmx -U service:jmx:rmi:///jndi/rmi://namenode-
host:8004/jmxrmi -O \
hadoop:service=NameNode,name=FSNamesystemState -A
UnderReplicatedBlocks \
-w 100 -c 1000 -username monitorRole -password secret
JMX OK - UnderReplicatedBlocks is 0
```

这个命令发布了一个 JMX RMI 与主机 *namenode-host* 在端口 8004 上的连接，并授权使用给定的用户名和密码。它读取对象“hadoop:service=NameNode,name=FSNamesystemState”的属性 UnderReplicateBlocks，并在终端上打印出它的值[②]。-w 和-c 选项定义了值的警告和重要级别：这些值通常在集群使用一段时间后再决定才比较合适。

在监控 Hadoop 集群的时候，在使用 Ganglia 的同时结合使用 Nagios 之类的警告系统是很常见的。Gagnlia 擅长高效收集大量度量信息并将它们制成图表，而 Nagios 及其类似系统擅长处理小部分度量信息，在达到一个临界值时发出警告。

---

① *http://java.sun.com/javase/6/docs/technotes/guides/management/agent.html*。

② 用 JConsole 找到想检测的 MBeans 的对象名也很方便。注意，数据节点度量的 MBeans 目前包含一个随机的标识符，因此除了 ad hoc 方式，很难以其他任何方式对它们进行监控。详情参见 *https://issues.apache.org/jira/browse/HADOOP-4482*。

10.3 维护

10.3.1 例行管理程序

元数据备份

如果名称节点的持久性元数据丢失或者被损坏，那么整个文件系统将会显示为不可用状态，所以备份这些文件相当重要。你可以保留多个不同时间的备份(一小时、一天、一星期或一个月)来防止文件崩溃，各个副本本身的，或者名称节点上运行的活动文件之间的。

一种进行备份的直接方式是写一个脚本，定期将第二名称节点的 *previous.checkpoint* 子文件夹(在 fs.checkpoint.dir 属性定义的文件夹下) 储存到另一个位置。脚本应该对副本的完整性进行额外的测试。具体做法是启动一个本地的名称节点守护进程，然后验证它是否已经将 *fsimage* 和 *edits* 文件成功读取到内存中(例如，通过浏览名称节点日志的成功信息)[①]。

数据备份

虽然 HDFS 被设计成能够可靠地储存数据，但正如在任何一个存储系统中一样，数据丢失在所难免，因此备份策略是必不可少的。由于 Hadoop 可以储存大量的数据，因此决定备份哪些数据，将备份存储在何处是一个挑战。这里的关键在于确定数据的优先级。优先级最高的是那些无法重新生成的数据和对业务很关键的数据，优先级低的数据，则是那些很容易重新生成的或因其商业价值有限而基本上属于用完即扔的数据，这类数据则可以选择不备份。

注意：千万不要错误地认为 HDFS 的副本可以替代备份。HDFS 中的 bug 可以引起副本的丢失。硬盘出错也会导致副本丢失。虽然 Hadoop 被特别设计为硬件出错不会导致数据丢失，但是数据丢失的可能性依然存在，特别是和软件 bug 或者人为失误综合在一起时。

对 HDFS 用户目录设置策略是很常见的。例如，可以设定空间配额分配和每晚备份的机制。无论策略是什么，都要确保用户知道它的内容，使其知道自己的预期。

① Hadoop0.21.0 自带离线映像查看器(Offline Image Viewer)，可以用来检查映像文件的完整性。

因为 distcp 工具可以并行复制文件，所以对于被分到其他 HDFS 集群（最好运行在不同版本的软件上，以预防由 HDFS bug 引起的丢失错误)或者其他 Hadoop 文件系统(例如 S3 或者 KFS)，它是一个理想的工具。还有一种办法是，可以使用一个完全不同的储存系统来保存备份，其中一种方法是从 HDFS 导出数据，详情参见“Hadoop 文件系统”小节。

文件系统检查(fsck)

建议在整个文件系统上定期(例如，每天)运行 HDFS 的 fsck 工具来查看丢失或者错误的块。详情参见“文件系统检查(fsck)”小节。

文件系统均衡器

文件系统均衡器定期运行均衡器工具来保持文件系统的数据节点的平衡。

10.3.2 委托节点和撤消节点

作为 Hadoop 集群的管理员，可能随时需要添加或者移除节点。例如，要增加集群的储存容量，需要运行新的节点。相反，有时可能有希望缩小集群等类似的想法，此时可以撤消某节点。有时，如果一个节点由于频繁发生错误或者运行非常缓慢，只能选择撤消使用它。节点一般同时运行数据节点和 tasktracker，它们都可以被委托或者撤消。

委托新的节点

虽然委托一个新的节点就像配置 *hdfs-site.xml* 指向名称节点和配置 *mapred-site.xml* 文件指向 jobtracker 进而启动数据节点和 jobtracker 守护进程一样简单，但我们最好能有一个授权节点列表。

允许任何机器都可以连接到名称节点并充当数据节点是有潜在的安全隐患，因为此类机器也许能够获得访问未授权数据的权限。另外，由于此类机器并非真正的数据节点，所以它不在你的控制之下，并且任何时候都可能停止运行，从而可能造成数据的丢失。(想像一下，如果有大量这样的节点接入，且块数据只在外来的节点上，情况会是怎样？)由于错误配置的缘故，即使在防火墙内部，这样的情景也是有风险的，因此，数据节点(和 tasktracker)应该在集群上进行明确管理。

被允许连接到名称节点的数据节点，是在 `dfs.hosts` 属性指定的文件中定义的。该文件储存在名称节点的本地文件系统上，包含有每个数据节点网络地址的行(如

数据节点所报告的——可以通过名称节点的网页用户界面来查看)。如果需要为数据节点定义多个网络地址，将它们放在一行，中间用空格来分隔即可。

类似地，可以连接到 jobtracker 的 tasktracker，是在 `mapred.hosts` 属性指定的这个文件中定义的。在大多数情况下，有一个共享文件，名为 *include file*，它被 dfs.hosts 和 `mapred.hosts` 两者引用，因为在集群上的节点会同时运行数据节点和 tasktracker 守护进程。

注意：`dfs.hosts` 和 `mapred.hosts` 属性定义的文件和 *slaves* 文件不同。前者被数据节点和 jobtracker 用来决定哪个工作者节点可以连接。*slaves* 文件被 Hadoop 控制执行脚本用来执行集群范围的操作，例如集群的重启。它从来不会被 Hadoop 守护进程使用。

要想在集群中添加新的节点，执行以下步骤即可。

1. 将新节点的网络地址加入 include 文件。
2. 使用以下命令更新带有新许可数据节点集的名称节点：

```
% hadoop dfsadmin -refreshNodes
```

3. 更新带有新节点的 *slaves* 文件，以便 Hadoop 控制脚本执行的后续操作，都包含这些新节点；
4. 启动新的数据节点；
5. 重新启动 MapReduce 集群；[①]
6. 检查在网页用户界面中的新的数据节点和 tasktracker。

HDFS 不会将块数据从旧的数据节点移动到新的数据节点来平衡集群。

要想达到集群均衡，你可以运行均衡器，详情请参见第 10 章。

撤消旧节点

虽然 HDFS 被设计用于容纳数据节点的错误，但这并不意味终止数据节点时不会有连带的副作用。例如在三级复制中，同时关闭三个在不同机架上的数据节点会使数据很容易丢失。撤消数据节点的方法是，向名称节点通知希望撤消的节点，使其能在数据节点关闭之前将块复制到其他数据节点。

有了 tasktracker，Hadoop 的容错性更高了。如果关闭正在运行任务的 tasktracker，

① 在写此书时，并没有刷新在 jobtracker 中许可的节点集的命令。可以设置 `mapred.jobtracker.restart.recover` 属性为 true 让 jobtracker 在重启后，恢复正在运行的作业。

jobtracker 会意识到错误并将任务重新分配到另一个 tasktracker。

撤消节点的过程由排除文件(exclude file)来控制。对于 HDFS，通过 dfs.hosts.exclude 属性来设置；对于 MapReduce，可以设置 mapred.hosts.exclude 属性。通常情况下，这些属性引用到同一个文件。Exclude 文件罗列了不允许连接到集群中的节点。

tasktracker 是否可以连接到 jobtracker，其规则十分简单：只有 include 文件中包含，但 exclude 文件不包含时，tasktracker 才可以连接到 jobtracker。没有定义的或空的 include 文件意味着所有节点都在 include 文件中。

对于 HDFS，规则则有一些不同。如果一个数据节点同时在 include 和 exclude 文件中，那么它可以连接，但只能撤消。表 10-4 总结了在 include 和 exclude 存放数据节点的情况。对于 tasktracker，一个未定义或者空的 include 文件意味着所有的节点都包含在其中。

表 10-4：HDFS 的 include 和 exclude 文件的优先级别

include 文件中出现该节点	exclude 文件中出现该节点	解释
否	否	节点可以连接
否	是	节点不可以连接
是	否	节点可以连接
是	是	节点可以连接和被撤消

要想从集群中移除节点，执行以下步骤即可。

1. 将拟撤消的节点之网络地址增加到 exclude 文件中。请不要在这时更新 include 文件。
2. 重新启动 MapReduce 集群来终止拟撤消的节点的 tasktracker。
3. 用以下命令更新具有新的许可的数据节点集的名称节点：

```
% hadoop dfsadmin -refreshNodes
```

4. 进入网页用户界面，然后检查拟撤消的数据节点的管理状态是否变为“撤消中”(Decommission In Progress)。然后把数据块复制到集群中的其他数据节点。
5. 所有数据节点报告其状态为“已撤消”(Decommission)后，所有数据块都被复制。此时便可以关闭已撤消的节点。
6. 从 include 文件中移除节点，然后运行以下命令：

```
% hadoop dfsadmin -refreshNodes
```

7. 从 *slaves* 文件中移除文件。

10.3.3 升级

升级 HDFS 和 MapReduce 集群需要一个周密的计划，我们主要考虑 HDFS 的升级。如果文件系统的布局改变，那么升级时会自动将文件系统的数据和元数据迁移到与新版本一致的格式。由于任何涉及数据迁移的操作都可能导致数据丢失，所以必须保证数据和元数据都有备份(详见“例行管理程序”)。

迁移计划中还应该包括在小型测试集群上进行试运行那些能承受其丢失的数据备份。试运行能让你熟悉该过程，根据集群和工具集的特定配置进行，以便在正式的集群上运行升级的之前解决所有的问题。在测试集群的同时，可以进行客户端的升级测试。

> **版本的兼容性**
>
> 所有 1.0 之前版本的 Hadoop 组件都有非常严格的版本兼容性要求。只有来自相同发布版本的组件才能保证相互的兼容性，这意味着整个系统——从守护进程到客户端——都需要同时更新。这需要集群有一段停机时间。
>
> 1.0 版本的 Hadoop 可以放松这些要求，例如旧的客户端可以与新的服务器交流(拥有相同的主发布号的情况下)。在后期的发布版本中，支持回滚升级，允许集群守护进程分阶段的升级，以便在集群更新期间客户端仍可以运行。

如果文件系统布局不改变，集群升级将非常简单：在集群上安装新的 HDFS 和 MapReduce 版本(同时也要在客户端安装)，关闭旧的守护进程，升级配置文件，然后启动新的守护进程，客户端使用新的库。这个过程是可逆的，因此升级后的版本回滚到以前版本同样也很简单。

每次成功升级后，应该执行一系列的最终清除步骤。

- 从集群上移除旧的安装和配置文件。
- 修复代码和配置中的每个错误警告。

HDFS 数据和元数据升级

如果使使用前述方法升级到新的 HDFS 版本，并且是一个不同的布局版本，名称节点会拒绝运行。它的日志中将产生以下类似信息：

```
File system image contains an old layout version -16.
An upgrade to version -18 is required.
Please restart NameNode with -upgrade option.
```

要想找出是否需要升级文件系统，最可靠的方法是在一个测试集群上进行试运行。

HDFS 升级将复制以前版本的元数据和数据。升级并不需要两倍的集群储存空间，因为数据节点使用硬连接来保留对同一个数据块的两个引用(当前版本和以前的版本)。这样一来，就可以在需要的时候轻松回滚到以前版本的文件系统。必须认识到一点，在已升级的系统上，对数据的任何改变都会在回滚结束后消失。

只能保留前一个版本的文件系统：不可以回滚多个版本。因此，要执行另一个对 HDFS 和元数据的升级，需要删除以前的版本，这个过程叫 *finalizing the upgrade*(确定更新)。一旦更新被确定，就不能再回滚到以前的版本。总之，可以忽略在升级期间的一些发布(例如，可以从版本 0.18.3 升级到 0.20.0 而不必先升级到 0.19.x)，但是在一些情况下，可能必须先升级中间的发布版本。发布版本的说明会说明是否要求中间版本。

只有健康的文件系统才可以升级。在运行升级之前，必须进行一个全面的 fsck。要想防止意外，可以保留包含系统中的所有文件和块的列表的 fsck 输出备份。这样一来，便可以在升级后将运行的输出与之进行对比。

在运行升级之前删除临时文件，这些文件包括 HDFS 上的 MapReduce 系统目录中的文件和本地临时文件。

完成这些准备工作之后，即可进行下述集群升级和文件系统的迁移过程。

1. 确保进行新的升级之前，已完成之前的任何升级。
2. 关闭 MapReduce，终止 tasktracker 上的所有孤儿任务进程。
3. 关闭 HDFS 并备份名称节点目录。
4. 在集群和客户端上安装新版本的 Hadoop HDFS 和 MapReduce。
5. 使用-upgrade 选项启动 HDFS。
6. 等待直到更新完成。
7. 在 HDFS 上执行健康检查。
8. 启动 MapReduce。
9. 回滚或者确定升级(可选)。

运行升级程序时，最好能从 PATH 环境变量中删除 Hadoop 脚本。这将让你确定在运行哪个版本的脚本。为新的安装目录定义两个环境变量是很方便的；在以下的指令中已经定义了 `OLD_HADOOP_INSTALL` 和 `NEW_HADOOP_INSTALL`。

要想执行升级，应运行下面的命令(这是升级步骤的第 5 步)：

```
% $NEW_HADOOP_INSTALL/bin/start-dfs.sh -upgrade
```

名称节点升级它的元数据，将以前的版本放入新建目录 *previous* 中：

```
${dfs.name.dir}/current/VERSION
                       /edits
                       /fsimage
                       /fstime
               /previous/VERSION
               /edits
               /fsimage
               /fstime
```

同样，数据节点升级它的储存目录，将旧的拷贝保存在 *previous* 的目录中。

升级过程并不是瞬间就能完成的，可以使用 *dfsadmin* 来检查升级的进度(升级中的事件同样放在守护进程的日志文件中；第 6 步)：

```
% $NEW_HADOOP_INSTALL/bin/hadoop dfsadmin -upgradeProgress status
Upgrade for version -18 has been completed.
Upgrade is not finalized.
```

这表明升级已经完成。在这个阶段，必须在文件系统上运行一些健康检查(第 7 步)(使用 *fsck* 来检查文件和块，这是基本的文件操作)。在运行一些检查(只读模式)时，可以让 HDFS 进入安全模式，以防止其他检查对它进行更改。

在完成更新之前，如果发现新的版本无法正确运行，可以选择将其回滚到以前的版本(步骤 9)。

注意： 回滚将文件系统状态恢复到运行升级之前的状态，同时所有的更改都被取消。换言之，它将回滚到前一个文件系统的状态，而不是降到文件系统过去的某个版本。

首先关闭新的守护进程：

```
% $NEW_HADOOP_INSTALL/bin/stop-dfs.sh
```

然后用-rollback 选项启动旧版本的 HDFS：

```
% $OLD_HADOOP_INSTALL/bin/start-dfs.sh -rollback
```

这个命令会使名称节点和数据节点用以前的副本替换它们当前储存目录下的内容。文件系统返回到原始状态。

确定升级(可选)。如果对新版本的 HDFS 感到满意，可以确定升级(第 9 步)，删除以前的储存目录。

注意：在升级确定之后，将无法回滚到前面的版本。

需要执行以下步骤之后，才能进行另一次升级：

```
% $NEW_HADOOP_INSTALL/bin/hadoop dfsadmin -finalizeUpgrade
% $NEW_HADOOP_INSTALL/bin/hadoop dfsadmin -upgradeProgress status
There are no upgrades in progress.
```

至此，HDFS 便完全升级到新的版本。

第 11 章

Pig 简介

Pig 提出了处理大型数据集的抽象层次。在 MapReduce 下，有 map 和 reduce 两个函数，并且找到匹配数据处理的模式往往需要很多 MapReduce 的步骤，这是非常复杂的。有了 Pig，数据结构变得更加丰富，通常会有多值和嵌套的情况，可以应用的数据之间的转换设置就会更强大，例如，包括连接在内的一些转换在 MapReduce 中就难以实现。

Pig 由以下两部分组成。

- 表达数据流的语言，称为 Pig Latin。
- 运行 Pig Latin 程序的执行环境。目前有两种环境：在单个 JVM 本地执行和在 Hadoop 集群上分布执行。

Pig Latin 程序由一系列操作或者转换组成，用于将输入数据生成输出。从总体上看，这些操作描述了一个数据流，Pig 执行环境将这个数据流转化成可执行的语句然后运行。实际上，Pig 将转换变成一系列的 MapReduce 作业，它使你能够专注于数据而不是执行本质，但作为程序员，大多不知道这一点。

Pig 是一种用于探索大型数据集的脚本语言。MapReduce 的缺点之一是开发周期很长。编写 mapper 和 reducer，编译和打包代码，提交作业和结果检索是一个耗时的工作，即使有了可以除去编译和打包这一步的流，仍旧耗费大量精力。Pig 的优点就是它能够通过从控制台发出六条 Pig Latin 的方法简单处理数 TB 的数据。由于 Pig 提供了几条用于反复检查程序中数据结构的命令，所以它非常支持程序员编写的查询语句。它甚至可以在输入数据具有代表性的子集上运行一个样本，因此在将其运用于整个数据集之前，可以看处理过程是否有误。

Pig 被设计成是可扩展的。处理路径中所有部分几乎都是可定制的：装载、存储、

过滤、分组、排序和链接都可以换用用户定义函数(UDF)。这些功能在 Pig 的内嵌数据模型上工作，所以它们可以与 Pig 的平台紧密结合在一起。另一个好处是与为 MapReduce 程序开发的库相比，UDF 可重用。

Pig 不适合所有的数据处理任务，但是，就像 MapReduce 的一样，它是专门用于数据的批处理。如果想在一个巨大的数据集中做一个只涉及小部分数据的查询，那么 Pig 并不适合，因为它会扫描整个或者很大一部分数据集。

Pig 也不执行 MapReduce 中的程序。这并不奇怪，因为 Pig 是一个 MapReduce 作业查询的编译系统，因此，不可避免地会带来一些系统开销。然而，值得高兴的是，在 Pig 团队的优化策略能降低这个开销而不需要更改 Pig 查询。

11.1 安装和运行 Pig

Pig 运行的是一个客户端应用程序。即使想在 Hadoop 集群上运行 Pig，也不需要在集群上安装其他东西：Pig 发起作业并与你工作站上的 HDFS(或其他 Hadoop 的文件系统)进行交互。

安装过程很简单。Java 6 是必不可少的(Windows 环境下，需要 Cygwin)。从 *http://hadoop.apache.org/pig/releases.html* 下载稳定版本，并在工作站合适的地方解压压缩包：

```
% tar xzf pig-x.y.z.tar.gz
```

在命令行路径上增加 Pig 的二进制目录也很方便。例如：

```
% export PIG_INSTALL=/home/tom/pig-x.y.z
% export PATH=$PATH:$PIG_INSTALL/bin
```

还需要设置指向一个合适的 Java 安装的 `JAVA_HOME` 环境变量。

尝试输入 `pig -help` 获取使用说明。

11.1.1 执行类型

Pig 有两个执行类型或模式：本地模式和 Hadoop 模式。

本地模式

在本地模式中，Pig 在一个单一的 JVM 上运行并且访问本地文件系统。这种模式

只适合小数据集或 Pig 测试使用。本地模式不使用 Hadoop。特别是它不使用 Hadoop 的本地作业执行器，而是将查询语句转化成实际计划自己去执行。

执行类型被设置为-x 或-exectype 选项。运行在本地模式时要设置本地选项：

```
% pig -x local
grunt>
```

这将启动 Grunt，即 Pig 交互式 shell，这将在后面进行详细解释。

在 Hadoop 模式中，Pig 会将查询转化成 MapReduce 的作业并在 Hadoop 集群上运行它们。集群可能是虚拟的或完全的分布式集群。Hadoop 模式(一个完全分布式集群)就是在大型数据集上运行 Pig 时所使用的东西。

若要使用 Hadoop 模式，需要告诉 Pig 所使用的 Hadoop 的版本以及集群运行的地点。Pig 版本将只针对 Hadoop 的特定版本。例如，Pig0.2.0 只能在 Hadoop 的 0.17.x 或 0.18.x 版本上运行。环境变量 PIG_HADOOP_VERSION 是用来告诉 Pig 它连接的 Hadoop 的版本。例如，下面所列的版本允许 Pig 连接到任何版本为 0.18.x 的 Hadoop：

```
% export PIG_HADOOP_VERSION=18
```

接下来，需要将 Pig 指向该群集的名称节点和 jobtracker。如果已经有一个 Hadoop 的确定的 fs.default.name 和 mapred.job.tracker 的网站文件(或文件)，只需将 Hadoop 的配置目录添加到 Pig 的类路径中：

```
% export PIG_CLASSPATH=$HADOOP_INSTALL/conf/
```

或者，可以在 Pig 的 conf 目录(这个目录可能也需要创建)中创建一个 pig.properties 文件，其中规定这两个属性。下面是一个伪分布式设置的例子：

```
fs.default.name=hdfs://localhost/
mapred.job.tracker=localhost:8021
```

一旦配置 Pig 连接到 Hadoop 的集群，便可以启动 Pig，设置-x 选项为 MapReduce，或者完全省略它，因为 Hadoop 的模式是默认的：

```
% pig
2009-03-29 21:22:20,489 [main] INFO
org.apache.pig.backend.hadoop.executionengine.
HExecutionEngine - Connecting to hadoop file system at:
hdfs://localhost/
2009-03-29 21:22:20,760 [main] INFO
org.apache.pig.backend.hadoop.executionengine.
```

```
HExecutionEngine - Connecting to map-reduce job tracker at:
localhost:8021
grunt>
```

如输出结果所示，Pig 会报告它已经连接到的文件系统和 jobtracker。

11.1.2 运行 Pig 程序

一共有三种执行 Pig 项目的方式，所有这些工作都适用于在本地和 Hadoop 的模式。

Script

Pig 可以运行一个包含 Pig 命令的脚本文件。例如，Pig script.pig 在本地文件 `script.pig` 上运行命令。另外，对于很短的脚本，可以使用`-e` 选项在命令行运行指定为字符串类型的脚本。

Grunt

Grunt 是一个运行 Pig 命令的交互式 shell。当 Pig 没有指定文件运行或者- e 选项不使用用时，Grunt 会启动。在 Grunt 中也可以运行或者执行 Pig 脚本。

Embedded

可以在 Java 中运行 Pig 程序，就像可以在 Java 使用 JDBC 运行 SQL 程序。详情可从 Pig wiki 的 *http://wiki.apache.org/pig/EmbeddedPig* 获悉。

11.1.3 Grunt

Grunt 有一个行编辑功能，它就像在 GNU Readline(用于在 Bash shell 和许多其他命令行的应用程序)中可以找到的编辑功能。例如，按 Ctrl-E 组合键，将光标移动到该行的末尾。Grunt 也记得命令历史等，可以使用 Ctrl- P 或按 Ctrl - N(上一步或下一步)或者向上或向下键将光标键撤回在历史缓存中的命令。

另一个便利的特点是 Grunt 自动补全(也称自动完成)的机制，按 Tab 键时，自动补全机制将尝试完成 Pig Latin 的关键字和函数。例如，考虑下面这个不完整的行命令：

```
grunt> a = foreach b ge
```

这时按下 Tab 键，ge 会自动扩展为 Pig Latin 的关键词 generate：

```
grunt> a = foreach b generate
```

可以通过创建一个名为 autocomplete 的文件定制一些完全单词，并将它放在 Pig 的类路径中(如 Pig 的安装目录中的 conf 目录)或者在调用 Grunt 的目录中的这个文

件。该文件应该每行有一个单词，单词不能包含任何空格。匹配是区分大小写的。添加一些常用文件路径(至 autocomplete)将会很有用(特别是因为 Pig 不执行文件名自动补全)或任何你创建的用户定义函数名。

可以用 help 命令获得命令的列表。运行完 Grunt，可以用 quit 命令退出。

11.1.4 Pig Latin 编辑器

PigPen 是一个 Eclipse 插件，为开发 Pig 程序提供了环境。它包括一个 Pig 的脚本文本编辑器、一个实例发生器(相当于 ILLUSTRATE 命令)以及一个在 Hadoop 集群上运行脚本的按钮。还有一个图形操作窗口，以图形的形式显示了可视化数据流的脚本。完整的安装和使用说明，请参阅 Pig 维基页面，网址为 *http://wiki.apache.org/pig/PigPen*。

还有其他编辑器含有 Pig Latin 语法的结构，包括 Vim 和 Text-Mate。详情可见 Pig 维基。

11.2 实例

让我们看一个简单的例子，通过用 Pig Latin 为天气温数据集编写程序来计算一年中的最高记录(就像第 2 章使用 MapReduce 那样)。完整的程序只有几行代码：

```
-- 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 == 0 OR quality == 1 OR quality == 4 OR quality == 5
    OR quality == 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
  MAX(filtered_records.temperature);
DUMP max_temp;
```

为了了解这是怎么回事，我们将使用 pig 的 Grunt 解释程序，这使我们能够进入代码行并且与程序交互以便了解它在干什么。在本地模式启动 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);
```

为简单起见，程序假定输入的是制表符分隔的文本，每一行都有 chararray、temperature 和 quality 字段。(后面可以看到，其实对于它接受的输入格式，Pig 比这更灵活。)此命令行描述了我们要处理的输入数据。year:chararray 符号描述字段的名称和类型；chararray 像一个 Java 字符串，一个 int 如同一个 Java 整型。加载操作有 URI(通用资源标识符)参数，这里我们只使用一个本地文件，但我们可以参考 HDFS 中的 URI。AS 子句(这是可选的)给出字段的名称，以方便在随后的声明中提到它们。

LOAD 操作符的结果，实际上任何 Pig Latin 操作符的结果，都是一个关系，这个关系只是一系列元组。元组就像一行在数据库表中的数据，包含多个以特定顺序排列的字段。在这个例子中，LOAD 函数产生一系列(年、气温、质量)显示输入文件的元组。以下列出的关系，一行为一个元组，元组在括号中是以逗号代表分隔符：

```
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
```

关系可以被赋予一个名称或别名，这样它们就可以被引用。此处，关系被赋予别名:records。可以使用 DUMP 操作检查其内容：

```
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

也可以在关系上使用 DESCRIBE 操作看到一个关系的结构，即关系的模式：

```
grunt> DESCRIBE records;
records: {year: chararray,temperature: int,quality: int}
```

这表明，记录有 chararray、temperature 和 quality 三个字段，这是我们在 AS 子句中给予的名字。在 AS 子句中也给予了字段类型。我们以后应在 Pig 中更详细的检查。

第二条语句删除那些缺气温(显示 9999 值)或者质量检查读取到不令人满意的记录。对于这个小数据集，没有记录被过滤：

```
grunt> filtered_records = FILTER records BY temperature != 9999 AND
```

```
>> (quality == 0 OR quality == 1 OR quality == 4 OR quality ==
5 OR quality == 9);
grunt> DUMP filtered_records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

第三个语句使用 GROUP 函数，按照 year 字段将记录关系组合起来。我们用 DUMP 看一下会产生什么：

```
grunt> grouped_records = GROUP filtered_records BY year;
grunt> DUMP grouped_records;
(1949,{(1949,111,1),(1949,78,1)})
(1950,{(1950,0,1),(1950,22,1),(1950,-11,1)})
```

我们现在有两行记录或元组，其中每个都是给每个年份的输入数据。每个元组的第一个字段是按照年份元组的字段，第二个字段是那一年的元组的包。一个包是一个没有顺序的元组的集合，在 Pig Latin 中由大括号表示。

通过这种方式分组数据，我们创建了一个每年的行，所以现在剩下的工作就是找到在每个包的元组的最高气温。在此之前，我们要了解 grouped_records 的关系结构：

```
grunt> DESCRIBE grouped_records;
grouped_records: {group: chararray,filtered_records: {year:
chararray,temperature: int,quality: int}}
```

这表明，Pig 将 group 的别名给予正在分组的字段，而且第二个字段与正在被分组的 filtered_records 关系有一样的结构。有了这些信息，我们可以尝试进行转变：

```
grunt> max_temp = FOREACH grouped_records GENERATE group,
>> MAX(filtered_records.temperature);
```

用 FOREACH 处理每一行生成派生行，并用一个 GENERATE 子句在每一个派生行中定义字段。在这个例子中，第一个字段只是年份组。第二个字段稍微复杂一些。在 filtered_records 中气温指的是在 grouped_records 中的 filtered_records 包中的 temperature 字段。MAX 是一个计算包中字段中最大值的内置函数。在这个例子中，它计算在每个 filtered_records 包里字段中的最高气温。让我们检查结果：

```
grunt> DUMP max_temp;
(1949,111)
```

```
(1950,22)
```

由此成功计算每年的最高气温。

生成例子

在这个例子中，我们只用带有一组少量行的小数据集样本，使其便于理解数据流以及有助于调试。创建一个 reduce 数据集是一门艺术，因为理想状态下，数据需要丰富到足以涵盖所有练习的查询(属性的完整性)情况，但是又要小到很容易被程序员理解(属性的简洁性)。使用随机样本一般并不奏效，因为连接和过滤操作倾向于删除所有随机数据，留下一个空的结果，这并不足以说明一般流程。

结合 ILLUSTRATE 操作，Pig 为生成相当完整和简明的数据集提供了工具。虽然不能为所有查询的例子(例如，它不支持 LIMIT、SPLIT 或嵌套 foreach 语句)，但它可以产生许多查询的有用例子。ILLUSTRATE 仅适用于关系只有一个模式时。

以下 ILLUSTRATE 的运行输出(为适应页面大小，格式稍有调整)：

```
grunt> ILLUSTRATE max_temp;
-------------------------------------------------------------------------------
| records      | year: bytearray | temperature: bytearray | quality: bytearray |
-------------------------------------------------------------------------------
|              | 1949            | 9999                   | 1                  |
|              | 1949            | 111                    | 1                  |
|              | 1949            | 78                     | 1                  |
-------------------------------------------------------------------------------
-------------------------------------------------------------------
| records      | year: chararray | temperature: int | quality: int |
-------------------------------------------------------------------
|              | 1949            | 9999             | 1            |
|              | 1949            | 111              | 1            |
|              | 1949            | 78               | 1            |
-------------------------------------------------------------------
----------------------------------------------------------------------------
| filtered_records      | year: chararray | temperature: int | quality: int |
----------------------------------------------------------------------------
|                       | 1949            | 111              | 1            |
|                       | 1949            | 78               | 1            |
----------------------------------------------------------------------------
------------------------------------------------------------------------------------
| grouped_records     | group: chararray | filtered_records: bag({year: chararray, |
                                             temperature: int,quality: int})       |
------------------------------------------------------------------------------------
|                     | 1949             | {(1949, 111, 1), (1949, 78, 1)}         |
------------------------------------------------------------------------------------
-------------------------------------------
| max_temp     | group: chararray | int   |
-------------------------------------------
|              | 1949             | 111   |
```

请注意，Pig 使用了一些原始数据(重要的是生成的数据集的真实性)，而且 Pig 创造一些新的数据。它注意到正在查询的特殊值 9999 并创建一个包含此值的元组来

实现 FILTER 语句。

总之，该 ILLUSTRATE 的输出是很容易完成的，并且它有助于你理解正在查询的内容。

11.3 与数据库比较

使用过 Pig 后，Pig Latin 看起来类似于 SQL。GROUP BY 和 DESCRIBE 这样的操作加深了这种印象。但是，一般说来，两种语言之间以及 Pig 和关系数据库之间有一些差异。

最显著的区别是，Pig Latin 是一个数据流编程语言，而 SQL 是一种声明式编程语言。换句话说，Pig Latin 程序是输入关系上一步一步操作的集合，其中每一步是一个单一的变化。相反，SQL 语句是定义输出的约束条件的集合。在许多方面，Pig Latin 编程就像关系数据库的查询规划层次一样，它解决了如何将声明式语句转化为步骤的问题。

关系数据库以预定义模式在表中存储数据。Pig 对它处理的数据是较宽松的：你可以在运行时定义一个模式，但它是可选的。本质上，它可以对任何来源的元组进行操作(虽然资源需要支持并行读取，例如在多个文件中)，其中一个 UDF 用来读取其原始代表数据中的元组。最常见的表示是一个带有分隔字段的文本文件，Pig 提供了这种格式的内置加载函数。不同于传统的数据库，Pig 没有加载数据到 RDM 那样的导入过程。处理过程第一步是从文件系统中(通常是 HDFS)加载数据。[①]

区别于 SQL 操作的是固定数据结构不同，Pig 支持复杂、嵌套的数据。另外，Pig 使用 UDF 和流操作紧紧与语言和 Pig 的嵌套数据结构结合在一起的功能使得 Pig Latin 比大多数 SQL 语言更富选择性。

有的关系数据库支持在线，低延迟查询的特点，在 Pig 中并没有，如事务和索引。如前所述，Pig 不支持随机读取或在几毫秒内顺序查询。它也不支持随机写入更新数据的一小部分，所有写操作是大量的、流写入的，就像 MapReduce 一样。

Hive 是介于 Pig 和传统关系数据库的 Hadoop 的子项目。和 Pig 一样，Hive 使用 HDFS 存储数据，但除此之外，仍有一些明显的差异。它的查询语言 Hive QL 基于 SQL，而且熟悉 SQL 的人在写 Hive QL 的时候基本没有什么问题。就像关系数据库一样，Hive 根据其管理模式将所有数据存储在表中，但它可以使模式与先前在 HDFS

① 或者像 Pig 哲学所言“猪任何东西都吃。”

中已有的数据协作，因此加载是可选的。Hive 与 Pig 都有一个特点就是不支持低延迟的查询。第 14 章的“Hadoop and Hive 在 Facebook 的应用”将对于 Hive 进一步讨论。

11.4 Pig Latin

本节给出了 Pig Latin 编程语言的语法和语义的不正式描述。[1]这并不意味着将给出完整的语言的参考文献，但是它足够让你对 Pig Latin 的架构有一个深刻的了解。[2]

11.4.1 结构

Pig Latin 程序由一系列语句组成。一条语句可以被看作是一个操作或命令[3]，例如，一个 GROUP 的操作是一种类型的语句：

```
grouped_records = GROUP records BY year;
```

下面 Hadoop 文件系统的列出命令是语句的另一个例子：

```
ls/
```

就像 GROUP 语句的例子一样，语句通常以分号结束。事实上，这是一个必须以分号结束语句的例子：省略它会是一个语法错误。另一方面，ls 命令不必用分号结束。作为一般准则，在 Grunt 交互使用中，语句或命令并不需要分号终止。这个组包括交互 Hadoop 的命令以及像 DESCRIBE 一样的诊断操作。末尾加分号永远不会错，如果有疑义，最简单的方法就是加分号。

为方便阅读而分行的语句必须用分号终止：

```
records = LOAD 'input/ncdc/micro-tab/sample.txt'
  AS (year:chararray, temperature:int, quality:int);
```

Pig Latin 有两种注释的形式。双连字符是单行注释。从第一个连字符开始这一行最后都会被 Pig Latin 解释器忽略：

---

① 不要被 Pig Latin 语言游戏混淆。通过将声母移动到单词的最后并加上“ay”英语单词也被翻译成 Pig Latin。例如，“Pig”变成“ig-pay”，“Hadoop 的”变成“Adoop-hay”。

② Pig Latin 没有正式的语言定义，但有一个全面的语言指南，可以在 Pig wiki 的链接 *http://wiki.apache.org/pig/*找到。

③ 在 Pig Latin 的文件中，有时这些术语被交替使用。例如，“GROUP 命令”，“GROUP 操作”和“GROUP 语句”。

```
-- My program
DUMP A; -- What's in A?
```

C 语言风格的注释更灵活，因为它区分开始和结束的注释块，以/ *和* /标记。它们可以跨行或在一个单行嵌入：

```
/*
* 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;
```

Pig Latin 有一些关键字在语言中有特殊意义而且不能用作标识符。其中包括操作符(LOAD，ILLUSTRATE)、命令(cat，ls)、表达式(matches，matches)和函数(DIFF，MAX)，下面将涉及所有这些。

Pig Latin 混合了区分大小写的规则。操作和命令不区分大小写(为使交互更灵活)，然而，函数名是区分大小写的。

11.4.2 语句

Pig Latin 程序执行时，每句语句轮流被解析。如果有语法错误，或像没有定义其他问题时，解释器将停止并显示错误信息。解释器为每个关系操作建立一个逻辑计划，它形成了 Pig Latin 程序的核心。语句的逻辑计划到目前为止将添加到该程序的逻辑计划中，然后解释器移到下一个语句。

特别注意，在构建程序的逻辑计划的时候，数据并不会进行处理。例如，再次从第一个例子考虑 Pig Latin 程序：

```
-- 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 == 0 OR quality == 1 OR quality == 4 OR quality == 5
OR quality == 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
  MAX(filtered_records.temperature);
DUMP max_temp;
```

当 Pig Latin 解析器看到包含 LOAD 语句的第一行，它就确认这是正确的语法和语义，并把它添加到逻辑计划，但它不会加载该文件中的数据(甚至检查该文件是否存在)。事实上，将它加载到哪里？内存中？即使它进入到内存，应该怎么处理数据？也许不是所有的输入数据都需要(例如，稍后的语句会过滤它)，因此加载它是毫无意义的。问题在于，在整个流程被定义之前，刚开始的处理是没有意义的。同样，Pig 使 GROUP 以及 `FOREACH ... GENERATE` 等语句生效，并将它们添加到逻辑时并没有执行它们。Pig 开始处理的语句是 DUMP 语句(STORE 语句也会触发处理)。在这时，逻辑计划编译成一个物理计划并执行。

注意：可以通过使用关系上的 EXPLAIN 命令(例如 `EXPLAIN max_temp`)看到由 Pig 创建的逻辑和物理计划上的关系。在 MapReduce 的模式，EXPLAIN 也会显示 MapReduce 的计划，该计划显示了物理操作如何分组为 MapReduce 任务。这是一个很好的方式了解 Pig 为你的查询运行了多少 MapReduce 任务。

表 11-1 列出了 Pig 中逻辑计划的关系操作。每个操作的详细描述可参见“数据处理操作”小节。

表 11-1：Pig Latin 关系操作

类别	操作	描述
加载和存储	LOAD	从文件系统或存储设备中加载数据到关系
	STORE	将关系保存到文件系统或其他存储设备中
	DUMP	在控制台打印关系
过滤	FILTER	从关系中删除不需要的行
	DISTINCT	从关系中删除重复行
	FOREACH...GENERATE	从关系中添加或删除字段
	STREAM	用外部程序改变关系
分组和连接	JOIN	连接两个或多个关系
	COGROUP	在两个或多个关系中为数据分组
	GROUP	在单个关系中为数据分组
	CROSS	创建两个或多个关系的交叉产品
排序	ORDER	按照一个或多个字段为关系排序
	LIMIT	限制一个关系的最大元组数
合并和分割	UNION	将两个或多个关系合并成一个
	SPLIT	将一个关系分割成一个或多个关系

还有其他类型的语句不会添加到逻辑计划中。例如，诊断操作、DESCRIBE、

EXPLAIN 和 ILLUSTRATE 等操作为了调试的目的被提供给用户与逻辑计划交互(见表 11-2)。DUMP 也是一种诊断操作，因为它只用来交互调试小的结果集，或者结合 LIMIT，从较大的关系中取得一些内容。STORE 语句应该被用在大于几行的输出结果，因为它将结果写入文件而不是控制台。

表 11-2：Pig Latin 诊断操作

操作	描述
DESCRIBE	显示关系模式
EXPLAIN	显示逻辑和物理规划
ILLUSTRATE	通过生成的输入子集显示逻辑规划的执行样本

Pig Latin 提供两种语句 REGISTER 和 DEFINE 以使人们有可能将用户定义的函数合并成 Pig 的脚本(见表 11-3)。

表 11-3：Pig Latin UDF 语句

语句	描述
REQISTER	Pig 运行时注册一个 JAR 文件
DEFINE	为 UDF、流脚本或者命令创建一个别名

由于它们不处理关系，命令不会添加到逻辑计划；反之，它们将立即执行。Pig 提供与 Hadoop 的文件系统、MapReduce 和一些功用命令(这使得在处理 Pig 之前或之后移动数据变得很方便，详见表 11-4)。

表 11-4：Pig Latin 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	将本地文件或目录复制到 Hadoop 文件系统
	copyToLocal	将 Hadoop 文件系体内的文件或目录复制到本地文件系统
	cp	将文件或目录复制到另一个目录下
	ls	列出文件
	mkdir	创建一个新的目录
	mv	将文件或目录移动到另一个目录下

续表

类别	命令	描述
	pwd	打印当前工作目录的路径
	rm	删除文件或目录
	rmf	强制删除一个文件或目录(文件或目录不存在时不会出错)
Hadoop MapReduce	kill	杀掉一道 MapReduce 作业
应用	help	显示已有命令或选项
	quit	退出解析器
	set	设置 Pig 选项

文件系统命令可以在任何 Hadoop 文件系统中的文件和目录上操作，它们与 Hadoop 的 fs 命令非常类似(这并不奇怪，因为两个都是围绕 Hadoop 的 FileSystem 接口简单包装的)。使用哪个 Hadoop 文件系统是由 Hadoop 的核心地址文件中的 fs.default.name 属性决定的。

这些命令大多是不言自明，除了 set(用来设置控制 Pig 的行为选项)。调试选项是用于在脚本中打开或关闭调试日志记录的(还可以在启动 Pig 时使用-d 或-debug 选项控制日志级别)。

```
grunt> set debug on
```

另一个有用的选项是 job.name 选择，它给了 Pig 作业有意义的名称，使其在共享的 Hadoop 的集群上运行时更容易选出你的 Pig MapReduce 的任务。如果 Pig 正在运行一个脚本(而不是在 Grunt 上的交互式查询)，它的任务名称默认为脚本的名称上的一个值。

11.4.3 表达式

表达式是通过评估产生值的工具。表达式可以在 Pig 中用作包含关系运算的语句的一部分。Pig 有丰富的表达式，其中许多都和其他编程语言中差不多如表 11-5 所示，其中带有简单描述和实例。我们将在本章中运用这些表达式。

表 11-5：Pig Latin 表达式

类别	表达式	描述	示例
常量	Literal	常值(也可参见表 11-6)	1.0,'a'
字段(按照位置)	$n	位置为 n(基于 0)上的字段	$0

续表

类别	表达式	描述	示例
字段(按照名字)	f	字段名 f	year
投影	c.$n,c.f	在容器 c(关系、包或元组)中的字段	records.$0,records.year
map 查找	m#k	在 map m 中与键 k 相关联的值	items#'Coat'
类型转换	(t)f	将字段 f 转换成类型 t	(int)year
运算	x+y,x-y	加法，减法	$1+$2,$1-$2
	x*y,x/y	乘法，除法	$1*$2,$1/$2
	x%y	取模，x 除以 y 的余数	$1%$2
	+x,-x	正，负	+1,-1
条件	x?y:z	三元二分条件，如果 x 为真则取 y，否则取 z	quality==0?0:1
比较	x= =y, x!=y	等于，不等于	quality == 0, temperature != 9999
	x > y, x < y	大于，小于	quality > 0, quality < 10
	x> =y, x<=y	大于等于，小于等于	quality >= 1, quality <= 9
	x matches y	模式匹配正则表达式	quality matches '[01459]'
	x is null	无	temperature is null
	x is not null	有	temperature is not null
boolean	x or y	逻辑或	q == 0 or q == 1
	x and y	逻辑与	q == 0 and r == 0
	not x	逻辑负	not q matches '[01459]'
函数	Fn(f1,f2,...)	在 f1,f2 等字段上应用函数	isGood(quality)
平坦化	FLATTEN(f)	从包或元组上删除嵌套	FLATTEN(group)

11.4.4 类型

前面已介绍 Pig 中的类型，如 `int` 和 `chararray` 一些简单类型。在这里，我们将更详细地讨论 Pig 的内置类型。

Pig 有四种数值类型：`int`，`long`，`float` 和 `double`，这与 Java 对应的类型是相同

的。还有一个 ByteArray 类型是类似 Java 使用来表达二进制数据的字节数组类型以及 chararray，它就像 java.lang.String 一样表示 UTF - 16 格式的文本数据，尽管它可以用 UTF -8 格式加载或存储。Pig 没有对应 Java 的 boolean[①] byte、short 或 char 这些原始类型。这些都是很容易用 Pig 的 int 类型或字符的 chararray 表示的。

数字、文本和二进制类型是简单原子类型。Pig Latin 也有三种代表嵌套结构的复杂类型：tuple、bag 和 map。Pig Latin 的类型都列于表 11-6。

表 11-6：Pig Latin 类型

类别	表达式	描述	示例
Numeric(数值)	int	32 位有符号整型	1
	long	64 位有符号整型	1L
	float	32 位浮点型	1.0F
	double	64 位浮点型	1.0
Text(文本)	chararray	在 UTF-16 格式的字符组	‘a’
Binary(二进制)	bytearray	字节数组	不支持
Complex(复杂)	tuple	任何类型的字段序列	(1,'pomegranate')
	bag	一个无序的元组集合，可能会有重复	{(1,'pomegranate'),(2)}
	map	键/值对集合。键必须是原子的，值可以是任意类型	[1#'pomegranate']

复杂类型通常从文件中加载或用关系操作构造。然而，请注意，在 Pig Latin 程序中创建常值时使用在表 11-6 中的字面形式。使用标准 PigStorage 加载时，在文件中原始形式通常是不同的。例如，表 11-6 中的某个包，在文件中的表示可能是 {(1,pomegranate),(2)}(注意此时没有引号)，有一个适当的模式，它可以当作单行单字段的关系被加载，它的值就是此包。

map 始终是从文件上加载的，因为在 Pig 中没有任何关系的操作可以产生 map。如果需要，可以写一个 UDF 生成 map。

虽然关系和包在概念上相同(一个无序元组的集合)，实际上，Pig 对待它们略有不同。一个关系是一个顶级构造，而一个 bag 不得不被包含在一个关系中。通常不必

① 虽然没有数据的布尔类型，Pig 有一个在测试条件时表达评估真或假的概念(如在 FILTER 语句中)。但是，Pig 不允许在一个字段中存储一个布尔表达式。

担心这个，但一些限制可能会让一些初学者失败。例如，它不能根据一个包的文本创建一个关系，所以下面的语句会失败：

```
A = {(1,2),(3,4)}; -- Error
```

在这种情况下最简单的解决方法是使用 LOAD 语句从文件中加载数据。

另一个例子是，不能将一个关系看作是包一样对待或者在一个新的关系中建立字段($ 0 指向 A 的第一个字段，使用位置符号)：

```
B = A.$0;
```

相反，必须使用关系操作把一个关系 A 转化成关系 B：

```
B = FOREACH A GENERATE $0;
```

也许 Pig Latin 将来的一个版本可以消除这些矛盾并且用同样的方式对待关系和包。

11.4.5 模式

Pig 中的关系可能有关联模式，它指定关系中的字段名称和类型。我们已经看到在 LOAD 语句中 AS 子句如何将模式附加到关系中：

```
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}
```

这次我们将 year 定义为一个整数，而不是 chararray 类型，即使是从同一个文件中加载的。如果我们需要对 year 进行操作，那么整型更合适(例如，将它变成一个时间戳)，而被用来作为一个简单的标识符时，chararray 表达可能更适当。Pig 声明哪些模式的灵活度与传统的 SQL 数据库的模式相反，SQL 数据库在数据加载到系统之前声明。Pig 专门用来分析无格式的输入文件，这些文件没有关联类型信息，因此，很自然的，它比在关系数据库中更晚选择字段类型。

也可以完全省略类型声明：

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>> AS (year, temperature, quality);
grunt> DESCRIBE records;
records: {year: bytearray,temperature: bytearray,quality: bytearray}
```

但是，如果用这种方法指定这样一个模式，需要指定各个字段。也无法在不指定名

字的情况下指定一个字段的类型。另一方面，模式完全是可选的，并且可以通过不指定一个 AS 子句就可以省略模式：

```
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}
```

在这种情况下，我们在模式中指定字段名称，year、temperature 和 quality。类型默认为最普遍的类型 byteArray 的，代表一个二进制字符串。

不必指定每个字段类型，可以保留一些默认为 bytearray 的字段，就像我们在这个声明中为 year 做的一样：

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt';
grunt> DESCRIBE records;
Schema for records unknown.
```

在没有模式的关系中的字段只能通过用位置符号引用：$0 指向关系中的第一个字段，$1 指向第二个字段，以此类推。它们都默认为 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}
```

虽然可以不为字段指定类型(特别是在写查询第一阶段)，但是为字段指定类型可以提高清晰度和Pig Latin程序的效率，一般推荐这样做。

注意：将模式声明为查询的一部分，这个做法灵活，但并不适合模式重用。相同的输入数据的一组 Pig 查询通常在每个查询中重复使用相同的模式。如果查询处理的许多字段，这种重复可以成为难以维持，因为 Pig(不像 Hive)没有在查询的外部有数据关联一个模式的方法。解决这个问题的一个办法是写你自己有封装模式的加载函数。详情参见 11.4.3 节。

数据验证和 null 值

SQL 数据库加载时会强制在一个表的模式中加约束。例如，试图将一个字符串加

载到一个被声明为数字类型的一列中将会失败。在 Pig 中，如果该值不能转换为模式中声明类型的，那么它将替代为空值。让我们看看它是如何工作的，如果有这样的天气数据，有一个字符“e”代替一个整数：

```
1950 0 1
1950 22 1
1950 e 1
1949 111 1
1949 78 1
```

Pig 通过对不合法的数据产生一个 null 来处理有问题的行，显示在屏幕上时会出现值缺失(而且也可以用 STORE 来保存)：

```
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 产生一个无效的字段的警告(这里没有显示)，但不会停止处理数据。对于大型数据集，这是非常普遍已损坏的、无效的或者仅仅是意外的数据，所以对每个不可解析的记录都增加修改功能是不可行的。相反，我们可以一口气找出所有的无效数据，便于我们采取相应的行动，或许可以修改我们的程序(因为它们表明我们已经犯了一个错误)，通过过滤出来(因为数据真的是无法使用)。

```
grunt> corrupt_records = FILTER records BY temperature is null;
grunt> DUMP corrupt_records;
(1950,,1)
```

请注意 is null 操作的用法，它类似于 SQL。事实上，我们从原来的记录中将包括如标识符和无法解析的值这样更多信息包含起来，以帮助我们分析错误的数据。

可以用下面的语句输出关系中的行的数量，从而找到很多损坏的记录：

```
grunt> grouped = GROUP corrupt_records ALL;
grunt> all_grouped = FOREACH grouped GENERATE group,
COUNT(corrupt_records);
grunt> DUMP all_grouped;
(all,1L)
```

另一种有用的技术是使用 SPLIT 操作将数据分为“好”与“坏”的两个关系，从而可以分开进行分析：

```
grunt> SPLIT records INTO good_records IF temperature is not null,
>> bad_records IF temperature is null;
grunt> DUMP good_records;
(1950,0,1)
(1950,22,1)
(1949,111,1)
(1949,78,1)
grunt> DUMP bad_records;
(1950,,1)
```

让我们回到例子中去，temperature 的类型被保留为未声明，损坏的数据不容易发现，因为它没有表现为 null：

```
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 == 0 OR quality == 1 OR quality == 4 OR quality ==
5 OR quality == 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)
```

在这种情况下会发生的事情是 temperature 字段被解释为 ByteArray，所以输入加载时，损坏的字段无法被检测到。传递到 MAX 函数时，temperature 字段的类型转换为 double，因为 MAX 只能是数值类型。损坏的字段表示为 double，因此它成为一个空值，MAX 自动将其忽略。最好的方法一般是为正在加载的数据声明一个类型，并在处理之前，让它们寻找关系中缺失的或损坏的数值，有时，损坏的数据会显示出比元组小，因为有些字段缺失了。可以通过使用 SIZE 功能过滤它们，内容如下：

```
grunt> A = LOAD 'input/pig/corrupt/missing_fields';
grunt> DUMP A;
(2,Tie)
(4,Coat)
(3)
(1,Scarf)
grunt> B = FILTER A BY SIZE(*) > 1;
grunt> DUMP B;
(2,Tie)
(4,Coat)
(1,Scarf)
```

模式合并

在 Pig 中，不用为每个数据流中的每个新的关系声明一个模式。在大多数情况下，Pig 可以通过考虑输入关系的模式猜出输出的关系操作所需要的模式。

如何将模式传递到新的关系中呢？一些关系操作不改变模式，例如 LIMIT 操作符(限制关系最多含有的元组数)，它产生的关系与其操作的关系具有相同的模式。对于其他操作而言，情况比较复杂。例如，UNION 可以将两个或两个以上的关系合并为一个，并试图合并输入的关系模式。如果模式由于不同的类型或字段数而不兼容，那么该 UNION 的模式是未知的结果。

可以使用 DESCRIBE 操作寻找出适合数据流中任何关系的模式。如果想重新定义一个关系的模式，可以使用带有 AS 子句的 FOREACH ...GENERATE 操作定义输入关系中一些或者所有字段的模式。详情参见 11.5 节。

11.4.6 函数

Pig 中的函数有以下五种类型。

求值函数

取一个或多个表达式，然后返回另一个表达式的函数。例如，内置的求值函数 MAX，它返回一个包的输入的最大值。有些求值函数是聚合函数，这意味着它们在一个数据包中操作产生一个标值，MAX 就是一个聚合函数。此外，许多聚合函数是代数，这意味着该函数的结果可以增量计算。在 MapReduce 的计算中，代数函数利用 combiner 进行更有效的计算(参见 2.4.2 节)。MAX 是一个代数函数，用来计算一个值集合中位数的函数则是一个非代数函数的例子。

过滤函数

求值函数的特殊类型会返回一个逻辑布尔类的结果。顾名思义，过滤函数被用在 FILTER 操作中以消除不必要的行。它们也可以用于其它运用布尔条件的关系操作，并在一般表达式中使用布尔表达式或条件表达式。一个内置的过滤函数的例子是 `IsEmp`，用于测试包或 map 中是否包含任何内容。

comparator

它是一个在元组对上可以强加顺序的函数。可以用一个 ORDER 子句指定 comparator 来控制排列顺序。

加载函数

指定如何从外部存储设备中将数据加载到关系中的函数。

存储函数

指定如何将关系中的内容保存到外部存储设备的函数。通常加载和存储函数执行相同类型。例如，`PigStorage`，从分隔的文本文件中的加载数据，也可以用同一格式存数据。

Pig 有一个内置函数的小集合，如表 11-7 所示。

表 11-7：Pig 内置函数

类别	函数	描述
Eval	AVG	计算包中的平均值
	CONCAT	连接两个字节组或两个字符组
	COUNT	计算包中的条目数
	DIFF	计算两个包中的设置差异。如果两个参数不是包，那么若它们相同，就返回包中的两个值；否则返回空包
	MAX	计算包中条目的最大值
	SIZE	计算类型的大小。数值型的大小总是一，对于字符组，它是字符数，对于字节组来说是字节数，对容器(元组、包和map)是条目数
	SUM	计算包中条目的总数
	TOKENIZE	将一个字符串分隔成组成它的单词
Filter	IsEmpty	测试包或 map 是否为空
Load/Store	PigStorage	用一个字段分隔文本格式加载或存储关系。每一行都用一个可配置的字段分隔符被分成字段，并在元组字段中保存。没有指定存储位置时，会保存到默认存储设备中

续表

类别	函数	描述
Load/Store	BinStorage	从二进制文件中下载或存储到二进制文件中。一个内部 Pig 格式用于 Hadoop 可写对象
	BinaryStorage	从二进制文件中下载关系或将关系存储到二进制文件中，该关系包含带有 bytearray 类型的值的单个字段的元组。bytearray 值的字节用 Pig 流被逐字保存
	TextLoader	通过写 toString()元组表达存储关系，一个一行。对于调试很有用

如果需要的功能不存在，可以自己编写。不过，在这样做之前，要看一下 Piggy Bank，即在 Pig 社区存放 Pig 函数的仓库。Pig 的维基页面(网址为 *http://wiki.apache.org/pig/PiggyBank*)写明了如何浏览并获得 Piggy Bank 的函数。如果 Piggy Bank 中没有需要的函数，可以自己编写(如果它足够普遍，可以考虑将它放到 Piggy Bank 中使其他人也可以受益)。这就称为用户定义函数，或者 UDF。

11.5 用户定义函数

Pig 的设计师意识到，能够插入自定义代码是非常重要的，但关键是平凡数据处理工作。为此，他们让用户可以轻松定义和使用用户定义的函数。

11.5.1 过滤 UDF

让我们通过写一个过滤函数来过滤质量不满意的气温记录来证明。以下代码：

```
filtered_records = FILTER records BY temperature != 9999 AND
(quality == 0 OR quality == 1 OR quality == 4 OR quality == 5
OR quality == 9);
```

改成以下形式：

```
filtered_records = FILTER records BY temperature != 9999 AND
isGood(quality);
```

这完成了两件事：它使 Pig 的脚本变得更加简洁，它将逻辑封装在一个地方，这样可以很容易地在其他脚本中重用。如果只是写一个特定查询，我们可能根本不必写一个 UDF。只有一遍又一遍地做同样的处理时，才有机会重用 UDF。

UDF 是用 Java 编写的，所有过滤函数是 FilterFunc 的子类，而 FilterFunc 本身是一个 EvalFunc 的子类。我们稍后将更详细地看看 EvalFunc，但目前只需注意，在本质上，EvalFunc 如下：

```
public abstract class EvalFunc<T> {
  public abstract T exec(Tuple input) throws IOException;
}
```

EvalFunc 唯一的抽象方法是 exec()，它需要一个元组，并返回(参数化)类型 T 的一个值。输入元组的字段由传递到函数的表达式组成，在这个例子中，返回一个整数。对于 FilterFunc，T 是布尔，所以这种方法应该返回 true，因为它的元组不应被过滤掉。

为实现质量过滤功能，我们写 IsGoodQuality 类，它扩展 FilterFunc 并实现了 exec()方法。见例 11-1。元组类本质上是有相关类型的对象列表。在这里，我们只关注第一个字段(因为该函数只有一个参数)，我们使用元组类的 get()方法提取索引。该字段是一个整数，如果它不是空，我们提取它，并检查它的值是否表示一个可以良好地读取的气温，返回适当的值，true 或 fale。

例 11-1：删除读取气温质量不尽人意的过滤函数 UDF

```
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);
    }
  }
}
```

要使用新的函数，我们首先编译它，并打包到 Jar 文件中(本书的示例代码，键入 ant pig 即可)。然后通过 REGISTER 操作告诉 Pig 相关的 JAR 文件，通过将本地路径赋值给文件名(而且不用包含在引号中)：

```
grunt> REGISTER pig.jar;
```

最后我们可以调用函数：

```
grunt> filtered_records = FILTER records BY temperature != 9999 AND
>> com.hadoopbook.pig.IsGoodQuality(quality);
```

Pig 通过将函数名看作 Java 类名并试图加载该名称的类的方式来进行函数调用。(顺便提一句，这就是为什么函数名是区分大小写的：因为 Java 类名是区分大小写的。)搜索类时，Pig 使用一个类加载器，其中包含已注册的 JAR 文件。在分布式模式下运行时，Pig 将确保 JAR 文件转移到集群上。

对于本例中的 UDF，Pig 在注册了的 JAR 文件中找了一个名为 com.hadoopbook.pig.IsGoodQuality 的类。

内置函数的处理方法是相同的，只有一点区别：Pig 搜索了一组内置的包名，所以函数调用不必是一个完全符合标准的名字。举例来说，MAX 函数其实是由包中 org.apache.pig.builtin 的类 MAX 执行的。这是 Pig 查询其中一个包，所以我们可以在 Pig 程序中写 MAX 而不是 org.apache.pig.builtin.MAX。

不能注册 Pig 中的包，但我们可以通过使用 DEFINE 操作定义来缩短函数名：

```
grunt> DEFINE isGood com.hadoopbook.pig.IsGoodQuality();
grunt> filtered_records = FILTER records BY temperature != 9999
AND isGood(quality);
```

如果要多次使用相同的脚本，为函数定义是一个好主意。如果想传递参数给该 UDF 的实现类的构造函数，这也是必要的。

改变类型

quality 字段被声明为整型时才开始过滤，但如果类型的信息不存在，那么 UDF 失败！这是因为这一字段是默认类型 ByteArray 的，由 DataByteArray 类表示。由于 DataByteArray 不是一个整数，所以无法转换。

显而易见的解决方法是在 exec()方法中将字段转化为整型。然而，有更好的办法，就是去告诉 Pig 的函数需要的字段类型。EvalFunc 的 getArgToFuncMapping() 方法正是为此提供的。我们可以重写它，并告诉 Pig 第一个字段应该是一个整数：

```
@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;
}
```

此方法返回对应的是传递给 exec()方法元组的每个字段的 FuncSpec 对象。这里有一个单一的字段，我们构建一个匿名 FieldSchema(名称被传为空，由于 Pig 在类型转换时会忽略名称)。通过 Pig 的数据类型类中的 INTEGER 常量可以指定该类型。

有了补充的函数，Pig 将尝试传递给函数的参数转换为整数。如果该字段不能被转换，则该字段传递为空。如果该字段为空，在 exec()方法总是返回 false。对于这种应用，此行为是恰当的，因为我们需要对难懂的 quality 字段的记录进行过滤。

下面是最后的使用新函数的程序：

```
-- max_temp_filter_udf.pig
REGISTER pig.jar;
DEFINE isGood com.hadoopbook.pig.IsGoodQuality();
records = LOAD 'input/ncdc/micro-tab/sample.txt'
  AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
isGood(quality);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
```

```
    MAX(filtered_records.temperature);
  DUMP max_temp;
```

11.5.2 求值 UDF

编写一个求值函数比写一个过滤函数略有进步。考虑一个从字符值中修去开头和结尾的空白 UDF(见例 11-2)，就像 `java.lang.String` 中的 `trim()`方法。我们将在后面使用此 UDF。

例 11-2：去除字符值前后空白的 EvalFunc UDF

```
public class Trim extends EvalFunc<String> {

  @Override
  public String exec(Tuple input) throws IOException {
    if (input == null || input.size() == 0) {
      return null;
    }
    try {
      Object object = input.get(0);
      if (object == null) {
        return null;
      }
      return ((String) object).trim();
    } catch (ExecException e) {
      throw new IOException(e);
    }
  }

  @Override
  public List<FuncSpec> getArgToFuncMapping() throws FrontendException {
    List<FuncSpec> funcList = new ArrayList<FuncSpec>();
    funcList.add(new FuncSpec(this.getClass().getName(), new Schema(
      new Schema.FieldSchema(null, DataType.CHARARRAY))));

    return funcList;
  }
}
```

`exec()`和 `getArgToFuncMapping()`方法很简单，犹如 `IsGoodQuality` `UDF` 的那些。

写一个求值函数时，需要考虑输出模式的样子。在下面的语句中，B 模式是由 `UDF`

的函数决定的：

```
B = FOREACH A GENERATE udf($0);
```

如果 UDF 创建标量字段的元组，那么 Pig 可以通过 map 决定 B 的模式。对于诸如包、元组或 map 之类的复杂类型，需要更多的帮助，应该执行 outputSchema()方法使 Pig 获得输出模式的信息。

Trim UDF 返回一个字符串，如下所示，Pig 将它翻译成字符型：

```
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 有开头和结尾空格的字符型字段。我们通过在 A 中的第一个字段中使用 Trim 函数来从 A 中创建 B(名为 fruit)。B 的字段可以很准确地推断为字符型。

11.5.3 加载 UDF

我们将演示一个自定义加载函数，它可以将纯文本的某些列读取成字段的 Unix cut 命令。它用法如下：

```
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)
```

传给 CutLoadFunc 的字符串是列的规范，每个逗号分隔的范围定义一个字段，它在 AS 子句中被分配名称和类型。让我们检验一下 CutLoadFunc 的执行，如例 11-3 所示。

例 11-3：一个纵列范围加载元组字段的 LoadFunc UDF

```
public class CutLoadFunc extends Utf8StorageConverter implements LoadFunc {

  private static final Log LOG = LogFactory.getLog(CutLoadFunc.class);

  private static final Charset UTF8 = Charset.forName("UTF-8");
  private static final byte RECORD_DELIMITER = (byte) '\n';

  private TupleFactory tupleFactory = TupleFactory.getInstance();
  private BufferedPositionedInputStream in;
  private long end = Long.MAX_VALUE;
  private List<Range> ranges;

  public CutLoadFunc(String cutPattern) {
    ranges = Range.parse(cutPattern);
  }

  @Override
  public void bindTo(String fileName, BufferedPositionedInputStream
    in,long offset, long end) throws IOException {
    this.in = in;
    this.end = end;

    // Throw away the first (partial) record - it will be picked up
    // by anotherinstance
    if (offset != 0) {
      getNext();
    }
  }

  @Override
  public Tuple getNext() throws IOException {
    if (in == null || in.getPosition() > end) {
      return null;
    }
```

```
    String line;
    while ((line = in.readLine(UTF8, RECORD_DELIMITER)) != null) {
      Tuple tuple = tupleFactory.newTuple(ranges.size());
      for (int i = 0; i < ranges.size(); i++) {
        try {
          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)));
        } catch (ExecException e) {
          throw new IOException(e);
        }
      }
      return tuple;
    }
    return null;
  }

  @Override
  public void fieldsToRead(Schema schema) {
    // Can't use this information to optimize, so ignore it
  }

  @Override
  public Schema determineSchema(String fileName, ExecType execType,
      DataStorage storage) throws IOException {
    // Cannot determine schema in general
    return null;
  }
}
```

Pig 使用 Hadoop 执行引擎的时候，mapper 正在进行数据加载，因此，输入可以被划分为可以被 mapper 独立处理的部分是很重要的。

在 Pig 中，加载函数被要求加载输入的一部分，这一部分指定为一个字节的范围。开始和结束偏移量通常不记录在边界上(因为它们对应的是 HDFS 块)，所以在初始化时，加载函数需要在 bindTo()方法中寻找下一个记录边界的起始点。Pig 运行时会重复调用 getNext()函数，而且加载函数会从流中读取元组直到 Pig 得到超出

它所要加载的一个字节范围的输入。这时，它返回空，表明没有更多的元组要读取。

CutLoadFunc 在构建时带有一个字符串，它将为每个字段指定纵列范围。解析字符串以及建立一个内部范围对象的列表的逻辑是封装在 Range 类中包含的范围，这里并没有显示(在本书配套示例代码中可以找到)。

在 CutLoadFunc 的 bindTo()方法中，可以找到下一个记录边界。如果是在流的开头部分，那么我们就知道流第一个记录边界的位置。否则，我们调用 getNext() 函数将放弃第一个部分行，因为它会在另一个 CutLoadFunc 实例中得到充分处理，就是在 CutLoadFunc 实例的字节范围内立即处理当前字节。在某些情况下，开始的偏移量可能会落到一行的边界上，但它无法发现这种情况，所以我们总是要放弃第一行。这不是一个问题，因为在输入流的位置严格超过结束偏移量之前，我们会一直读取行。也就是说，如果结束偏移量在行的边界上时，我们将再读一行。这样，我们可以肯定，没有任何一行被加载函数忽略。(这种情形很像在第 7 章的“TextInputFormat”描述和说明的。)

getNext() 的执行作用是把输入文件的行转化为 Tuple 对象。它通过 TupleFactory 的方法做到这点，TupleFactory 是 Pig 类用来创建 Tuple 实例的。newTuple()方法会创建一个有所需字段的数量的新元组，而且这个数量正是 Range 类的数量，而且 Range 对象将通过每行的子串决定字段的位置。

如果行数少于范围要求，我们将需要思考用另外的方法。一种选择是抛出一个异常，并停止进一步的处理。如果你的应用程序无法容忍不完整或有损坏的记录，那么这种选择是恰当的。在许多情况下，最好返回一个有空字段的元组，让 Pig 脚本将不完整的数据处理为它认为合适的数据。我们在这里采取的方法是，如果范围的结束位置超过了行的结束位置时退出循环，此时，我们退出目前的字段以及元组中随后的所有字段，并将它们设置为默认值为 NULL。

运用模式

现在，让我们考虑正在被加载字段的类型。如果用户指定了一个模式，则字段需要转换成有关类型。然而，这是由 Pig 惰性执行的，因此加载器应该要一直用 DataByteArray 类型创建 bytearray 类型的元组。加载函数仍然有机会进行转换，但是，它必须为此执行一大堆转换函数：

```
public interface LoadFunc {

  // Cast methods
  public Integer bytesToInteger(byte[] b) throws IOException;
  public Long bytesToLong(byte[] b) throws IOException;
```

```
    public Float bytesToFloat(byte[] b) throws IOException;
    public Double bytesToDouble(byte[] b) throws IOException;
    public String bytesToCharArray(byte[] b) throws IOException;
    public Map<Object, Object> bytesToMap(byte[] b) throws IOException;
    public Tuple bytesToTuple(byte[] b) throws IOException;
    public DataBag bytesToBag(byte[] b) throws IOException;

    // Other methods omitted
  }
```

CutLoadFunc 不自己执行这些方法，因为它扩展了 Pig 的 Utf8Stora geConverter，它提供了标准的从 UTF-8 编码数据类型到 Pig 数据类型的转换方法。

在某些情况下，加载的函数本身可以决定模式。举例来说，如果我们加载了像 XML 或 JSON 一样的自我描述数据型的函数，我们可以通过查看数据为 Pig 自动创建一个模式。另外，加载函数可以用另外一种方法决定模式，如外部文件，或在它的构造函数中传递信息。为了支持这种情况，加载函数可以提供 determineSchema()函数的执行来返回一个模式。但请注意，如果用户在 LOAD 的 AS 子句中提供了一个模式，那么它决定的模式要比加载函数的 Schema()方法所决定的模式更占有优先权。

对于 CutLoadFunc，我们在 determineSchema()中返回空，所以只有用户提供时才会有模式。

最后，LoadFunc 有 fieldsToRead()方法，它能够让加载函数找出查询要求的列。这可以成为一个有用的列存储优化，因此，加载器只加载查询所需要的列。CutLoadFunc 没有显著的可以加载到子集的方法，因为它读取每个元组整行，所以我们忽略此信息。

带有 Slicer 的高级加载

为了进一步控制数据加载的过程，除了 LoadFunc 以外，加载函数还可以使用 Slicer 接口。Slicer 的执行可以自由选择任何方式使用传递到 LOAD 的地址信息：它并不一定要是一个 Hadoop 路径，但它必须可以被解析成数据库引用，或有其他方法。此外，由 Slicer 决定将输入数据集分割成什么样的 Slice 的对象，而 Slice 由单个 MapReduce mapper 来处理，因此自定义 Slicer 可以采用不同于 Pig 标准的切割行为来划分输入数据，这样可以将文件分割成 HDFS 的块一样大小的部分。例如，为处理大的图片或视频文件的 Slicer 可能在每个部分创建一个文件。

可以在 Pig 维基页面(网址为 *http://wiki.apache.org/pig/UDFManual*)进一步了解如何写 Slicer。

11.6 数据处理操作符

11.6.1 加载和存储数据

在整个这一章，我们已经看到如何在 Pig 中从外部存储设备上加载处理数据。存储结果也是很直观。下面是一个使用 Pig-Storage 将元组存储为纯文本的例子，纯文本是用冒号作为分隔符的：

```
grunt> STORE A INTO 'out' USING PigStorage(':');
grunt> cat out
Joe:cherry:2
Ali:apple:3
Joe:banana:2
Eve:apple:7
```

11.6.2 过滤数据

一旦已经将数据加载到关系，下一步通常就是过滤掉不感兴趣的数据。通过提前在处理管线上执行过滤，你可以最大限度地减少通过该系统的流动数据量，这样可提高效率。

FOREACH .. GENERATE

我们已经了解如何使用带有简单表达式和 UDF 的 FILTER 操作符从一个关系中删除行，且学会了使用 FOREACH...GENERATE 操作关系的每一行。它可以用来删除字段或产生新字段。在这个例子中，我们两件事都做：

```
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)
```

这里，我们已经创建了有三个字段的新关系 B。它的第一个字段是关系 A 的第一个字段($ 0)的投影。B 的第二个字段是关系 A 中加入一个 1($1)后的第三个字段。

B 的第三个字段是一个常量字段(B 中的每一行具有相同的第三个字段)，它是值为 Constant 的字符数组。

FOREACH...GENERATE 操作有一个支持更复杂处理的嵌套形式。在下面的例子，我们为气温数据集计算各种统计数据：

```
-- year_stats.pig
REGISTER pig.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;
```

使用我们早些时候开发的 cut UDF，我们从输入数据集中将一些字段加载到记录关系中去。接下来，我们按照 year 对记录进行分组。我们要注意设置 reducer 数量的关键字 PARALLEL，这在集群上运行时至关重要。然后，我们使用一个嵌套 FOREACH...GENERATE 操作处理每个组。第一个嵌套的语句通过 DISTINCT 操作创建一个关系，它为气象站区分 USAF 标识符。第二个嵌套的语句通过 FILTER 操作和一个 UDF 创建一个经过“良好”读取的关系。最后嵌套的语句是一个 GENERATE 语句(一个嵌套的 FOREACH...GENERATE 必须有一个 GENERATE 语句作为嵌套的结束语句)，它产生了创建嵌套的块的关系。

在一些年份的数据上运行它，我们会得到以下数据：

```
(1920,8L,8595L,8595L)
(1950,1988L,8635452L,8641353L)
(1930,121L,89245L,89262L)
(1910,7L,7650L,7650L)
(1940,732L,1052333L,1052976L)
```

字段是 year，唯一气象站的数目，良好数据的总数和读数的总数。我们可以看到气象站和读数如何随时间上升。

STREAM

STREAM 操作允许你使用外部程序或者脚本在关系中传递数据。它的命名与 Hadoop 流的命名类似，Hadoop 流也为 MapReduce 提供了类似的能力(参见 2.5 节)。

STREAM 可以使用带有参数的内置命令。这里有一个例子，它使用 Unix 的 cut 命令来提取 A 中的每个元组的第二个字段。注意，命令和参数都被封装在反引号内：

```
grunt> C = STREAM A THROUGH `cut -f 2`;
grunt> DUMP C;
(cherry)
(apple)
(banana)
(apple)
```

STREAM 操作使用为程序的标准输入和输出流序列化或者反序列化关系。A 的元组被转换为用标签分隔开的一行传递给脚本。该脚本的输出是一次读取一行，并且在标签上分割并创建输出关系 C 的新元组。可以使用 DEFINE 命令提供自定义的序列化函数和反序列化函数。

编写自定义处理脚本时，Pig 流是最强大的。以下 Python 脚本过滤了损坏的气温记录：

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

import re
import string
import sys

for line in sys.stdin:
  (year, temp, q) = string.strip().split(line)
  if (temp != "9999" and re.match("[01459]", q)):
    print "%s\t%s" % (year, temp)
```

需要将脚本运到群集上才能使用它。为此需要使用一个 DEFINE 子句，它也会为 STREAM 命令创建一个别名。STREAM 语句然后可以指向这个别名，如以下 Pig 脚本显示：

```
-- max_temp_filter_stream.pig
DEFINE is_good_quality `is_good_quality.py`
  SHIP ('src/main/ch11/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;
```

11.6.3 数据的分组和联接

在 MapReduce 上联接数据集需要程序员做一部分工作(参见 8.3 节)，对于联接操作，Pig 提供了很好的内置支持。由于 Pig(和 MapReduce)擅长分析的大型数据集是非规范化，所以在 Pig 中使用联连操作不如 SQL 中那么频繁。

联接

让我们看一个内联接的例子。考虑到关系 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)
```

我们为每个相同的数字类型的字段上联接两个关系：

```
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)
```

这是一个典型的内联接，每个在两个关系中的匹配都会对应结果中的每一行。(这实际上是等值联接因为联接的谓词是等于。)结果字段是由所有输入关系中的所有的字段组成的。如果所有被联接的关系太大以至于不能放入内存，那么应该使用一般联接操作。如果关系中的一个小到足以放在内存中，那么有一种称为碎片复制联接的特殊联接类型，它可以通过将小输入数据分布到所有 mappers 中去的方式执行，并通过一个在内存中的关于更大关系的查找表来执行一个。有一个特殊的语法用于告诉 Pig 使用碎片复制联接：

```
grunt> C = JOIN A BY $0, B BY $1 USING "replicated";①
```

第一个关系必须大，后跟一个或多个小的关系。

COGROUP

JOIN 总是给出一个扁平的结构：一组元组。该 COGROUP 语句类似于 JOIN，但它创建一组嵌套的输出元组。如果想在随后的语句中使用此结构，这是有用的：

```
grunt> D = COGROUP A BY $0, B BY $1;
grunt> DUMP D;
(0,{},{(Ali,0)})
(1,{(1,Scarf)},{})
(2,{(2,Tie)},{(Joe,2),(Hank,2)})
(3,{(3,Hat)},{(Eve,3)})
(4,{(4,Coat)},{(Hank,4)})
```

COGROUP 为每一个唯一分组键生成了一个元组。每个元组的第一个字段是键，其余的字段是带有匹配键的关系中的元组的包。第一个包包含关系 A 中的有相同的键的匹配元组。同样，第二包包含关系 B 中的有相同的键的匹配元组。

如果关系中的某个特定键没有匹配的键，那么关系的包是空的。举个例子，由于没有人买了围巾(ID 为 1)，在元组中的第二个包的那个行是空的。这是一个外联接，它是 COGROUP 的默认类型。它可使用关键字 OUTER 来让语句变得明确，并使这一 COGROUP 语句与前相同：

---

① Pig 的联接实现算法除了普通的哈希联接，还有碎片复制联接、合并联接和倾斜联接。

```
D = COGROUP A BY $0 OUTER, B BY $1 OUTER;
```

可以用 INNER 关键字来限制有空包的行，它赋予了 COGROUP 内联接的语义。INNER 关键字应用于每个关系，所以下面的例子是在关系 A 没有匹配时才限制行(这里去掉了不知名的产品 0)：

```
grunt> E = COGROUP A BY $0 INNER, B BY $1;
grunt> DUMP E;
(1,{(1,Scarf)},{})
(2,{(2,Tie)},{(Joe,2),(Hank,2)})
(3,{(3,Hat)},{(Eve,3)})
(4,{(4,Coat)},{(Hank,4)})
```

我们通过将这个结构变简单来找出谁买了关系 A 中的每一样东西：

```
grunt> F = FOREACH E GENERATE FLATTEN(A), B.$0;
grunt> DUMP F;
(1,Scarf,{})
(2,Tie,{(Joe),(Hank)})
(3,Hat,{(Eve)})
(4,Coat,{(Hank)})
```

可以运用 COGROUP，INNER 和 FLATTEN(用于去除嵌套)模仿 JOIN：

```
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)
```

`JOIN A BY $0, B BY $1.`的结果是相同的。

如果联接键是由几个字段组成，可以在 JOIN 或 COGROUP 语句的 BY 子句中指定它们。请确保在每个 BY 子句的字段数量是相同的。

这里还有一个 Pig 的联接例子，在脚本中，在由输入值控制的一段时间内给每个气象站计算最高气温：

```
-- max_temp_station_name.pig
REGISTER pig.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,
  com.hadoopbook.pig.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';
```

我们使用较早开发 cut UDF 加载一个有站点 ID 和名称的关系(USAF 和 WBAN 标识符)以及具有所有天气记录的一个关系和有站点 ID 的标识。我们将过滤的天气记录用站点 ID 分组，并在加入站点前，合计最高气温。最后，我们找出在最终结果中需要的字段：USAF，WBAN，气象站名和最高气温。以下是 20 世纪 10 年代的一些结果：

```
228020 99999 SORTAVALA         322
029110 99999 VAASA AIRPORT     300
040650 99999 GRIMSEY           378
```

这个查询可以用一个片段重复联接来使其变得更有效，因为气象站的元数据是很小的。

CROSS 语句

Pig Latin 包括向量积操作(也称为笛卡儿积)，其中将关系中的每一个元组联接到第二个关系的每个元组上(如果提供更多关系的话，那么联接那些关系上的元组)。输出的大小是输入大小的积，从而可能使输出值非常大：

```
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)
```

在处理大型数据集时，应该尽量避免产生大小为原来平方倍(或更糟)的中间产物。计算整个数据集的向量积是不太需要的。

我们自然想到一个例子：计算文档语料各文档的相似度，需要保证对每一份文件是成对产生才可计算其相似性。但是，如果一开始就有大多数的文件对相似度为零的情况(也就是说，它们是无关的)，可以找一种更好的算法。

在这种情况下，核心思想是将重点放在我们用来计算相似度的实体上(例如，文件中的条目)，使之成为该算法的中心。实际上，也可以删除文件(停用词)中不利于区分的条目，这进一步降低了问题空间。使用这种技术分析大约 100 106 万个的文件会产生 10 亿(109)中间对[1]，而不是用单纯的或者使用没有停用词删减的方法(生成输入的向量积)产生万亿(1012)中间对。

① "Paurwuse Document Similarity in Large Collections with Map Reduce", Elsayed, Lin,, Oard(2008 年，学院园，博士：马里兰大学)。

GROUP 语句

虽然 COGROUP 为两个或两个以上的关系中的数据分组，但是 GROUP 语句只为单个关系中的数据分组。GROUP 不仅支持用键的平等性分组，你也可以使用一个表达式或用户定义函数作为组键。例如，考虑下面的关系 A：

```
grunt> DUMP A;
(Joe,cherry)
(Ali,apple)
(Joe,banana)
(Eve,apple)
```

第二个字段用字符数来分组：

```
grunt> B = GROUP A BY SIZE($1);
grunt> DUMP B;
(5L,{(Ali,apple),(Eve,apple)})
(6L,{(Joe,cherry),(Joe,banana)})
```

GROUP 创建了一个关系，其第一个字段是分组字段，它被命名为组。第二个字段是一个包，它包含与原始关系(本例是 A)有相同模式的被分组的字段。

还有两个特殊的分组操作：ALL 和 ANY。ALL 将单个组的关系中的所有元组都组合起来，就像 GROUP 一样是一个常值：

```
grunt> C = GROUP A ALL;
grunt> DUMP C;
(all,{(Joe,cherry),(Ali,apple),(Joe,banana),(Eve,apple)})
```

请注意，这种 GROUP 语句形式没有 By。ALL 分组通常用于在计算关系中的元组数。

ANY 关键字用于随机在关系中分组，在取样过程中这非常有用。

11.6.4 数据的排序

Pig 中的关系是无序的。可以参考关系 A：

```
grunt> DUMP A;
(2,3)
(1,2)
(2,4)
```

行将用什么样的顺序进行处理并没有保证，尤其是用 DUMP 和 STORE 获得 A 中

的内容时，行可能写成任何顺序。如果想强制输出一个顺序，可以用 ORDER 操作以一个或者多个字段排列关系。默认的排序顺序将比较相同类型的字段进行自然排序，而且不同的类型有一个任意的但是确定的顺序(例如，一个元组永远比一个包“小”)。使用 VSING 子句指定一个 UDF 可以产生不同的排序，此 UDF 继承 Pig 的 ComparisonFunc 类。

下面的例子以第一个字段的升序以及第二个字段的降序来排列 A，

```
grunt> B = ORDER A BY $0, $1 DESC;
grunt> DUMP B;
(1,2)
(2,4)
(2,3)
```

在已经排好序的关系上所进行的任何进一步的处理并不能保证保留它的顺序。例如：

```
grunt> C = FOREACH B GENERATE *;
```

尽管关系 C 并没有与关系 B 相同的内容，但它的元组可以用 DUMP 或 STORE 以任何顺序发出。正因为这个原因，在得到输出之前执行 ORDER 操作是很正常的。

LIMIT 语句在限制结果数时是很有用的，它是得到关系样本的一种快速和粗略的方法。原型(ILLUSTRATE 命令更喜欢产生更具有代表性的样本数据)。在 ORDER 语句之后，可以很快运用它获取前 n 个元组。通常，LIMIT 会从关系中选择任意 n 个元组，但是 ORDER 语句之后，顺序很快会被保留(这是一个例外情况，它在处理关系时不保留它的顺序)。

```
grunt> D = LIMIT B 2;
grunt> DUMP D;
(1,2)
(2,4)
```

如果限制的大小比关系中的元组数大，则返回所有元组(因此限制没有任何影响)。

使用 LIMIT 可以提高查询性能，因为 Pig 尝试在处理管道中尽早运用限制，可以减少需要处理的数据量。为此，如果不是对整个输出感兴趣，应该一直使用 LIMIT。

11.6.5 数据的合并和分割

有时候想将一些关系合并成一个，此时可以使用 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 的合并，由于关系是无序的，所以关系 C 元组的顺序是不确定的。此外，可以将两个有不同模式或不同字段数的两个关系合并起来，就像前面一样。Pig 尝试在 UNION 操作时将关系中的模式合并起来。在这种情况下，它们是不相容的，所以 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，你的脚本需要能够处理有不同的字段和/或者类型的元组，SPLIT 操作与 UNION 相反，它将一个关系分割成两个或两个以上的关系。

11.7 Pig 实践提示与技巧

在开发和运行 Pig 程序时，有些实用技巧有必要知道。本节介绍其中的一些。

11.7.1 并行

在 Hadoop 模式下运行时，需要告诉 Pig 每道作业需要多少个 reducer。你可以使用操作符的 PARALLEL 子句运行在 reduce 阶段，它包括了所有的分组和联接操作(GROUP，COGROUP，JOIN，CROSS)以及 DISTINCE 和 ORDER 操作符。默认情

况下，reducer 的数量是 1(就像为 MapReduce 的)，因此在大型数据集运行时，设置并行度是很重要的。下面的行为 GROUP 将 reducer 的数目设置为 30：

```
grouped_records = GROUP records BY year PARALLEL 30;
```

一个好的 reduce 任务数的设置是，比集群的 reduce 槽稍小一点。

11.7.2 参数替换

如果有一个运行在常规基础上的 Pig 脚本，自然能想到运行有不同参数的脚本。例如，日常运行的脚本可以使用的日期确定哪些输入文件被运行了。Pig 支持参数替换，在脚本中的参数可以用在运行时提供的值取代。参数被记为以一个$字符为前缀的标识符，例如$ input 和$output，用于指定下面脚本的输入和输出路径：

```
-- max_temp_param.pig
records = LOAD '$input' AS (year:chararray, temperature:int,
quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
  (quality == 0 OR quality == 1 OR quality == 4 OR quality == 5
OR quality == 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
  MAX(filtered_records.temperature);
STORE max_temp into '$output';
```

在启动 Pig 时，参数可以用-param 选项指定，每个参数都有一个：

```
% pig \
    -param input=/user/tom/input/ncdc/micro-tab/sample.txt \
    -param output=/tmp/out \
    src/main/ch11/pig/max_temp_param.pig
```

也可以把参数放到一个文件中，并使用-param_file 选项将它传递给 Pig。例如，我们可以像前面的命令一样通过将参数定义放在一个文件中来获得相同的结果。

```
# Input file
input=/user/tom/input/ncdc/micro-tab/sample.txt
# Output file
output=/tmp/out
```

调用Pig就会变成：

```
% pig \
```

```
-param_file src/main/ch11/pig/max_temp_param.param \
src/main/ch11/pig/max_temp_param.pig
```

可以使用-param_file 反复指定多个参数文件。也可以结合使用-param 和-param_file 选项，如果有参数在一个参数文件中的同一行命令中定义，那么命令行的后一个值有优先权。

动态参数

对于使用-param 选项提供的参数，很容易通过运行一个命令或脚本使数值变成动态的。许多 Unix shell 支持将命令替换为在反引号中封装的命令，我们可以用它来使输出目录基于日期：

```
% pig \
        -param input=/user/tom/input/ncdc/micro-tab/sample.txt \
        -param output=/tmp/`date "+%Y-%m-%d"`/out \
        src/main/ch11/pig/max_temp_param.pig
```

通过执行一个 shell 中封装的命令并使用 shell 的输出作为替代值，Pig 也支持参数文件中的反引号。如果该命令或脚本以非零退出状态退出，会有错误报出而且执行会停止。在参数文件中支持的反引号是一个有用的功能，如果它们被定义在文件或命令行上，它意味着参数可以以同样的方式定义。

参数替换处理

在脚本运行前的预处理步骤中会发生参数替换。通过用-dryrun 选项执行 Pig 产生的预处理，可以看到替换发生。在纯粹的运行模式，Pig 执行参数替换并生成带有替换值的原始脚本，但不执行该脚本。在正常模式上运行之前，可以检查生成的脚本，并检查该替换是否明智(因为它们是动态生成的)。

在本书写作时，Grunt 并不支持参数替换。

第 12 章

Hbase 简介

Jonathan Gray 和 Michael Stack

12.1 HBase 基础

HBase 是一种构建在 HDFS 之上的分布式、面向列的存储系统。在需要实时读写、随机访问超大数据集时，可以使用 Hbase 这一 Hadoop 应用。

尽管已经有许多数据存储和访问的策略和实现方法，但事实上大多数解决方案，特别是一些关系类型的，在构建时并没有考虑到超大规模和分布式的特点。许多商家通过复制和分区的方法来扩充数据库使其突破单个节点的界限，但这些功能通常都是事后增加的，安装和维护都很复杂。同时，也会影响 RDBMS(关系数据库管理系统)的特定功能，例如联接、复杂的查询、触发器、视图和外键约束这些操作在大型的 RDBMS 上代价相当高，甚至根本无法实现。

HBase 从另一角度来处理伸缩性问题。它通过线性方式从下到上增加节点来进行扩展。HBase 不是关系型数据库，也不支持 SQL，但是它有自己的特长，这是 RDBMS 不能处理的：Hbase 巧妙地将大而稀疏的表放到商用服务器集群上。

HBase 公认的用例是 webtable，这种表存储爬取网页及其属性(如 language 和 MIME 类型)，以网页的 URL 作为键。webtable 有数以亿计的行。MapReduce 作业在 Webtable 上不断地执行，得到统计数据，增加新的 MIME 类型的列，并为之后

的搜索引擎建立索引而进行文本内容解析。当用户随机点击网地址的缓存页面属性时，随机网页实时显示，同时，不同速率的爬行器随之更新该表。

背景

HBase 项目是由 Powerset 公司的 Chad Walters 和 Jim Kellerman 从 2006 年底开始创建的。它基于当时一篇刚刚发表的、由 Chang 等人所发表的 Google 论文"Bigtable：A Distributed Storage System for Structured Data"(*http://labs.google.com/papers/bigtable.html*)。在 2007 年 2 月，Jim Kellerman 接着推进了之前 Mike Cafarella 在一个主要运行系统上所缺的代码。

HBase 的最初发布是作为 Hadoop 0.15.0 的一部分，从 2008 年初，HBase 成为 Hadoop 项目的一个子项目(*http://hadoop.apache.org/hbase or http://hbase.org*)。自 2007 年末以来，Powerset 已经将 HBase 作为产品使用，HBase 的其他使用商包括 WorldLingo、Streamy.com、OpenPlaces、Yahoo！集团和 Adobe。

12.2 概念

这一小节，我们将快速概述 HBase 概念，然后使用一个熟悉的例子帮助你理解后文。[①]

12.2.1 数据模型速览

应用程序将数据存储到带有标签的表中。表由行和列组成。表的单元格是行和列坐标的交集，它们是有版本号的。在默认情况下，版本号是在单元格插入时由 HBase 自动分配的时间戳。表的单元格的内容是一个未解释的字节数组。

表行的键也是字节数组，因此从理论上来说，无论是 string 还是 long 类型的二进制表示，甚至是序列化的数据结构，都可以作为行的键。表用行键，即表的主键，对表中的行进行排序。在默认情况下，排序是以字节为序，所有的表都通过表的主键进行访问。[②]

---

① 若需了解详情，请参见 HBase 维基页面 HBase Architecture，网址如下：*http://wiki.apache.org/hadoop/Hbase/*

② 尽管有些 HBase 的应用可以支持二级索引，但是它们只是辅助而并非核心的，并可能只能用于自己的项目。

每行的列被分组，形成列族(column families)。所有列族成员都有相同的前缀，所以，列 temperature:air 和 temperature:dew_point 都是 temperature 列族的成员，同理，station:identifier 是 station 列族的成员。[①]列族名的前缀必须由可打印字符组成，后半部分可以是任意字节。

表的列族必须在事先作为表架构定义的一部分被声明，但是可以根据需要增加新的列族成员。例如，只要列族 station 已经存在于需要更新的表中，客户端就可以提供一个新的列 station:address 进行更新并保存其值。

从物理上看，所有列族成员在文件系统中被存储在一起。因此，之前我们说 HBase 是一种面向列的存储，其实更确切的说法是面向列族的存储。因为它的调整和存储规格都是在列族这一级别定义的，所以这也表明所有列族成员都有常规的访问模式和大小特性。

HBase 表和 RDBMS 表大同小异，只有单元格有版本号，对行进行了排序，同时只要事先有列族，客户就能动态增加列。

区域

HBase 自动把表横向切分为不同区域(region)，每个区域包含表所有行的一个子集。区域依此确定：第一行(包含第一行)及最后一行(不包含最后一行)，加上一个随机生成的区域标识。刚开始时，表只含单个区域，但随着区域扩大到超过设置的阈值，便以行为分界线，划分成大小差不多的两个新区域。在第一次划分开始前，所有正在载入的数据将会存入原始控制区域的主机服务器。随着表的不断增大，表拥有的区域也在不断增加。区域是分散在 HBase 集群上的单元，因此对于任何一个服务器，再大的表都可以由服务器集群来处理。它们通过管理整个区域某部分的节点来管理整个表。在任何时候，已排序区域的在线集合都包含表的所有内容。

锁定

无论有多少列组成行级事务，行的更新都是原子的。这使锁定模型比较简单。

12.2.2 实现

就像 HDFS 和 MapReduce 都是由客户端、从节点和协调主节点组成的一样——HDFS 中的名称节点(NameNode)和数据节点(DataNodes)，MapReduce 中的

① 在 HBase 中，冒号(:)一般用来区别不同列族，它是被硬编码的。

jobtracker 和 tasktracker——HBase 由一个主节点(master)协调一个或多个区域服务器(regionserver)从节点组成(参见图 12-1)。HBase 主节点负责引导初始安装，分配区域给已注册的区域服务器，恢复区域服务器的故障。主节点负载较轻。区域服务器负责 0 到多个区域，响应客户端的读写请求。同时，它们也负责通知 HBase 主节点，区域要分裂成新的子区域的信息，以便 Hbase 主节点管理父区域的下线及替代(父区域的)子区域的分派。HBase 依赖于 Zookeeper(第 13 章)，且在默认情况下，它使用一个 Zookeeper 实例为集群状态提供授权。[①]

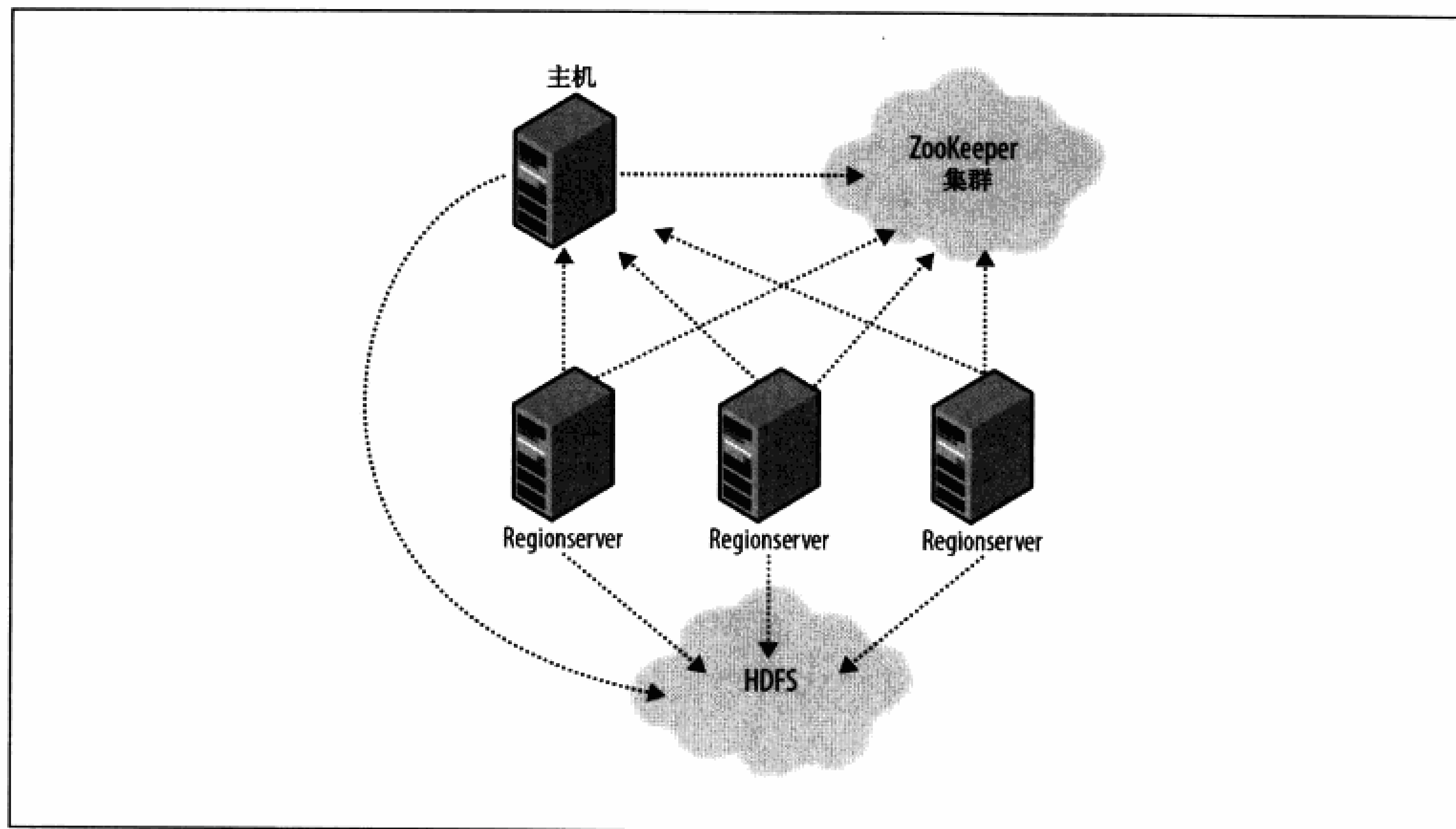


图 12-1：HBase 集群的成员

就像数据节点(datanode)和任务跟踪节点(tasktracker)被列在 Hadoop 的 conf/slaves 文件中一样，区域服务器从节点被列在 HBase 的 conf/regionservers 文件中。启动和停止脚本和 Hadoop 中的相似，同样使用基于 SSH 的远程命令运行机制。集群站点配置由 HBase 的 conf/hbase-site.xml 和 conf/hbase-env.sh 文件完成，它们的格式与 Hadoop 父项目对应的相同(参见第 9 章)。

注意：我们发现一个共通点，无论是服务还是类型，HBase 都直接使用或子类化 Hadoop 父项目的实现。如果这不可能，HBase 就会尽可能遵循 Hadoop 模型。例如，由于 HBase 使用 Hadoop 配置系统，因此其配置文件具有相同的格式。这意味着，用户可以利用对 Hadoop 的了解进一步探索 HBase。HBase 只有在增加其特殊功能时才会偏离此原则。

① 同样，可以对 HBase 进行配置，使其能使用现有的 ZooKeeper 集群。

HBase 通过 Hadoop 文件系统的 API 进行数据维护。由于文件系统接口的实现多种多样(有为本地文件系统的，为 KFS 文件系统的，也有为 Amazon 的 S3 的，为 HDFS 的)，HBase 可以选择其中任何一种。由于大部分应用都使用 HDFS 的接口，因此除非有特别说明，默认情况下 HBase 都采用本地文件系统。在第一次安装 HBase 时，这种本地文件系统相对较好，但是在 HBase 集群第一次配置时，通常都需要指明 HBase 使用的 HDFS 集群。

运行 HBase

HBase 的内部有两个目录表：*-ROOT-* 和 *.META.*。HBase 通过这两个表来维护当前列表、状态、最新历史和集群上所有活跃区域的位置。-ROOT-表包含.META.表所在区域列表。.META.表包含所有用户空间区域列表。这些表的表项以区域的起始行作为键。如前所述，行键已排序，因而要查找包含特定行的那个区域，就是找出其键大于或等于查找的行键的第一项。随着区域的变迁，划分、关闭/启用、删除、由区域负载均衡器(region load balancer)进行重新部署或由于区域服务器崩溃而被重新部署，在这些情况下，日志表都会进行相应的更新，使集群上的所有区域状态保持最新。

当新的客户端连接到 Zookeeper 集群时，首先必须知道-ROOT-的地址。客户端根据-ROOT-分析出.META.区域的地址，而.META.区域则覆盖需要查询的行。接着，客户机查询找到的.META.区域以确认用户空间区域及其地址。然后客户端即可直接与主机区域服务器交互。

为了节省每次行操作的三轮查找，客户端缓存了它们已知的所有-ROOT-和.META.的地址和用户空间区域的起始行和结束行。这样一来，它们不用再回.META.表，就能根据自己的缓存查询主机区域。直到出错时，客户端才停止使用缓存项。当区域被移起时，会出错，此时客户端将再次根据.META.表查询新的地址。接下来，如果.META.也被移走，则再查询-ROOT-。

对于区域服务器的写操作，它首先被载入提交日志(commit log)，再被加入内存缓存(in-memory cache)中。当缓存满后，缓存的内容就被提交给文件系统并清空。

提交日志由 HDFS 管理，因此就算区域服务器瘫痪，它也仍然存在。主机得知一个区域服务器不能再访问后，它会按区域划分这一“死亡”区域服务器的提交日志。在重新分配时，“死亡”区域服务器上的区域在打开进行工作之前，会选取刚划分但尚未确认的编辑的文件，先恢复到出错前的状态，再重新运行。

在读取操作时，先查询区域的分布式缓存，如果有足够多的版本可满足查询需要，则返回，否则，从新到旧，按顺序查询缓冲文件，直到找到满足要求的版本，或查

询所有缓冲文件也没有找到符合要求的。一旦缓冲文件到达一定数目，后台处理(background process)就会将它们压缩并整个将它们重新写入，因为一次读操作查询的文件越少，它的效率就越高。在压缩时，超过最大配置值的版本、删除文件和过期的单元格都将被清除。在区域服务器监控器中运行的一个独立进程会在文件大小超出配置的最大值时，刷新文件大小。

12.3 安装

从 HBase 的发布网址(*http://hadoop.apache.org/hbase/release.html*)下载稳定版本并解压到文件系统。例如：

```
% tar xzf hbase-x.y.z.tar.gz
```

由于是基于 Hadoop 的，所以首先要告知 Hbase Java 在系统上的安装地址。如果已经将 JAVA_HOME 的环境变量设置为指向适当的 Java 安装路径，就使用该配置，不需要进行更多配置。如果先前没有设置，则可以通过编辑 HBase 的 conf/hbaseenv.sh 来设置应用于 HBase 的 Java 安装，并将 JAVA_HOME 变量(示例参见附录 A)设定为 1.6.0 版本的 Java。

注意：HBase 和 Hadoop 一样需要 Java 6。

考虑到方便，可以将 HBase 的二进制目录添加到命令行路径中。例如：

```
% export HBASE_INSTALL=/home/hbase/hbase-x.y.z
% export PATH=$PATH:$HBASE_INSTALL/bin
```

要想得到 HBase 选项列表，输入以下命令：

```
% hbase
Usage: hbase <command>
where <command> is one of:
shell run the HBase shell
master run an HBase HMaster node
regionserver run an HBase HRegionServer node
rest run an HBase REST server
thrift run an HBase Thrift server
zookeeper run a Zookeeper server
migrate upgrade an hbase.rootdir
or
CLASSNAME run the class named CLASSNAME
Most commands print help when invoked w/o parameters.
```

测试驱动程序

启动一个 HBase 实例，使用本地文件系统的/tmp 目录作为持久化存储，输入以下命令：

```
% start-hbase.sh
```

这将触发两个后台程序：一个是在本地系统上的独立的 HBase 实例，默认情况下，HBase 会写到/tmp/hbase-${USERID}；另一个是一个 Zookeeper 实例(存储集群状态 Zookeeper 实例)。[①]

为管理 HBase 实例，输入以下命令启动 HBaseshell：

```
% hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Version: 0.19.1, r751874, Thu Mar 12 22:54:22 PDT 2009
hbase(main):001:0>
```

这将触发 JRuby IRB 解析器，它包含附加的 Hbase 特有命令。输入 help 再按回车键，显示可用 shell 命令列表。这个表很长，列出的每个命令都包含一个用法示例。这些命令使用 Ruby 进行排版，指定列表和字典。帮助屏幕底部有一个快速教程。

现在我们创建一个简单的表，增加一些数据，然后再将它们清除。

要创建表，必须为表命名并设置它的架构。一个表的架构包括表的属性及表的列族列表。列族也有自己的属性，是在架构定义时按顺序生成的。列表属性包括：组内容是否被文件系统压缩，一个单元保存几个版本。架构可以在脱机后编辑，具体做法是使用 shell 的 disable 命令，使用 alter 命令进行一些必要的更改，最后用 enable 命令将表上传到服务器。

要创建一个名为 test 的表，带有唯一列族 data，并使用表和列族的默认属性，输入以下命令即可：

```
hbase(main):007:0> create 'test', 'data'
0 row(s) in 4.3066 seconds
```

注意：如果前面的命令没有成功运行，shell 显示错误信息和栈跟踪，则说明没有安装成功。可检查 HBase 日志库下的主机日志(master log)查看出错原因。

---

① 在独立模式下，HBase 主机和区域服务器都运行在相同的 JVM 下。

指定一个架构时，如果需要增加表和列族属性，请参见 help 命令的输出结果。

为了证明已经成功创建新的表，可运行 list 命令。这会输出用户空间下所有的表：

```
hbase(main):019:0> list
test
1 row(s) in 0.1485 seconds
```

要在 data 列族中三个不同的行和列中插入数据并列出表的内容，如下即可：

```
hbase(main):021:0> put 'test', 'row1', 'data:1', 'value1'
0 row(s) in 0.0454 seconds
hbase(main):022:0> put 'test', 'row2', 'data:2', 'value2'
0 row(s) in 0.0035 seconds
hbase(main):023:0> put 'test', 'row3', 'data:3', 'value3'
0 row(s) in 0.0090 seconds
hbase(main):024:0> scan 'test'
ROW COLUMN+CELL
row1 column=data:1, timestamp=1240148026198, value=value1
row2 column=data:2, timestamp=1240148040035, value=value2
row3 column=data:3, timestamp=1240148047497, value=value3
3 row(s) in 0.0825 seconds
```

注意，我们在没有改变架构的情况下，新增加了三列。[①]

要想删除表，必须在删除它之前关掉它：

```
hbase(main):025:0> disable 'test'
09/04/19 06:40:13 INFO client.HBaseAdmin: Disabled test
0 row(s) in 6.0426 seconds
hbase(main):026:0> drop 'test'
09/04/19 06:40:17 INFO client.HBaseAdmin: Deleted test
0 row(s) in 0.0210 seconds
hbase(main):027:0> list
0 row(s) in 2.0645 seconds
```

运行以下语句关闭 HBase 实例：

---

① 若要快速加载一百万条 1k 的行，可像这里一样运行 PerformanceEvaluation。这将启动一个客户端运行 PerformanceEvaluatio sequentialWriter 脚本。它将生成一张名为 TestTable 的单列族表并发布它。要想进一步了解 PerformanceEvaluation 脚本，请参见 HBase 的维基页面(*http://wiki.apache.org/hadoop/Hbase/PerformanceEvaluation*)：

```
% hbase org.apache.hadoop.hbase.PerformanceEvaluation sequentialWriter 1
```

```
% stop-hbase.sh
```

若想了解怎样安装分布式 HBase 并在 HDFS 上应用它，请参见 HBase 文档的 Getting Started 部分(网址为 *http//:hadoop.apache,org/hbase/docs/carrent/api/overview-sumnary.html overuew_description*)。

12.4 客户端

有许多客户端选项用于与 HBase 集群交互。

12.4.1 Java

HBase 和 Hadoop 都是用 Java 写的。在本章后面的代码将展示如何向 HBase 表上传数据以及如何读取和更新 HBase 表中的数据。这些与现有表交互的例子中，可以使用 HBase 的 `HTable` 类获取和更新 HBase 表的实例方法。还可以使用另一类，即 `HBaseAdmin` 来管理集群(增加表、删除表以及启动表)。这两个类都在 `org.apache.hadoop.hbase.client` 包里。

MapReduce

`org.apache.hadoop.hbase.mapred` 包里的 HBase 类和实用工具简化了 HBase 的使用，使其作为 MapReduce 作业的资源提供方和(或)接受方。`TableInputFormat` 类用来划分区域边界，使 map 在单个区域上执行。`TableOutputFormat` 类则是将 reduce 以后的结果写入 HBase。在例 12-1 中，`RowCounter` 类可以在 HBase 的 `mapred` 包中找到。它运行一个 map 任务，计算有多少行使用 `TableInputFormat`。

例 12-1：MapReduce 应用，计算 HBase 表的行数

```
public class RowCounter extends Configured implements Tool {
  // Name of this 'program'
  static final String NAME = "rowcounter";

  static class RowCounterMapper
  implements TableMap<ImmutableBytesWritable, RowResult> {
    private static enum Counters {ROWS}

    public void map(ImmutableBytesWritable row, RowResult value,
        OutputCollector<ImmutableBytesWritable, RowResult> output,
```

```
      Reporter reporter)
    throws IOException {
      boolean content = false;
      for (Map.Entry<byte [], Cell> e: value.entrySet()) {
        Cell cell = e.getValue();
        if (cell != null && cell.getValue().length > 0) {
          content = true;
          break;
        }
      }
      if (!content) {
        // Don't count rows that are all empty values.
        return;
      }
      // Give out same value every time. We're only interested in
      the row/key reporter.incrCounter(Counters.ROWS, 1);
    }

    public void configure(JobConf jc) {
      // Nothing to do.
    }

    public void close() throws IOException {
      // Nothing to do.
    }
  }

  public JobConf createSubmittableJob(String[] args) throws IOException {
    JobConf c = new JobConf(getConf(), getClass());
    c.setJobName(NAME);
    // Columns are space delimited
    StringBuilder sb = new StringBuilder();
    final int columnoffset = 2;
    for (int i = columnoffset; i < args.length; i++) {
      if (i > columnoffset) {
        sb.append(" ");
      }
      sb.append(args[i]);
    }
    // Second argument is the table name.
    TableMapReduceUtil.initTableMapJob(args[1], sb.toString(),
      RowCounterMapper.class, ImmutableBytesWritable.class,
      RowResult.class, c);
```

```
    c.setNumReduceTasks(0);
    // First arg is the output directory.
    FileOutputFormat.setOutputPath(c, new Path(args[0]));
    return c;
  }

  static int printUsage() {
    System.out.println(NAME +
      " <outputdir> <tablename> <column1> [<column2>...]");
    return -1;
  }

  public int run(final String[] args) throws Exception {
    // Make sure there are at least 3 parameters
    if (args.length < 3) {
      System.err.println("ERROR: Wrong number of parameters: " +
        args.length);
      return printUsage();
    }
    JobClient.runJob(createSubmittableJob(args));
    return 0;
  }

  public static void main(String[] args) throws Exception {
    HBaseConfiguration c = new HBaseConfiguration();
    int errCode = ToolRunner.run(c, new RowCounter(), args);
    System.exit(errCode);
  }
}
```

这个类实现了Tool(详见5.2.2节)和Hbase TableMap的接口(org.apache.hadoop.mapred.Mapper 的一个特例，用来设置 TableInputFormat 传来的 map 输入类型)。createSubmittableJob()方法用来解析配置参数，在命令行上传递，以创建表和运行 RowCounter 的列。它还调用了 TableMapReduceUtil.initTableMapJob() 实用方法，其中包含设置使用的 map 类，设置输入格式为 TableInputFormat 等。map 很简单，它检查所有的列。如果所有的列都为空，就不算这一行；否则 Counters.ROWS 递增 1。

12.4.2 REST 和 thrift

Hbase 还带有 REST 和 thrift 接口。这有利于与非 Java 语言编写的程序进行交互。

无论使用哪种方法，Java 服务器都有一个 HBase 客户端实例，代理 HBase 集群内部和外部的 REST 和 thrift 的请求。这个额外的工作代理请求和响应意味着一点：相较于直接用 Java 的客户端，这些接口的运行更慢。

REST

要运行 REST，首先使用以下命令：

```
% hbase-daemon.sh start rest
```

这将启动一个服务器实例，默认端口为 60050，对它进行后台处理，并将服务器的所有输出记到 HBase 日志目录下的日志文件。

客户机可以请求被设置为 JSON 或 XML 的响应，这取决于客户端的 HTTP Accept 头是怎样设置的。参见 REST 的维基页面查询发出 REST 客户端请求的文档和相关示例。

要停止 REST 服务器，输入以下命令：

```
% hbase-daemon.sh stop rest
```

thrift

同样，可以通过为 thrift 客户端开启一个服务器来启用 thrift 服务，输入以下代码：

```
% hbase-daemon.sh start thrift
```

这将启动服务器实例，默认端口为 9090，对它进行后台处理并将服务器的所有输出记入 HBase 日志目录下的日志文件。HBase 的 thrift 文档[①]标识了使用衍生类的 thrift 的版本。HBase thrift IDL 可以在 HBase 源代码 src/java/org/apache/hadoop/hbase/thrift/Hbase.thrift 中找到。

要停止 thrift 服务器，输入以下命令：

```
% hbase-daemon.sh stop thrift
```

12.5 示例

尽管 HDFS 和 MapReduce 对于处理大数据集的批量操作是非常好的工具，但是它

① *http://hadoop.apache.org/hbase/docs/current/api/org/apache/hadoop/hbase/thrift/package-summary.html*。

们并不提供高效读写单条记录的方法。在这个例子中，我们将探索 HBase 怎样弥补这一缺点。

前几章提到的现有气象数据集包含数以万计的站点 100 多年的观察数据，而且数据还在无限增加中。在这个例子中，我们建立一个简单的 Web 接口，使用户能在不同站点间切换，并且可以按时间顺序浏览他们的历史气温观察数据。在这个例子中，假设数据集较大，观察数据可达到十亿，同时更新气温的速度非常快，每一秒都有成百上千的更新数据从世界各地的气象站涌来。除此之外，Web 应用必须最大程度显示即时接收到的观察数据。

第一个大小需求就决定了无法使用一个简单的 RDBMS 实例，所以使用 HBase 作为备选存储。第二个潜在需求也排除了一般的 HDFS。一个 MapReduce 作业可以建立初始索引，使所有的观察数据能够被随机访问，但一旦有更新的数据到达需要进行数据维护，这一点，HDFS 和 MapReduce 就爱莫能助了。

12.5.1 架构

在我们的例子中，有两个表。

Stations(站点)

此表维护站点数据。stationid 作为行键。以 info 作为列族，它为站点信息提供键/值(key/val)字典。给键定义名称、地址和描述。此表是静态的，在这种情况下，info 组和一个典型的 RDBMS 表设计非常相似。

Observations(观察数据)

此表包含气温观察数据。将 stationid 和逆序时间戳作为行键。将 data 作为列族，包含 airtemp 列，将观察数据作为列值。

我们对于架构的选择取决于我们想怎样从 Hbase 高效读取数据。行和列按字典顺序升序存储，虽然可以进行二级索引和正则表达式匹配，但这样会带来性能损失。[①]你必须知道自己想怎样最高效地查询数据，这对于实现高效的查询和访问数据非常重要。

对于 station 表，显然应选择使用 stationid 作为键，因为我们通常都使用站点 ID 来获取特定站点的信息。observations 表则使用一个末尾附加时间戳的组合键。这样可以将一个特定站点所有的观察数据统一分组，并使用逆序时间戳

---

① 捆绑式 HBase 二级索引机制利用了非常强大但性能较弱的 TransactionalHBase。如果需要良好的性能，目前还是建议你使用自己的二级索引表或利用一个外部索引来执行，比如 Lucene。

(Long.MAX_VALUE - epoch)，以二进制形式存储。每个站点的观察数据将按最新数据放在前面的顺序排序。

可在 shell 中按照以下方式定义表：

```
hbase(main):036:0> create 'stations', {NAME => 'info', VERSIONS
=> 1}0 row(s) in 0.1304 seconds
hbase(main):037:0> create 'observations', {NAME => 'data',
VERSIONS => 1}0 row(s) in 0.1332 seconds
```

在两种情况下，我们感兴趣的只是表单元格的最新版本，因此将 VERSIONS 设为 1。

12.5.2 加载数据

如果有相对较少的气象站，则可以利用任何一个可用的接口，轻松插入静态数据。

然而，如果要加载数十亿的观察数据，这种导入方法通常极其复杂，并且需要数据库长时间运行，尽管如此，MapReduce 和 HBase 的分布式模型还是可以让我们充分利用集群。将行输入数据复制到 HDFS 上，同时运行 MapReduce 作业读取输入数据并写到 HBase，完全读入再写入到所有节点上。最初，运行导入时，表只有一个区域，因此写操作只在一个服务器上进行，但随着导入进展和表的划分，区域将随着写入操作被分布在集群上。

例 12-2 演示了一个 MapReduce 示例作业，它将观察数据导入到 HBase，输入文件与前几章例子中使用的相同。

例 12-2：MapReduce 应用，将气温数据从 HDFS 导入一个 HBase 表

```
public class HBaseTemperatureImporter extends Configured implements Tool {

  // Inner-class for map
  static class HBaseTemperatureMapper<K, V> extends MapReduceBase
    implements
      Mapper<LongWritable, Text, K, V> {
    private NcdcRecordParser parser = new NcdcRecordParser();
    private HTable table;

    public void map(LongWritable key, Text value,
      OutputCollector<K, V> output, Reporter reporter)
    throws IOException {
      parser.parse(value.toString());
      if (parser.isValidTemperature()) {
```

```
        byte[] rowKey = RowKeyConverter.makeObservationRowKey
          (parser.getStationId(),
          parser.getObservationDate().getTime());
        BatchUpdate bu = new BatchUpdate(rowKey);
        bu.put("data:airtemp", Bytes.toBytes
         (parser.getAirTemperature()));
        table.commit(bu);
      }
    }

    public void configure(JobConf jc) {
      super.configure(jc);
      // Create the HBase table client once up-front and keep it
      // around rather than create on each map invocation.
      try {
        this.table = new HTable(new HBaseConfiguration(jc),
          "observations");
        } catch (IOException e) {
          throw new RuntimeException("Failed HTable construction", e);
        }
      }
  }

  public int run(String[] args) throws IOException {
    if (args.length != 1) {
      System.err.println("Usage: HBaseTemperatureImporter <input>");
      return -1;
    }
    JobConf jc = new JobConf(getConf(), getClass());
    FileInputFormat.addInputPath(jc, new Path(args[0]));
    jc.setMapperClass(HBaseTemperatureMapper.class);
    jc.setNumReduceTasks(0);
    jc.setOutputFormat(NullOutputFormat.class);
    JobClient.runJob(jc);
    return 0;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new HBaseConfiguration(),
      new HBaseTemperatureImporter(), args);
    System.exit(exitCode);
  }
}
```

HBaseTemperatureImporter 有一个名为 HBaseTemperatureMapper 的内部类，就像第 5 章的 MaxTemperatureMapper 类一样。这个外部类实现 Tool 接口，并执行安装过程以启动内部类 HBaseTemperatureMapper。HBaseTemperatureMapper 和 MaxTemperatureMapper 一样，接受相同的输入和进行相同的解析(使用第 5 章介绍的 NcdcRecordParser)来检查气温数据的合法性，但不像 MaxTemperatureMapper 那样会增加合法气温到输出收集器，而是将它们增加到 HBase 观察数据表上的 data:airtemp 列。在 configure()方法中，我们再一次针对观察数据表增加了一个 HTable 实例，并在之后 map 对 HBase 调用时进行使用。

使用的行键是 RowKey Converter 类的 makeObservationRowKey()方法中，根据站点 ID 和观察时间来创建的。

```
public class RowKeyConverter {

  private static final int STATION_ID_LENGTH = 12;
  /**
   * @return A row key whose format is: <station_id>
     <reverse_order_epoch>
   */
  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 reverseOrderEpoch = Long.MAX_VALUE - observationTime;
    Bytes.putLong(row, STATION_ID_LENGTH, reverseOrderEpoch);
    return row;
  }
}
```

这个转换利用了站点 ID 是一个固定长度字符串这一事实。MakeObservationRowKey()方法中使用的 Bytes 类来自 Hbase 实用工具。它包括了字节数组与普通 Java 和 Hadoop 类型之间的转换。在 makeObservationRowKey()中，Bytes.putLong()方法用来填充键字节数组。Bytes.SIZEOF_LONG 常量用于计算行键数组长度和在数组中定位。

我们可以通过以下代码运行程序：

```
% hbase HBaseTemperatureImporter input/ncdc/all
```

优化提示

- 注意观察，在所有客户端上导入数据时都与表的区域逐一对应（在单个节点上），然后再进行下一个节点，以此类推，而不是在所有区域上平均分布载入的数据。这通常是由已排好序的输入之间的一些交互和划分器的工作方式所造成的。在插入前，随机排序行键可能会对此有帮助。在我们的例子中，已给定 stationid 值的分布和 `TextInputFormat` 的划分方法，所以上传时会被充分地分布。[1]
- 每项任务只能获取一个 `HTable` 实例。实例化一个 Htable 实例会带来系统开销，因此如果每次插入都要实例化，可能会对性能造成负面影响。所以我们在 `configure()` 步骤中进行 `HTable` 的安装。
- 在默认情况下，每个 `HTable.Commit(BatchUpdate)` 实际执行的是插入，不会有任何缓存。可以使用 `HTable.setAutoFlush(false)` 禁用 HTable 自动刷新功能，然后设置可配置写缓冲区的大小。提交的插入任务填满写缓冲区时，它会被刷新。记住，在每项任务的最后，必须手动调用 `HTable.flushCommits()` 以保证缓冲区中没有未被清除的文件。为此，可以重载 mapper 中的 `close()` 方法。
- HBase 包含 TableInputFormat 和 TableOutputFormat 以帮助提供和获取 HBase 资源的 MapReduce 作业(参见例 12-1)。写前面示例的一种方法是，使用第 5 章的 MaxTemperatureMapper，另外再增加一个新的 reducer 任务，获得 MaxTemperature-Mapper 的输出，并通过 TableOutputFormat 将它提供给 HBase。

12.5.3 Web 查询

我们直接使用 Hbase 的 Java API 来实现 Web 应用。现在你应该清楚选择架构和存储格式的重要性吧。

最简单的查询是获取静态的站点信息。这类查询在传统数据库中是很简单的，但 HBase 提供了额外的操作和灵活性。使用 info 组作为键/值(key/value)字典(列名作为键，列值作为值)，代码如下：

---

[1] 如果一个表是新的，它将只有一个区域，并且在最初阶段所有更新都会被加载到这个区域上直到它分裂，即使行键是随机分布的。这就意味着上传的速度最初比较缓慢，直到有充足的区域被分布从而使所有集群成员都参与上传。不要将这种现象和先前描述的混为一谈。

```
public Map<String, String> getStationInfo(HTable table, String
    stationId)
      throws IOException {
  byte[][] columns = { Bytes.toBytes("info:") };
  RowResult res = table.getRow(Bytes.toBytes(stationId), columns);
  if (res == null) {
    return null;
  }
  Map<String, String> resultMap = new HashMap<String, String>();
  resultMap.put("name", getValue(res, "info:name"));
  resultMap.put("location", getValue(res, "info:location"));
  resultMap.put("description", getValue(res, "info:description"));
  return resultMap;
}

private static String getValue(RowResult res, String key) {
  Cell c = res.get(key.getBytes());
  if (c == null) {
    return "";
  }
  return Bytes.toString(c.getValue());
}
```

在这个例子中，getStationInfo()获取了一个 HTable 实例和一个站点 ID。为了获取站点信息，我们使用 HTable.getRow()，需要从行获取多列记录时，可使用这个方法。传递一个列族名 info：而不是具体的列名，将返回这个组所有列[①]。getRow()的结果显示在 RowResult 中，作为以行键作为数据成员的 SortedMap 的执行结果。它由列名编码为一个字节数组，值是 cell 数据结构，有一个时间戳且单元格的内容在字节数组中。getStationInfo()方法将 RowResult Map 转为一个更友好的以 String 为键和值的 Map。

我们已经看到在使用 HBase 时，需要用到工具函数。现在，越来越多的东西正从 HBase 抽象出来，以处理底层的交互。但更重要的是了解它们的工作原理和选择不同存储方式有何差异。

在关系数据库上使用 HBase 的一个好处是不需要事先定义列。因此，尽管每个站点现在可能至少有三个属性，但在将来可能有成百上千的选择，那时，我们不需要改变架

① 列族的名称为不含限定符的列名。冒号是必需的。如果只传递 info 作为列族名，而不是 info：，HBase 会报错，指出列族名格式不正确。

构就可以插入数据。当然，你需要修改读写代码。示例中的代码将被修改，使其用 RowResult.entrySet()循环，而不是在 RowResult 中显式地逐一取值。

我们将用 HBase 扫描器在我们的 Web 应用中检索观察数据。

这里，我们根据 Map<ObservationTime, ObservedTemp>的结果而使用 NavigableMap<Long, Integer>，因为它已排序且有一个 descendingMap()方法，所以我们可以按升序或降序来访问观察数据。代码如例 12-3 所示。

例 12-3：从一个 HBase 表获取一系列气象站观察数据

```
public NavigableMap<Long, Integer> getStationObservations(HTable table,
    String stationId, long maxStamp, int maxCount) throws IOException {
  byte[][] columns = { Bytes.toBytes("data:airtemp") };
  byte[] startRow = RowKeyConverter.makeObservationRowKey(stationId,
    maxStamp);
  RowResult res = null;
  NavigableMap<Long, Integer> resultMap = new TreeMap<Long, Integer>();
  byte[] airtempColumn = Bytes.toBytes("data:airtemp");
  Scanner s = table.getScanner(columns, startRow);
  int count = 0;
  try {
    while ((res = s.next()) != null && count++ < maxCount) {
      byte[] row = res.getRow();
      byte[] value = res.get(airtempColumn).getValue();
      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 {
      s.close();
  }
  return resultMap;
}
/**
 * Return the last ten observations.
 */
public NavigableMap<Long, Integer> getStationObservations(HTable
    table,String stationId) throws IOException {
  return getStationObservations(table, stationId, Long.MAX_VALUE, 10);
}
```

getStationObservations()方法用来获取站点 ID 的和由 max Stamp 和

maxCount 定义最大行数的范围。注意，返回的 NavigableMap 现在实际是按时间顺序降序排列的。如果想按照升序读取，需要使用 NavigableMap.descendingMap()。

以 Long.MAX_VALUE - stamp 顺序排序的优势在本例中也许并不明显。它的优势体现在你希望根据一个给定的上下限获得最新观察数据时。这种情况在 Web 应用中经常出现。如果观察数据按实际时间戳来排序，我们就能获得给定上下限的最老数据。获得最新数据意味着要获取所有数据，之后再到最后面进行查找。这便是需要从 RDBMS 迁移到 HBase 的一个重要原因。

扫描器

HBase 扫描器(Scanner)就像传统数据库中的游标或 Java 迭代器一样，区别在于(除外后者)它们在使用后必须关闭。扫描器有序地返回行。使用者通过调用 HTable.getScanner()获取一个 HBase 表的扫描器。大量重载形式允许用户传递开始扫描的行和停止扫描的行，并在行结果中显示这一行中的列。同时在服务器端运行一个过滤器(可选)。[①]扫描器接口不在 Java 的文档中，如下所示：

```
public interface Scanner extends Closeable, Iterable<RowResult> {
  public RowResult next() throws IOException;
  public RowResult [] next(int nbRows) throws IOException;
  public void close();
}
```

你可以查询下一行或几行的结果。每一次 next()调用都会返回区域服务器一次，因此一次获取多行可以大量减少性能损耗。[②]

12.6 HBase 与 RDBMS 的比较

HBase 和其他面向列的数据库一样，通常会与更传统的、普通的关系数据库或 RDBMS 作比较。它们的实现和所要完成的任务截然不同，虽然有许多不同之处，但对它们就相同问题的可能解决方案进行比较还是合理的。

---

① 要想进一步了解 HBase 的服务器端过滤机制，请参见 *http://hadoop.apache.org/hbase/docs/current/api/org/apache/hadoop/hbase/filter/package-summary.html*。

② hbase.client.scanner.caching 配置选项的默认值为 1。扫描器将一次获取所包含的大量结果，并将它们发到客户端，再返回到服务器继续获取下一批的结果，直到批处理全部完成。缓冲值越高，扫描就越快，但会占用客户端更多内存。

如前所述，HBase 是分布式的、面向列的数据存储系统。它延续了 Hadoop 留下的工作，提供在 HDFS 之上的随机读写。它从底层开始设计并侧重于每个方向的扩展问题：行数的长度(数十亿)、列数的宽度(数百万)以及在成千上万的商用节点上的自动横向分区和复制。表的架构是物理存储的映射，构建一个用于高效数据结构串行化、存储和检索的系统。这样一来，任务就落在应用程序开发者身上，看他是否能恰当使用这样的存储和检索模式。

严格来说，RDBMS 是一个严格遵守 Codd 12 法则(*http://en.wikipedia.org/wiki/Codd%27s_12_rule*)的数据库。典型的 RDBMS 是模式固定的，它拥有 ACID 特性和精密的 SQL 查询引擎，是面向行的数据库。它强调强一致性、参照完整性、物理层的抽象以及通过 SQL 语言的复杂查询。你可以很容易地创建一个二级索引，执行复杂的外部和内部联接、计数、求和、排序和分组，也可以将数据分页到几个表、行和列中。

对于大多数中小型应用程序来说，这种方案是无可替代的，它灵活、成熟，同时有开源 RDBMS 解决方案的强大功能(比如 MySQL 和 PostgreSQL)。然而，如果需要对数据集大小或读写并发性或是两者兼有的伸缩性要求，很快你就会发现这时 RDBMS 的性能损失非常大，使分布本身非常困难。这种伸缩性的 RDBMS 通常会打破 Codd 法则，放宽 ACID 约束，数据库管理员的传统智慧也变得无济于事，与此同时还失去了使关系数据库如此便捷而成为首选的绝大部分优点。

12.6.1 成功的服务

这里有一个典型的 RDBMS 伸缩性怎样实现的梗概。以下列表假设了一个成功扩展的服务。

- *初始公开启动*
  从本地工作站迁移到共享的、有一个精心定义模式的远程托管 MySQL 实例。
- *服务变得更广泛，大量读操作向数据库涌来*
  增加分布式缓存来缓存普通的查询。读操作不再是严格的 ACID；缓存的数据肯定会过期。
- *服务持续增长，大量写操作向数据库涌来*
  通过强化过的服务器纵向延伸 MySQL，服务器有 16 个核、128 GB 的 RAM 和几个 15 k RPM 的硬盘驱动器，成本很高。
- *新功能增加了查询的复杂性，联接操作过多*
  反规范化代码来减少联接操作。(DBA 学校没有教我们！)

- *不断增加的扩展使服务器不堪负荷，处理速度变得很慢*

  停止进行服务器端的所有计算。
- *一些查询仍然很慢*

  周期性地事先具体化复杂的查询，并在多数情况下停止联接操作。
- *读操作还好，但是写操作变得越来越慢。*

  放弃二级索引和触发器。(没有索引？)

此时，要解决伸缩性问题仍没有一个明确的方案。无论如何，都需要水平扩展。你可以试着从最大的表中分出一些小的类型，或者寻找一些商用解决方案提供多主机的能力。

数不胜数的应用、业务和网站都具有很好伸缩性、容错性和建立在 RDBMS 上的分布式数据系统，倾向于使用许多原有的策略。但是，最终拥有的不再是一个真正的 RDBMS，为了折衷和解决复杂问题，它们牺牲了原有的特征和便捷。任何形式的从机备份或外部缓存都会给非规范化数据带来不一致性。低效率的联接和二级索引意味着几乎所有查询都变成主键查找。一个可多写入设置可能意味着根本没有真正的联接操作，分布式事务处理成为一场噩梦。管理一个完全独立的缓存集群，也使网络拓扑变得极为复杂。即使使用这种系统，采取折衷策略，你仍会担心主机会崩溃以及几个月内可能有 10 倍于现在数据和负载的残酷事实。

12.6.2 HBase

引入 HBaes 之后，会有以下特性。

- *无真正的索引*

  行是顺序存储的，每行中的列也一样。因此不会有索引膨胀的问题，而且插入操作的性能与表的大小无关。
- *自动分区*

  随着表的增长，它们会自动被分割成区域，分布在所有可用的节点上。
- *创建新节点时自动地线性扩展*

  增加一个节点，将它加入现有的集群，然后运行区域服务器。区域将自动再平衡，因而负载会均匀分布。
- *商用硬件*

  集群建立在$1000～$5000 的节点上，而不是$50 000 的节点。RDBMS 非常消耗 I/O，而 I/O 又是硬件消耗中最贵的一种。
- *容错性*

  大量节点降低了每一个节点的相对重要性，因此没有必要担心某个节点的故障时间。

- *批量处理*

  MapReduce 集成实现了充分的并行性，根据局部性原则将作业分布到数据上。

如果你还在因为担心数据库(正常运行时间，伸缩性或者速度)而无法睡眠，应该好好考虑一下从 RDBMS 的世界跳到 Hbase。使用一个可行的伸缩性方案，而不是像过去那样为此不遗余力地砸钱。有了 Hbase，软件是免费的，硬件是便宜的，分布性是固有的，这是一条捷径，不会再有黑盒阻挡你前进的脚步了。

12.6.3 用例：streamy.com 的 Hbase

`streamy.com` 是一个实时的新闻聚合器和一个社交共享平台。由于有强大的功能集合，我们开始使用的是在 PostgreSQL 上的一个复杂实现。它是一个非常棒的产品，拥有很大的社区和一个非常漂亮的代码库。我们尝试了书中的所有办法来使数据库随着扩展保持速度，为了满足需求而花大量精力直接修改代码。起初，我们利用了所有 RDBMS 的优势，然而最终渐渐发现，我们不得不放弃所有这一切。与此同时，我们整个团队成为了 DBA。

我们成功解决了大部分问题，但是其中两个问题最终引领我们找到了另一种解决方案，来到 RDBMS 之外的世界。

Streamy 抓取了成千上万的 RSS 订阅，聚集了几千万条目。除了要存储它们，更复杂的查询之一就是要从一个资源集合中读取按时间排序的所有条目的列表。最终，它可以运行几千个资源，并且用单次查询就能获取所有的条目。

大批量条目表

起先，这是一个单条目的表，但是大量的二级索引使得插入和更新都变得非常慢。我们开始着手将条目分成几个一对一的链接表，用来存储其他信息，使静态区域从动态区域中分离出来，根据它们的查询方法来分区，同时不断地将所有表都反规范化。即使进行了这些改变，单个的更新还是需要重新写整个记录，因此条目的追踪统计数据非常难以扩展。重写记录和一直保持更新索引都是我们所使用的 RDBMS 一些固有的特性，是不可能被消除的。当时是自然分区，这并不难，但随着时间推移，问题的复杂性很快就失控。我们需要另一种解决方案！

超大排序合并

对按时间排序的列表进行排序合并在 Web 2.0 应用中是非常常见的。一个 SQL 查询示例子可能如下所示：

```
SELECT id, stamp, type FROM streams
  WHERE type IN ('type1','type2','type3','type4',...,'typeN')
  ORDER BY stamp DESC LIMIT 10 OFFSET 0;
```

假设 id 是 streams 的主键，stamp 和 type 有二级索引，RDBMS 查询设计器就会对查询做如下处理：

```
MERGE (
  SELECT id, stamp, type FROM streams
    WHERE type = 'type1' ORDER BY stamp DESC,
  ...,
  SELECT id, stamp, type FROM streams
    WHERE type = 'typeN' ORDER BY stamp DESC
) ORDER BY stamp DESC LIMIT 10 OFFSET 0;
```

问题出现了，我们只排序了前 10 条 ID 记录，但事实上查询设计器实现了整个合并，进而在最后再做其他条件限定。对每个类型进行一个简单的堆排序就可以找到前 10 个 ID 后就“提前退出”。在我们的例子中，每个类型都可以有上万个 ID，因此要实现整个列表排序合并是非常缓慢的，并且没必要。事实上，我们还写了一个自定义的 PL/Python 脚本，使用一系列查询来执行堆排序，查询如下所示：

```
SELECT id, stamp, type FROM streams
  WHERE type = 'typeN'
  ORDER BY stamp DESC LIMIT 1 OFFSET 0;
```

如果我们以 typeN 结束(它是堆中下一个最近的)，可运行另一个查询：

```
SELECT id, stamp, type FROM streams
  WHERE type = 'typeN'
  ORDER BY stamp DESC LIMIT 1 OFFSET 1;
```

在几乎所有的例子中，这种方法都胜于原始 SQL 实现和查询设计器策略。使用 Python 处理方法比使用 SQL 的最坏情况要快一个数量级。我们逐渐发现我们都需要比查询设计器更聪明。

同样地，此时我们真的需要另一种解决方案。

有了 HBase 的日子

基于 RDBMS 的系统通常能正确实现我们的需求，问题在于可伸缩性。开始重点考虑伸缩性和性能而不是正确性时，这种简捷性就结束，继而开始在可能的所有地方进行优化。一旦开始实现自己的解决方案来解决数据问题，RDBMS 的开销和复杂性就会成为障碍。存储层的抽象和 ACID 约束都是巨大的障碍和花费，不可能在每

次扩展时都能无限支付下去。Hbase 就是一个分布的、面向列的、排序的 map 存储，仅此而已，唯一需要从用户那里抽象出来的部分就是分布情况，这正是我们不想处理的。在另一方面，商业逻辑是非常专业化和最优化的。尽管 Hbase 不能解决我们所有的问题，但是比起从前，已经可以更好地解决这些问题，并依赖 Hbase 来扩展我们的存储，而不是逻辑。能专注于我们的应用和逻辑，而不是数据扩展本身，这确实是一个非常大的突破。

目前，我们已经有上亿行和上万列。更令人兴奋的是，我们可以放心地存储几十的行和几百万的列了。

12.7 实践

下面讨论用户使用比基本示例更复杂的 Hbase 实例时的一些常见问题。

12.7.1 版本

首先确保运行的是 Hadoop 和 Hbase 兼容的版本。兼容版本有它们共有的主版本和副版本编号。尽管 Hbase0.18.0 不能与 Hadoop0.19.0 版本集群进行对话(它们的副版本号不同)，但是 Hbase0.19.2 可以在 Hadoop0.19.1 的 HDFS 上运行。如果幸运，不兼容的版本会抛出异常，指出版本不符。如果它们不能有效地相互通信来传递版本，你会发现你的 Hbase 集群将无限期地挂起，直到被重启。如果 Hbase 或 Hadoop 较老的版本没有卸载干净，还存在于 CLASSPATH 中，在升级时就可能发生不匹配异常或 Hbase 挂起。

12.7.2 Hbase 和 HDFS 的爱与恨

Hbase 对 HDFS 的使用与 MapReduce 对它的使用非常不同。通常，在 MapReduce 中，HDFS 文件是打开的，它们的内容由 map 任务传递，然后再关闭。在 Hbase 中，数据文件从集群启动开始一直保持着打开状态，这样避免了我们在每一次访问时打开文件的消耗。因此 Hbase 遇到的问题不是 MapReduce 客户端通常遇到的。

- *文件描述信息块的用尽*

  由于我们在一个加载的集群上让文件保持打开状态，所以不久就会达到系统和 Hadoop 的使用极限。举例来说，假设我们有一个拥有三个节点的集群，每个节点都运行一个数据节点实例和一个区域服务器，同时我们正在向一个分布在

100 个区域、拥有 10 个列族的表上传数据。假设每一个列族平均有两个刷新文件。计算一下，每次我们都有 100*10*2，即 2000 个文件打开着。这个会加到其他由扫描器、Java 库消耗的信息描述符的总数上。每一个打开的文件至少消耗一个远程数据节点的描述符。每个处理器默认的文件描述符个数是 1024，当我们超过了这个上限(ulimit)，我们将在日志中看到“Too many open files”的报错，但是通常我们最先看到的是 Hbase 的不确定行为。固定的查询增加了 ulimit 文件描述符的总数。[①]可以确信 Hbase 在查找区域服务器日志的开始几行时，有足够的文件描述符。它将重要内容发送到正在使用的 JVM，以及类似于文件描述符的上限 ulimit 这样的环境设置上。

- *数据节点的线程用尽*

  同样的，Hadoop 数据节点也有一个同时运行 256 个线程的上限。假设有先前项目中引用过的相同的表数据，很容易发现为什么我们会较早到达上限。将它分配在数据节点中，那么在写操作时，每一次建立与文件块的连接都要消耗一个线程。[②]如果查看数据节点日志，会发现一个报错，类似于“xceiverCount 258 exceeds the limit of concurrent xcievers 256”但是同样的，在进入日志前，你会发现 Hbase 运行不正常。将 dfs.datanode.max.xcievers(注意属性名不要拼错)增加到 HDFS 中，然后重启集群。[③]

- *不良块*

  如果在访问时，服务器负载已经很重，那么由长期运行的区域服务器控制的 DFSClient 会将文件块标记为不良。由于数据块在默认情况下会被复制三次，因此区域服务器 DFSClient 会移到下一个副本。但是如果这个副本在重负载期间被访问，则三个副本中的两个数据块会被标为不良。如果第三个块也变为不良，我们就会在区域服务器日志中看见这样的报错“No live nodes contain current block”(没有活动的节点包含当前块)。在启动时，随着区域的开启和部署，会出现许多变化和竞争。在最极端的情况下，“No live nodes contain current block”(没有活动的节点包含当前块)会很快出现。注意，该配置需要设置在 Hbase DFSClient 能看得到的地址：设置在 hbase-site.xml 中或通过链接

① 参见 Hbase 常见问题解答，了解如何提高集群的上限(*http://wiki.apache.org/hadoop/Hbase/FAQ*)。

② 参见 Hadoop 和 HDFS 中 HADOOP-3856 的异步 IO 处理。

③ 参见 Hbase 故障排除指南，进一步了解此问题。网址为 *http://wiki.apache.org/hadoop/Hbase/Troubleshooting*。

hadoop-site.xml(或近期版本的 hdfssite.xml)将其设置到 HBase 的 conf 目录。[①]

12.7.3 用户界面

Hbase 通过在主机上运行 Web 服务器来提供运行集群状态的视图。默认情况下，在 60010 端口监听。主机用户界面显示了一些基本属性的列表，如软件版本、集群负载、请求等级、集群表的列表和参与的区域服务器。在主机用户界面上点击一个区域服务器，就能连到运行在该区域服务器上的 Web 服务器。它列出了这个服务器上的区域以及基本的指标，例如消耗资源量和请求等级。

12.7.4 度量

Hadoop 有一个度量体系，可以每隔一段时间向 context 发送一些重要信息。启用 Hadoop 度量，将其绑定到 Ganglia，这样能够扩展视图，看到集群当前和最近发生的事情。Hbase 同样有它自己的度量：请求等级、重要信息数量和资源使用数等。它们可以由 Hadoop 的上下文获得。参见 Hbase conf 文件下的 hadoop-metrics.properties 文件。[②]

12.7.5 架构设计

Hbase 表和 RDBMS 里的表很像，唯一不同的是它的单元被标有版本，行是经过排序的，并且列可以被客户端随时增加到已有的对应列族格。在设计 Hbase 的架构时，这些因素都要考虑在内。在设计架构时，另一个需要记住的特性是，为面向列(组)的存储方式(如 Hbase)定义的属性，要能够容纳宽的、稀疏的表，且不增加开销。[③]

联接

在 Hbase 中，没有本地数据库连接能力，但是超宽表可以做到这一点，因此也就不需要从二级或三级表中进行联接。宽行有时可以容纳对应于特定主键的所有数据。

---

① 参见 Hbasae 故障排除指南，进一步了解此问题。网址为 *http://wiki.apache.org/hadoop/Hbase/Troubleshooting*。

② 确实，尽管该文件用来建立 Hbase 度量，但它以 Hadoop 命名。

③ Daniel J. Abadi 编写的“Column-Stores for Wide and Sparse Data”，网址为 *http://ab.csail.mit.edu/projects/cstore/abadicidro.pdf*。

行的键

得花一些时间来设计行的键。在本章气象数据的例子中，混合型行键有站点作为前缀，使其能够通过站点来对气温进行分组。逆序的时间戳后缀使气温可以按从最新到最老的时间顺序扫描。一个灵活的混合型键可以使集群数据的访问更容易。

设计混合型键时，可能只能从 0 开始给组件标号，这样行键才能被正确排序。否则，在只考虑字节排序的情况下，会遇到 10 排在 2 前面的情况(02 排在 10 前面)。

如果键是整数，一般使用二进制表示，而不是使用数字的字符串表示，它占用的空间更少。

第 13 章

ZooKeeper 简介

在这本书中，到现在我们已经学习了大规模数据的处理。而本章的内容有些不同，介绍的是使用 Hadoop 的分布式协作服务来建立传统的分布式应用，即 ZooKeeper。

要写一个分布式应用是非常困难的，主要原因是因为局部故障。一个消息通过网络在两个节点之间传递时，网络如果发生故障，发送方并不知道接收方是否收到了这个消息。它可能在网络故障前就收到了，也可能没有收到，又或者可能接收方的进程死了。发送方了解情况的唯一方法就是再次连接接收方，并向它进行询问。这就是局部故障：我们根本不知道操作是否失败。

ZooKeeper 并不能阻止局部故障的发生，因为它们的本质是分布式系统。它当然也不会隐藏局部故障。[①]ZooKeeper 的目的是提供一些工具集，用来建立安全处理局部故障的分布式应用。

ZooKeeper 还有如下特性。

简易

ZooKeeper 的核心就是一个精简的文件系统，它提供一些简单的操作以及一些附加的抽象(例如排序和通知)。

---

① 这是 J. Waldo 等人所提过的，“分布式计算笔记”(1994)，*http://research.sun.com/techrep/1994/smli_tr-94-29.pdf*。其中提到，分布式编程与本地编程截然不同，而且这种区别很难简单地用书面定义。

易表达

ZooKeeper 的原型是一个丰富的集合，它们是一些已建好的块，可以用来构建大型的协作数据结构和协议，例如：分布式队列、分布式锁以及一组对等体的选举。

高可用性

ZooKeeper 运行在一些机器集合上，同时它们被设计成可用性较高的，因此应用程序可以依赖它。ZooKeeper 可以帮助你的系统避免单点故障，从而建立一个可靠的应用程序。

简化松散耦合交互

ZooKeeper 的交互支持参与者之间并不了解对方。举例来说，ZooKeeper 可以被当作一种公共机制，使得即使进程彼此不知道对方的存在(或是网络详情)也可以相互发现并且交互。对等方可能甚至不是同步的，因为一个进程可能在 ZooKeeper 上留下信息，直到它被关闭以后才被另一个进程读到。

ZooKeeper 是一个库

ZooKeeper 提供了一个开源的、共享的执行存储，以及通用协作模式的方法，分担了每个程序员写通用协议的负担(这些通常很难写正确)。随着时间的推移，人们可以增加和改进这个库来满足自己的需求。

ZooKeeper 的执行能力也非常强。在 ZooKeeper 诞生的地方—— Yahoo！，它的吞吐量标准已经达到了大约每秒 10 000 基于写操作的工作量。对于读操作的工作量来说，它的吞吐量标准还要高几倍。

13.1 ZooKeeper 的安装和运行

第一次尝试 ZooKeeper 时，最简单的是使用一个 ZooKeeper 服务器，在单机模式下运行，比如可以在开发机器上运行。ZooKeeper 要求用 Java 6 运行，因此必须确保已经安装。在 Windows 上运行 ZooKeeper(Windows 只能作为开发平台，而不是应用平台)，也需要安装 Cygwin。

从 Apache Zookeeper 发布页面下载一个 ZooKeeper 的稳定版本，*http://hadoop.apache.org/zookeeper/releases.html*，将它解压到对应的地址：

```
% tar xzf zookeeper-x.y.z.tar.gz
```

ZooKeeper 提供一些二进制用来运行并与服务交互。将包含二进制的这个库加到命令行路径也很方便：

```
% export ZOOKEEPER_INSTALL=/home/tom/zookeeper-x.y.z
```

```
% export PATH=$PATH:$ZOOKEEPER_INSTALL/bin
```

运行 ZooKeeper 服务前，需要创建一个配置文件。这个配置文件一般命名为 zoo.cfg，并且位于 conf 的子目录(你也可以将它放在/etc/zookeeper 下，或者在由 ZOOCFGDIR 环境变量定义的目录下)。示例如下：

```
tickTime=2000
dataDir=/Users/tom/zookeeper
clientPort=2181
```

这是一个标准的 Java 属性文件，在这个例子中被定义的三个属性是在单机模式下运行 ZooKeeper 的最少要求。简单来说，tickTime 是 ZooKeeper 中的基本时间单元(指以毫秒为单位)，dataDir 是 ZooKeeper 存储持续数据的本地文件系统的地址，clientPort 是 ZooKeeper 监听客户端连接的端口(通常选择 2181)。需要将 dataDir 改变成适合具体系统的设置。

有合适的配置定义后，我们现在可以启动一个本地 ZooKeeper 服务器了：

```
% zkServer.sh start
```

为了检查 ZooKeeper 是否在运行，将 ruok 命令("Are you OK？")发送到使用 nc(telnet 也可以)的客户端：

```
% echo ruok | nc localhost 2181
Imok
```

ZooKeeper 回应了"I'm OK."。还有其他的四字命令可以用来与 ZooKeeper 交互。它们大多是查询：dump 列出会话列表和当前 znode；envi 列出服务器属性；reqs 列出未响应的请求；stat 列出服务数据和连接着的客户端。也可以更新 ZooKeeper 的状态：srst 重新设置服务数据，并且可以满足从运行 ZooKeeper 服务器的主机发来的停止关闭着的 ZooKeeper 的要求。

使用 JMX 支持，可以满足更广泛的 ZooKeeper 监听需求，相关说明可参见 ZooKeeper 文档，网址为 *http://hadoop.apache.org/zookeeper/*。

13.2 范例

假设一组服务器，向客户端提供一些服务。我们希望客户端能够定位其中一台服务器，使其能够使用这些服务，挑战之一就是维护这组服务器列表。

此成员列表明显不能存在网络中的单个节点上，因为如果那个节点发生故障，就意

味着是整个系统的故障(我们希望这个列表有很高的可用性)。假设我们有了一个健壮的方法来存储此表，仍然有别的问题存在：当它出故障，该怎样将此表从这个服务器上移走。一些进程需要负责移走故障服务器，但我们必须注意，不可以是服务器本身的进程，因为那时它们已经不再运行！

我们所描述的并不是一个被动的分布式数据结构，而是主动的，而且当一些外部事件出现时，它还可以改变 entry 的状态。ZooKeeper 提供这种服务，我们来看看怎样用它来建立这种组成员制应用(正如先前提到的)。

13.2.1 ZooKeeper 中的组成员制

理解 ZooKeeper 的一种方法是将它视为一个提供高可用性的文件系统。它没有文件和目录，但是有一个统一概念的节点，叫 znode，作为数据(如文件)以及其他 znode(比如目录)的容器。znode 来自于一个层次(级)命名空间。传统的建立成员列表的方法是以小组的名称创建一个父 znode，同时子 znode 使用的是组成员(服务器)的名称。如图 13-1 所示。

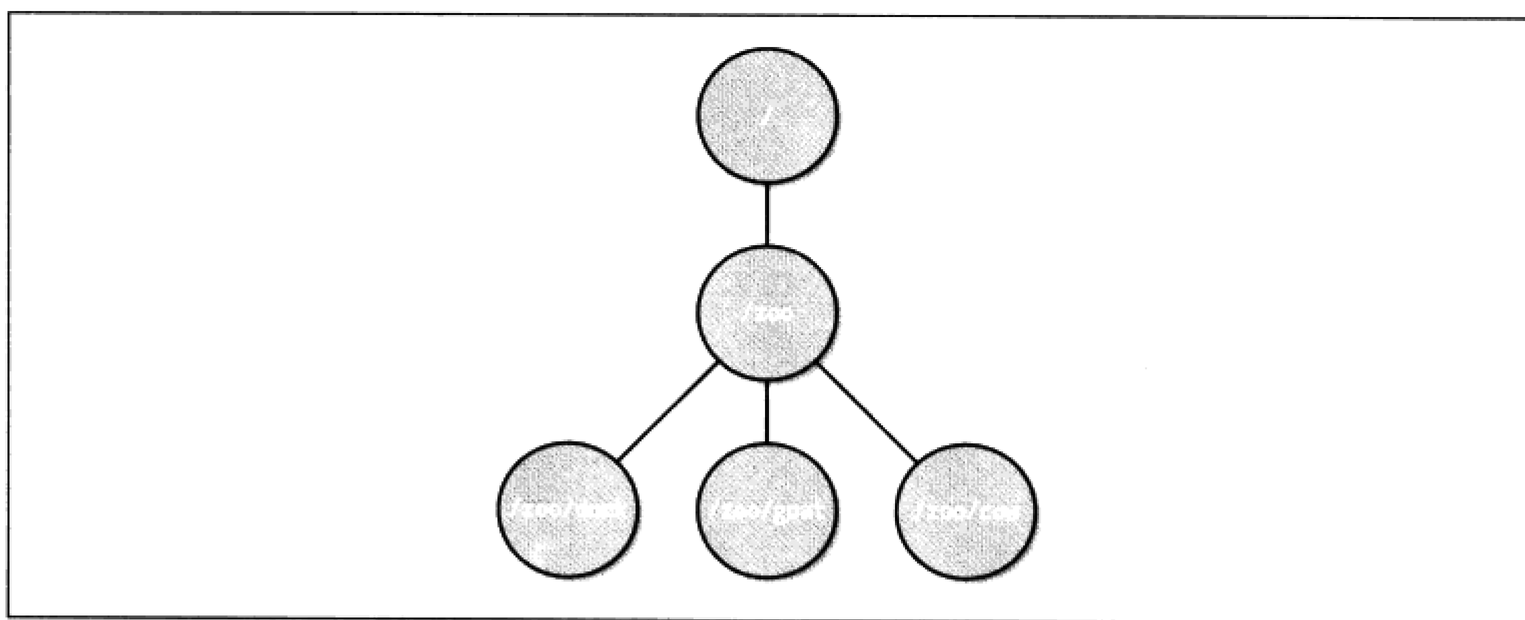


图 13-1：ZooKeeper 的 znode

在这个例子中，我们不在任何一个 znode 中存数据，而是存储在应用中，你可以想像在 znode 中存储关于成员的数据，比如主机名。

13.2.2 创建组

下面要写一个为组创建一个 znode 的程序(本例中为/zoo)来介绍 ZooKeeper 的 Java API。参见例 13-1。

例 13-1：创建一个 znode，代表 ZooKeeper 中的一个组

```
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();
  }
}
```

main()方法运行后，创建了一个 CreateGroup 实例，并且调用了它的 connect() 方法。此方法实例化一个新的 ZooKeeper 对象，它是客户端 API 的主类，同时维护着客户端与 ZooKeeper 服务的连接。这个构造函数有三个参数：第一个是 ZooKeeper 服务的主机地址(以及可选端口，默认为 2181)；[①]第二个是每毫秒的会话超时(我们设置成 5 秒)，之后我们会详细介绍；第三个是 Watcher 对象的实例。Watcher 对象接收 ZooKeeper 的响应，并通知它各种事件。在这个例子中，CreateCroup 是一个 Watcher，因此我们将它传递给 ZooKeeper 的构造函数。

当一个 ZooKeeper 实例被创建后，它启动一个线程连接到 ZooKeeper 服务。对构造函数的响应返回得很快，因此在使用 ZooKeeper 对象前等待连接建立非常重要。我们利用 Java 的 CountDownLatch 类(在 java.util.concurrent 包中)来阻塞，直到 ZooKeeper 实例准备好。这就是 Watcher 进来的地方。Watcher 接口只有一个操作：

```
public void process(WatchedEvent event);
```

客户端连接到 ZooKeeper 以后，Watcher 的 process()方法会被调用，并收到一个事件，表明连接已完成。在收到连接事件时(由 Wathcer.Event.KeeperState 枚举型表示，并带有值 SyncConnected)，我们就使用 CountDownLatch 中的 countDown()操作减掉一个计数。Latch 被创建时，计数为 1，代表需要在释放所有等待线程前发生事件的数量。在调用一次 countDown()函数后，此计数器会归零，await()操作返回。

当 connect()操作已经返回后，下一个被调用的 CreateGroup 操作是 create()方法。在这个方法中，我们使用 create()方法在 ZooKeeper 实例上创建一个新的 ZooKeeper 的 znode。它含有的参数是路径(由一个 string 表示)、znode 的内容(一个字节数组，这里是 null)、一个访问控制列表(缩写为 ACL，在这里是一个完全打开的 ACL，允许任何客户端对 znode 进行读写)以及需要被创建的 znode 属性。

znodes 可能是临时的或是永久的。一个临时性 znode 在创建它的客户端断开连接后(可能是明确的断开连接，也可能是客户端由于种种原因中断)就会被 ZooKeeper 服务删除。相反，一个永久型 znode 不会在客户端断开连接后被删除。我们希望代表一个组的 znode 能够比创建它的程序的生命周期更长，因此我们创建一个永久型 znode。

---

① 对于一个复合的 ZooKeeper 服务来说，这个实参是由逗号分隔的服务器列表(主机和可选端口)的集合。

create()操作返回的值是由 ZooKeeper 创建的路径。我们用它来表示路径已经成功创建的消息。下面我们来看看，由 create()返回路径的方法，与我们关注顺序的 znode 时被传递到此操作的方法有什么不同。

为了看这个程序的运行，我们需要使 ZooKeeper 在本地机器上运行，然后再输入：

```
% export CLASSPATH=build/classes:$ZOOKEEPER_INSTALL/*
:$ZOOKEEPER_INSTALL/lib/*:\ $ZOOKEEPER_INSTALL/conf
% java CreateGroup localhost zoo
Created /zoo
```

13.2.3 加入组

这个应用的下一部分就是一个将成员注册到组里的程序。每一个成员会作为程序运行并加入组一个组。当程序退出时，它必须从这个组上移除。我们可以在 ZooKeeper 的命名空间下，创建一个临时性的 znode 来代表它。

JoinGroup 程序执行了这一操作，见例 13-2。创建并加入 ZooKeeper 实例的这一逻辑已经实现在一个基础类 ConnectionWatcher 中，见例 13-3。

例 13-2：加入组的程序

```
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);
  }
}
```

例 13-3：一个帮助类，等待到 ZooKeeper 的连接建立

```
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 `join()` 操作中，它创建了一个临时性的 znode 作为组 znode 的子节点，然后通过 sleep 模拟着做一些工作直到进程被强制终止。之后你将看到，在被终止以后，临时性的 znode 将被移除出 ZooKeeper。

13.2.4 列出组成员

现在我们需要一个程序来找出一个小组里的成员(参见例 13-4)。

例 13-4：用来列出组成员的程序

```
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()操作中，我们通过一个 znode 路径调用 getChildren()操作，结合查看标志来找到这个 znode 的子节点的路径，将它们打印出来。当 znode 的状态改变时，在 znode 上放查看标记会导致注册过的 Watcher 被触发。尽管在这里我们不使用它，观察一个 znode 的子节点可以允许程序获得连接或离开该小组的成员，或是小组被删除的消息。

小组的 znode 不存在时抛出异常，我们将捕获 KeeperException.NoNodeException。

来看看 ListGroup 的运行。正如预期的一样，由于我们还没有加任何成员，所以这个 zoo 小组是空的：

```
% java ListGroup localhost zoo
No members in group zoo
```

我们可以使用 JoinGroup 来增加一些成员。因为它们不会自己中断(由于 sleep 声明)，所以我们将它们作为后台进程发布。

```
% java JoinGroup localhost zoo duck &
```

```
% java JoinGroup localhost zoo cow &
% java JoinGroup localhost zoo goat &
% goat_pid=$!
```

最后一行保存着运行这个程序的 Java 进程的进程 ID，它增加 goat 作为一个成员。我们需要记住这个 ID，这样才能在以后检查了所有成员后停止这个进程：

```
% java ListGroup localhost zoo
goat
duck
cow
```

要移除一个成员，我们就停止它的进程：

```
% kill $goat_pid
```

在几秒以后，它将会因为进程的 ZooKeeper 会话被中断而从这个小组中消失(超时时间被设置为 5 秒)，同时与它相关的临时性节点也将被删除：

```
% java ListGroup localhost zoo
duck
cow
```

让我们回过头来看看现在我们了解了些什么。我们可以建立一张参与到分布式系统的小组节点的列表。这些节点可能彼此并不知道对方。如果有想要使用此列表里的节点来完成任务的服务，它可以在节点不知道它存在的情况下发现它们。

最后要注意，与节点通信时，组成员制并不能处理网络故障。即使节点是一个组成员，通信也可能失败。我们在面对这种故障的时候还是要使用常规方法来处理。

ZooKeeper 命令行工具

ZooKeeper 有一个命令行工具用来与 ZooKeeper 命名空间进行交互。我们可以用它来列出/zoo znode 下的 znode，如下所示：

```
% zkCli.sh localhost ls /zoo
Processing ls
WatchedEvent: Server state change. New state: SyncConnected
[duck, cow]
```

不需要参数就可以使用这个命令来获取用法。

13.2.5 删除一个组

为了更好地理解这个例子，让我们来看看怎样删除一个组。ZooKeeper 类提供一个带有路径和版本号的 delete()方法。ZooKeeper 只在要删除的 znode 的版本号与已经定义过的版本号一样时才会删除该 znode。乐观锁机制使客户端能够发现 znode 修改的冲突。你可以不用管版本核对，也不用管 znode 原来的版本号，而使用版本号-1 来删除该 znode。

在 ZooKeeper 中没有递归删除操作，因此必须在删除父节点前先将子节点删除。我们在 DeleteGroup 类中完成了这一操作，它可以在删除一个组时删除它所有的成员(例 13-5)。

例 13-5：删除一个组及其成员的程序

```
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，可以删除早期创建的 zoo 小组：

```
% java DeleteGroup localhost zoo
% java ListGroup localhost zoo
Group zoo does not exist
```

13.3 ZooKeeper 服务

ZooKeeper 是一种高可用的、高性能的协作服务。在这一小节，我们来看一看这个服务拥有的特性：它的模型、操作以及执行。

13.3.1 数据模型

ZooKeeper 维护着一个分级节点树，znode。一个 znode 存储着数据并且有一个相关的 ACL。由于 ZooKeeper 是为协作而设计的(通常使用小数据文件)，不是大容量数据存储，因此任何一个 znode 的数据存储量的上限是 1 MB。

数据访问是原子的。客户端读取存储在 znode 内的数据，这一步所做的不仅仅是读取一部分数据，数据将被整体传送(或者读取失败)。同样的，一个写操作将替换与该 znode 相关的所有数据。ZooKeeper 保证了写操作成功或者失败，不可能出现部分数据被客户端写入。ZooKeeper 不支持追加操作、这些特性与 HDFS 截然不同，HDFS 是被设计成大容量数据存储的，流式数据访问的，并且提供追加操作。

Znode 以路径标注，在 ZooKeeper 中被表示成由反斜杠分隔的 Unicode 字符串，如同 Unix 中的文件路径。路径必须是绝对的，因此它们必须由反斜杠字符开头。除此以外，它们必须是唯一的，也就是说每一个路径只有一个表示，因此这些路径不能改变。举例来说，在 Unix 中，一个路径为/a/b 的文件等价于路径为/a/./b 的文件，因为“.”指的是它正在与当前路径的目录进行交互。在 ZooKeeper 中“.”没有这个特殊的意思，而是一个合法的路径元素(对于“..”表示当前目录的上级目录也是一样)。

路径元素是由 Unicode 字符组成的，但是也有一些约束(在 ZooKeeper 参考文档中有详细说明)。字符串“zookeeper”是保留字，因此不能用来作为路径元素。特别的，ZooKeeper 使用/zookeeper 子树来存储管理信息，如一些配额信息。

注意，路径并不是 URI，它们是由 java.lang.String 表示在 Java API 中的，而不是由 Hadoop Path 类(或是 java.net.URI 类)。

Znode 有一些对于建立分布式应用非常有用的特性，我们将在后文讨论。

临时性 znode

znode 可以分为两种：临时的或永久的。znode 类型是在创建时确定的，并且不能改变。一个临时性 znode 是在创建它的客户端的会话结束时由 ZooKeeper 删除。相反，一个永久型 znode 并不依赖于客户端会话，而且只在客户端明确删除它的时候才会被删除(不一定是创建它的那个客户端)。一个临时性的 znode 不应该有子节点，即使是临时性的子节点。

即使临时性的 znode 绑定在客户端会话上，它们对所有的客户端都是可见的(当然依赖于它们的 ACL 机制)。

临时性的 znode 对于建立那些需要知道什么时候某些分布式资源可以使用的应用非常有效。本章先前提到的例子就使用临时性 znode 实现了组成员服务，因此任何进程都可以在任何特定时刻发现组成员。

序号

一个顺序的 znode 由 ZooKeeper 给定一个序号作为其名称的一部分。如果一个 znode 在创建时设置了顺序标志，那么一个单调递增计数器的值将会附加到它的名字上。

举例来说，如果客户端请求以/a/b-为名创建一个顺序的 znode，那么 znode 事实上可能以/a/b-3 的名字被创建[①]。如果之后另一个顺序型 znode 以/a/b-名被创建，那么它将被给定一个唯一的名字，含有一个更大的计数器值，如/a/b-5。在 Java API 中，真正给予顺序型 znode 的路径将会作为 create()调用的返回值传递给客户端。

序号可以用来加强分布式系统全局上的事件顺序，同时客户机也可以使用它们来知道顺序。在 13.4.3 节，将介绍怎样使用顺序型 znode 来建立共享锁。

Watch

在 znode 有改变时，Watch 使客户端了解相应的信息。Watch 由 ZooKeeper 服务的操作来设置，同时由服务的其他操作来触发。举例来说，一个客户端可能调用了 znode 上的 exists 操作，同时在这个节点上加了一个 Watch。如果这个 znode 不存在，那么 exists 操作将会返回 false。如果在一段时间后，这个 znode 由另一台客户

① 一般来说(但不是必须的)顺序型节点的路径名尾部都带有一个破折号，使得序号(被应用程序)便于阅读和语法分析。

端创建了，那么 Watch 将被触发，通知第一台客户端 znode 被创建的消息。后文将清楚说明什么操作触发了这一切。

Watcher 只被触发一次。[1]为了获得多次提醒，客户端需要再次注册 Watch。在先前的例子中，如果客户端想要进一步获得这个 znode 存在的信息(如当它被删除时接到通知)，它需要再次调用 exists 操作来设置一个新的 Watch。

第 13 章的“配置服务”小节有一个例子，它证明了怎样使用 watches 在集群中更新配置。

13.3.2 操作

在 ZooKeeper 中有 9 个基本操作，在表 13-1 中列出。

表 13-1：ZooKeeper 服务中的操作

操作	描述
create	创建 znode(父 znode 必须已经存在)
delete	删除 znode(znode 没有子节点)
exists	测试 znode 是否存在并获取它的元数据
getACL，setACL	为 zode 获取/设置 ACL
getChildren	获取 znode 的所有子节点的列表
getData，setData	获取/设置 znode 的相关数据
sync	使客户端的 znode 视图与 ZooKeeper 同步

更新 ZooKeeper 的操作是有限制的。delete 或 setData 必须明确需要更新的 znode 的版本号(在先前调用的 exists 可以找到)。如果版本号不匹配，更新将会失败。更新操作是非阻塞式的，因此客户端如果失去了一个更新(由于另一个进程在同时更新这个 znode)，它可以在不阻塞其他进程执行的情况下，选择重新尝试或进行其他操作。

尽管 ZooKeeper 可以被看作是一个文件系统，但是它出于便利，摒弃了一些文件系统的操作原语。因为文件非常小并且是整体读写的，所以不需要提供打开、关闭或是寻地操作。

① 会话事件的回复不需要再次注册。

注意：sync 操作与 POSIX 文件系统中的 fsync()操作并不一样。正如先前所提到的，在 ZooKeeper 中，写操作是原子的，而且一个成功的写操作必须保证是被写到大部分 ZooKeeper 服务器的永久存储上。然而，对于 ZooKeeper 来说，读操作是可以滞后最新 ZooKeeper 服务的状态的，同时 sync 操作允许客户端自身更新。这个问题将在 13.3.4 节详细说明。

API

有两种核心语言绑定在 ZooKeeper 客户端上，一个是 Java，另一个是 C。对于 Perl、Python 和 REST 客户端来说，还有 contrib。对每个绑定来说，可以选择同步或异步来执行操作。我们已经看到了异步的 Java API。下面是 exists 操作的签名，它返回了一个封装有 znode 元数据的 Stat 对象，如果 znode 不存在，就返回 null。

```
public Stat exists(String path, Watcher watcher) throws
    KeeperException,InterruptedException
```

等价的异步操作也在 ZooKeeper 类中可以找到，如下：

```
public void exists(String path, Watcher watcher, StatCallback
cb, Object ctx)
```

在 Java API 中，由于操作的结果由回调过程来传递，因此所有的异步方法都是 void 的返回类型。调用方传递一个回调过程来执行，它的方法将在从 ZooKeeper 得到响应后被调用。在这个例子中，回调过程来在 StatCallback 接口中，如下所示：

```
public void processResult(int rc, String path, Object ctx, Stat
stat);
```

rc 参数指的是在 KeeperException 中被定义的返回码。非零码代表异常，stat 参数将为 null。Path 和 ctx 参数指的是从客户端传递给 exists()方法的相关参数，并且可以被用来确定回调过程来回应的是哪个请求。当路径没有给定充分的内容可以用来区别请求时，ctx 参数可以是客户端使用的任意对象。如果不需要，可以被设为 null。

事实上有两个 C 共享库。一个是单线程库 zookeeper_st，只支持异步 API，而且专门用于 pthread 库不可获得或是静止的平台服务。大部分开发人员都使用多线程库 zookeeper_mt，因为它既支持同步 API，也支持异步 API。若要详细了解怎样建立并使用 CAPI，请参考 ZooKeeper 分布式的 src/c 库中的 README 文件。

选择同步还是异步 API？

两种 API 都提供相同的功能，因此使用哪一种主要由自己决定。举例来说，如果你有事件驱动程序模型的话，那么异步 API 更合适。

异步 API 使你能管线化请求，在某些用例中可以提供更好的吞吐量。想像一下，如果你想读取大批 znode 并且独立地处理它们。使用同步 API 的话，那么每个读操作都将会阻塞直到它返回的那一刻；相反，使用异步 API 的话，可以非常快地连续执行所有的异步读操作，并且在它们返回时在不同的线程中处理响应。

Watch 触发器

读操作 `exists`、`getChildren` 和 `getData` 都被设置了 watch，并且这些 watch 都由写操作来触发：`create`、`delete` 和 `setData`。ACL 操作并不参与到 watch 中。当 watch 被触发时，watch 事件被生成，它的类型由 watch 和触发它的操作共同决定。

- `exists` 操作上的 watch 在被监视的 znode 创建、删除或数据更新时被触发。
- `getData` 操作上的 watch 在被监视的 znode 删除或数据更新时被触发。在创建时不能触发，因为只有 znode 一定存在，`getData` 操作才会成功。
- `getChildren` 操作上的 watch 在被监视的 znode 的子节点创建或删除时，或是当这个 znode 自身删除时被触发。可以通过查看 watch 事件类型来区分是 znode 还是它的子节点被删除：`NodeDelete` 表示 znode 被删除，`NodeChildrenChanged` 表示子节点被删除。

表 13-2 总结了所有组合。

表 13-2：Watch 创建操作及其相应的触发器

创建 watch	Watch 触发器 create znode	 child	 delete znode	 child	 setData
exists	NodeCreated		NodeDeleted		NodeDataChanged
getData			NodeDeleted		NodeDataChanged
getChildren		NodeChildrenChanged	NodeDeleted	NodeDeletedChanged	

watch 事件包括了事件所涉及的 znode 的路径，因此对于 `NodeCreated` 和 `NodeDeleted` 事件来说，根据路径就可以简单地区分是哪个 znode 被创建或是被删除了。为了查询在 `NodeChilderenChanged` 事件后哪个子节点被改变了，需要再次调用 `getChildren` 来获得新的 `children` 列表。同样地，为了查询 `NodeDataChanged` 事件后产生的新数据，需要调用 `getData`。在两种情况下，znode 可能在获取 watch 事件或执行读操作这两种状态下切换，在写应用程序时必须记住这一点。

ACL

创建一个 znode 会产生一个 ACL 列表，它决定着谁可以对 znode 进行某些操作。

ACL 依赖于验证，客户端通过这一进程将自己绑定到 ZooKeeper 上。ZooKeeper 提供了几种验证模式。

digest
　　客户端由用户名和密码验证。

host
　　客户端由主机名验证。

ip
　　客户端由 IP 地址验证。

在与 ZooKeeper 建立会话之后，客户端会验证自己。验证是可选的，尽管 znode 的 ACL 可能会请求一个被验证过的客户端，这时，客户端必须验证自己来访问节点。下面有个 digest 模式验证的例子，它使用了用户名和密码：

```
zk.addAuthInfo("digest", "tom:secret".getBytes());
```

ACL 是由验证模式、对应模式的身份以及一套许可的组合。举例来说，如果我们想要客户端在 example.com 域中对 znode 进行读访问，那么我们要对这个 znode 的 ACL 进行设置，使用 host 模式，带有 example.com 的 ID 和 READ 许可。在 Java 中，我们如下创建 `ACL` 对象：

```
new ACL(Perms.READ, new Id("host", "example.com"));
```

表 13-3 列出了完整的许可集合。注意，`exists` 操作并不受 ACL 许可控制，因此任何客户端都可以调用 `exists` 来查询 znode 的 `stat` 或是找到事实上并不存在的 znode。

表 13-3：ACL 许可

ACL 许可	允许的操作
CREATE	create(一个子 znode)
READ	getChildren
	getData
WRITE	setData
DELETE	delete(一个子 znode)
ADMIN	setACL

在 ZooDefs.Ids 类中有一些重定义的 ACL，其中包括 OPEN_ACL_UNSAFE，它给每个客户端所有的许可(除了 ADMIN 许可以外)。

除此之外，ZooKeeper 有一个可置换验证机制，它允许在需要情况下加入第三方验证系统。

13.3.3 执行

ZooKeeper 服务可以以两种模式运行。在单机模式下，只有一个 ZooKeeper 服务器，便于用来测试(甚至可以用来嵌入单元测试)。但是它没有高可用性和恢复性的保障。在工业界，ZooKeeper 以复合式模式运行在一组叫作 ensemble 的集群上。ZooKeeper 通过复制来获得高可用性，同时，只要 ensemble 中的大部分机器都运作，就可以提供服务。举例来说，在一个 5 节点 ensemble 中，可以在任何两台机器故障的情况下服务仍在运作，因为仍有三台机器在运作。注意，6 节点 ensemble 也只可以承受两台机器的故障，因为三台故障后还剩三台，不能满足 6 台中大部分不出故障的要求。因此我们通常为一个 ensemble 选择奇数台机器。

ZooKeeper 的思想非常简单：它所需要做的就是保证对 znode 树的每一次修改都复制到 ensemble 中的大部分机器上去。如果机器中的小部分出故障了，那么至少有一台机器将会恢复到最新状态，其他的则保存着副本，直到最终到达最新状态。

这个简单想法的实现并不是没有意义的。ZooKeeper 采用了 Zab 协议，它分为两个阶段，并且可能被无限地重复。

阶段 1：领导者选举

在 ensemble 中的机器要参与一个选择特殊成员的进程，这个成员叫作领导者，其他机器则叫作跟随者。在大部分(或指定额数)的跟随者与它们的领导者同步了状态以后，这个阶段才算完成。

阶段 2：原子广播

所有的写操作请求被传送给领导者，并通过广播将更新信息告诉跟随者。当大部分跟随者执行了修改后，领导者就提交更新操作，客户端将得到更新成功的回应。为获得一致性的协议被设计为原子的，因此无论修改失败与否，它都分两阶段提交。

ZooKeeper 使用 Paxos 吗？

不。ZooKeeper 的 Zab 协议与著名的 Paxos 算法不一样(Leslie Lamport, “Paxos Made Simple” ACM SIGACT News[分布式计算专栏] 32, 4 [总期号 121, 12 月 2001] 51 58)。Zab 与其是有相似之处，但在它操作的个别方面有着不同，比如依赖 TCP 进行消息顺序保证。

在 Benjamin Reed 和 Flavio Junqueira 的 “A simple totally ordered broadcast protocol” 一文中描述了 Zab。(2008 年，第二届关于大规模分布式系统和中间件专题研讨会的议程[LADIS’08]，美国纽约州的纽约城高地的 IBM 华生研究中心。出版在 ACM 国际会议议程系列，2009 ACM 报刊。ISBN: 978-1-60558-296-2)。

Google 的 Chubby Lock Service((Mike Burrows, “The Chubby Lock Service for Loosely-Coupled Distributed Systems,2006 年 11 月，*http://labs.google.com/papers/chubby.html*)是与 ZooKeeper 有着共同目标的，基于 Paxos 的服务。

如果领导者出故障了，剩下的机器将会再次进行领导者 election，并在新领导者被选出前继续执行任务。如果在不久以后老的领导者恢复了，那么它将以跟随者的身份继续运行。领导者选举非常快，由发布的结果所知，大约是 200 毫秒[①]，因此在选举时性能不会有明显的减慢。

所有在 ensemble 中的机器在更新它们内存中的 znode 树之前会先将更新信息写入磁盘。读操作请求可能由任何机器服务，同时，由于它们只涉及内存查找，因此非常快。

13.3.4 一致性

了解 ZooKeeper 实现的本质能够帮助我们了解服务的一致性保证。在 ensemble 中的领导者和跟随者非常聪明，跟随者通过更新号来滞后领导者，结果导致了只要大

① 由 Yahoo！报道，*http://hadoop.apache.org/zookeeper/docs/current/zookeeperOver.html*。

部分而不是所有的 ensemble 元素确认更新，就能被提交了。对于 ZooKeeper 来说，一个较好的智能模式是将客户端连接到跟着领导者的 ZooKeeper 服务器上。客户端可能被连接到领导者，但是它不能控制它，而且在如下情况时，甚至可能不知道。[①]参见图 13-2。

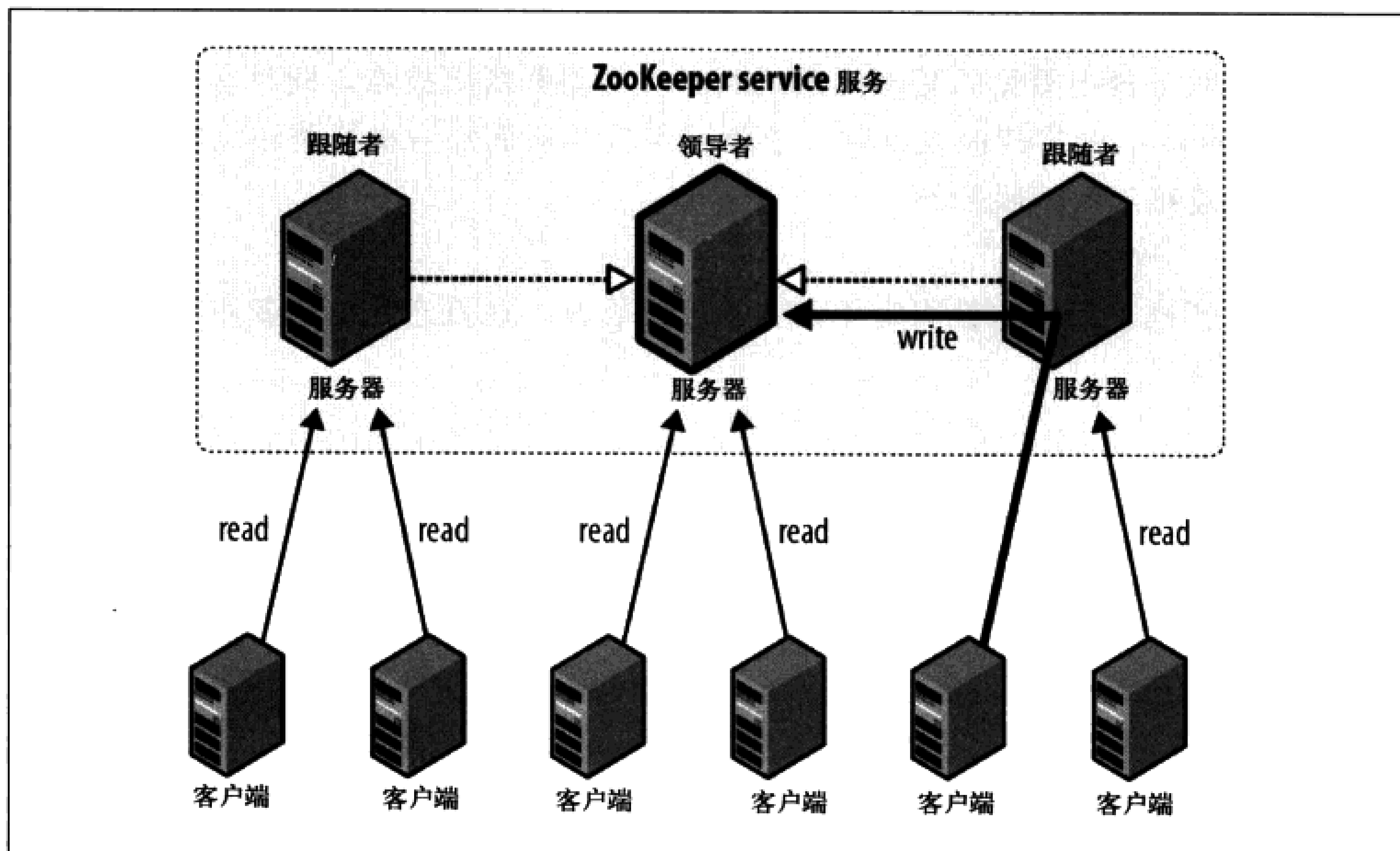


图 13-2：跟随者执行读操作，领导者提交写操作 2732

每一个对 znode 树的更新都会给定一个唯一的全局标识，叫 zxid(表示“ZooKeeper 事务 ID”)。更新是被排序的，因此如果 zxid 的 z1 比 z2 小，那么 z1 就比 z2 先执行。对于 ZooKeeper 来说，这是分布式系统中排序的唯一标准。

下面的内容来自 ZooKeeper 的设计，用来保证数据的一致性流。

顺序的一致性

任何一个客户端的更新都按它们发送的顺序排序，也就意味着如果一个客户端将 znode z 的值更新为值 a，那么在之后的操作中，它会将 z 更新为值 b，在客户端发现 z 带有值 b 之后，就不再会看见带有值 a 的 z(前提是对于 z 没有其他更新)。

① 可以通过配置 ZooKeeper 使领导者不接受客户端的连接。在这种情况下，它唯一做的就是协助更新，通过将 leaderServe 的属性设置成 no 实现。建议所应用的 ensemble 大于三台服务器。

原子性

更新不是成功就是失败，这意味着如果更新失败了，没有客户端会知道。

单系统映像

无论客户端连接的是哪台服务器，它与系统看见的视图是一样的。这就意味着，如果一个客户端在相同会话时连接了一台新的服务器，它将不会再看见比在之前服务器上看见的更老的系统状态。当服务器出故障，同时客户端尝试连接 ensemble 中的其他机器时，故障服务器的后面那台机器将不会接受连接，直到它连接到故障服务器。

容错性

一旦更新成功后，它就固定了，将不再被修改，这就意味着更新将会避免服务器的故障。

合时性

在任何客户端的系统视图上的时间间隔都是有限制的，因此它在超过几十秒的时间内不会过期。这也就意味着，服务器不会让客户端看一些过时的数据，而是关闭，强制客户端转到一个更新的服务器上。

由于性能的原因，读操作由 ZooKeeper 服务器的内存提供，而且不参与写操作的全局排序。这一特性可能会导致来自使用 ZooKeeper 外部机制交流的客户端与 ZooKeeper 状态的不一致。

举例来说，客户端 A 将 znode z 的值 a 更新为 a’，A 让 B 来读 z，B 读到 z 的值是 a 而不是 a’。这与 ZooKeeper 的保证机制是相容的(不允许的情况叫作“同步一致的交叉客户端视图”)。为了避免这种情况的发生，B 在读取 z 的值之前，应该先调用 z 上的 `sync`。`Sync` 操作强制 B 连接上的 ZooKeeper 服务器与 leader 保持一致，这样，当 B 读到 z 的值时，它将成为 A 设置的值(或者是之后的值)。

注意：容易混淆的是，sync 操作只能被异步调用。这样操作的原因是你不需要等待它的返回，因为 ZooKeeper 保证了任何接下去的操作将会发生在 sync 在服务器上执行以后，即使操作是在 sync 完成前被调用的。

13.3.5 会话

ZooKeeper 客户端与 ensemble 中的服务器列表配置相一致。在启动时，它尝试与表中的一个服务器相连接。如果连接失败了，它就尝试表中的其他服务器，以此类推，直到它最终连接到其中一个，或者当 ZooKeeper 的所有服务器都无法获得时，连接失败。

一旦与 ZooKeeper 服务器连接成功，服务器会创建与客户端的一个新的对话。每个会话都会有超时时段，这是应用程序在创建它时设定的。如果服务器没有在超时时段内得到请求，它可能中断这个对话。一旦对话被中断了，它可能不再被打开，而且任何与会话相连的临时性节点都将丢失。尽管因为会话的长时期性，会话中断相对来说很少发生，但是对于应用程序来说要适当地处理它是非常重要的(具体做法参见 13.4.2 节)。

无论什么时候会话持续空闲长达一定时间，都会由客户端发送 ping 请求保持活跃(犹如心跳)。(Ping 由 ZooKeeper 客户端库自动发送，所以你的代码不需要担心如何保持会话。)时间段要选得足够小以监测服务器故障(由读操作超时反应)，并且能在会话超时时间段内重新连接到另一个服务器。

time 参数

在 ZooKeeper 中有几个 time 参数。tick time 是 ZooKeeper 中的基本时间长度，为 ensemble 里的服务器所使用，用来定义对于交互运行的调度。其他设置以 tick time 的名义定义，或者至少由它约束。举例来说，如果会话超时了，可能小于 2 tick 或者大于 20。如果想设置超出这一范围的会话超时，它会下降到小于这一范围的数值。

常见的 tick time 被设置成 2 秒(2000 毫秒)，这就意味着允许会话超时的时间在 4～40 秒之间。在选择会话超时时需要进行适当的考虑。

较小的会话超时会更快得察觉机器故障。在组成员制的例子中，会话超时就是故障机器被移出小组的时间。但是，也不要把会话超时设置得太小，因为繁忙的网络会导致包的延时或是非意图性的会话中断。在这种情况下，机器会显示“flap”：断开，并再在一段时间内反复地重新连接小组。

创建更复杂的临时性状态的应用程序应该支持更长的会话超时，因为重新构建的代价会更昂贵。在一些情况下，我们可以让应用程序在一定会话超时时间内能够重启，并且避免会话过期。(这可能更适合于执行维护或是升级)每个会话都由服务器给定一个唯一的身份和密码，而且如果是在建立连接时被传递给 ZooKeeper 的话，它能够恢复会话(只要它没有过期)。所以应用程序可以安全关闭，同时因为存储了会话的身份和密码，它可以重新获得这个身份和密码并恢复会话。

你应该将这些特性视为一种可以避免会话过期的优化，它并不能代替用来处理会话过期。会话过期可能出现在机器突然故障时，或是由于任何原因导致的应用程序安全关闭了，但在会话中断前没有重启。

作为常规的原则，ZooKeeper 的 ensemble 越大，会话超时应该越长。连接超时、读取超时和 ping 周期都作为一个函数定义在 ensemble 的服务器中，因此当 ensemble 扩大时，这些时间会减小。如果连接经常断开，就试着增加超时。可以监测 ZooKeeper 公制——比如使用 JMX 监测请求响应时间的统计数据。

13.3.6 状态

ZooKeeper 对象的转变通过其生命周期中的不同状态(参见图 13-3)。可以使用 getState()方法在任何时候查询它的状态：

```
Public States getState()
```

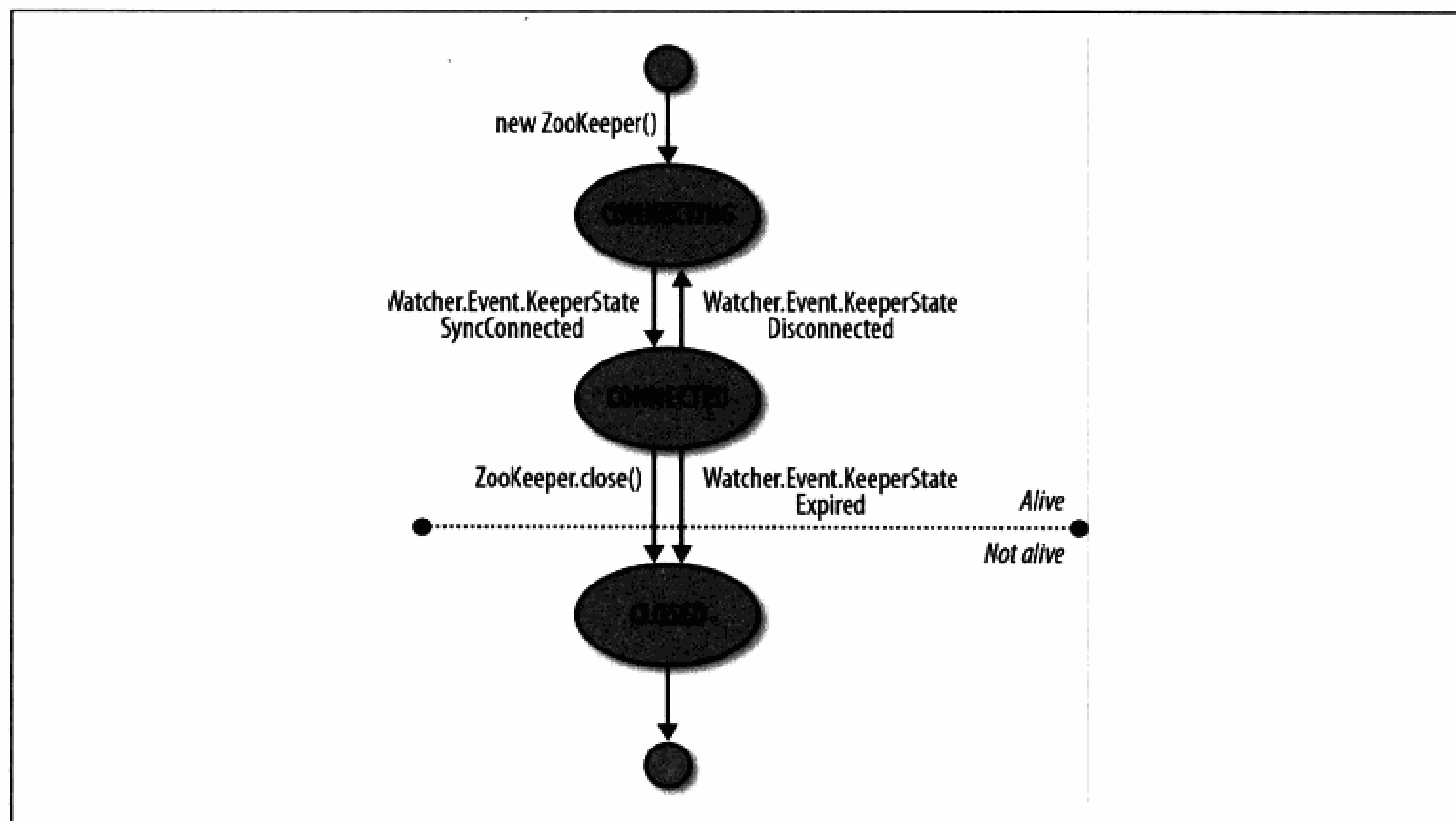


图 13-3：ZoooKeeper 状态事务

states 是枚举型，代表 ZooKeeper 对象可能所处的不同状态。(不管这个枚举型的名字，一个 ZooKeeper 的实例可能一次只处于一个状态。)一个新建的 ZooKeeper 实例正在与 ZooKeeper 服务器建立连接时，是处于 CONNECTING 状态的。一旦连接建立好以后，它就变成 CONNECTED 状态了。

使用 ZooKeeper 的客户端可以通过注册 watcher 的方法来获取状态转变的消息。一旦进入 CONNECTED 状态了，watcher 将获得一个 KeeperState 值为 SyncConnected 的 WatchedEvent。

注意：ZooKeeper 的 watcher 对象有两个职责：它可以被用来了解 ZooKeeper 状态的改变(如这里描述的一样)，也可以被用来了解 znode 的改变(参见 13.3.3 节)。被传递给 ZooKeeper 对象构造函数的(默认的)watcher 被用作监测状态的改变，而 znode 的变化既可以使用 watcher 的专门的实例(通过将其传递到对应的读操作上)，或者如果使用了带有布尔型标志的、用来具体化是否使用 watcher 的读操作形式的话，也可以共享默认的 watcher。

ZooKeeper 实例可能失去或重新连接到 ZooKeeper 服务，在 CONNECTED 和 CONNECTING 状态中切换。如果连接断开，watcher 得到了一个 Disconnected 事件。需要注意的是，这些状态的迁移是由 ZooKeeper 实例自己初始化的，如果连接断开，它将自动尝试重新连接。

如果任何一个 close()方法被调用，或是会话由 Expired 类型的 KeeperState 提示过期时，ZooKeeper 实例可能会转变成第三种状态 CLOSED。一旦处于 CLOSED 状态，ZooKeeper 对象将不再是活动的了(可以使用 states 的 isAlive()方法进行测试)，而且不能被重用。客户端必须建立一个新的 ZooKeeper 实例才能重新连接到 ZooKeeper 服务。

13.4 使用 ZooKeeper 建立应用程序

深入了解部分 ZooKeeper 之后，让我们回过来，开始学习怎样用它来写一些有用的应用程序。

13.4.1 配置服务

配置服务是分布式应用程序需要的最基本的服务之一，这样一来，配置的基本信息就能被集群中的机器分享。在最简单的一层中，ZooKeeper 可以作为用来配置的高可用性存储，使应用程序的参与者可以恢复或者更新配置文件。使用 ZooKeeper 监测，就可以创建活动的配置服务，有需要的客户端可以以此来了解配置的改变。

让我们来写一个这样的服务。首先让我们做一些假设来简化这个应用(通过一些操作可以移除它们)。第一，我们需要存储的唯一配置值是字符串，而且键正好在 znode 的路径中，因此我们可以使用 znode 来存储键/值对。第二，客户端一次只能更新一个。这个模型适合于想要它的工作者需要跟踪更新信息的主机(如 HDFS 的名称节点)。

我们将代码封装在 ActiveKeyValueStore 的类中：

```
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()方法的约束是一个给定值的写到 ZooKeeper 上的键。它通过使用 exists 操作先测试 znode，再执行适当的操作，来隐藏创建一个新的 znode 和使用新值更新已经存在的 znode 之间的区别。还有需要提及的细节是需要将字符串值转换成字节数组，我们只要使用带有 UTF-8 编码的 getBytes()方法就能完成。

为了详细说明 ActiveKeyValueStore 的用法，考虑带有值的 ConfigUpdater 类来更新配置特性。参见例 13-6。

例 13-6：一个应用程序，用来以随机次数更新 ZooKeeper 的特性

```
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 有 ActiveKeyValueStore 连接到 ConfigUpdater 构造函数的 ZooKeeper 中。run()方法不断循环着，用随机值以随机次数更新/config znode。

下面让我们来看看怎样读取/config 的配置特性。首先我们向 ActiveKeyValue-Store 增加一个读操作：

```
  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()方法带有路径、Watcher 和一个 Stat 对象。Stat 对象由 getData()的值充满，被用来将信息传递回 caller。这样一来，caller 就能获得 znode 的数据和元数据。尽管这样，我们仍会传递一个 null Stat，因为我们对元数据并不感兴趣。

作为服务的使用者，ConfigWatcher(参见例 13-7)创建了 ActiveKeyValueStore，同时在启动以后，调用 store 的 read()方法(在它的 displayConfig()方法里)将参数传递给作为 watcher 的自己，同时显示了它读到的配置的初始值。

例 13-7：一个应用程序，监测 ZooKeeper 中的更新，并将其打印到控制台

```
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);
  }
}
```

ConfigUpdater 更新 znode 时，ZooKeeper 使用 EventType.NodeDataChanged 事件类型使 watcher 运行。ConfigWatcher 通过读取和显示 config 的最新版本在它的 process()方法中执行。

由于 watch 是一次性信号，每次我们调用 ActiveKeyValueStore 的 read()方法时都会通知 ZooKeeper 有新的 watch——这样可以保证我们能看到将来的更新。另外，我们不能保证获得每一次的更新，因为在收到 watch 事件和下一次读取之间，znode 可能已经更新了，而且可能是很多次，同时因为客户端在那段时间没有注册的 watch，所以它不会被通知。对于这个配置服务来说，这个不是问题，因为客户

端只关心属性的最新值，它优先于之前的值，不过还是要注意潜在的限制。

让我们来看看执行的代码。在终端窗口中启动 ConfigUpdater：

```
% java ConfigUpdater localhost
Set /config to 79
Set /config to 14
Set /config to 78
```

接着在另一个窗口马上启动 ConfigWatcher：

```
% java ConfigWatcher localhost
Read /config as 79
Read /config as 14
Read /config as 78
```

13.4.2 可恢复的 ZooKeeper 应用

分布式计算的错误见解中，第一条就说到[①]：“网络是可靠的。”随着它们的建立，程序至今已经假设了一个可靠的网络，因此当它们在一个真正的网络上运作时，可能会因为种种原因出故障。让我们来检查一下可能的故障原因，看看如何补救，使我们的程序在面对故障时能变得可恢复。

每一个 Java API 中的 ZooKeeper 操作在它的 throw 子句中声明了两种类型的异常：InterruptedException 和 KeeperException。

InterruptedException

如果操作被中断，InterruptedException 就会被抛出。对于取消阻塞方法，有一个标准的 Java 机制，它调用了被调用阻塞方法的线程上的 interrupt() 函数。成功的取消将导致一个 InterruptedException。ZooKeeper 遵循这一标准，因此你可以用这种方法来取消一个 ZooKeeper 操作。使用 ZooKeeper 的类或库通常要传播 InterruptedException，这样它们的客户端才能够取消它们的操作。[②]

InterruptedException 并不表示故障，而是操作已经被取消了，因此在配置应用程序中，可以通过传播这个异常使应用程序中断。

---

① 参见 *http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing*。

② 详情参见 Brian Goetz 的文章 Dealing with InterruptedException，网址为 *http//www.ibm.com/developerwrks/java/librry/j-jtp05236*。

KeeperException

如果 ZooKeeper 服务器发送错误信号或者服务器发生通信故障时，KeeperException 将被抛出。KeeperException 有各种各样的子类来应对不同的出错。举例来说，KeeperException.NoNodeException 就是 KeeperException 的一个子类，在一个不存在的 znode 上执行操作时将会被抛出。

每一个 KeeperException 的子类都有涵盖错误信息的对应代码。举例来说，对于 KeeperException.NoNodeException，这个代码就是 KeeperException.Code.NONODE(一个枚举型值)。

接下来有两个方法来处理 KeeperException：一个是捕获 KeeperException 并测试它的代码来确定补救的办法，另一个是捕获等价的 KeeperException 子类，在每个 catch 块中采取适当的操作。

KeeperException 分成三种明确的种类。

状态异常 状态异常出现在操作由于没有被应用到 znode 树上而出错时。状态异常通常因为另一个进程正在改变 znode 而发生。举例来说，一个带有版本号的 setData 操作会因为 znode 被其他进程先做了更新而失败，并抛出 KeeperException.BadVersionException，因为版本号不能匹配。程序员通常知道到这种类型的冲突是可能发生的，因此他们会编码来避免它。

一些状态异常可能会表示程序出错，比如 KeeperException.NoChildrenForEphemeralsException，它会在尝试创建一个临时性 znode 的子节点时被抛出。

可恢复异常 可恢复异常是指那些应用程序可以在相同 ZooKeeper 会话中恢复的异常。一个可恢复异常由 KeeperException.ConnectionLossException 表示，说明与 ZooKeeper 的连接丢失了。ZooKeeper 将会重新连接，而且在大多数情况下这种重新连接将会成功，并且能保证会话的完整性。

然而，ZooKeeper 不能辨别是否因为 KeeperException.ConnectionLossException 被应用而失败的操作。这是一个部分故障的例子(我们在本章的开头介绍过)。因此责任就落在了程序员身上，去处理这些不确定因素，使得应用程序能对此采取应对措施。

在这方面，对 idempotent(等幂)和 nonidempotent(非等幂)操作做一个区分非常有

用。Idempotent 操作是那些可以以相同结果被一次或多次应用的操作，比如读请求或是无条件的 setData。它们可以简单地被重新尝试。

nonidempotent 操作不能被随意地重新尝试，因为多次应用它的结果与一次使用是两样的。程序需要能够检测它的更新是由 znode 路径名中的内嵌信息实施的还是它自己的数据。当我们关注锁服务的执行时，将在 13.4.3 节讨论怎样处理这些失败的 nonidempotent 操作。

不可恢复异常 在一些情况下，ZooKeeper 会话会因为超时或是被关闭(两种情况都会有 KeeperException.SessionExpiredException)，或者可能因为验证失效((KeeperException.AuthFailedException)而变成非法的。在任何情况下，所有与会话相关的临时性节点将会丢失，因此应用程序需要在与 ZooKeeper 重新连接前重新建立它的状态。

一个可靠的配置服务

回到 ActiveKeyValueStore 的 write()方法，前文讲到它由 exists 操作组成，由 create 或 setData 跟踪：

```
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);
  }
}
```

从整体看，write()方法是等幂的，所以我们可以无条件地重新尝试它。这里有一个 write()方法的修改版本，以循环来重新尝试。它被设置成最大数目的 retry(MAX_RETIRES)，并在每次尝试间休息 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());
        }
      } catch (KeeperException.SessionExpiredException e) {
        throw e;
      } catch (KeeperException e) {
        if (retries++ == MAX_RETRIES) {
          throw e;
        }
        // sleep then retry
        TimeUnit.SECONDS.sleep(RETRY_PERIOD_SECONDS);
      }
    }
  }
```

这个代码谨慎地没有重新尝试 KeeperException.SessionExpiredException，因为当会话到期时，ZooKeeper 对象进入 CLOSED 状态，从此它将永远不能重新连接(参考图 13-3)。我们简单地重新抛出异常[①]，并使调用者创建一个新的 ZooKeeper 实例，使得整个 write()方法可以重新尝试。一个简单的创建新实例的方法是创建一个新的 ConfigUpdater(事实上是我们重新命名 ResilientConfigUpdater)来从过期的会话中恢复：

```
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;
    }
```

① 另一种写代码的方法是拥有一个独立的只为 KeeperException 的 catch 块以及一个用来查看代码是否有值 KeeperException.Code.SESSIONEXPIRED 的测试。两种方法因为都以相同的方法执行，所以可以根据自己的喜好来选择。

```
    }
  }
```

处理会话过期的可选方法之一是寻找 watcher 中（在这个例子中是 ConnectionWatcher）Expired 类型的 KeeperState，并在它被察觉时创建一个新的连接。这样的话，即使我们收到 KeeperException.SessionExpiredException，也只要保持 write() 方法的重试，因为重新连接最终还是会建立。不管我们采用怎样的方法从过期会话中恢复，最重要的是它与连接断开的故障是不同的，需要不同的处理方法。

注意： 事实上还有另一个我们忽视的故障类型。当 ZooKeeper 对象被创建时，它尝试连接到 ZooKeeper 服务器。如果连接失败或是超时，它将尝试 ensemble 中的另一台服务器。如果试过 ensemble 中的所有服务器，还是无法连接，那么它令抛出 IOException。ZooKeeper 所有服务器都无法访问的可能性很低，尽管如此，一些应用程序可能循环地尝试这一操作直到 ZooKeeper 能够被获得。

这只是处理重试的一种方法——还有其他很多方法，比如在固定每次重试时使用指数的退避。Hadoop Core 中的 org.apache.hadoop.io.retry 包涵盖了一些有用的方法，你可以用来向代码增加可以重用的重试逻辑，对于建立 ZooKeeper 应用来说也可能有用。

13.4.3 锁服务

分布式锁是一种机制，用来提供一组进程间的互斥。在任何时间里，只有一个进程可以持有锁。分布式锁可以为大型分布式系统中的选举所使用，leader 就是在任何时间点都持有锁的进程。

注意： 不要将 ZooKeeper 自己的选举与普通的选举服务混淆，前者可以由 ZooKeeper 原型构建。ZooKeeper 自己的选举不向公众开放，不像我们在这里描述的普通类型的选举服务一样。这里所说的普通类型的选举服务为分布式系统所使用，需要由主机进程的同意。

为了使用 ZooKeeper 执行分布式锁，我们使用顺序型 znode 来标识争夺锁的进程的顺序。这种思想很简单：首先指明一个 lock znode，一般描述了被锁定的实体，叫作/leader；接着想要获得锁的客户端创建顺序型的临时 znode 作为这个 lock znode 的子节点。在任何时刻，拥有最小序列号的客户端获得锁。举例来说，如果两个客户端在几乎相同的时间创建了 znode、/leader/lock-1 和 /leader/lock-2，那么创建

/leader/lock-1 的客户端拥有锁，因为它的 znode 有较低的序列号。ZooKeeper 服务是顺序的仲裁者，因为是由它来分配序列号的。

这个锁可以通过删除 znode /leader/lock-1 被简单地删除；或者，如果客户端进程死亡了，它将会因为是临时性 znode 而被删除。创建/leader/lock-2 的客户端将会拥有锁，因为它有接下去最小的顺序号。它将由 watch 被通知已经拥有了锁，watch 在创建的 znode 消失时启动。

获得锁的伪代码如下。

1. 在 lock znode 下创建一个名为 lock-的临时性顺序 znode，记住它实际的路径名(`create` 操作的返回值)。
2. 获得 lock 的 children 并设置 watch。
3. 如果在 1 中创建的 znode 的路径名有一个在 2 返回的 children 的最小号码，那么 lock 就能被得到。退出。
4. 等待设置在 2 中的 watch 返回消息，回到第 2 步。

集群影响

尽管这个算法是正确的，但是仍有一些问题。首先，执行会遭受集群影响。设想一下，成百上千的客户端都尝试着获取锁，每个客户端都在 lock znode 上放一个 watch 事件来观察它 children 的变化。每次当锁被释放或者另一个进程开启 lock 响应进程时，watch 事件就会启动，每个客户端就会收到消息。这种“集群影响”指的是大量客户端同时收到相同的事件，却只有一小部分客户端可以真正处理。在这种情况下，只有一个客户端可以成功地获得锁，而且维护和发送 watch 事件到所有客户端的进程会导致通信阻塞，这会给 ZooKeeper 服务器带来很大的压力。

为了避免这种集群影响，我们需要提高发送消息的条件。执行锁的主要依据是仅当子 znode 之前的顺序号都消失时，客户端被通知，而不是任何子 znode 被删除(或是创建)时。在我们的例子中，如果客户端创建了 znodes /leader/lock-1，/leader/lock-2，和 /leader/lock-3，那么持有 /leader/lock-3 的客户端只有当 /leader/lock-2 消失时才需要被通知，而不是在/leader/lock-1 消失或是新 znode /leader/lock-4 被增加时。

可恢复的异常

另一个使用锁算法的问题是它不能处理当 create()操作因为连接丢失而失败的情况。回忆下在这种情况下，我们并不知道这个操作是成功或是失败。创建一个顺序型 znode 是一个非等幂操作，因此我们不能简单地复试，因此如果第一个 `create()`

成功了，我们将有一个再也不能被删除的(至少直到客户端会话结束)孤立的znode。死锁可能是最不幸的结果。

问题是在重新连接后，客户端不能辨别它是否创建了子 znode。我们在 znode 名称中嵌入一个标识符，如果遭遇连接断开，它不是创建其他子 znode，而是查看锁节点任何节点是否在它们的名字中有这个标识符。如果没有节点在它的名字中有这个标识符，那么客户端就能安全地创建一个新的顺序型的子 znode。

客户端会话的标识符是一个长整型，对 ZooKeeper 服务来说是唯一的，因此对于要确定连接断开事件的客户端来说也是理想的。会话标识符可以通过调用 `ZooKeeper` Java 类中的 `getSessionID()` 方法获得。

临时性顺序 znode 应该以 lock-*<sessionId>*-形式的名称创建，因此当序号被 ZooKeeper 附加时，名称会变成 lock-*<sessionId>*-*<sequenceNumber>*。序号对于父节点来说是唯一的，但对子节点来说不是，所以这种方法允许子 znode 识别它们的创建者以及创建的顺序。

不可恢复的异常

如果一个客户端的 ZooKeeper 会话过期，由客户端创建的临时性 znode 将会被删除，同时交出锁，或是至少失去客户端等待获得锁的资格。使用锁的应用程序应该意识到它不再持有锁了，要清除它的状态，并通过创建一个新的锁对象再次启动，再尝试获得它。注意，是应用程序控制着进程，而不是锁执行，因此它不能二次猜测应用程序需要怎样清除它的状态。

执行

要正确执行分布式锁是一个非常精细的工作，因为要覆盖所有的故障模式是没有意义的。ZooKeeper 带有 Java 中的叫作 `WriteLock`(来自 ZooKeeper 3.2.0 onward)的成品品质的锁执行，对客户端来说使用非常方便。

13.4.4 更多分布式数据结构和协议

还有许多分布式数据结构和协议可以用来建立 ZooKeeper，比如栅栏、队列和两阶段提交。需要注意的一件有趣的事是这些都是同步协议，即使我们使用非同步 ZooKeeper 原型(比如通知)来构建它们。

ZooKeeper 网站(*http://hadoop.apache.org/zookeeper/*)在伪代码中描述了这样的几个

数据结构和协议。在写这本书时，标准实现还不能作为 ZooKeeper 的一部分获得，但在不久之后它们将被增加到代码库中。

BookKeeper

BookKeeper 是一个高可用、可靠的日志服务。它可以被用来提供预写式日志，这是一种保证存储系统中数据完整性的传统方法。在使用预写式日志的系统中，每次写操作之前它都将被写入事件日志中。使用这种方法，我们不需要在每次写操作之后将数据写入永久存储中，因为当系统故障时，最新的状态会由重放事件日志来恢复，其中包括任何没有应用的写操作。

BookKeeper 客户端创建了名叫 ledger 的日志，而且每个附加到 ledger 的记录叫作 ledger entry，是简单的字节数组。Ledger 由 bookie 管理，它们是复制 ledger 数据的服务器。注意，ledger 数据并不存储在 ZooKeeper 中，只有元数据被存储。

一般来说，富有挑战的是让使用预写式日志的系统能够健壮地面对写入事件日志的节点故障。这通常由复制事件日志以相同的方式处理。举例来说，Hadoop 的 HDFS 名称节点将它的修改日志写到多个磁盘上，其中一个通常是安装 NFS 的磁盘。然而，在处理基本故障时仍然是手工的。通过提供日志作为高可用的服务，BookKeeper 保证了故障的透明化，因为它可以承受 bookie 服务器的丢失。

BookKeeper 由 ZooKeeper 分布式的 contrib 目录提供，可以在此发现它的更多用法。

13.5 工业界中的 ZooKeeper

在工业界，你要让 ZooKeeper 以复制模式运行。这里将讨论运行 ZooKeeper 服务器的 ensemble 需要关心的问题。然而，这里的介绍不是很透彻，因此你可以参考 ZooKeeper 管理员手册(*http://hadoop.apache.org/zookeeper/docs/current/*)了解详细的最新指令，包括支持的平台、推荐的硬件、维护的步骤以及配置属性。

13.5.1 恢复力及性能

ZooKeeper 机器需要最小化机器和网络故障的影响。通常，这意味着服务器应该通过机架、电源供应和转换来传播，这样一来，任何服务器的故障不会导致 ensemble 失去它的大部分服务器。拥有 ensemble 中所有服务器间的低响应时间连

接对于 ZooKeeper 来说有影响，因此 ensemble 应该被限制在一个单数据中心。

ZooKeeper 是一个高可用系统，而且它因为能够及时地执行它的函数而显得非常重要。因此 ZooKeeper 应该运行在只负责 ZooKeeper 的机器上。有其他应用程序竞争资源会造成 ZooKeeper 性能显著下降。

配置 ZooKeeper，使它能保留在不同磁盘上的由快照驱动的事务日志。默认情况下，两者都运行在由 `dataDir` 属性定义的目录中，不过要为 `dataLogDir` 指定一个地址，事务日志才能被写在那儿。通过拥有它自己的设备(不只是一部分)，ZooKeeper 服务器可以最大化它将日志项目写到磁盘的速率，不用寻址，它将按顺序执行。所有的写操作都经过领导者，写操作的吞吐量是不能由增加服务器而扩大的，所以写操作尽可能快非常重要。

如果进程与磁盘交换，性能将会受到不利的影响。这可以通过设置 Java 堆大小和减少机器上物理内存的可用空间来避免。ZooKeeper 脚本将从它的配置文件中查询一个叫 java.env 的文件，它可以用来改变堆大小(以及其他需要的 JVM 参数)，通过设置 `JVMFLAGS` 环境变量。

13.5.2 配置

每个在 ZooKeeper 服务器 ensemble 中的服务器都有一个在 ensemble 中唯一的数字标识符，而且必须在 1～255 之间。服务器号被定义在名为 myid 文件的纯文本中，该文件由 `dataDir` 属性指定。

设置每个服务器号只是工作的一半。我们还需要给所有服务器所有的标识符以及 ensemble 中其他服务器的网络地址。ZooKeeper 配置文件必须为每一个服务器留出一行，以如下形式：

```
server.n=hostname:port:port
```

n 的值由服务器号代替。有两种端口设置：第一个是跟随者使用的连接到领导者的端口，第二个是用来选举领导者的。这里有一个为了机复制的 ZooKeeper 配置的例子：

```
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
```

服务器在三个端口监听：2181(用来监听客户端连接)；2888(如果它们是领导者，用来监听跟随者连接)；3888(用来在选举领导者阶段监听其他服务器的连接)。当ZooKeeper 服务器启动时，它读取 myid 文件来判断这是哪个服务器，然后读取配置文件来确定它需要监听哪个端口，以及 ensemble 中其他服务器的网络地址。

连接到这个 ZooKeeper ensemble 的客户端应该使用 `zookeeper1：2181，2181，zookeeper2:2181，zookeeper3:2181` 作为 `ZooKeeper` 对象构造函数里的主机字符串。

在复制模式中，有两个额外的强制性属性 `initLimit` 和 `syncLimit`，两者由不同的 `tickTime` 计时。

`InitLimit` 是指允许跟随者连接到领导者可以与其同步的时间。如果大部分跟随者在这段时间不能同步，那么领导者将会放弃它的领导地位，另一个选举领导者事件将会发生。如果这种情况经常发生(因为它被记入日志了，所以你可以发现是否是这种情况)，就表明设置太低。

`syncLimit` 是允许一个跟随者与领导者同步的时间。如果跟随者在这段时间不能同步，那么它就会自动重启。被附属到这个跟随者的客户端将连接到另一个领导者。

这些是启动并运行一个 ZooKeeper 服务器集群所需要的最少的设置。还有很多配置的选项，尤其是与改善性能有关的选项，都被记录在 ZooKeeper 管理员手册中。

第 14 章

案例研究

14.1 Hadoop 在 Last.fm 的应用

14.1.1 Last.fm：社会音乐革命

成立于 2002 年，Last.fm 是一个互联网广播和音乐社区网站，向它的用户提供多种服务，如免费音乐流下载，音乐和事件评论，个性化的排行榜，等等。每月大约有 2500 万人使用 Last.fm，产生了大量需要处理的数据。其中一个例子是用户传递表明他们正在收听的歌曲信息(这称为“scrobbing”)。此数据由 Last.fm 处理和存储，所有用户可以直接访问(以图表的形式)，并且用来判断用户的音乐喜好和广泛性以及艺术家和音乐相似度。

随着 Last.fm 服务的发展和用户数从数千至数百万的增长，存储、处理和管理所有传入的数据变得越来越具有挑战性。幸运的是，Hadoop 很快变得足够稳定，并随着它能解决的问题数量变得清晰而被积极采用。2006 年在 Last.fm 中第一次使用 Hadoop 并且在随后的几个月投入生产。下面是 Last.fm 采用 Hadoop 的几个原因。

- 分布式文件系统提供了对存储在其中的数据的冗余备份(例如，网络日志和用户收听数据)并且无额外花费。
- 需要时可通过增加较便宜的硬件来简单地实现可扩展性。
- 每次当 Last.fm 限制财政资源时，费用是合理的(免费)。
- 开源代码和活跃的社区意味着 Last.fm 可以任意修改 Hadoop 来添加自定义功

能和补丁。

- Hadoop 提供了一个灵活的框架通过相对容易的学习曲线来运行分布式计算算法。

Hadoop 现已成为 Last.fm 基础设施的重要组成部分，目前包含两个 Hadoop 集群，跨 50 台机器，300 个内核和 100 TB 的硬盘空间。数以百计的日常作业运行在集群上执行操作，如日志分析，A/B 测试评价，AdHoc 处理和图表生成。本案例研究将侧重于排行榜生成的处理过程，因为这是 Hadoop 在 Last.fm 上的第一个应用，并且说明了较之其他方法，Hadoop 在应付巨型数据集时的强大和灵活之处。

14.1.2 使用 Hadoop 生成排行榜

Last.fm 使用用户生成的音乐收听数据生成多种不同类型的排行榜，如每个国家和每个用户的每周单曲排行榜。一些 Hadoop 程序用来处理收听数据并生成这些排行榜，这些程序每天、每周或每月为基本单位运行。图 14-1 说明了这些数据是如何在网站上显示的，在本例中是一个每周流行单曲排行榜。

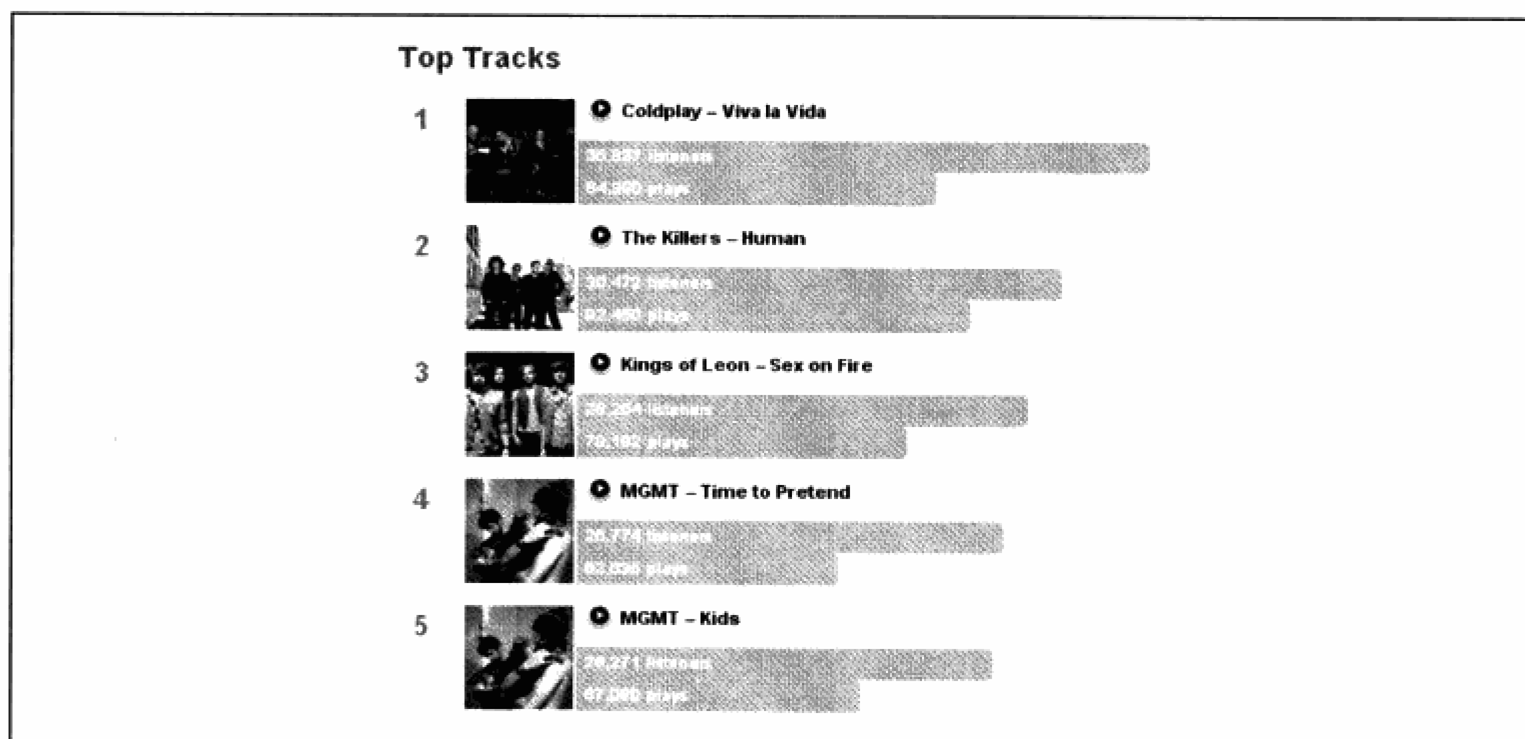


图 14-1：Last.fm 流行歌曲排行榜

Last.fm 的收听数据主要从两个来源之一得到。

- 用户播放自己的音乐(如从 PC 或其他设备上收听一个 MP3 文件)，这些信息通过 Last.fm 官方客户端应用程序或数百个第三方应用程序之一发送到 Last.fm 上。
- 用户到 Last.fm 的一个 Internet 广播电台在线收听。可通过 Last.fm 播放器或网

站来获得音乐流和提供给用户的其他功能，例如：允许用户喜爱、跳过或删除她收听的歌曲。

在处理收到的数据时，我们对用户提交的单曲(上述的第一个来源，从这里开始我们称为 scrobble)和在 Last.fm 电台上收听的单曲(前面提到的第二个来源，从这里开始称为 radio listen 电台收听)进行区分。这种区分是非常重要的，可防止 Last.fm 的推荐系统产生反馈回路，而推荐系统只基于 scrobble。在 Last.fm 上 Hadoop 最基本的作业之一是获得传入的收听数据并归为同一格式以用来在 Last.fm 网站上显示以及作为其他 Hadoop 程序的输入。这是通过下节描述的例子“单曲统计”程序实现的。

14.1.3 单曲统计程序

单曲收听数据提交到 Last.fm 时，它经历了一个验证和转换阶段，其最终的结果是一些空格分隔的文本文件，包含用户 ID、单曲 ID、单曲被 scrobble 的次数、在电台上单曲被收听的次数以及它被跳过的次数。表 14-1 包含收听数据的例子，它将用于下一个例子作为单曲统计程序的输入(实际数据规模在 GB，并且包括更多这里为简单起见而省略的字段)。

表 14-1：收听数据

UserId	TrackId	Scrobble	Radio	Skip
111115	222	0	1	0
111113	225	1	0	0
111117	223	0	1	1
111115	225	1	0	0

这些文本文件是提供给单曲统计程序的原始输入，其中包括两个计算各种值的作业和一个合并结果的作业(见图 14-2)。

unique listener 作业通过计算一个用户第一次收听该单曲并忽略同一用户的其他收听次数来计算收听该单曲的不同听众的总人数。sum 作业通过所有用户所有收听记录来统计该单曲收听总次数、scrobble 次数、电台收听次数和跳过次数。虽然这两个作业的输入格式是相同的，但需要两个不同的作业，因为 Unique Listener 作业负责按每首单曲每个用户发出值，而 Sum 作业按每首单曲发出值。最后的“合并”作业负责合并这两个作业的中间输出产生最后的结果。运行该程序的最终结果每首单曲如下：

- 不同听众人数。

- 单曲被 scrobble 的次数。
- 单曲在电台中收听的次数。
- 单曲收听总次数。
- 单曲在电台中被跳过的次数。

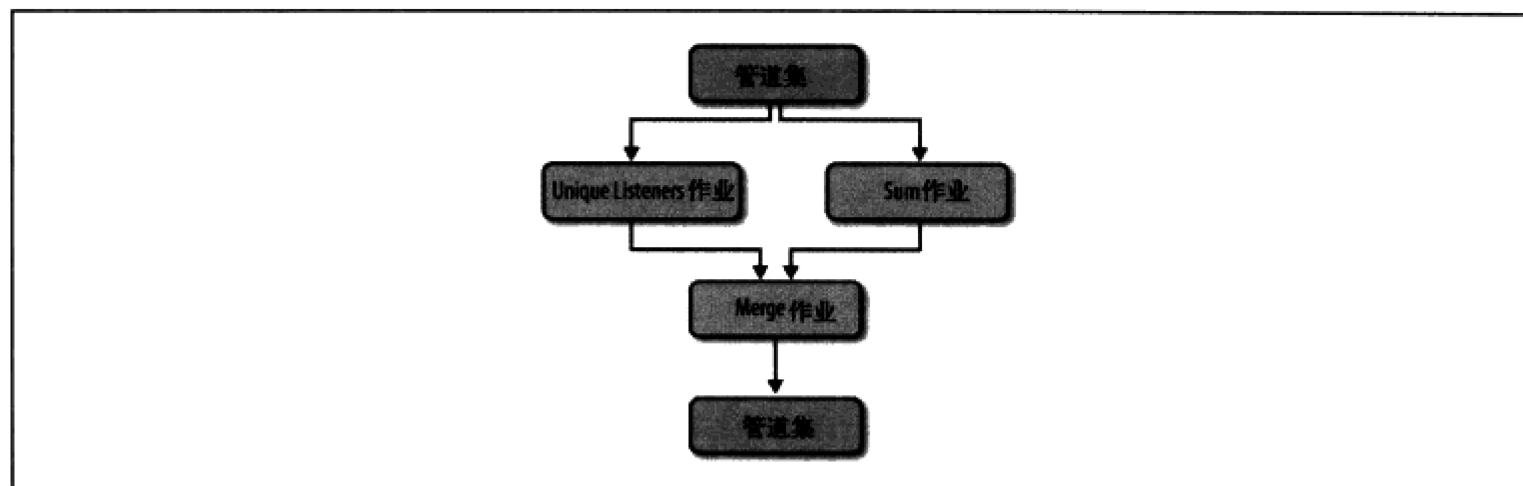


图 14-2：单曲统计作业

每个作业和它的 MapReduce 过程将在后面详细描述。请注意提供的代码片段由于空间有限被简化了，需要下载完全的代码清单，请参考本书“前言”。

计算不同听众人数

Unique Listener 作业为每首单曲计算不同听众的人数。

UniqueListenersMapper `UniqueListenersMapper` 处理空格分隔的原始收听数据并发出与每首单曲相关的用户 ID：

```
public void map(LongWritable position, Text rawLine,
OutputCollector<IntWritable,
    IntWritable> output, Reporter reporter) throws IOException {

  String[] parts = (rawLine.toString()).split(" ");

  int scrobbles = Integer.parseInt(parts[TrackStatisticsProgram.
COL_SCROBBLES]);
  int radioListens = Integer.parseInt(parts[TrackStatisticsProgram.
COL_RADIO]);
// if track somehow is marked with zero plays - ignore
if (scrobbles <= 0 && radioListens <= 0) {
    return;
  }
  // if we get to here then user has listened to track,
```

```
  // so output user id against track id
  IntWritable trackId = new IntWritable(
    Integer.parseInt(parts[TrackStatisticsProgram.COL_TRACKID]));
  IntWritable userId = new IntWritable(
    Integer.parseInt(parts[TrackStatisticsProgram.COL_USERID]));
  output.collect(trackId, userId);
}
```

UniqueListenersReducer UniqueListenersReducer 收到每首单曲 ID 的用户 ID 列表，并将这些 ID 放到一个集合中以消除所有重复。每首单曲 ID 对应的集合的大小之后被发出(例如：不同听众人数)。如果某个键有许多值时，将所有 reduce 值存储到一个集合可能会有内存不足的风险。这在实际中并不会发生，但是要克服这一点，一个额外的 MapReduce 的步骤可以用来删除所有重复的值或者可以使用二次排序。(有关详细信息，请参阅 8.2 节。)

```
public void reduce(IntWritable trackId, Iterator<IntWritable> values,
    OutputCollector<IntWritable, IntWritable> output, Reporter reporter)
    throws IOException {

  Set<Integer> userIds = new HashSet<Integer>();
  // add all userIds to the set, duplicates automatically removed
(set contract)
  while (values.hasNext()) {
    IntWritable userId = values.next();
    userIds.add(Integer.valueOf(userId.get()));
  }
  // output trackId -> number of unique listeners per track
  output.collect(trackId, new IntWritable(userIds.size()));
}
```

表 14-2 显示了该作业输入数据的例子。表 14-3 显示了 map 输出，表 14-4 显示了 reduce 输出。

表 14-2：作业输入

Line of file	UserId	TrackId	Scrobbled	Radio play	Skip
LongWritable	IntWritable	IntWritable	Boolean	Boolean	Boolean
0	11115	222	0	1	0
1	11113	225	1	0	0
2	11117	223	0	1	1
3	11115	225	1	0	0

表 14-3：mapper 输出

TrackId	UserId
IntWritable	IntWritable
222	11115
225	11113
223	11117
225	11115

表 14-4：reducer 输出

TrackId	#listeners
IntWritable	IntWritable
222	1
225	2
223	1

统计单曲总数

sum 作业相对简单，它只是统计我们对每首单曲感兴趣的值。

SumMapper 输入数据仍是原始文本文件，但在这种情况下是完全不同的处理。所需的最终结果是一个与单曲相关的总数(不同听众人数、播放次数、scrobble 次数、电台收听次数和被跳过次数)。为了简单起见，我们使用一个中间对象 `TrackStats` 来保存这些值，`TrackStats` 由 Hadoop Record I/O 生成，它实现了 `WritableComparable` 接口(因此它可作为输出)。mapper 创建一个 `TrackStats` 对象，并为在文件中的每一行设置值，除了不同听众人数被留空(最后的合并作业会将其填补)：

```
public void map(LongWritable position, Text rawLine,
    OutputCollector<IntWritable, TrackStats> output, Reporter
    reporter)throws IOException {

  String[] parts = (rawLine.toString()).split(" ");
  int trackId = Integer.parseInt(parts
    [TrackStatisticsProgram.COL_TRACKID]);
  int scrobbles = Integer.parseInt(parts
    [TrackStatisticsProgram.COL_SCROBBLES]);
  int radio = Integer.parseInt(parts
    [TrackStatisticsProgram.COL_RADIO]);
```

```
    int skip = Integer.parseInt(parts
      [TrackStatisticsProgram.COL_SKIP]);
    // set number of listeners to 0 (this is calculated later)
    // and other values as provided in text file
    TrackStats trackstat = new TrackStats(0, scrobbles + radio,
      scrobbles, radio, skip);
    output.collect(new IntWritable(trackId), trackstat);
  }
```

SumReducer 在这种情况下，reducer 执行的功能类似于 mapper，它统计每首单曲的数据并返回一个总数：

```
public void reduce(IntWritable trackId, Iterator<TrackStats>
    values,OutputCollector<IntWritable, TrackStats> output,
    Reporter reporter)
    throws IOException {

  TrackStats sum = new TrackStats(); // holds the totals for
      this track
  while (values.hasNext()) {
    TrackStats trackStats = (TrackStats) values.next();
    sum.setListeners(sum.getListeners() + trackStats.getListeners());
    sum.setPlays(sum.getPlays() + trackStats.getPlays());
    sum.setSkips(sum.getSkips() + trackStats.getSkips());
    sum.setScrobbles(sum.getScrobbles() + trackStats.getScrobbles());
    sum.setRadioPlays(sum.getRadioPlays() +
      trackStats.getRadioPlays());
  }
  output.collect(trackId, sum);
}
```

表 14-5 显示了作业的输入数据(与 Unique Listener 作业相同)。表 14-6 显示了 map 输出，表 14-7 显示了 reduce 输出。

表 14-5 作业输入

Line	UserId	TrackId	Scrobbled	Radio play	Skip
LongWritable	IntWritable	IntWritable	Boolean	Boolean	Boolean
0	11115	222	0	1	0
1	11113	225	1	0	0
2	11117	223	0	1	1
3	11115	225	1	0	0

表 14-6：map 输出

TrackId IntWritable	#listeners IntWritable	#plays IntWritable	#scrobbles IntWritable	#radio plays IntWritable	#skips IntWritable
222	0	1	0	1	0
225	0	1	1	0	0
223	0	1	0	1	1
225	0	1	1	0	0

表 14-7：reduce 输出

TrackId IntWritable	#listeners IntWritable	#plays IntWritable	#scrobbles IntWritable	#radio plays IntWritable	#skips IntWritable
222	0	1	0	1	0
225	0	2	2	0	0
223	0	1	0	1	1

合并结果

最后的作业需要合并前两个作业的输出：每首单曲的不同听众人数和其统计数据。为了能够合并这些不同的输入，需要使用两个不同的 mapper(每种输入类型一个)。这两个中间作业被配置为在不同的路径中写入结果，而 `MultipleInputs` 类就用于指定哪个 mapper 程序将处理哪个文件。下面的代码显示了作业的 `JobConf` 如何设置做到这一点：

```
MultipleInputs.addInputPath(conf, sumInputDir,
  SequenceFileInputFormat.class, IdentityMapper.class);

MultipleInputs.addInputPath(conf, listenersInputDir,
    SequenceFileInputFormat.class, MergeListenersMapper.class);
```

也可以使用一个 mapper 程序来处理不同的输入，但这个例子中解决办法更方便，更简练。

MergeListenersMapper 该 mapper 程序是用来处理 `UniqueListenerJob` 的输出，即每首单曲的不同听众人数。它用类似 `SumMapper` 中的方式创建了一个

TrackStats 对象，但是这一次，它仅填充了每首歌曲不同听众人数而将其他值留空：

```
public void map(IntWritable trackId, IntWritable
    uniqueListenerCount,OutputCollector<IntWritable, TrackStats>
    output, Reporter reporter)
    throws IOException {
  TrackStats trackStats = new TrackStats();
  trackStats.setListeners(uniqueListenerCount.get());
  output.collect(trackId, trackStats);
}
```

表 14-8 显示了一些 mapper 的输入，相应的输出显示在表 14-9。

表 14-8：MergeListenersMapper 输入

TrackId	#listeners
IntWritable	IntWritable
222	1
225	2
223	1

表 14-9：MergeListenersMapper 输出

TrackId	#listeners	#plays	#scrobbles	#radio	#skips
222	1	0	0	0	0
225	2	0	0	0	0
223	2	0	0	0	0

IdentityMapper　IdentityMapper 处理 SumJob 的输出，即 TrackStats 对象。因为没有额外的处理需求，它直接发出输入数据(见表 14-10)。

表 14-10：IdentityMapper 输入/输出

TrackId	#listeners	#plays	#scrobbles	#radio	#skips
IntWritable	IntWritable	IntWritable	IntWritable	IntWritable	IntWritable
222	0	1	0	1	0
225	0	2	2	0	0
223	0	1	0	1	1

SumReducer　前面两个 mapper 发出的值为相同类型：每首单曲一个 TrackStats 对象，并分别填入了不同值。最后一个 reduce 阶段可以重用前面介绍的

SumReducer 对每首单曲创建一个 TrackStats 对象，总结所有的值并发出它(见表 14-11)。

表 14-11：最后 SumReducer 输出

TrackId	#listeners	#plays	#scrobbles	#radio	#skips
IntWritable	IntWritable	IntWritable	IntWritable	IntWritable	IntWritable
222	1	1	0	1	0
225	2	2	2	0	0
223	1	1	0	1	1

最后的输出文件之后被累计和复制到服务器上，那里的网络服务将数据提供给 Last.fm 网站显示。图 14-3 显示了这方面的一个例子，该例子显示了一首单曲的总听众人数和播放次数。

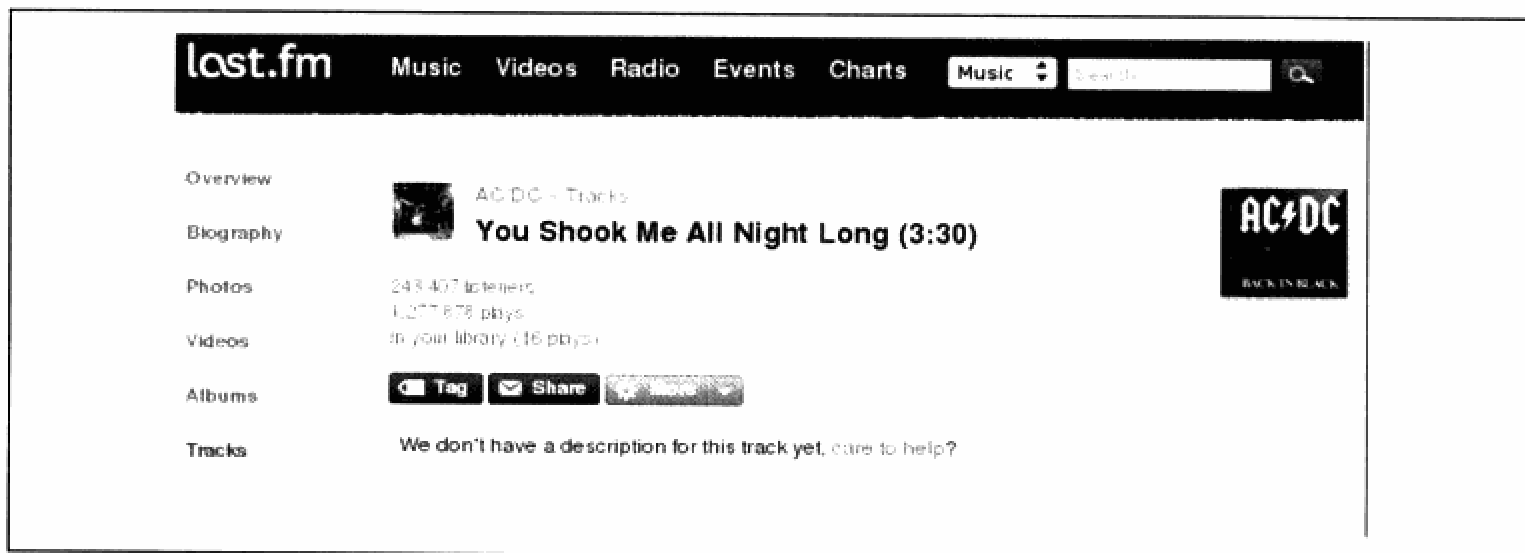


图 14-3：TrackStats 结果

14.1.4 小结

Hadoop 已成为 Last.fm 基础设施的重要组成部分，用于生成和处理从网络日志到用户收听数据等各种数据集。这里介绍的例子已经大大简化以便传达主要概念，在实际应用中的输入数据具有更复杂的结构并且处理它的代码也更为复杂。Hadoop 本身，虽然作为产品使用已足够成熟，但仍然在被积极开发，并且每周都由 Hadoop 社区增加新功能和改进。在 Last.fm 我们很高兴能成为社区的一部分，作为代码和思想的贡献者，同时也作为一种开源技术伟大作品的最终用户。

(作者Adrian Woodhead和Marc de Palol)

14.2 Hadoop 和 Hive 在 Facebook 的应用

14.2.1 简介

Hadoop 可以用来组建核心后台和近实时的计算基础设施。它还可以用来存储和归档大量数据集。在本案例中，我们将探讨后台数据架构和 Hadoop 在其中发挥的作用。我们将描述虚拟的 Hadoop 的配置，Hive 的潜在用途——一个建立在 Hadoop 之上的开源数据仓库和 SQL 基础结构——不同类型的企业和产品应用已使用此基础结构构建。

14.2.2 Hadoop 在 Facebook 的应用

历史

在 Facebook，需要处理和存储的日志和维度数据已随着网站使用率的增加而爆炸式递增。在这种环境下数据处理平台的一个关键要求是能够迅速同步扩展。此外，工程资源是有限的，该系统应该非常可靠并且易于使用和维护。

起初，Facebook 的数据仓库完全是由一个 Oracle 实例执行的。当我们开始涉及可扩展性和性能问题之后，我们开始调查是否有开源技术可用于我们的环境。作为调查的一部分，我们部署了一个相对较小的 Hadoop 的实例，并开始将一些核心数据集放到实例中。Hadoop 是有吸引力的，因为雅虎为批处理的需要而在内部使用它，同时也因为我们熟悉谷歌推广的 MapReduce 模式之简单性和可扩展性。

我们最初的原型是非常成功的：工程师们喜欢这种在合理时限内处理大量数据的能力，这种我们从没有过的能力。他们还喜欢能够使用他们偏爱的编程语言来处理(使用 Hadoop 流)。将我们的核心数据集放在一个集中数据存储上也十分方便。大约在同一时间，我们开始开发 Hive。通过使用大多数工程师和分析师熟悉的 SQL 语言形式来表达常用的计算，使得用户在 Hadoop 集群中更容易处理数据。

因此，集群的大小和使用跨越式增长，而今天 Facebook 正运行世界第二大 Hadoop 集群。在写作本书时，我们在 Hadoop 中已有超过 2 PB 的数据并且每天加载超过 10 TB 的数据。我们的 Hadoop 实例有 2400 内核和约 9 TB 内存，并且每天在许多节点上运行有 100%的利用率。我们能够迅速扩展该集群以适应增长，并且我们已经能够利用开源特点在必要时修改 Hadoop 以满足我们的需要。通过一些 Hadoop 核心组成部分和开源的 Hive(如今已是一个 Hadoop 的子项目)，我们反过来对开源做出了贡献。

用例

在 Facebook 中至少有四个相互关联但不相同的类在 Hadoop 上使用。

- 在大量数据上产生每日每时摘要。在公司内这些统计用于各种不同目的。
  - 工程和非工程部门使用基于这些摘要的报告推动产品决策。这些摘要包括用户增长报告、页面浏览量和用户平均网站停留时间。
  - 提供关于在 Facebook 上运行的广告的表现数。
  - 一些网站功能的后台处理，如：你可能喜欢的人或喜欢的应用程序。
- 在历史数据之上运行 adhoc 作业。这些分析帮助我们解答来自产品群体和执行团队的问题。
- 为我们的日志数据集作为一个长期归档存储。
- 为查找特定属性的日志记录事件(这里日志是由该属性索引的)，它用来维持网站的完整性和保护用户不受垃圾邮件的骚扰。

数据架构

图 14-4 显示了我们架构的基本组成部分和其中的数据流。

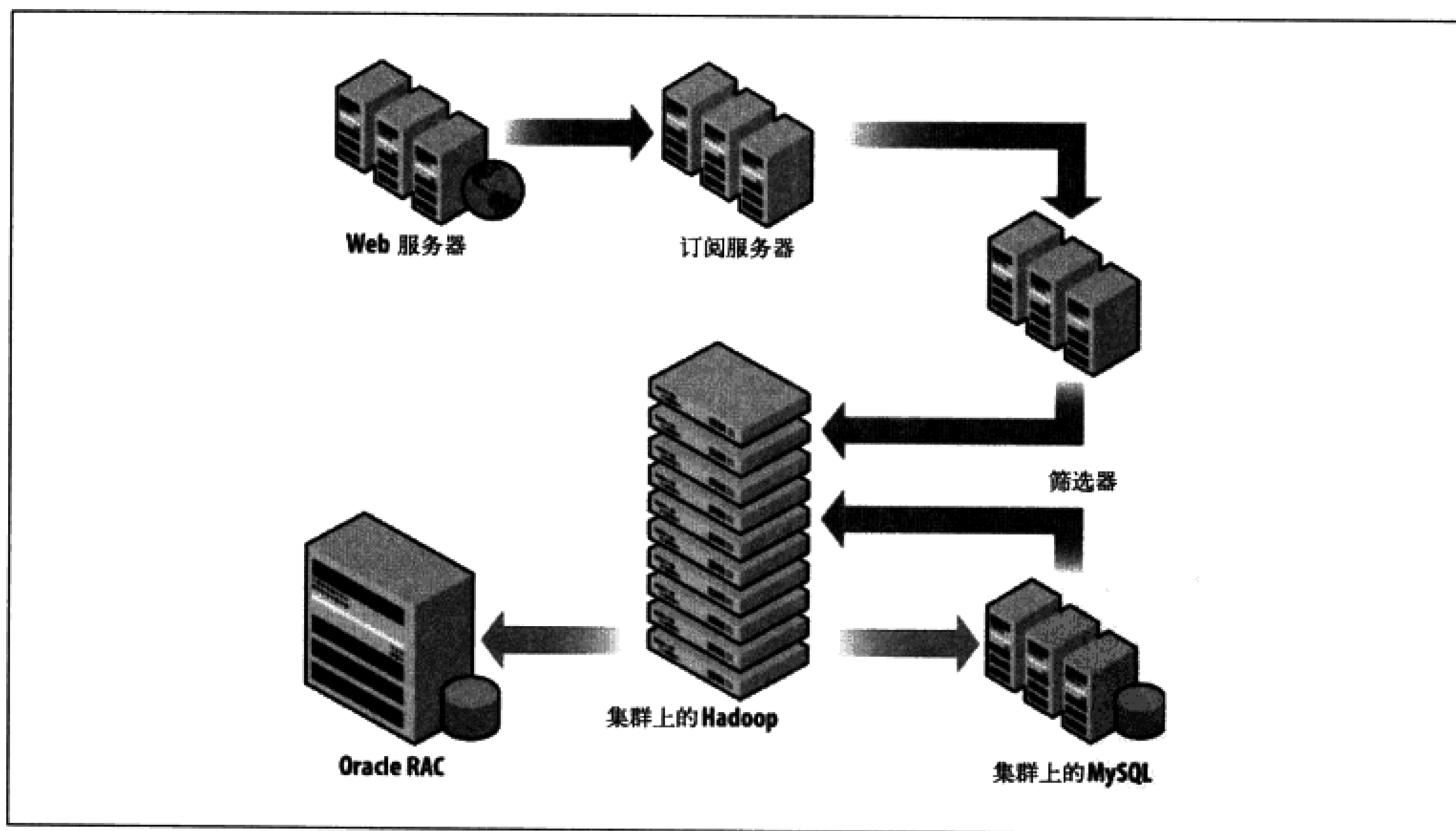


图 14-4：Facebook 的数据仓库架构

如图 14-4 所示，下列组件用于处理数据。

Scribe

日志数据由 Web 服务器以及内部的服务(例如后台搜索)生成的。我们用 Scribe，一个在 Facebook 中开发的开源日志收集服务，它将每天还含有数十 TB 量的数百个日志数据集存储到一些 NFS 服务器上。

HDFS

日志数据的很大一部分被复制到一个中心 HDFS 实例。维度数据每日也从我们内部的 MySQL 数据库中得到并复制到 HDFS。

Hive/Hadoop

我们使用 Hive(在 Facebook 中开发的一个 Hadoop 子项目)在 HDFS 中收集的所有数据上建立了一个数据仓库。HDFS 中的文件，包括来自 Scribe 的日志数据和来自 MySQL 层的维度数据，可作为表和逻辑分区。由 Hive 提供的类似 sql 的查询语言与 MapReduce 一并使用来创建/发布各种摘要和报告，以及对这些表进行历史分析。

工具

在 Hive 最上层的基于浏览器的接口允许用户只需点击几下鼠标便可组建和开展 Hive 查询(这将启动 MapReduce 作业)。

传统的关系型数据库

我们使用 Oracle 和 MySQL 数据库发布这些摘要。在这里的数据量相对较小，但查询率高并且需要实时响应。

DataBee

一个内部的 ETL 流程软件，用于提供在所有数据处理作业上可靠的批处理的一个常用框架。

复制作业不断将来自存储 Scribe 数据的 NFS 层的数据复制到 HDFS 集群。NFS 设备安装在 Hadoop 层，复制进程在 Hadoop 集群上作为 map-only 作业(map-only 作业是指该 MapReduce 作业仅含 map 阶段，没有 reduce 阶段。)运行。这使得复制进程易于扩展，也使它们能容错。目前，我们以这种方式每天从 Scribe 复制超过 6 TB 的数据到 HDFS。我们每天还从 MySQL 层下载多达 4 TB 的维度数据到 HDFS。这在 Hadoop 集群上也非常方便，只需 map-only 作业从 MySQL box 中复制数据。

Hadoop 配置

我们的 Hadoop 部署背后的思想是统一化。我们使用一个 HDFS 实例，并且绝大多数处理由一个 MapReduce 集群(运行一个 jobtracker)完成。原因非常简单。

- 通过操作一个集群，我们可以最大限度地降低运营费用。
- 数据不需要重复。前面描述的所有用例都可在一个地方得到所有数据。
- 在所有部分使用相同的计算集群获得了巨大的效率。
- 我们的用户是在协作环境中工作的，所以在服务质量方面的要求并不繁琐(至今)。

我们还有一个共享的 Hive 元数据存储(使用 MySQL 数据库)，可以存储在 HDFS 中有关所有 Hive 表的元数据。

14.2.3 虚拟案例研究

在本节中，我们将介绍一些大型网站中常见的典型问题，这些问题通过传统的仓储技术是难以解决的，因为涉及的成本和规模都过高。在这种情况下，Hadoop 和 Hive 可以提供一个更具扩展性，更有效的解决办法。

广告客户的见解和表现

Hadoop 最常见的用途之一是从大量数据上产生摘要。最典型的是大型广告网络，如 Facebook 的广告网络、谷歌的 AdSense 以及许多其他提供给广告客户有关其广告的标准统计，帮助广告客户有效地调整他们的活动。在大型数据集上计算广告表现数目是一个非常数据密集型操作，Hadoop 和 Hive 的可扩展性和成本优势可以真正帮助实现在合理时间和花费内计算这些数目。

许多广告网络提供给广告客户标准化的基于 CPC 和基于 CPM 的广告单元。CPC 广告是每次点击付费(cost-per-click)广告：广告客户络支付给广告网络的金额取决于用户访问网站时该广告获得的点击次数。另一方面，CPM 广告收取广告客户的金额是与用户在网站上看到该广告的次数成比例的。除了这些标准化的广告单元外，在过去几年里，为个别用户量身定做的内容更为灵活的广告在网络广告业中也越来越常见。雅虎通过 SmartAds 做到这一点，而 Facebook 提供给它的广告客户 Social Ads。后者允许广告客户从用户的朋友网络中嵌入信息。例如，耐克公司的广告可能提到用户的朋友，他最近成了耐克迷并在 Facebook 中与他的朋友共享这一信息。此外，Facebook 也提供给广告客户互动广告单元，其中，用户可以通过评论或播放内嵌视频更有效地与广告互动。一般来说，在线广告网络有多种广告提供给广告客户，这种多样性也增加了广告用户想要了解有关他们活动的各种表现数目的另一个方面。

就最基本而言，广告客户很想了解看到或点击广告的不同用户人数以及总次数。对于更多的动态广告，他们甚至可能还想了解在广告单元中出现的动态信息种类或用

户参与广告互动方式的统计。例如，一个特定的广告可能给 3 万不同用户显示了 10 万次。同样，一个嵌在互动广告的视频可能已被 10 万不同用户观看。此外，通常每个广告、活动和账户都有这些表现数目报告。一个账户可能有多个活动，每个活动可能有多个广告运行在网络上。最后，这些数目通常由广告网络按不同时间间隔报告。典型的时间间隔有每天，每周滚动，每月，每月滚动，有时甚至是活动的整个生命周期。此外，广告客户也会查看通过其他数据划分方式得到的这些数目的地域统计，如在亚太地区某特定广告的总观看人数或点击人数的比例。

很显然，有四个主要方面层次结构：账户；活动和广告层面；时间段；互动类型和用户层面。

其中最后一个用于报告不同数量的，而其他三个是报告层面的。用户层面也用来创建广告观看者和点击者的地域状况统计。所有这些信息集合起来便可以帮助广告客户调整其活动，以提高他们在广告网络中的效益。除了这些管线的多面性，如果没有像 Hadoop 这样用于大规模广告网络的技术，数据处理量和每天数据增长的速度就会使其难以扩展。例如在这里所写的，在 Facebook 中用于处理为广告表现数目的广告日志量每天大约有 1 TB(非压缩)。该日志量从 2008 年 1 月以来增长了 30 倍，而当时每天只有 30 GB 左右。Hadoop 与硬件同步扩展的能力成为了只需轻微调整作业配置这些管线便可跟上数据增长步伐的能力背后的一个主要因素。通常，这些配置更改涉及增加处理这些管线密集部分的 Hadoop 作业的 reducer 的数量。目前该阶段最大运行 400 个 reducer(从 2008 年 1 月的 50 个 reducer 增长了 8 倍)。

专案分析和产品反馈

除了定期报告，另一个数据仓库解决方案的主要用例是能够支持专案分析和产品反馈解决方案。例如，任何典型网站产品调整时，产品经理或工程师通常需要通过用户参与以及该功能点击率了解新功能的影响。产品团队甚至可能希望对该变化在不同区域和国家上的影响做更深入的分析。例如，这一变化是否在美国增加了用户点击率，或者是否在印度减少了用户参与。通过使用 Hadoop 的 Hive 和常规 SQL 可以实现许多这一类型的分析。点击率的衡量可以很容易地表示为与功能相关的链接的点击和看法的链接。该信息可以和地域信息连接起来以计算产品变化对不同地区的影响。随后人们可以通过聚集来计算不同地域的平均点击率。所有这些都很容易在 Hive 中用一系列 SQL 查询表达。(这将生成多个 Hadoop 作业)。只要有一个评估需求，就可以通过 Hive 自带的抽样功能在用户样本集上运行同样的查询。有些分析需要使用到自定义 map 和 reduce 脚本与 Hive SQL 的结合，而这也很容易插入到 Hvie 查询中。

一个更为复杂分析的例子是估计过去一整年每分钟登录到网站的用户数的峰值。这

将涉及抽样页面浏览日志(因为一个流行网站的总页面浏览数据是巨大的)，按时间将其分组，然后通过一个自定义 reduce 脚本找到不同时间点的新用户数。这是一个很好的例子，它需要 SQL 和 MapReduce 两者来解决最终用户问题并且显示了使用 Hive 实现它非常容易。

数据分析

Hive 和 Hadoop 可易用于对数据分析应用程进行训练和计分。 这些数据分析应用可以跨越多个领域，例如流行网站、生物信息公司和石油勘探公司等。该应用在网络广告行业的一个典型例子是预测一个广告的什么特性能使其更容易受到用户的注意。训练阶段通常涉及确认响应指标和预测特性。在该情况下，点击率是一个不错的衡量广告效益的指标。一些广告中令人感兴趣的特性可能是它所属的行业、广告内容、广告在页面上的位置等。Hive 可用于组合训练数据，然后将它们复制到一个数据分析引擎(通常是 R 或用户在 MapReduce 编写的程序)。在这种特定情况下，不同广告表现数目和特性可以在 Hive 中结构化为表。人们可以容易地对数据抽样(由于 R 只能处理有限的数据量，因此抽样是必须的)并使用 Hive 查询执行适当的聚集和连接操作以组合一个反馈表，该表包含最重要的决定广告效益的特性。然而，由于抽样会丢失信息，一些比较重要的数据分析应用程序使用并行实现体，它们是使用 MapReduce 框架的流行数据分析内核的实现体。

虽然绝大部分数据分析任务不执行每天计分，但一旦模型被训练了，它可以被部署用于每天计分。其中许多实际上是特别任务并且需要可以用来作为产品设计过程输入的一次性分析。

14.2.4 Hive

概览

开始使用 Hadoop 时，我们很快就对它的伸缩性和可用性印象深刻。然而，我们对其广泛运用心存顾虑，主要是因为用 Java 编写 MapReduce 程序的复杂性(以及培训用户编写程序的成本)。我们知道，很多公司的工程师和分析师了解 SQL 语句作为一种查询和分析数据工具，并且他们中的很多人精通一些脚本语言(如 PHP 和 Python)。因此，当务之急是需要开发一种软件来弥补用户精通的语言和 Hadoop 程序要求的语言之间的差距。

同时很明显的是，我们很多的数据集是结构化的并且是容易划分的。这些要求的结果自然是一个系统，该系统能将数据建模为表和划分，同时还可以提供一个类似

SQL 的语言用于查询和分析。同样重要的是能够在查询中插入自定义的 MapReduce 程序，这些程序可以由用户选择的编程语言编写。该系统就是 Hive。Hive 是一个建立在 Hadoop 最上层的数据仓库基础设施并且作为一种主要工具，用于查询存储在 Facebook 的 Hadoop 上的数据。在下面的章节中，我们将详细描述该系统。

数据组织

所有数据集上的数据被连续地组织起来并且被压缩、划分和排序存储：

压缩

几乎所有的数据集都存储为使用 gzip 编码的顺序文件。旧的数据集使用 bzip 编码被再次压缩，bizp 编码提供了比 gzip 更大的压缩。bzip 的比 gzip 慢，但旧数据的访问频率更少，而且就磁盘空间上的节省而言，这种性能损失是值得的。

分区

大部分数据集是按日期分区的。单个分区被加载到 Hive，Hive 将每个分区加载到一个单独的 HDFS 目录中。在多数情况下，分区是简单地基于与 scribe 日志文件关联的日期戳的。但是，在某些情况下，我们根据时间戳来扫描数据和整理数据，该时间戳可从一个日志条目中得到。更进一步，我们也将可以按多种属性分区数据(例如，国家和日期)。

排序

表内的每个分区往往是按唯一 ID(如果有的话)排序(和散列划分)的。这有几个主要优势。

- 很容易在数据集上运行抽样查询。
- 我们可以在排序的数据上建立索引。
- 在数据集上可以非常有效地执行包含唯一 ID 的聚集和联接操作。

加载数据到这个长期格式是由每日 MapReduce 作业完成的(这与近实时数据导入过程是不同的)。

查询语言

Hive 查询语言与 SQL 语言非常类似。它有传统的 SQL 结构，如 join，group by，where，select，from 子句及 from 子查询。它试图将 SQL 命令转换成一系列 MapReduce 作业。除常见 SQL 语句，它还有一系列其他扩展，如查询本身能指定自定义 mapper 和 reducer 脚本，能够扫描一次数据后插入多个表、分句、HDFS 或本地文件，能够在样本而不是完整的数据集上运行查询(在测试查询时这种能力是

相当有用的)。Hive 的元数据存储用表存储元数据，并向 Hive 编译器提供元数据，为将 SQL 命令转换为 MapReduce 作业。通过分区，map 端聚合和其他功能，编译器会尽力创建可以优化查询运行的计划。

使用 Hive 的数据管线

此外，就用 SQL 表达数据管线而言 Hive 提供了必要的灵活性，能以一种简单、快捷的方式将这些管线组合在一起。这对于目前仍在不断发展和扩大的组织和产品而言特别有用。许多在处理数据管线时所需要的操作是能被理解 SQL 操作，如联接、分组和唯一聚合。由于 Hive 这种能将 SQL 转化为 Hadoop MapReduce 作业的能力，创建和维护这些管道变得相对容易一些。在本节我们用一个虚拟广告网络例子来说明 Hive 的这些方面，并展示如何用 Hive 计算一些常见的广告客户需要的总结报告。作为一个例子，假设一个在线广告网络在 Hive 中将广告信息存储在一张名为 dim_ads 的表中并且将其广告显示信息存储在一张名为 impression_logs 的表中，后者按日期划分，2008-12-01 的每日显示数(按活动既有唯一又有总和，由广告网络例行提供给广告客户)在 Hive 中的表达如下：

```
SELECT a.campaign_id, count(1), count(DISTINCT b.user_id)
FROM dim_ads a JOIN impression_logs b ON(b.ad_id = a.ad_id)
WHERE b.dateid = '2008-12-01'
GROUP BY a.campaign_id;
```

这也是典型的 SQL 语句，人们可以在其他关系型数据库(如 Oracle 和 DB2)中使用。

为了计算来自之前已连接的数据的同一广告和账户的每日显示数，Hive 提供了执行多种分组的能力，如下所示查询(类似 SQL 但不是严格的 SQL)：

```
FROM(
  SELECT a.ad_id, a.campaign_id, a.account_id, b.user_id
  FROM dim_ads a JOIN impression_logs b ON (b.ad_id = a.ad_id)
  WHERE b.dateid = '2008-12-01') x
INSERT OVERWRITE DIRECTORY 'results_gby_adid'
  SELECT x.ad_id, count(1), count(DISTINCT x.user_id) GROUP BY
x.ad_id
INSERT OVERWRITE DIRECTORY 'results_gby_campaignid'
  SELECT x.campaign_id, count(1), count(DISTINCT x.user_id)
GROUP BY x.campaign_id
INSERT OVERWRITE DIRECTORY 'results_gby_accountid'
  SELECT x.account_id, count(1), count(DISTINCT x.user_id) GROUP
BY x.account_id;
```

鉴于 Hive 中已加入优化，查询可以转换成一系列 Hadoop MapReduce 作业，这些作业能够随着数据倾斜而伸缩。从本质上讲，连接被转换成一个 MapReduce 的作业，三个分组被转换为四个 MapReduce 作业，第一个作业按 unique_id 部分整合。这特别有用，因为 impression_logs 按 unique_id 分布要比按 ad_id 分布均匀得多(通常在一个广告网络，一些广告占据主导地位，因为他们对用户显示更为均匀)。因此，按 unique_id 部分聚合计算使得管线能更加均匀地将作业分发到 reducer。同样的模板只要在查询中改变日期谓词便可用于计算不同时期内的表现数目。

然而，计算生命周期更加棘手，就像使用之前描述的策略一样，人们必须扫描 impression_logs 表中的所有分区。因此，为了计算生命周期，一个更加可行的策略是每天在一个中间表的一个分区上按 ad_id，unique_id 组存储一个生命周期计数。该表内的数据连同后几天的 impression_logs 可用于逐步产生广告表现数目生命周期。举例来说，为了获得 2008 年 12 月 1 日的每日显示数，可使用 2008 年 11 月 30 日的中间表分区。用于实现该功能的 Hive 查询如下：

```
INSERT OVERWRITE lifetime_partial_imps PARTITION(dateid='2008-
12-01')
SELECT x.ad_id, x.user_id, sum(x.cnt)
FROM (
  SELECT a.ad_id, a.user_id, a.cnt
  FROM lifetime_partial_imps a
  WHERE a.dateid = '2008-11-30'
  UNION ALL
  SELECT b.ad_id, b.user_id, 1 as cnt
  FROM impression_log b
  WHERE b.dateid = '2008-12-01'
) x
GROUP BY x.ad_id, x.user_id;
```

此查询计算了 2008 年 12 月 1 日的部分总和，这可用于计算在 2008 年 12 月 1 日以及 2008 年 12 月 2 日的数目(此处未显示)。该 SQL 实际上被转换为一个 Hadoop MapReduce 作业，该作业在已结合的输入流上计算分组。此 SQL 可以跟在以下 Hive 查询之后，它计算不同分组的实际数量(类似于在每日管线上的一样)：

```
FROM(
  SELECT a.ad_id, a.campaign_id, a.account_id, b.user_id
  FROM dim_ads a JOIN impression_logs b ON (b.ad_id = a.ad_id)
  WHERE b.dateid = '2008-12-01') x
INSERT OVERWRITE DIRECTORY 'results_gby_adid'
  SELECT x.ad_id, sum(x.cnt), count(DISTINCT x.user_id) GROUP BY
```

```
x.ad_id
INSERT OVERWRITE DIRECTORY 'results_gby_campaignid'
  SELECT x.campaign_id, sum(x.cnt), count(DISTINCT x.user_id)
GROUP BY x.campaign_id
INSERT OVERWRITE DIRECTORY 'results_gby_accountid'
  SELECT x.account_id, sum(x.cnt), count(DISTINCT x.user_id)
GROUP BY x.account_id;
```

Hive 和 Hadoop 是批处理系统，它们提供计算数据的延迟不如一般的关系型数据库(如 Oracle 或 MySQL)。因此，在许多情况下，将 Hive 和 Hadoop 生成的摘要加载到一个传统的关系型数据库，通过不同的 BI 工具甚至是一个网站入口将数据提供给用户仍然是很有用的。

14.2.5 存在的问题及未来的工作

公平共享

Hadoop 集群通常运行一系列日常生产作业，这些日常作业与不同优先级、大小的 Adhoc 作业一样需要在合理时间内完成计算。在典型的设置下，这些作业往往在晚上运行，这期间来自用户运行的 Adhoc 作业的干扰很小。然而，如果没有足够的保障，大型 Adhoc 作业和生产作业之间的重叠往往是不可避免的，而这会影响生产作业的延迟。ETL 处理也包含几个近实时作业，这些作业必须在小时段内完成(其中包括从 NFS 服务器上复制 Scribe 数据的进程，以及一些数据集的每小时摘要)。这也意味着，一个无赖作业可能导致整个集群的崩溃并且将生产进程置于危险之中。

由 Facebook 开发并回馈于 Hadoop 的公平共享的 Hadoop 作业调度程序提供了这些问题的解决方案。它为 Adhoc 作业池保留了必须的计算资源，同时让所有人能使用闲置资源。它还可以在这些池之间公平地分配计算资源以防止大型作业占据集群资源。内存在集群中是最抢手的资源之一。我们对 Hadoop 做了一些修改，使得如果 jobTracker 内存资源不足，则取消 Hadoop 作业的提交。这可以使得用户进程运行限制在一个合理的内存范围中，而且可以在适当的地方设置一些监测脚本，以防止 MapReduce 作业影响运行在同一个节点上的 HDFS 守护进程(这主要是由于高内存消耗)。日志目录都存储在单独的磁盘分区并定期清理，同时我们认为把 MapReduce 中间存储也放在单独的磁盘分区也是很有用的。

空间管理

容量管理仍然是一个巨大的挑战，因为利用率和数据增长的速度非常快。许多有着

不断增长数据集的新兴公司也有同样的痛苦。在许多情况下，这些数据实际上是临时性的。在这种情况下， 可以使用 Hive 的保留设置并且可以按 bzip 格式再次压缩旧数据以节省空间。虽然从磁盘存储观点来看配置基本上是对称的，但增加一层单独的高存储密度设备层来保存旧数据可能是有益的。这会使在 Hadoop 中存储归档数据变得便宜。然而，获得这些数据应该是透明的。我们目前正在研究数据归档层，它使得统一所有处理旧数据方面的操作成为可能。

Scribe-HDFS 整合

目前，如前所述，Scribe 向一些 NFS 文件服务器中写入数据，而在 NFS 文件服务器中这些数据将被自定义复制作业取出并分发到 HDFS。我们正在努力使 Scribe 能直接写入到另一个 HDFS 的实例。这将使 Scribe 的扩展和管理变得很容易。由于对 Scribe 需要有高利用率，它的目标 HDFS 实例可能与生产 HDFS 实例不同(由于用户作业，它与任何负载/停机问题是不同的)。

Hive 的改进

Hive 仍在积极发展。一系列核心特性正在研究中，如对 order by[①]和 having 语句的支持，更多的聚合函数，更多的内置函数，日期时间，数据类型，等等。同时，一些性能优化也正在研究中，如谓词推测和公共子表达式简化。在一体化方面，正开发 JDBC 和 ODBC 驱动程序以整合 OLAP 和 BI 工具。通过所有这些优化，我们希望能释放 MapReduce 和 Hadoop 的能力，并且使它在 Facebook 中更接近非工程社区。欲了解更多有关该项目的资料，请访问 *http://hadoop.apache.org/hive/*。

(作者 Joydeep Sen Sarma 和 Ashish Thusoo)

14.3 Hadoop 在 Nutch 搜索引擎

14.3.1 背景

Nutch 是一个用以建立可扩展的互联网爬虫和搜索引擎的框架。这是 Apache 软件基金会的一个项目和 Lucene 的一个子项目，它处于 Apache 2.0 协议中。

我们不会深入研究网络爬虫的内部，本案例研究只是表明 Hadoop 如何应用于搜索

① Hive 在 0.3.1 版已加入 order by 功能，但 0.3.1 版未发布。实际上在 0.4.0 发布版上正式出现。

引擎来实现各种复杂的处理任务。有兴趣的读者可以在该项目的官方网站(*http://lucene.apache.org/nutch*)上找到大量针对 Nutch 的资料。我只想说，要想创建和维护一个搜索引擎，需要以下子系统。

页数据库

该数据库存储了爬虫已知的所有网页及其状态的记录，如上一次访问该网页的时间，其提取状态、刷新间隔和内容校验等。在 Nutch 术语中，这个数据库被称为 CrawlDb。

要抓取的网页列表

由于爬虫定期更新它们对网络的认知，因此它们下载新的网页(以前看不到的)或刷新它们认为已经过期的网页。Nutch 将此称为候选网页列表，它们用于抓取 fetchlist。

原始网页数据

网页内容是从远程站点下载的，并按之前解释过的形式，即作为一个字节数组存储在本地。在 Nutch 中这个数据被称为网页内容。

网页数据解析

之后，用合适的解析器对网页内容进行解析——Nutch 提供多种流行文件格式的解析器，如 HTML，PDF，Open Office 和 Microsoft Office，RSS 及其他格式。

链接图数据库

这个数据库对计算计算基于链接的网页排名是必要的，如 PageRank。对于 Nutch 所知的每个 URL，它包含了一个列表，该列表包含指向它的其他 URL 及其相关的锚文本(从 HTML 的`<a href="..">`和锚文本`</a>`元素中获得)。这个数据库被称为 LinkDb。

全文搜索索引

这是一个建立在收集到的网页元数据和提取的纯文本内容上的典型的倒序索引。它通过优秀的 Lucene 库实现(*http://lucene.apache.org/java*)。

我们之前简单提过，Hadoop 开始是作为 Nutch 的一个组件，是为了提高其可扩展性并解决由集中的数据处理模型造成的明显的性能瓶颈。Nutch 也是第一个移植到框架中的公开的概念验证应用程序，该框架就是后来的 Hadoop，而将 Nutch 的算法和数据结构移植到 Hadoop 所需的工作竟然如此之小。这可能促进了 Hadoop 以后作为一个单独子项目的发展，给除了 Nutch 以外的应用提供一个可重用的框架。

目前，几乎所有 Nutch 工具都是通过运行一个或多个 MapReduce 作业来处理数据。

14.3.2 数据结构

Nutch 维护着几个主要的数据结构，它们都利用了 Hadoop I/O 类和格式。根据这些数据的用途和它被创建后的访问方式，它们使用 Hadoop map 文件或序列文件存放。

由于这些数据是由 MapReduce 作业产生和处理的(这会运行一些 map 和 reduce 任务)，其磁盘上的布局与常见 Hadoop 输出格式相对应，即 `MapFileOutputFormat` 和 `SequenceFileOutputFormat`。因此确切地说，数据被保存在几个局部 map 文件或序列文件，其数量与在作业中创建这些数据的 reduce 任务数量相同。为简单起见，后文会忽略这个区别。

CrawlDb

CrawlDb 以`<url, CrawlDatum>`的 map 文件形式存储每个 URL 的状态，其中键使用 `Text` 类，值使用 Nutch 特定的 `CrawlDatum` 类(实现了 `Writable` 接口)。

为了提供对记录的快速随机访问(有时当用户想检查 CrawlDb 单个记录时，这对于诊断很有用)这些数据被存储在 map 文件上，而不是序列文件。

CrawlDb 最初使用 Injector 工具创建，它只是将代表初步 URL 列表(称为种子列表)的纯文本转换成如前所述格式的 map 文件。随后，它被更新为从已抓取和解析的网页上获取的信息，详见后文。

LinkDb

该数据库存储 Nutch 所知的每个 URL 的传入链接信息。它是一个`<url, Inlinks>`的 map 文件，其中 Inlinks 是一个 URL 和锚文本数据列表。值得注意的是，这一信息不会在页面收集期间立即可得，但相反信息是可得的，即一个网页的外向链接。将这种关系逆向处理是由不久会描述的一个 MapReduce 作业实现的。

段

段在 Nutch 的说法是对应于获取和解析一批 URL 的。图 14-5 阐述了如何创建和处理段。

段(实际上是文件系统的一个目录)包含以下部分(只是包含 `MapFileOutputFormat` 或 `SequenceFileOutputFormat` 数据的子目录)。

content

包含已下载网页的原始数据，以`<url, Content>`的 map 文件形式存储。这里 Nutch 使用了一个 map 文件，因为它需要快速随机访问以呈现页面缓存视图。

crawl_generate

包含将要抓取的 URL 列表以及从 CrawlDb 获取的它们目前的状态，以<url, CrawlDatum>的序列文件形式存储。这个数据使用序列文件，首先是因为它是顺序处理，其次是因为我们无法满足 map 文件已排序键顺序不变的要求。我们需要尽可能远地传播属于同一主机的 URL，以减少每个目标主机的负载，这也就是说，记录或多或少是随机排序的。

crawl_fetch

包含抓取的状态报告，即它是否成功，响应代码是什么等。存储在<url, CrawlDatum>的 map 文件中。

crawl_parse

每个成功抓取并解析的页面的外向链接列表存储在这里，这样 Nutch 可以通过了解新 URL 扩大它的抓取边界。

parse_data

解析期间收集的元数据及一个页面的外向链接列表。此信息对于稍后建立逆向图(传入链接—向内链接)是至关重要的。

parse_text

页面的纯文本版本，适合在 Lucene 索引。这些被存储为一个<url, ParseText>的 map 文件，以便在构建摘要(摘录)以显示搜索结果列表时，Nutch 可以迅速访问它们。

Generator 工具运行时(图 14-5 的 1)，新段从 CrawlDb 创建，最初只包含要抓取的 URL 列表(即 crawl_generate 子目录)。当该列表被几个步骤处理之后，该段收集来自一系列子目录中的处理工具的输出数据。

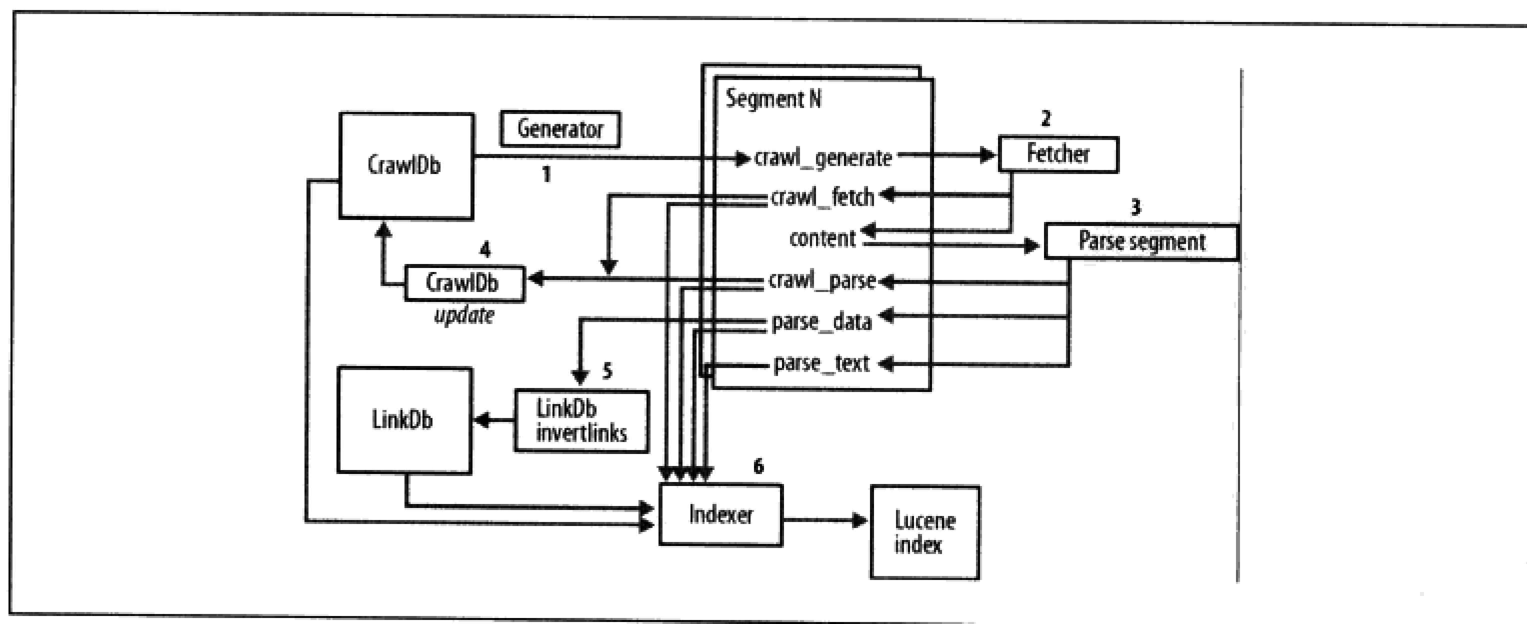


图 14-5：段

例如，一个名为 Fetcher 的工具填充内容，它从抓取列表的 URL 下载原始数据(2)。此工具还在 crawl_fetch 中保存状态信息，以便用于以后在 CrawlDb 中更新页面状态。

该段的其余部分被段解析工具填充(3)，它读取内容部分，选择合适的基于声明(或检测)为 MIME 类型的内容解析器，并将解析结果保存为三个部分：crawl_parse，parse_data 和 parse_text。这些数据之后将用于以新信息更新 CrawlDb(4)和创建 LinkDb(5)。

段将被保留到目前在内的所有网页都过期。Nutch 有一个可配置的最长时间限制，过了该限制之后网页将被强行选中重新抓取。这有助于操作者逐步淘汰所有超过此限制的段(因为他可以确定到那个时候该段内的所有网页都会被重新抓取)。

段数据主要用于创建 Lucene 索引([6]——主要是 parse_text 和 parse_data 部分)，但它也提供了一个用于快速检索纯文本和原始内容数据的数据存储机制。前者是必要的，以便 Nutch 可以生成片段(匹配查询最佳的文档文本的片段)，后者提供呈现页面“缓存视图”的能力。在这两种情况下，数据可以从 map 文件中直接访问以响应片段生成或缓存内容的请求。实际上，即使对于大量存储，从 map 文件中直接访问数据的性能已相当足够。

14.3.3 Nutch 中 Hadoop 数据处理精选实例

下文显示了一些 Nutch 工具的相关细节来说明如何将 MapReduce 模式应用于 Nutch 中的实际数据处理任务。

链接反向

抓取过程中收集的 HTML 页面包含 HTML 链接，其中可能指向自身(内部链接)也可能指向其他页面。HTML 链接是从源页面指向目标页面。见图 14-6。

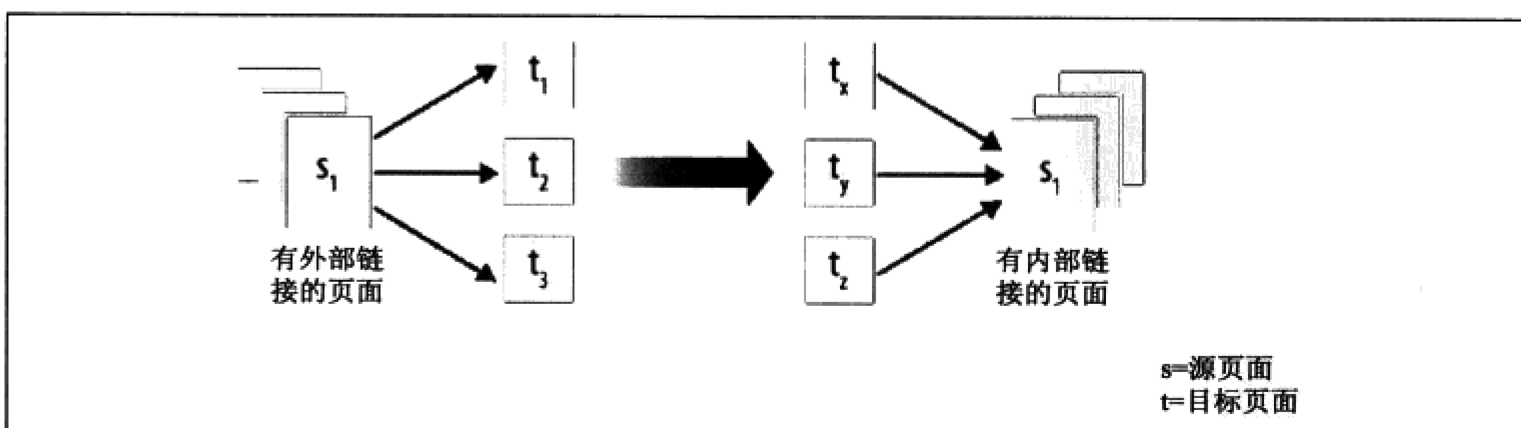


图 14-6：链接反向

然而，计算一个网页的重要性(或质量)的大部分算法需要相反的信息，即哪些网页包含指向当前页的外向链接。此信息在抓取时不能立即得到。此外，同时考虑到内向链接的锚文本有助于索引处理，以便该锚文本在语义上丰富当前页面的文本。

如前所述，Nutch 收集外向链接信息，然后用这些数据建立一个 LinkDb，其中以内向链接和锚文本形式包含了反向链接数据。

本节介绍 LinkDb 工具实现的大体轮廓。为了很多阐述清晰，细节被忽略了(例如 URL 规范化和过滤)。留下的部分给出了一个典型的例子来阐述为什么 MapReduce 模式非常适合运行搜索引擎所需的关键数据转换处理。大型搜索引擎需要处理大量网络图数据(许多网页上包含很多内向/外向链接)，而 Hadoop 提供的并行性和容错性使这成为可能。此外，很容易用如下所示的 map-sort-reduce 原语表达链接反向。

下面的代码片断介绍了 LinkDb 工具的作业初始化：

```
JobConf job = new JobConf(configuration);
FileInputFormat.addInputPath(job, new Path(segmentPath, "parse_data"));
job.setInputFormat(SequenceFileInputFormat.class);
job.setMapperClass(LinkDb.class);
job.setReducerClass(LinkDb.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Inlinks.class);
job.setOutputFormat(MapFileOutputFormat.class);
FileOutputFormat.setOutputPath(job, newLinkDbPath);
```

我们可以看到，这项作业的源数据是已抓取的 URL 列表(键)和相应的包含每个网页外向链接信息的 ParseData 记录，它是一个外向链接数组的形式。一个外向链接包含目的 URL 和锚文本。

该作业的输出也是一个 URL 列表(键)，但值是内向链接的实例，它只是一个专门的包含目的 URL 和锚文本的内向链接集合。

也许令人惊讶，URL 通常以纯文本格式存储和处理而不是 `java.net.URL` 或 `java.net.URI` 实例。有几个原因：从下载的内容中提取的 URL 通常需要规范化(例如，将主机名转换为小写，解决相对路径问题)；常常损坏或无效或涉及不支持的协议。许多规范化和过滤操作能更好地表达为跨越 URL 几个部分的文本模式。此外，为了链接分析的目的，我们可能仍然要处理和统计无效的 URL。

现在让我们仔细看看 `map()`和 `reduce()`的实现——在这里，它们简单到足够在同一个类里实现：

```
public void map(Text fromUrl, ParseData parseData,
    OutputCollector<Text, Inlinks> output, Reporter reporter) {
  ...
  Outlink[] outlinks = parseData.getOutlinks();
  Inlinks inlinks = new Inlinks();
  for (Outlink out : outlinks) {
  inlinks.clear(); // instance reuse to avoid excessive GC
  String toUrl = out.getToUrl();
    String anchor = out.getAnchor();
    inlinks.add(new Inlink(fromUrl, anchor));
    output.collect(new Text(toUrl), inlinks);
  }
}
```

从该清单中可以看到，对于每个 Outlink 我们的 map()都会产生一对<toUrl, Inlinks>，其中 Inlinks 只包含一个含有 fromUrl 和锚文本的 Inlink。该链接的方向已被反向。

之后，这些只含一个元素的 Inlinks 在 reduce()中被聚集：

```
public void map(Text fromUrl, ParseData parseData,
    OutputCollector<Text, Inlinks> output, Reporter reporter) {
  Inlinks inlinks = new Inlinks();
  while(values.hasNect()){
    result.add(vaues.next());
  }
  output.collect(tour, result);
}
```

从该代码很明显可以看出，我们已经得到我们所期望的——即一个包含所有指向 toUrl 的 fromUrls 以及其锚文本的列表。反向处理已经完成。

然后使用 MapFileOutputFormat 保存这些数据，并且这些数据成为了 LinkDb 的新版本。

fetchlist 抓取列表的生成

现在让我们来看看一个更复杂的用例。Fetchlists 从 CrawlDb 产生(这是一个<url, crawlDatum>的 map 文件以及包含此 URL 状态的 crawlDatum)，它们包含准备抓取的 URL，这些 URL 之后将由 Nutch Fetcher 工具处理。Fetcher 本身就是一个 MapReduce 应用程序(稍后描述)。这意味着输入数据(被划分为 N 个部分)将被 N 个 map 任务处理，Fetcher 使得 SequenceFileInputFormat 不再分隔数据超过已有输

入划分数量。我们刚才简单地提到 fetchlist 需要通过一种特殊的方式产生，以使 fetchlist 每个部分的数据(也因此在每个 map 任务中处理)符合特定要求。

1. 所有来自同一主机上的 URL 最终需要在同一个分区。这是必需的，因为这样 Nutch 可以容易地实现内部 JVM 主机级模块化，以避免大量目标主机。
2. 来自同一主机的 URL 距离应尽可能远(例如：很好地与来自其他主机的 URL 混合)，以尽量减少主机级阻塞。
3. 不应该有来自同一台主机超过 x 数量的 URL，这样有着许多 URL 的大型网站就不会控制小网站(来自小网站的 URL 仍然有机会被安排抓取)。
4. 高分 URL 应优先于低分 URL。
5. 在 fetchlist 中总共最多有 y 个 URL。
6. 输出划分的数量应该符合 map 抓取任务的最佳数目。

在这种情况下，需要两个 MapReduce 作业来满足所有图 14-7 所示的要求。同样，在下面的清单中，为简明起见，我们要跳过这些步骤的某些细节。

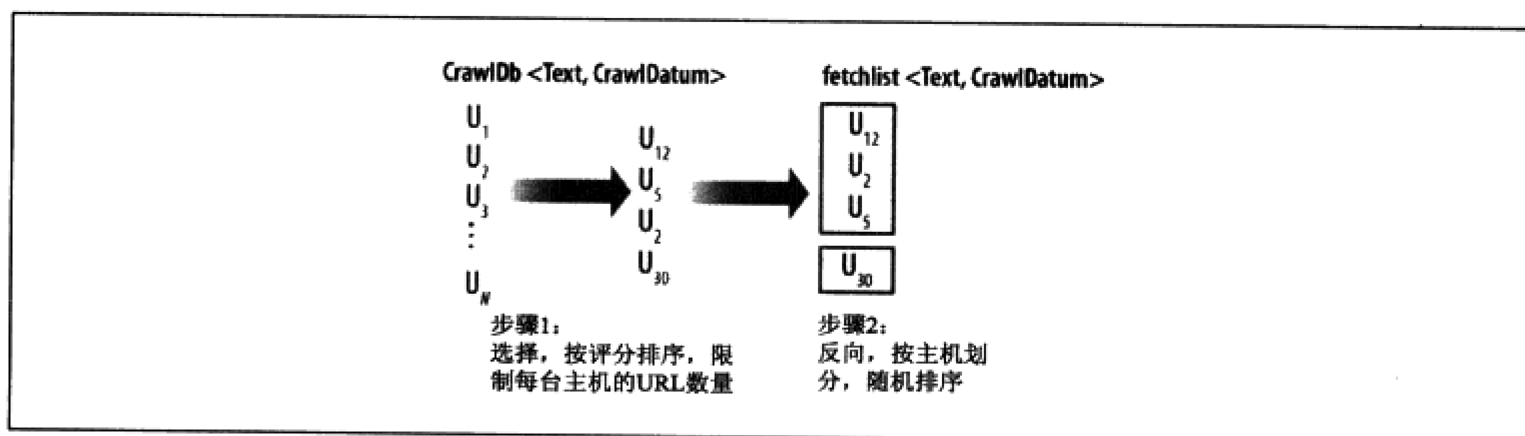


图 14-7：fetchlist 的生成

步骤 1：选择，按评分排序，限制每台主机的 URL 数量 在这一步，Nutch 运行一个 MapReduce 作业来选择被认为是有资格被抓取的 URL，并按评分对其进行排序(分配给每个 URL 的浮点值，例如，一个 PageRank 值)。输入数据来自 CrawlDb，这是一个`<url, datum>`map 文件。这个作业的输出是一个`<score,<url, datum>>`序列文件，按照评分降序排列。

首先，让我们看看作业启动：

```
FileInputFormat.addInputPath(job, crawlDbPath);
job.setInputFormat(SequenceFileInputFormat.class);
job.setMapperClass(Selector.class);
job.setPartitionerClass(Selector.class);
job.setReducerClass(Selector.class);
FileOutputFormat.setOutputPath(job, tempDir);
```

```
job.setOutputFormat(SequenceFileOutputFormat.class);
job.setOutputKeyClass(FloatWritable.class);
job.setOutputKeyComparatorClass(DecreasingFloatComparator.class);
job.setOutputValueClass(SelectorEntry.class);
```

Selector 类实现三个函数：mapper，reducer 和 partitioner。最后一个函数特别有趣：Selector 使用一个自定义的划分函数将来自同一主机的 URL 分配给同一 reduce 任务，这样我们能满足前面列表上的标准 3-5。如果我们不改写默认的划分函数，来自同一主机的 URL 最终将在输出的不同分区中，而我们将无法跟踪和限制总计数，因为 MapReduce 任务之间没有沟通。像现在这样，所有属于同一主机的 URL 将最终被同一 reduce 任务处理，这意味着我们可以控制每台主机被选择的 URL 数量。

可以很容易地实现一个自定义的划分函数，使得在同一任务内的需要处理的数据最终在同一分区中。让我们先看看 Selector 类如何实现划分函数接口(其中只包括一个方法)：

```
/** Partition by host. */
public int getPartition(FloatWritable key, Writable value, int
numReduceTasks) {
  return hostPartitioner.getPartition(((SelectorEntry)value).url,
key, numReduceTasks);
}
```

该方法返回一个从 0～numReduceTasks - 1 的整数。它只是简单地将键替换为来自 SelectorEntry 的原始 URL，以将 URL(而不是评分)传递给一个 PartitionUrlByHost 实例，PartitionUrlByHost 实例计算了分区数量：

```
/** Hash by hostname. */
public int getPartition(Text key, Writable value, int
numReduceTasks) {
  String urlString = key.toString();
  URL url = null;
  try {
    url = new URL(urlString);
  } catch (MalformedURLException e) {
    LOG.warn("Malformed URL: '" + urlString + "'");
  }
  int hashCode = (url == null ? urlString :
url.getHost()).hashCode();
  // make hosts wind up in different partitions on different runs
  hashCode ^= seed;
```

```
    return (hashCode & Integer.MAX_VALUE) % numReduceTasks;
  }
```

正如所看到的代码段，计算分区数量功能只是基于 URL 的主机部分，这意味着所有属于同一主机的 URL 最终将在同一个分区中。

这个作业的输出按评分降序排列。由于在 CrawlDb 中有很多记录分数相同，所以我们不能使用 MapFileOutputFormat，因为我们将违反 map 文件的严格以键顺序不变的要求。

细心的读者会发现，我们必须使用其他的东西而不是原来的键，但我们仍希望保留原始键/值对，这里我们使用一个 Selector Entry 类将原始键/值对传递给下一个处理步骤。

Selector.reduce()跟踪着 URL 总数以及每个主机的 URL 最大数目，并且简单地舍弃多余的记录。请注意，执行总数限制必然只是近似的。我们是按照总限额除以 reduce 任务数量来计算当前任务的限额。但是，我们不确定任务内是否会得到平均数量的 URL。事实上，在多数情况下由于主机之间 URL 的分配不均，它是不会发生的。但是，对于 Nutch 来说这种近似就足够了。

步骤 2：反向，按主机划分，随机排序 在上一步中，我们最终得到的是<score, selectorEntry>序列文件。现在我们必须产生一个<url, datum>序列文件并满足刚才描述的标准 1，2 和 6。此步骤的输入数据是步骤一产生的输出数据。

下面的片断显示了这个作业的启动：

```
FileInputFormat.addInputPath(job, tempDir);
job.setInputFormat(SequenceFileInputFormat.class);
job.setMapperClass(SelectorInverseMapper.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(SelectorEntry.class);
job.setPartitionerClass(PartitionUrlByHost.class);
job.setReducerClass(PartitionReducer.class);
job.setNumReduceTasks(numParts);
FileOutputFormat.setOutputPath(job, output);
job.setOutputFormat(SequenceFileOutputFormat.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(CrawlDatum.class);
job.setOutputKeyComparatorClass(HashComparator.class);
```

SelectorInverseMapper 类简单地舍弃当前键(评分值)，提取原始 URL 并且用它

作为键，用 SelectorEntry 作为值。细心的读者可能会问，为什么我们不更进一步，同时提取原始 CrawlDatum 并用它作为值，稍后详细解释。

这项作业的最终输出是一个<Text, CrawlDatum>序列文件，但 map 阶段的输出使用了<Text, SelectorEntry>。我们必须使用 setMapOutputKeyClass()和 setMapOutputValueClass()的 setter 来指明我们为 map 输出使用了不同的键/值类，否则，Hadoop 假定我们使用为 reduce 输出声明的相同的类(该冲突往往会导致作业失败)。

用 PartitionUrlByHost 类对 map 阶段输出进行分区，以便再次将同一主机的 URL 分配到相同的分区中。这满足要求 1。

一旦数据从 map 到 reduce 任务中被打乱，它将按照输出键比较器被 Hadoop 排序，在这里，比较器是 HashComparator。这个类使用一个简单的散列方案将 URL 按照来自同一主机的 URL 最不可能放在一起的方式混合起来。

为了满足要求 6，我们将 reduce 任务数目设置为 Fetcher map 任务所需的数目(前面提到的 numParts)，牢记每个 reduce 分区将用于以后创建单个 Fetcher map 任务。

PartitionReducer 类负责最后一步，即将<url,selectorEntry>转换为<url, crawlDatum>。一个使用 HashComparator 的惊人的副作用是几个网址可能被散列到相同的哈希值，而 Hadoop 调用 reduce()方法仅传递它们的第一个键，所有其他键将被认为是相同的而丢弃。为什么我们要在 SelectorEntry 记录中保留所有 URL 的原因现在变得很清晰了，因为现在我们可以从迭代值中提取它们了。这里是该方法的实现：

```
public void reduce(Text key, Iterator<SelectorEntry> values,
    OutputCollector<Text, CrawlDatum> output, Reporter reporter)
throws IOException {
  // when using HashComparator, we get only one input key in
case of hash collisions
  // so use only URLs extracted from values
  while (values.hasNext()) {
    SelectorEntry entry = values.next();
    output.collect(entry.url, entry.datum);
  }
}
```

最后，reduce 任务的输出存储为一个在 Nutch 段目录内 crawl_generate 子目录中的 SequenceFileOutputFormat。这个输出满足从 1～6 的所有条件。

Fetcher：一个多线程的 MapRunner 实践

Nutch 的 Fetcher 应用负责从远程站点下载网页内容。因此，为了减少抓取 fetchlist 花费的时间，该进程利用一切机会并行执行是很重要的。

目前 Fetcher 中已有一级并行——输入的 fetchlist 的多个部分被分配到多个 map 任务。但这在实践中是不够的：从不同的主机顺序下载 URL(见前面的 HashComparator)将是一个巨大的时间浪费。由于这个原因，Fetcher map 任务使用多个工作线程处理数据。

让我们从作业的启动开始：

```
job.setSpeculativeExecution(false);
FileInputFormat.addInputPath(job, "segment/crawl_generate");
job.setInputFormat(InputFormat.class);
job.setMapRunnerClass(Fetcher.class);
FileOutputFormat.setOutputPath(job, segment);
job.setOutputFormat(FetcherOutputFormat.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(NutchWritable.class);
```

首先，我们关闭其他执行。我们不能运行多个下载来自同一主机的内容的 map 任务，因为这将违反主机级别的负载限制(例如并发请求的数量和每秒请求数量)。

接着，我们使用自定义的 InputFormat 实现类来防止 Hadoop 将输入数据分割成更小块从而创建比已有输入划分更多的 map 任务。这再次确保了我们控制了主机级别的访问限制。

使用自定义 OutputFormat 实现类来存储输出数据，它使用 NutchWritable 值中包含的数据创建了几个输出 map 文件和序列文件。该 NutchWritable 类是 GenericWritable 的子类，它可以传递之前声明过的几个不同的 Writable 类实例。

Fetcher 类实现了 MapRunner 接口，并且我们将该类作为作业的 MapRunner 的实现。这里列出了代码的相关部分：

```
public void run(RecordReader<Text, CrawlDatum> input,
    OutputCollector<Text, NutchWritable> output,
                 Reporter reporter) throws IOException {
  int threadCount = getConf().getInt("fetcher.threads.fetch", 10);
  feeder = new QueueFeeder(input, fetchQueues, threadCount * 50);
  feeder.start();
```

```
    for (int i = 0; i < threadCount; i++) { // spawn threads
      new FetcherThread(getConf()).start();
    }
    do {                                 // wait for threads to exit
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {}
      reportStatus(reporter);
    } while (activeThreads.get() > 0);
  }
```

Fetcher 事先读取了许多输入记录，它使用了 QueueFeeder 线程将输入记录放到一系列单台主机队列中。然后，几个 FetcherThread 实例启动了，它们从主机队列中取出队列元素，而同时 QueueFeeder 不断读取输入数据填满队列。每个 FetcherThread 从任意非空队列内取出队列元素。

与此同时，map 任务的主线程原地等待所有线程完成工作。它定期向 Hadoop 框架报告其状态，防止 Hadoop 认为该任务已完结并结束它。一旦所有队列元素都得到处理，循环便结束且控制权交回 Hadoop，它便认为此 map 任务完成了。

索引：使用自定义 OutputFormat

这是一个不会产生序列文件或 map 文件输出的 MapReduce 应用程序的例子，相反，该应用程序的输出是一个 Lucene 索引。同样，由于 MapReduce 应用程序可能由若干 reduce 任务组成，该应用程序的输出可能由若干部分 Lucene 索引组成。

Nutch Indexer 工具使用 CrawlDb，LinkDb 和 Nutch 段中的信息(抓取状态，解析状态，网页元数据和纯文本数据)，所以这项作业的设置部分包括添加几条输入路径：

```
FileInputFormat.addInputPath(job, crawlDbPath);
FileInputFormat.addInputPath(job, linkDbPath);
// add segment data
FileInputFormat.addInputPath(job, "segment/crawl_fetch");
FileInputFormat.addInputPath(job, "segment/crawl_parse");
FileInputFormat.addInputPath(job, "segment/parse_data");
FileInputFormat.addInputPath(job, "segment/parse_text");
job.setInputFormat(SequenceFileInputFormat.class);
job.setMapperClass(Indexer.class);
job.setReducerClass(Indexer.class);
FileOutputFormat.setOutputPath(job, indexDir);
job.setOutputFormat(OutputFormat.class);
```

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(LuceneDocumentWrapper.class);
```

一个 URL 所有的相应记录是分散在这些输入部分的，需要将它们结合起来才能创建 Lucene 文件并添加到索引中。

Indexer 中的 `Mapper` 实现只是将输入数据，不论其来源和实现类，封装在一个 `NutchWritable` 中，因此 reduce 阶段可能会收到不同来源、使用不同类的数据，但从 map 和 reduce 步骤中仍然能够只声明一个输出值类(如 `NutchWritable`)。

Reducer 实现遍历了所有属于同一个键(URL)的值，分解数据(取出 `CrawlDatum`, `CrawlDb CrawlDatum`, `LinkDb Inlinks`, `ParseData` 和 `ParseText`)，并使用该信息构建一个 Lucene 文件，该文件之后被封装到 `LuceneDocumentWrapper` 并收集。除了所有文本内容(来自于纯文本数据或元数据)，该文件还包含一个类似 PageRank 的评分信息(从 CrawlDb 数据中获得)。Nutch 用这个评分设置 Lucene 文件的附加值。

OutputFormat 的实现是该工具中最令人感兴趣之处：

```
public static class OutputFormat extends
    FileOutputFormat<WritableComparable, LuceneDocumentWrapper> {

  public RecordWriter<WritableComparable, LuceneDocumentWrapper>
      getRecordWriter(final FileSystem fs, JobConf job,
                      String name, final Progressable progress)
                      throws IOException {
  final Path out = new Path(FileOutputFormat.getOutputPath(job), name);
  final IndexWriter writer = new IndexWriter(out.toString(),
                     new NutchDocumentAnalyzer(job), true);

  return new RecordWriter<WritableComparable, LuceneDocumentWrapper>() {
    boolean closed;
    public void write(WritableComparable key,
        LuceneDocumentWrapper value)
        throws IOException {             // unwrap & index doc
    Document doc = value.get();
    writer.addDocument(doc);
    progress.progress();
  }

  public void close(final Reporter reporter) throws IOException {
    // spawn a thread to give progress heartbeats
    Thread prog = new Thread() {
```

```
          public void run() {
            while (!closed) {
              try {
                reporter.setStatus("closing");
                Thread.sleep(1000);
              } catch (InterruptedException e) { continue; }
                catch (Throwable e) { return; }
              }
            }
          };

        try {
          prog.start();
          // optimize & close index
          writer.optimize();
          writer.close();
        } finally {
          closed = true;
        }
      }
    };
  }
```

需要一个 `RecordWriter` 实例时，`OutputFormat` 便打开一个 `IndexWriter` 创建一个新的 Lucene 索引。然后，对于 reduce 方法收集的每个新输出记录，它从 `LuceneDocumentWrapper` 中将 Lucene 文件分解出来并将其添加到索引中。

当一个 reduce 任务完成后，Hadoop 会尝试关闭 `RecordWriter`。在这种情况下，关闭过程可能需要很长时间，因为我们想在关闭之前对索引进行优化。在此期间，由于没有进度更新，Hadoop 可能认为任务被挂起并试图杀死它。为此，我们首先启动一个后台线程保证进度更新，然后继续执行索引优化。一旦优化完成后，我们停止进展更新线程。现在，输出索引已被创建、优化、关闭并准备在搜索应用中被使用。

14.3.4 小结

这里的 Nutch 简要概述必然省略了许多细节，如错误处理、记录、URL 过滤和规范化，处理重定向或其他形式的“别名”网页(如镜像)，删除重复内容，计算 PageRank 评分等，可以在该项目官方网页和维基(*http://wiki.apache.org/nutch*)找到更多资料。

今天，Nutch 被许多组织和个人用户所使用。不过，经营搜索引擎需要在硬件、集

成、定制和索引的维护上大量投资，因此多数情况下，Nutch 用来构建商业垂直或具体领域的搜索引擎。

Nutch 在积极地发展，并且该项目密切注意着 Hadoop 的新版本。因此，它将继续成为现实生活中一个使用 Hadoop 为核心并有优异成果的实用案例。

(作者Andrzej Białecki)

14.4 Hadoop 用于 Rackspace 的日志处理

Rackspace Hosting 一直为企业提供管理系统，并延续了这种精神，Mailtrust 在 2007 年秋季成为 Rackspace 的邮件托管部分。 Rackspace 目前在数百台服务器上为超过 100 万用户和数千家公司托管电子邮件。

14.4.1 需求/存在的问题

在系统中转移由 Rackspace 客户生成的邮件会产生一个很大的“纸”记录，它以大约每天 150 GB 的各种格式日志的形式存在。这非常有益于聚合数据，用于发展的目的以及了解客户如何使用我们的应用，这些记录也是用于解决系统问题。

如果电子邮件无法投递或客户无法登录，我们的客户支持团队能够找到有关该问题的足够信息来开始调试过程，这是至关重要的。为了使人们快速发现信息，我们离不开机器上的或原始格式的日志。相对的，我们使用 Hadoop 做了大量处理，最终得到客户支持可以查询的 Lucene 索引。

日志

数量最多的两种日志格式是由 Postfix 邮件传输代理和 Microsoft Exchange Server 产生的。所有在我们系统传输的邮件都会在某一时刻到达 Postfix 并且大部分信息在多个 Postfix 服务器中传输。Exchange 环境必然是独立的，而一个 Postfix 机器类扮演一个附加保护层的角色，并使用 SMTP 在每个环境托管的邮箱之间传输信息。

这些消息在多台机器上传输，但每个服务器只知道足够的关于邮件目的地的信息用以转移到下一个负责的服务器。因此，为了建立一个信息的完整历史记录，我们的日志处理系统需要从系统全局出发。这就是 Hadoop 对我们帮助极大的地方：随着系统的增长，日志量也同样增长。为了使我们日志处理逻辑保持不变，我们必须确保它可扩展，而 MapReduce 便是非常适合这种增长的框架。

14.4.2 简史

我们的记录处理系统的早期版本是基于 MySQL 的，但当我们获得越来越多的日志机器后，我们到达了单台 MySQL 服务器可以处理的范围极限。数据库模式已经相当非规范化，这可以使得 shard 没有那么困难，但此时 MySQL 的分区支持仍非常薄弱。与其以 MySQL 为中心实现我们自己的 shard 和处理解决方案，我们选择使用 Hadoop。

14.4.3 选择 Hadoop

只要在关系型数据库中 shard 数据，就会损失 SQL 中执行分析数据集的很多好处。Hadoop 使用我们为小数据集使用的相同算法，使我们能够更易并发地处理所有数据。

14.4.4 收集和存储

日志收集

这些生成我们处理的日志的服务器分布在多个数据中心，但目前我们有且只有一个 Hadoop 集群，它位于这些数据中心的其中一个(见图 14-8)。为了整合日志并将它们放入集群中，我们使用 Unix 系统日志替换 syslog-ng 和一些简单的脚本来控制在 Hadoop 中文件的创建。

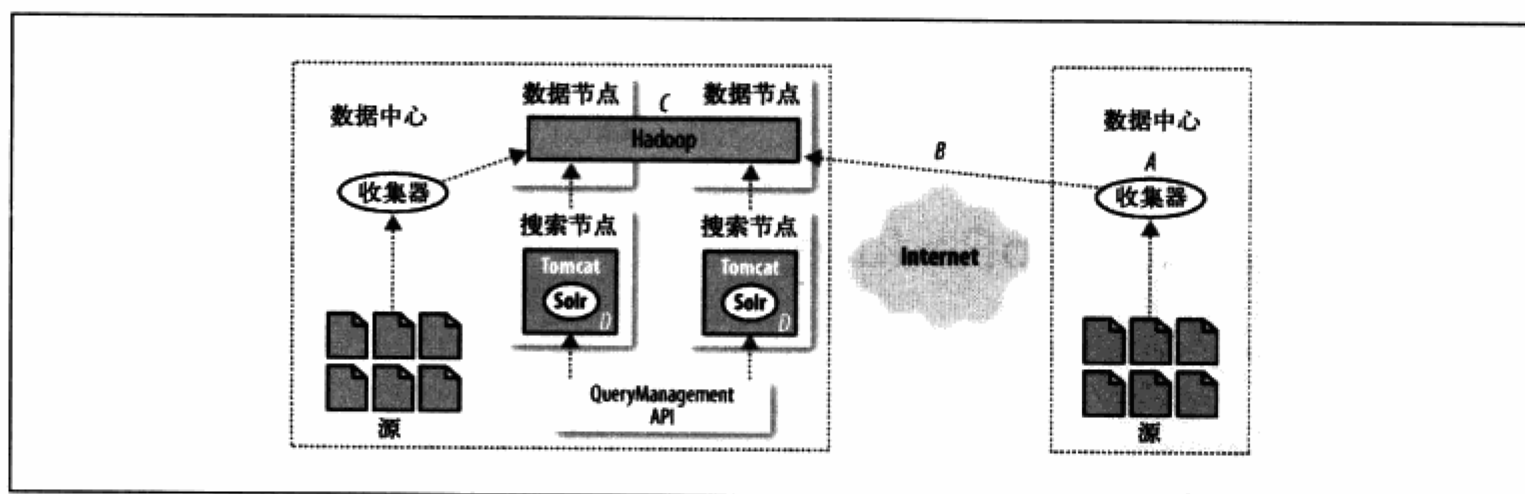


图 14-8：Rackspace 中的 Hadoop 数据流

在一个数据中心中，syslog-ng 被用来将日志从源机器传输到一组负载平衡的收集器中。在收集器(collector)中，每个类型的日志聚合为一个单一的流，用 gzip 适当

压缩(图 14-8 步骤 A)。来自远程收集器中的记录可以通过 SSH 隧道跨数据中心传输到在本地 Hadoop 群集中的收集器(步骤 B)。

一旦压缩日志到达本地收集器，它可以被写入 Hadoop(步骤 C)。目前，我们使用一个简单的 Python 脚本来缓冲输入数据到磁盘，并定期使用 Hadoop 命令行接口将数据放入 Hadoop 集群中。当缓冲达到了 Hadoop 块大小的倍数或已过去足够长的时间，该脚本将日志缓冲复制到 Hadoop 中的输入文件夹。

这种安全地整合来自不同数据中心的日志的方法是在 SOCKS 支持被添加到 Hadoop 前开发的，SOCKS 支持是通过 `hadoop.rpc.socket.factory.class.default` 参数和 `SocksSocketFactory` 类添加的。通过直接从远程收集器中使用 SOCKS 支持和 HDFS 的 API，我们可以消除一个磁盘写以及系统的复杂性。我们计划在未来的开发中用这些特性来替换。

一旦原始日志已放置到 Hadoop 上，它们便准备好被 MapReduce 作业处理。

日志存储

我们 Hadoop 集群目前包含 15 个数据节点，每个数据节点内有商业 CPU 和 3 个 500 GB 的磁盘。我们给需要保存 6 个月存档时期的文件设置默认副本因子为 3，给其他文件设置默认副本因子为 2。

Hadoop 名称节点使用与数据节点完全相同的硬件。为了提供相当的高可用性，我们用两个次级名称节点以及一个虚拟 IP，该 IP 可以很容易地通过 HDFS 快照指出三台机器中的任意一台。这意味着在故障情况下，根据在次级名称节点中快照的年龄，我们有可能失去了 30 分钟的数据。这对于我们的日志处理应用是可以接受的，但其他 Hadoop 的应用程序可能通过使用 namenode 映像的共享存储来要求无损故障。

14.4.5 日志的 MapReduce

处理

悲哀的是，在分布式系统中唯一标识符实际上很少是独一无二的。所有的电子邮件有一个(假定)唯一标识符称为 message-id，它们是由其最初的主机产生的，但一个坏客户端可以很容易地重复发送。此外，由于 Postfix 的设计者无法信任 message-id 能唯一标识消息，他们提出一个独立的编号叫作 queue-id，它只能保证本地机器上消息生命周期的唯一性。

虽然 message-id 往往是一个消息的明确标识符，但在 Postfix 日志上，它需要使用 queue-id 来找到 message-id。看例 14-1 的第二行(已格式化过以更好地适合页面)，你将看到十六进制字符串 1DBD21B48AE，这便是日志行指向的消息的 queue-id。因为当消息被收集时(间隔几小时)，关于它的信息(包括其 message-id)是作为单独行输出的，因此我们的解析代码有必要保持有关消息的状态。

例 14-1：Postfix 日志行

```
Nov 12 17:36:54 gate8.gate.sat.mlsrvr.com postfix/smtpd[2552]:
connect from hostname
Nov 12 17:36:54 relay2.relay.sat.mlsrvr.com postfix/qmgr[9489]:
1DBD21B48AE:
from=<mapreduce@rackspace.com>, size=5950, nrcpt=1 (queue active)
Nov 12 17:36:54 relay2.relay.sat.mlsrvr.com postfix/smtpd[28085]:
disconnect from
hostname
Nov 12 17:36:54 gate5.gate.sat.mlsrvr.com postfix/smtpd[22593]: too
many errors
after DATA from hostname
Nov 12 17:36:54 gate5.gate.sat.mlsrvr.com postfix/smtpd[22593]:
disconnect from
hostname
Nov 12 17:36:54 gate10.gate.sat.mlsrvr.com postfix/smtpd[10311]:
connect from
hostname
Nov 12 17:36:54 relay2.relay.sat.mlsrvr.com postfix/smtp[28107]:
D42001B48B5:
to=<mapreduce@rackspace.com>, relay=hostname[ip], delay=0.32,
delays=0.28/0/0/0.04,
dsn=2.0.0, status=sent (250 2.0.0 Ok: queued as 1DBD21B48AE)
Nov 12 17:36:54 gate20.gate.sat.mlsrvr.com postfix/smtpd[27168]:
disconnect from
hostname
Nov 12 17:36:54 gate5.gate.sat.mlsrvr.com postfix/qmgr[1209]:
645965A0224: removed
Nov 12 17:36:54 gate2.gate.sat.mlsrvr.com postfix/smtp[15928]:
732196384ED: to=<m
apreduce@rackspace.com>, relay=hostname[ip], conn_use=2, delay=0.69,
delays=0.04/
0.44/0.04/0.17, dsn=2.0.0, status=sent (250 2.0.0 Ok: queued as
02E1544C005)
Nov 12 17:36:54 gate2.gate.sat.mlsrvr.com postfix/qmgr[13764]:
732196384ED: removed
```

```
Nov 12 17:36:54 gate1.gate.sat.mlsrvr.com postfix/smtpd[26394]:
NOQUEUE: reject: RCP
T from hostname 554 5.7.1 <mapreduce@rackspace.com>: Client host
rejected: The
sender's mail server is blocked; from=<mapreduce@rackspace.com>
to=<mapred
uce@rackspace.com> proto=ESMTP helo=<mapreduce@rackspace.com>
```

从 MapReduce 角度来看，每个日志行是一个单一的键/值对。在阶段 1，我们需要将所有行与单个 queue-id 键映射在一起，然后简化它们以确定日志消息值表示 queue-id 是否已完成。

同样，一旦我们有一个消息的完成的 queue-id，我们需要在阶段 2 对 message-id 进行分组。我们对完成的 queue-id 和它的 message-id 进行 map，以 message-id 作为键，以它的日志行列表作为值。在 reduce 阶段，我们确认所有 message-id 的 queue-id 表示信息是否已离开我们的系统。

邮件日志 MapReduce 作业的这两个阶段及其 `InputFormat` 和 `OutputFormat` 一起形成了一个阶段事件驱动架构(SEDA)类型。在 SEDA，应用程序被分成多个“阶段”，这是由队列分离的。在 Hadoop 中，队列可以是一个作为 MapReduce 作业消耗来源的 HDFS 中的输入文件夹或是一个 MapReduce 生成的在 Map 和 Reduce 阶段之间的隐含队列 。

在图 14-9 中，阶段之间的箭头表示队列，虚线箭头表示隐含的 MapReduce 队列。每个阶段都可以将一个键/值对通过这些队列发送(SEDA 称之为事件或消息)到另一阶段。

阶段 1：map 在我们邮件日志处理作业的第一阶段，该 map 阶段的输入可以是行数键和日志消息值也可以是 queue-id 键和 log-message 值数组。第一种输入类型是当我们处理来自输入文件队列中的原始日志文件时产生的，第二种类型是一种中间格式，它表示一个 queue-id 的状态，该 queue-id 我们已经准备处理，但因为它没完成而重新放入队列中。

注意： 为了实现这一双重输入，我们实现了一个 Hadoop 的 InputFormat，它代理底层 SequenceFileRecordReader 或者 LineRecordReader 的工作，这取决于输入 FileSplit 的文件扩展名。这两个输入格式在 HDFS 来自不同的输入文件夹(或队列)。

阶段 2：reduce 在这个阶段，reduce 阶段确定 queue-id 是否有足够的已完成的行。如果 queue-id 已完成，我们以 message-id 为键，一个 `HopWritable` 对象为值将其输出。否则，将 queue-id 设为键，并且日志行数组需要与下一原始数据集合

映射起来。这将持续到我们完成 queue-id，或直至超时。

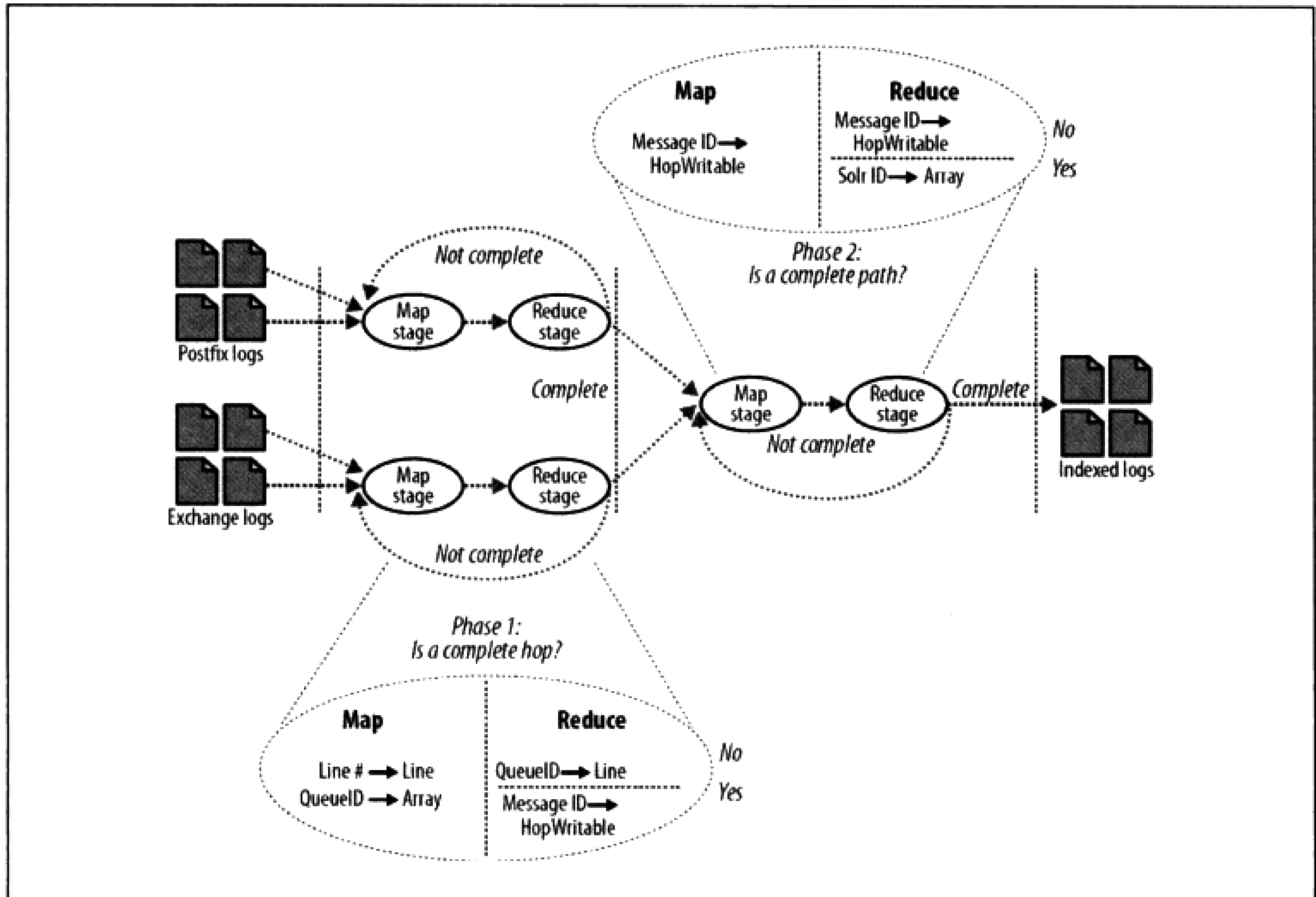


图 14-9：mapReduce 链

该 HopWritable 对象是一个 POJO，实现 Hadoop 的 Writable 接口。它从单台服务器的角度完整地描述了一个消息，包括发送地址和 IP，将邮件传递到其他服务器的意图以及典型的邮件头信息。

该分区输出与一个 OutputFormat 实现一起完成，该 `OutputFormat` 实现与我们的双重 `InputFormat` 有些对称。我们的 `MultiSequenceFileOutputFormat` 是在 Hadoop API 的 r0.17.0 版本中添加 `MultipleSequenceFileOutputFormat` 之前就已实现的，但满足了相同的目标：我们需要 reduce 输出根据其键的特点写到不同的文件中。

阶段 3：map 在邮件日志处理作业的下一阶段，输入是 message-id 键及来自上一阶段的 `HopWritable` 值。这个阶段不包含任何逻辑：相反，它只是使用标准 `SequenceFileInputFormat` 和 `IdentityMapper` 将来自第一阶段的输入组合起来。

阶段 4：reduce 在最后的 reduce 阶段，我们希望了解是否所有我们为 message-id 收集的 `HopWritable` 都代表一个在我们系统中的完整的消息路径。一个消息路径

本质上是一个有向图(通常是无环，如果服务器配置错误，它可能包含循环)。在此图中，一个顶点是一个服务器，可标记为多种 queue-id。边就是服务器间传送邮件的意图。对于这个处理，我们使用 JgraphT 图库。

对于输出我们再次使用 `MultiSequenceFileOutputFormat`。如果 reducer 认为一个 message-id 的所有 queue-id 创建了一个完整的消息路径，那么消息便被序列化并排队以提供给 `SolrOutputFormat` 处理。否则，该消息的 `HopWritable` 便被排队以提供给阶段 2 处理：map 阶段再次处理下一批 queue-id。

该 `SolrOutputFormat` 包含一个嵌入的 Apache Solr 实例——最初由 Solr Wiki 推荐——产生一个在本地磁盘上的索引。之后关闭 `OutputFormat` 将包括把磁盘索引压缩到输出文件的最终目的地。相较于使用 Solr 的 HTTP 接口或直接使用 Lucene 这种方法的优势如下：

- 我们可以强制执行 Solr 模式。
- map 和 reduce 仍然保持幂等。
- 索引负载从搜索节点中删除了。

目前，我们使用默认 `HashPartitioner` 类来决定哪些任务将得到特定键，这意味着键是半随机分配的。在未来系统的重复使用中，我们希望实现一个新的 partitioner 以发送地址进行划分(而不是我们最常见的搜索字词)。一旦索引被 sender 划分，我们可以使用该地址的哈希表以确定索引合并或查询的位置，而我们的搜索 API 将只需要与有关的节点通信。

近期搜索合并

在一系列 MapReduce 阶段完成后，一系列不同的机器被通知到这些新索引，并能够获得它们来合并。这些搜索节点运行 Apache Tomcat 和 Solr 实例来主管完成的索引以及获得及合并索引到本地磁盘(如图 14-8 步骤 D)。

sharding 查询/管理 API 是一个 PHP 代码层，用以在所有的搜索节点上处理 sharding 输出索引。我们使用了一个简单的连续散列的实现来决定哪些搜索节点负责每个索引文件。目前，索引先按其创建时间然后按它们散列过的文件名被 shard，但我们计划在未来某阶段用发送地址散列代替文件名散列(见阶段 2：reduce)。

因为 HDFS 中已经处理了 Lucene 索引的副本，因此没有必要在 Solr 中保持多个副本可用。相反，在故障情况下，搜索节点是完全被删除的，并且其他节点负责合并索引。

搜索结果 有了这个系统，我们已经从日志生成中取得了 15 分钟的周转时间为我们的客户支持团队提供搜索结果。

我们的搜索 API 支持完整的 Lucene 查询语法，所以我们经常看到这样复杂的查询：

```
sender:"mapreduce@rackspace.com" -recipient:"hadoop@rackspace.com"
  recipient:"@rackspace.com" short-status:deferred
timestamp:[1228140900 TO 2145916799]
```

查询返回的结果是一个完整的序列化的消息路径，它表明单个服务器和收件人是否收到邮件。我们目前将路径显示为二维图(图 14-10)，用户可以与之交互以扩展兴趣点，但这些数据的可视化方面有很多改进的空间。

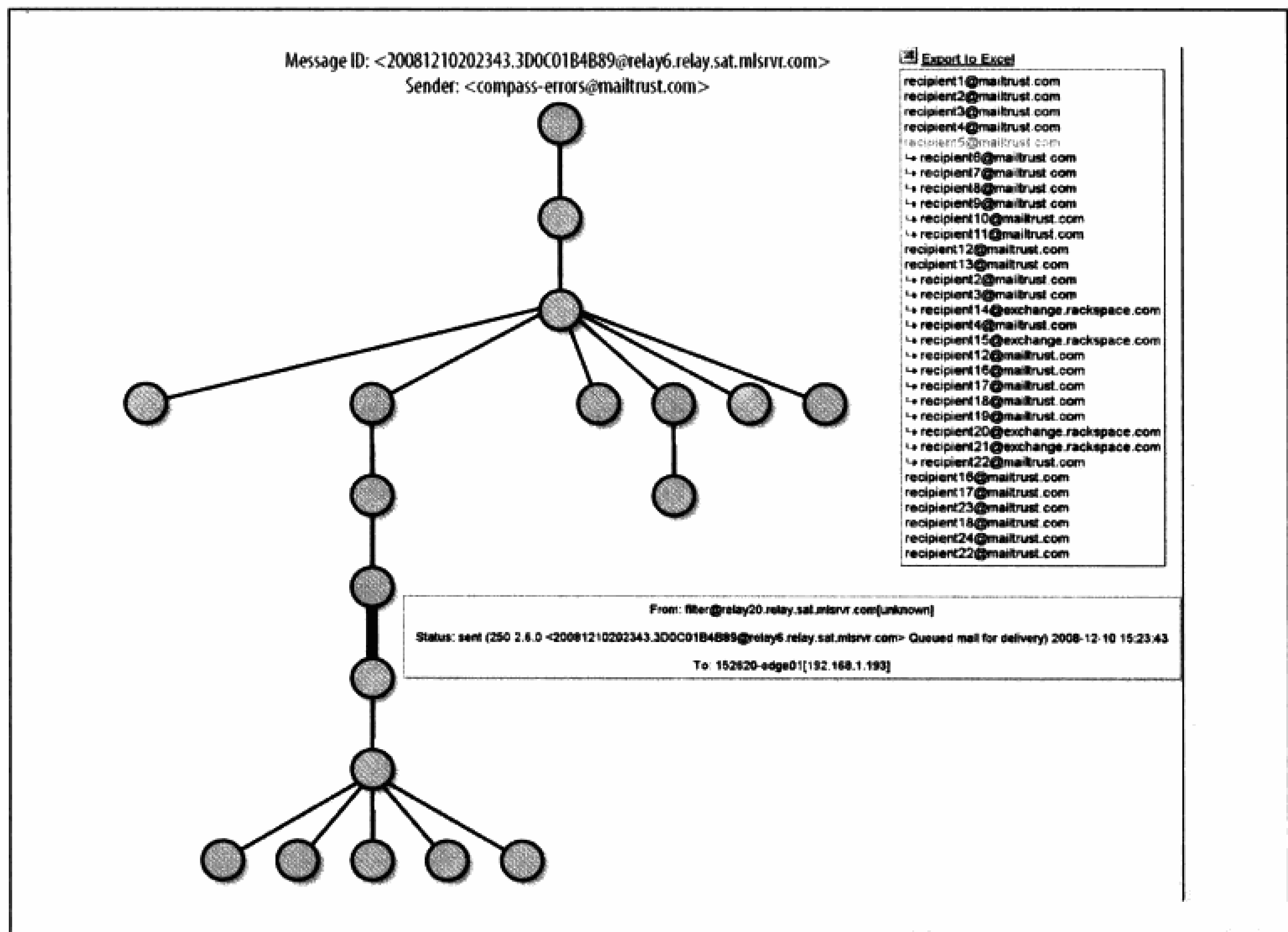


图 14-10：数据树

为分析归档

除了给客户支持提供短期搜索，我们对执行日志数据分析也很感兴趣。

每天晚上，我们以当天的索引作为输入运行一系列 MapReduce 作业。我们实现了

SolrInputFormat，它可以获得并解压缩索引，并放出每个文件作为一个键/值对。有了这个 InputFormat，我们可以遍历一天所有消息的路径，并回答几乎任何关于我们的邮件系统的问题，具体如下。

- 每个域数据(病毒、垃圾邮件、连接和收件人)。
- 最有效的垃圾邮件规则。
- 特定用户产生的负载。
- 反弹消息的原因。
- 连接的地域来源。
- 特定机器之间的平均时延。

由于我们在 Hadoop 中有几个月的压缩索引存档，即使我们每晚日志摘要已被删除，我们还可以解决历史问题。例如，我们最近想要确定每月发送 IP 地址中最多的地址，我们只需用一个简单的一次性的 MapReduce 作业完成。

(作者Stu Hood)

14.5 Cascading 项目

Cascading 是一个开源的 Java 库和应用程序接口(API)，为 MapReduce 提供一个抽象层。它允许开发人员创建复杂的，运行于 Hadoop 集群上的任务关键数据处理应用。

Cascading 项目始于 2007 年夏天。首次公开发行版本 0.1 在 2008 年 1 月推出。 1.0 版本发布于 2009 年 1 月。二进制文件，源代码和附加模块可以从项目网站 *http://www.cascading.org/*下载。

“map”和“reduce”操作提供强大的原语。但是，它们往往在一个错误的粒度水平上，创建复杂的、高组合性的、可以被不同开发者之间共享的代码。此外，当面临现实世界的问题时，许多开发人员发现很难从 MapReduce 角度“思考”。

为了解决第一个问题，Cascading 用简单的字段名称和数据元组模型来替代 MapReduce 中使用的“键”和“值”，其中一个元组是一个简单的值列表。至于第二个问题，Cascading 直接去掉了 map 和 reduce 作业，转而引入更高层次的抽象：Function，Filter，Aggregator 和 Buffer。

该项目首次公开发布的同时，其他替代也相继出现，但 Cascading 旨在补充它们，考虑这些替代框架需要的前后条件或其他期望。

例如，在其他几个 MapReduce 工具中，必须事先预格式化、过滤或导入数据到

Hadoop 的文件系统(HDFS)中才能运行应用程序。该准备数据的步骤必须在编程抽象以外执行。相反，Cascading 提供了方法能将你的数据作为编程抽象的组成部分来准备和管理。

本案例研究从 Cascading 的主要概念介绍开始，然后以概览 ShareThis 如何在其基础设施中利用 Cascading 来结束。请参阅在项目网站上的 Cascading 用户指南获得更深入的 Cascading 处理模型介绍。

14.5.1 字段、元组和管道

MapReduce 模型使用键和值将输入数据连接到 map 函数，map 函数连接到 reduce 函数，reduce 函数连接到输出数据。

因此理论上，键和值不仅绑定了 map 到 reduce，还有 reduce 到下一个 map 然后再到下一个 reduce 等(图 14-11)。也就是说，键/值对来源于通过 map 和 reduce 作业链的输入文件和流。实现足够的这些成链的 MapReduce 应用后，就可以看到一组反复使用的明确的键/值操作来修改键/值数据流。

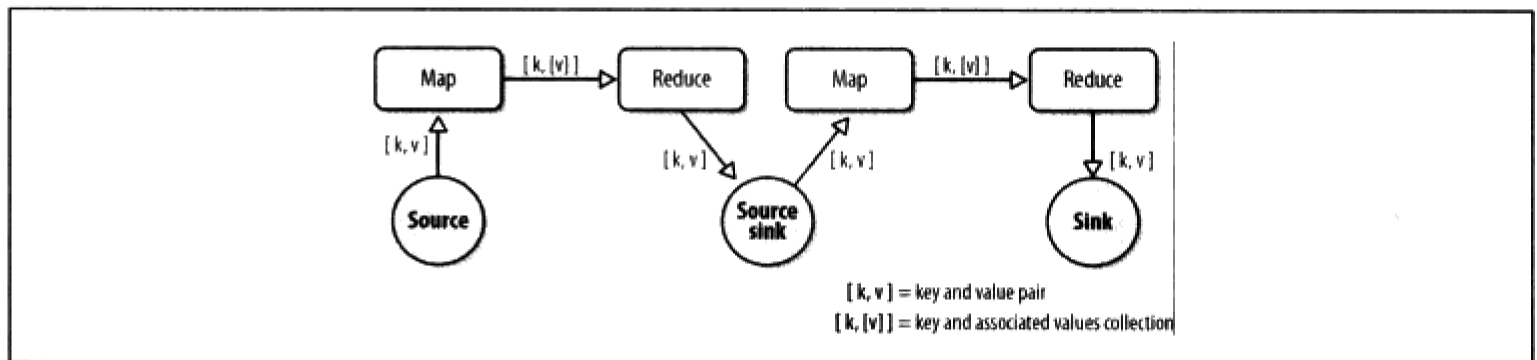


图 14-11：MapReduce 的计数和排序

Cascading 对其进行了简化，抽象了键和值并用元组代替它们，元组具有相应的字段名称，类似于在关系型数据库中的表和列名的概念。并在处理过程中，这些字段和元组的流通过由管道连接的用户自定义的操作中被操作(图 14-12)。

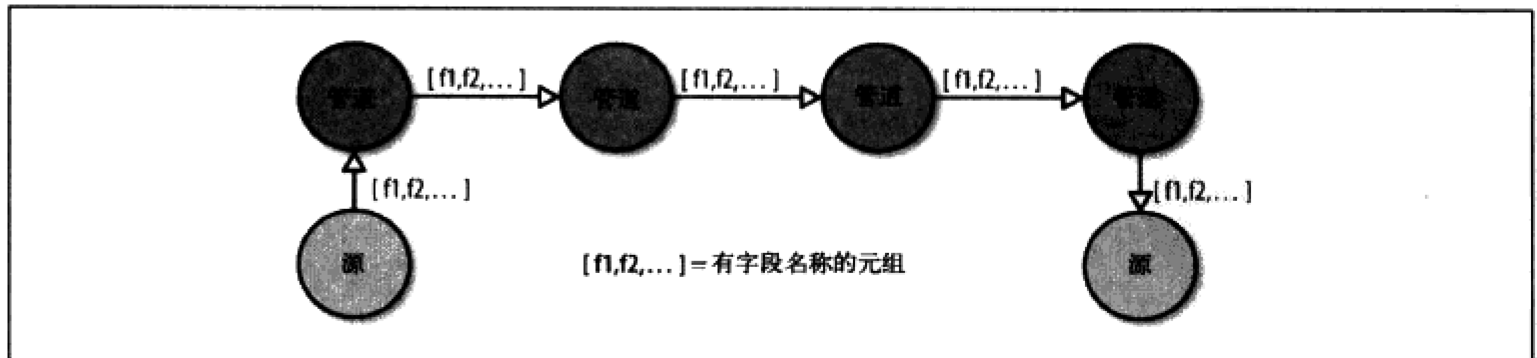


图 14-12：字段和元组联接的管道

因此，MapReduce 键和值简化如下：

字段

字段是它们的字符串名(如“first_name”)和数值型的位置(如 2 和-1 分别代表第三位和最后位)或者是两者结合的一个集合，很像列名。因此，字段用于在元组中声明值的名称以及从元组中按名称选择值。后者更类似于 SQL 的 select 调用。

元组

元组只是 `java.lang.Comparable` 对象的数组。元组非常类似于数据库行或记录。

这样，map 和 reduce 操作就被抽象成一个或多个管道实例(图 14-13)。

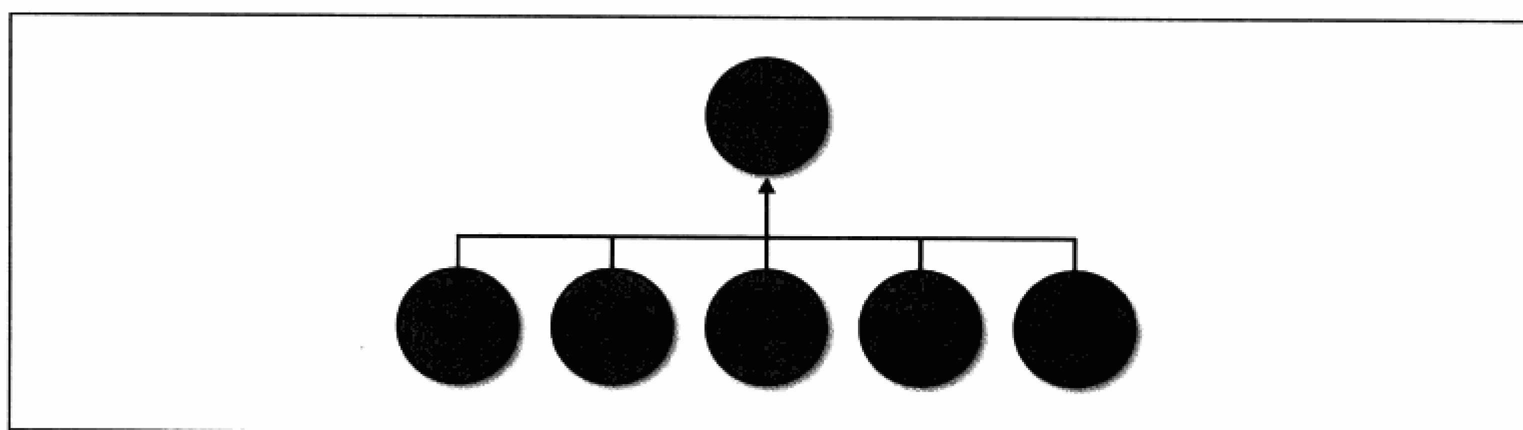


图 14-13：管道类型

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(将在不久之后描述)。

GroupBy

`GroupBy` 管道按分组字段对元组进行分组，就像执行 SQL 的 `group` 语句。如果多个输入元组流都有相同的字段名称，它也可以将这些流合并成为一个的单一流。

CoGroup

`CoGroup` 管道不仅按共同字段名连接多个元组流，还按共同分组字段对这些元组进行分组。可以在两个或两个以上元组流上使用所有标准的联接类型(内联接，外联接等)和自定义联接。

Every

`Every` 管道一次处理一个元组组，该组是由 `GroupBy` 或 `CoGroup` 管道分组的。它可以对组应用 Aggregator 或 Buffer 操作。

SubAssembly

`SubAssembly`管道允许在单个管道内嵌套集合，从而可以嵌套在更复杂的集合里。开发者可以将所有这些管道链接成“管道集”，其中每个集可以有很多输入元组流(source[源])和很多输出元组流(sink[汇])(见图 14-14)。

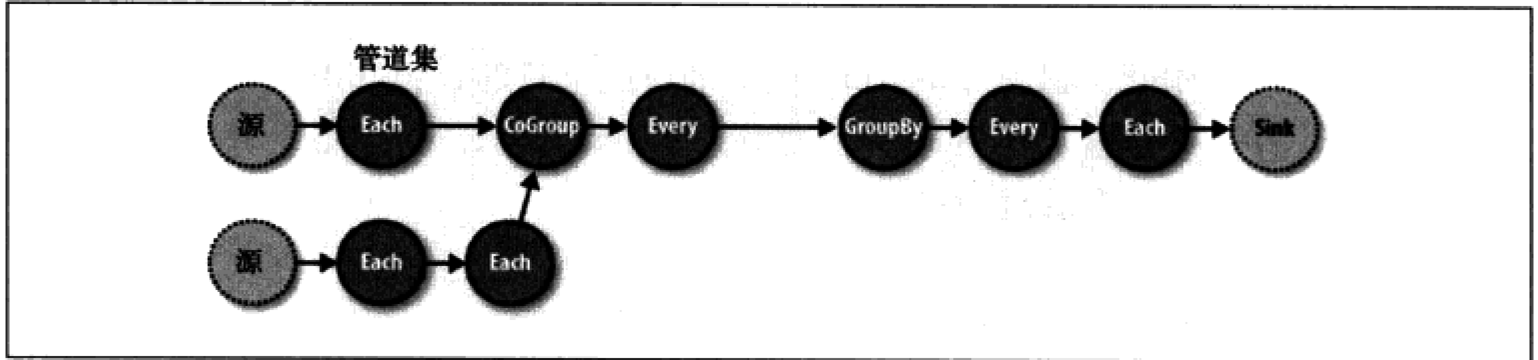


图 14-14：一个简单的管道集

从表面上看这似乎比传统的 MapReduce 模式更复杂，而且也承认这里有比 map、reduce、键和值更多的概念。但实际上，传统 MapReduce 中还有更多的概念必须协同工作以提供不同的行为。

例如，如果一个开发者需要给 reducer 值“二次排序”，她将需要实现 map，reduce、一个“混合”键(有两个键嵌套在父键里)、值、分区器、一个“输出值分组”比较器、和“输出键”比较器，所有这些都将以不同方式结合在一起，并且很可能在以后的应用中不能重用。

而在 Cascading，这将只是一行代码：`new GroupBy(<previous>, <grouping fields>, <secondary sorting fields>)`，而在这前面就是前一个管道。

14.5.2 操作

如前所述，Cascading 区别于 MapReduce，它引入了适用于单个元组或元组成组的替代操作(图 14-15)。

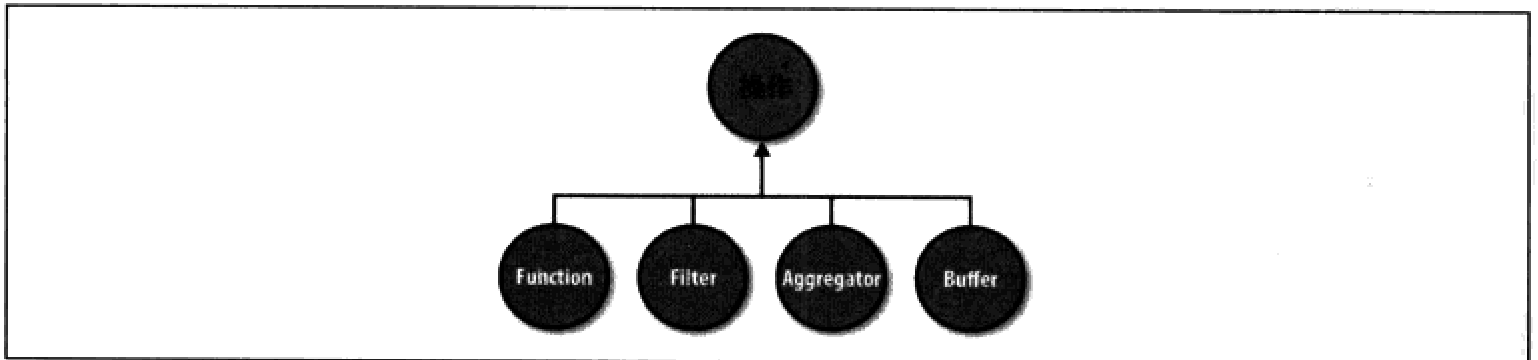


图 14-15：操作类型

Function(函数)

Function 操作单个输入元组，每个输入可能会返回零个或多个输出元组。Function 应用于 `Each` 管道。

Filter(过滤器)

Filter 是一种特殊类型的函数，它返回一个布尔值表示当前输入元组是否应该从元组流中被删除。虽然 Function 可以达到这个目的，但 Filter 是为该情况优化过的，并且许多 Filter 可以按“逻辑”Filter 如与，或，异或，非等来分组，以迅速创建更为复杂的过滤操作。

Aggregator(聚合器)

`Aggregator` 为元组组执行一些操作，这里元组组按一组共有字段值被分组。例如，所有具有相同“last-name”值的元组。常用共同 Aggregator 操作有 `Sum`，`Count`，`Average`，`Max` 和 `Min`。

Buffer(缓冲)

Buffer 类似于 `Aggregator`，唯一的区别是它被优化为一个在特定分组中所有元组之上的“滑动窗口”。当开发者需要有效地在一组有序的元组中插入缺少的值(如丢失的日期或时期)或创建运行平均值时，这是非常有用的。需要处理元组时，通常会选择 `Aggregator` 操作，因为许多 `Aggregator` 可以非常高效地链接在一起，但有时可能 Buffer 是该作业的最佳工具。

管道集创建后，操作被绑定到管道(图 14-16)。

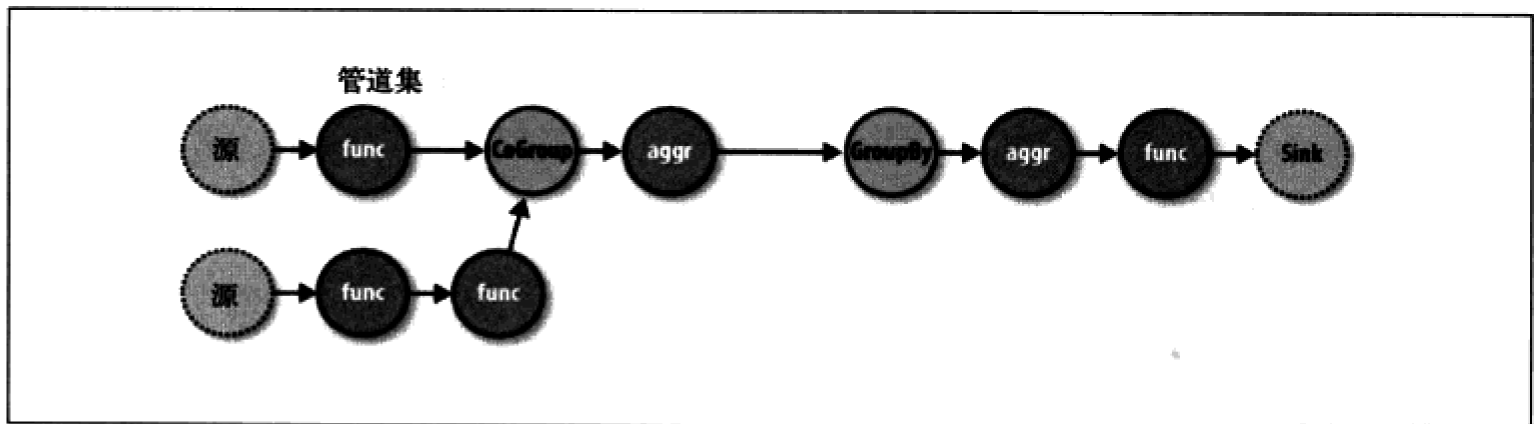


图 14-16：操作的集合

Each 和 Every 管道提供了一个简单机制实现在传递给其子操作之前从输入元组中选择部分或全部值。同时还有一个合并操作结果与原始输入元组来创建输出元组的简单机制。由于不涉及细节，这使得每个操作只关心元组值和字段参数而不是当前输入元组的所有字段集合。随后，操作就像 Java 方法一样可以跨应用重用。

例如，在 Java 中，一个声明为 `concatenate(String first, String second)` 的

方法比 `concatenate(Person person)`更抽象。在第二种情况下，`concatenate()` 函数必须“知道” `Person` 对象，在第一种情况中，它并不知道数据是从哪里来的。Cascading 操作也不例外。

14.5.3 Tap、Sheme 和 Flow

在之前的很多图中有涉及“source”和“sink”的概念。在 Cascading，所有数据被读出或写入到 `Tap` 实例，但通过 `Scheme` 对象被转换成元组实例或从元组实例中转换而来。

Tap

  Tap 负责如何及在哪里获取数据部分。例如，数据是在 HDFS 上还是本地文件系统上？在 Amazon S3 还是通过 HTTP？

Scheme

  `Scheme` 负责读取原始数据并将其转换为一个元组并且/或者将元组写入原始数据，其中“原始”数据可以是行文本，Hadoop 二进制序列文件，或者某种专有的格式。

请注意，Tap 不属于管道集，因此它们不是管道类型。

但当它们作为集群可执行文件时，它们便被连接到管道集。当管道集连接必要数量的 source 和 sink Tap 实例时，我们得到了一个 `Flow`。当管道集连接必要数量的 source 和 sink Tap 实例，并且 Tap 发出或得到管道希望的字段名称时，`Flow` 就被创建了。也就是说，如果一个 `Tap` 发出一个元组以及字段名称 line(通过读入 HDFS 中的一个文件中的数据)，那么管道集头必须也期望一个 line 值。否则，连接管道集和 Tap 的过程将立即出错而失败。

因此，管道集实际上是数据流程定义，而不是自身“可执行”。它们必须连接到 source 和 sink Tag 实例才可以运行于群集上。这种 Tag 和管道集的分离正是 Cascading 如此强大的幕后功臣。

如果认为管道集类似于一个 Java 类，那么 `Flow` 就像一个 Java 对象的实例(图 14-17)。也就是说在同一应用程序中，同一管道集可以被多次实例化为新 Flow 而无需担心它们之间的任何干扰。这使得管道集可以像被标准 Java 库一样创建和共享。

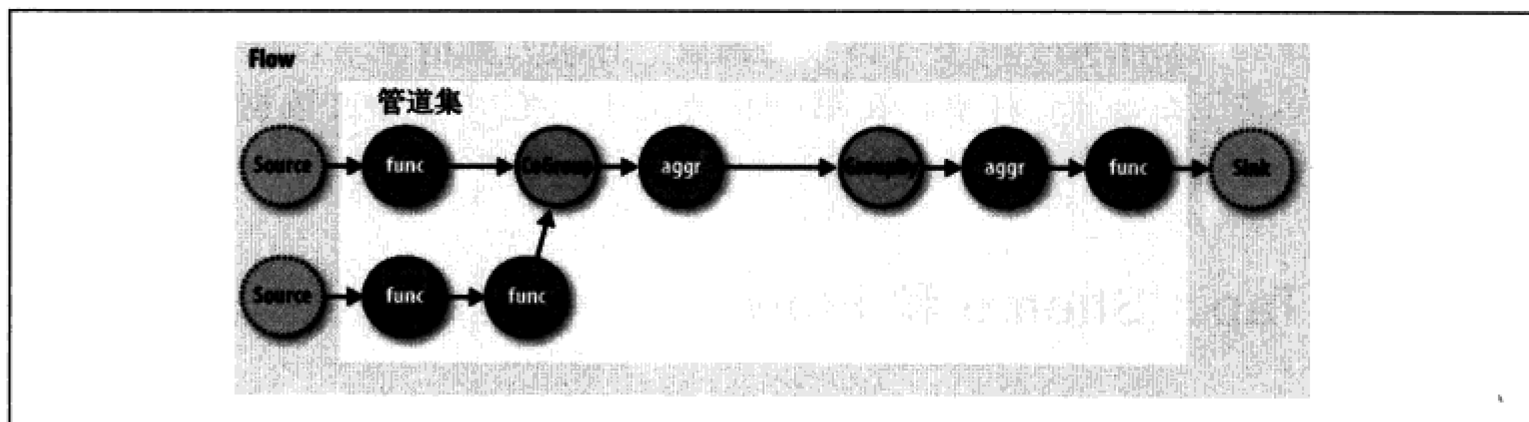


图 14-17：Flow

14.5.4 Cascading 实践

知道什么是 Cascading 之后，也可了解它的工作机制和用 Cascading 编写的应用程序是什么样。见例 14-2。

例 14-2：字数统计和排序

```
Scheme sourceScheme =
  new TextLine(new Fields("line"));❶
Tap source =
  new Hfs(sourceScheme, inputPath);❷

Scheme sinkScheme = new TextLine();❸
Tap sink =
  new Hfs(sinkScheme, outputPath, SinkMode.REPLACE);❹

Pipe assembly = new Pipe("wordcount");❺

String regexString = "(?<!\\pL)(?=\\pL)[^ ]*(?<=\\pL)(?!\\pL)";
Function regex = new RegexGenerator(new Fields("word"), regexString);
assembly =
  new Each(assembly, new Fields("line"), regex);❻

assembly =
  new GroupBy(assembly, new Fields("word"));❼

Aggregator count = new Count(new Fields("count"));
assembly = new Every(assembly, count);❽

assembly =
  new GroupBy(assembly, new Fields("count"), new Fields("word"));❾
```

```
FlowConnector flowConnector = new FlowConnector();
Flow flow =
  flowConnector.connect("word-count", source, sink, assembly);⑩

flow.complete();⑪
```

❶ 我们创建一个新的 Scheme，它可以读取简单的文本文件，并为 line 字段的每行发出一个新的元组，就如 Field 实例声明的一样。

❷ 我们创建一个新的 Scheme，它可以写入简单的文本文件，输入是一个含有任意数量的字段/值的元组。如果有一个以上的值，它们将在输出文件中以制表符分割。

❸和❹ 我们创建 source 和 sink Tag 实例分别指向输入文件和输出目录。该 sink Tap 将覆盖任何已存在的文件。

❺ 我们构建我们的管道集头，并将其命名为“wordcount”。该名称用来绑定 source 和 sink Tap 到管道集。多头或多尾需要唯一的名称。

❻ 我们构建一个 Each 管道和一个函数，该函数将遇到的每个单词解析“line”字段为一个新的元组。

❼ 我们构建一个 GroupBy 管道，它将为每个在字段 word 中的不同值创建一个新的元组组合。

❽ 我们构建一个 Every 管道和一个 Aggregator，Aggregator 将统计在每个不同单词组的元组数量。该结果将被保存到 count 字段。

❾ 我们构建一个 GroupBy 管道，它将为每个在字段 count 中不同的值创建一个新元组组合，并对字段 word 中的每个值进行次要排序。该结果将是一个 count 和 word 值列表并且 count 升序排序。

❿和⓫ 我们将管道集和它的 source 和 sink 连接到一个 Flow，然后在集群上执行 Flow。

在这个例子中，我们给在输入文件中遇到的单词统计数量并为计数自然排序(升序)。如果有一些具有相同 count 值的单词，将这些单词自然排序(按字母顺序排列)。

这个例子中一个明显的问题是有些单词可能有大写字母，例如“the”和句子开头的“The”。因此，我们可能会想增加一个新的操作以使所有的单词为小写，但我们认识到，今后所有需要解析来自文件的单词的应用程序应该会有同样的行为，所以我们决定创建一个可重用的 SubAssembly 管道，就像我们在传统应用中会创建子程序一样(见例 14-3)。

例 14-3：创建一个 SubAssembly

```
public class ParseWordsAssembly extends SubAssembly❶
  {
  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); ❷
    previous = new Each(previous, new Fields("word"), expression);

    setTails(previous);❸
  }
}
```

❶ 我们创建 `SubAssembly` 类，它本身就是一种管道。

❷ 我们创建一个 Java 表达式函数，它将在名为"word"字段中的 `String` 值上调用 `toLowerCase()`。我们还必须传入表达式期望的"word"的 Java 类型，在这里是 `String`。(*http://www.janino.net/*使用到了这些。)

❸ 我们必须告诉 SubAssembly 超类我们的管道子集在哪里结束。

首先，我们创建一个 `SubAssembly` 管道作为我们"解析单词"管道子集。由于这是一个 Java 类，它可以重用于任何其他应用程序，只要有传入 word 字段(例 14-4)。请注意，有使这个函数更通用的办法，详情请参见 Cascading 用户指南。

例 14-4：用 SubAssembly 扩展的字数统计和排序

```
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);❶
```

```
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();
```

❶ 我们用 `ParseWordsAssembly` 管道替换上个例子中的 Each 管道。

最后我们仅用新的 `SubAssembly` 管道替换上个例子中的 `Every` 和单词解析函数。该嵌套可以根据需要继续深入。

14.5.5 灵活性

退一步看看这种新的模式给了我们什么。或者更好的说法是它解决了什么问题。

我们不再考虑 MapReduce 作业，或 Mapper 和 Reduce 接口实现，以及如何绑定或链接 MapReduce 作业到在它们之前的作业。在运行时，Cascading“规划者”实现了最佳方式来划分管道集到 MapReduce 作业并管理它们之间的连接(图 14-18)。

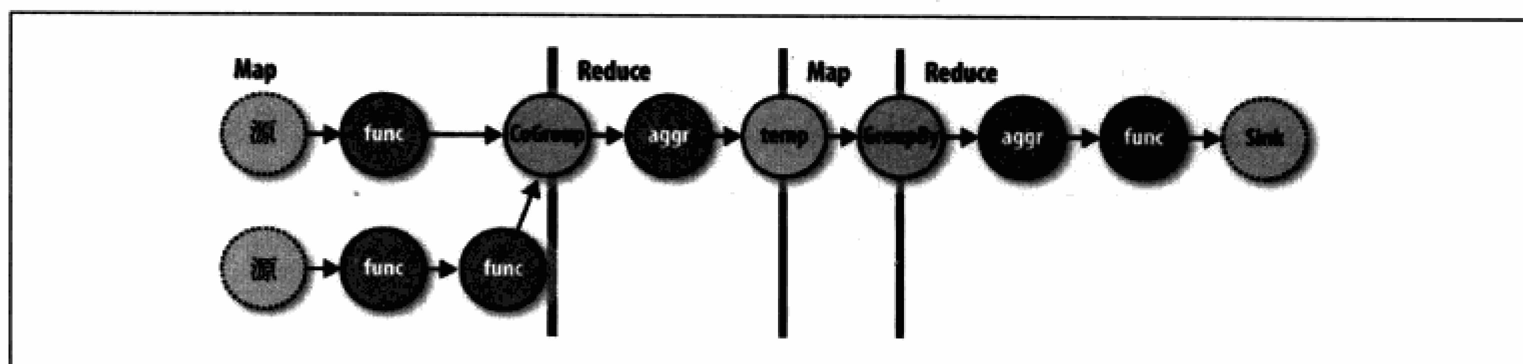


图 14-18：如何将 Flow 转换为链式 MapReduce 作业

正因为如此，开发人员可以构建任意粒度的应用。他们可以从一个仅仅过滤日志文件的小应用程序开始，但随后可以根据需要反复构建更多的特性到应用中。

由于 Cascading 是一个 API 而不是像 SQL 字符串的语法，因此更为灵活。首先，开发人员可以使用他们喜爱的语言像是 Groovy，JRuby，Jython，Scala 或其他来创建领域特定语言(DSL)(见项目网站的例子)。其次，开发人员可以扩展 Cascading 的不同部分，诸如让自定义 Thrift 或 JSON 对象能读写，并让它们能传入传出元组流。

14.5.6 Hadoop 和 Cascading 在 ShareThis 的应用

ShareThis 是一个共享网络，可以更简单地共享任何在线内容。只需要单击网页上的按钮或浏览器插件，ShareThis 便允许用户从任何地方在线无缝接入他们的链接和网络并通过电子邮件、IM、Facebook、Digg 和手机短信等分享内容，不需要离开当前页面。出版商可以部署 ShareThis 按钮来开发服务的全球共享能力以吸引流量，刺激病毒活跃性，并跟踪在线内容分享。ShareThis 还通过减少网页上的混乱和提供在社交网络、加盟团体和社区即时分发内容的方式简化社交媒体服务。

正因为 ShareThis 用户通过网上部件共享网页和信息，事件流会不断进入 ShareThis 的网络。这些事件首先被过滤和处理，然后交给不同的后台系统，包括 AsterData，Hypertable 和 Katta。

这些事件量可能是巨大的，大到传统系统无法处理。由于恶意系统的“注入式攻击”，浏览器缺陷或故障部件，这些数据也可能很“脏”。为此，ShareThis 选择部署 Hadoop 作为预处理和前台到后台系统的业务流程。他们还选择使用 Amazon 网络服务，在弹性计算云(EC2)上托管他们的服务器，在简单存储服务(S3)上提供长期存储，着眼于弹性 MapReduce。

在本文中，我们将集中在“日志处理管道”(图 14-19)。该日志处理管道只是获取 S3 存储段内的数据，进行处理(后文将详述)，并将结果存储到另一个存储段。简单队列服务(SQS)用于协调标记数据处理运行开始和结束的事件。下行其他进程获取数据加载 AsterData，从 Hypertable 获取 URL 列表得到网页抓取源，或获得抓取的网页数据用 Katta 创建 Lucene 索引。请注意，Hadoop 是 ShareThis 架构的核心。它用来协调架构组件之间数据的处理和转移。

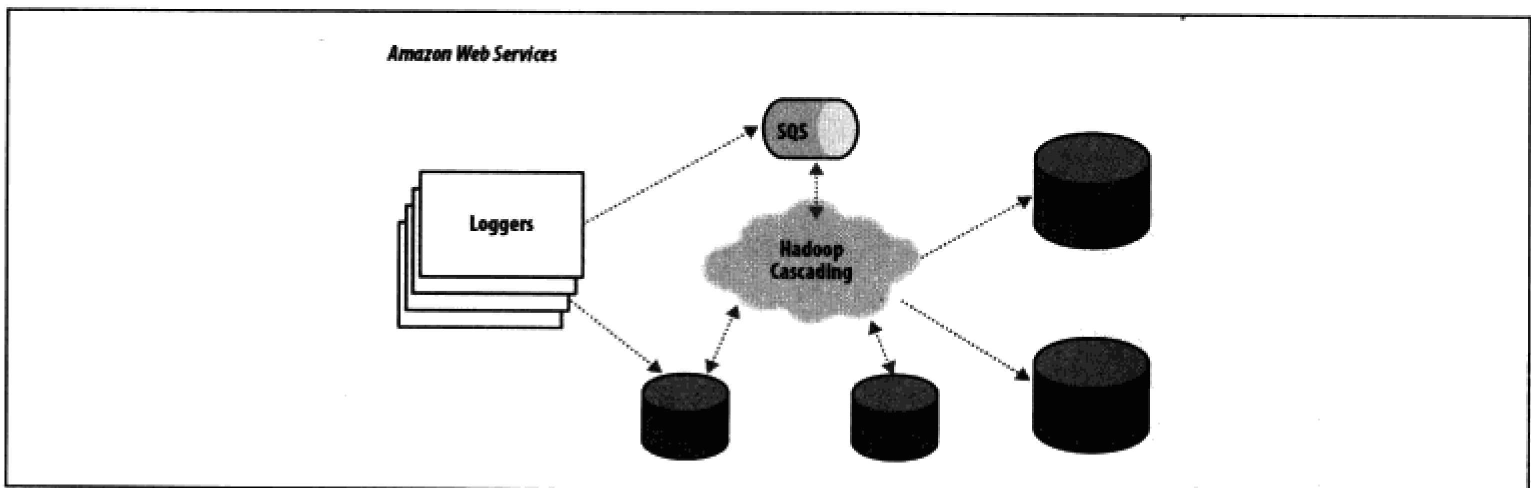


图 14-19：ShareThis 日志处理管道

由 Hadoop 作为前台，所有的事件日志在被加载到 AsterData 集群或被任何其他组

件使用之前可以通过一套规则来被解析、过滤、清洁和组织。AsterData 是一个群集的数据仓库，它可以支持大型数据集，并允许复杂的使用标准的 SQL 语法的专门查询。ShareThis 整理和准备在 Hadoop 集群上传入的数据集，然后加载该数据到 AsterData 集群用于专门分析和报告。尽管 AsterData 也可以做，但是使用 Hadoop 作为处理管道中的第一阶段是很有意义的，它可以抵消对主要数据仓库的负载。

Cascading 被选中作为主要的数据处理 API 来简化开发过程，记录数据是如何在架构组件间被协调的，并提供面向开发人员的这些组件的接口。这表明了它不同于更传统的 Hadoop 用例，基本上只是查询存储的数据。相反，Cascading 和 Hadoop 一起可以提供更好、更简单的完整解决方案、终端到终端的结构，因此对于用户来说，更有价值。

对于开发人员，Cascading 使其可以开始于一个简单的单元测试(通过实现 `cascading.ClusterTestCase` 的子类)，它可以进行简单的文本解析，然后再进入更多处理规则的层次上，同时保持应用程序逻辑上的组织性用于维护。Cascading 通过一些方式实现这种组织。首先，独立操作(Function，Filter 等)可以独立编写和测试。其次，应用被分割成阶段：一个作为解析，一个作为规则，以及最后阶段作为装箱/收集数据，所有都是通过之前描述的 `SubAssembly` 基类。

来自 ShareThis 日志记录器中的数据看上去非常像 Apache 日志以及日期/时间戳、共享 URL、引荐 URL 以及一点元数据。为了在分析下行流中使用该数据，这些 URL 需要被拆箱(解析查询字符串数据和域名等)。因此，顶层 SubAssembly 被创建用以封装解析，并且如果数据对于解析足够复杂，那么子 SubAssembly 就被嵌套在其中用来处理特定的字段。

应用规则也是同样的做法。当每个元组通过 *`SubAssembly`* 规则时，如果任何规则被触发那么它就被标记为“坏”的。除了“坏”的标签，一个表示该记录坏的原因描述被添加到元组内以用于以后的审查。

最后，partitioner`SubAssembly` 被创建来做两件事。第一，将元组流分成两部分，“好”数据流和“坏”数据流。第二，partitioner 将其拆分成间隔的数据，如每隔一小时。为此，只有需要两个操作。首先用从流中已经存在的时间戳值创建间隔，接着使用间隔和好/坏元数据创建一个目录路径，例如“05/good /”，其中“05”是凌晨 5 点，“good”是指元组通过了所有的规则。这个路径将被 Cascading TemplateTap 使用，TemplateTap 是一个特殊的 Tap，它可以基于元组中的值动态地输出元组流到不同的位置。在这种情况下，TemplateTap 使用 path 值来创建最终的输出路径。

开发人员还创建了第四个 SubAssembly，它在单元测试期间应用 Cascading Assertion。这些断言双重检查了规则和解析 SubAssemblies 的作业。

在例 14-5 的单元测试中，我们看到 partitioner 没有被测试，但它将被添加到另一个没有显示出来的集成测试中。

例 14-5：单元测试 Flow

```
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, 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}$"));
}
```

对于集成和部署，许多在 Cascading 中的功能可以容易地与外部系统集成并且有更好的容错处理。

在生产中，所有的 SubAssembly 都被连接起来成为 Flow，但不仅仅是 source 和 sink Tap，trap Tap 也在其中。通常，当从远程 Mapper 或 Reducer 任务中的一个操作抛出异常，Flow 就会失败并结束所有它管理的 MapReduce 作业。当 Flow 陷入时，被发现的任何异常以及导致异常的数据被保存到与当前陷阱有关的 Tap。之后下个元组将被处理而没有停止 Flow。有时，你希望你的 Flow 遇到错误便失效。但在这种情况下，ShareThis 开发人员知道他们可以回去查看“失败”的数据并更新

自己的单元测试，而同时生产系统保持运行。丢失几个小时的处理时间远比丢失一些坏的记录更糟糕。

使用 Cascading 的事件监听器，可以将 Amazon SQS 集成进来。当一个 Flow 完成后，便发送消息通知其他系统已有数据准备好可从 Amazon S3 提取。如果失败，则有不同的消息被发送以提醒其他进程。

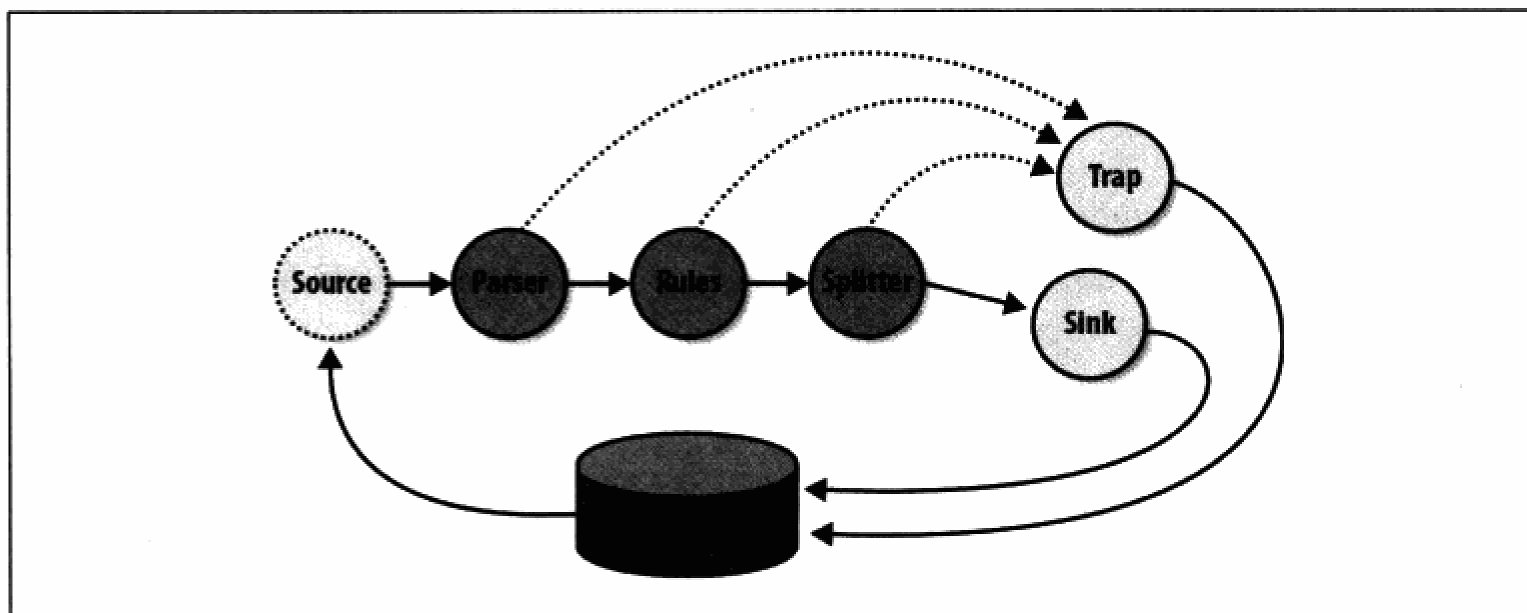


图 14-20：ShareThis 日志处理 Flow

剩余的下行流进程从不同的独立集群上的日志处理管道停止处继续。日志处理管道现在每天运行一次，所以没有必要为了这 23 小时保持 100 个节点的集群。因此它被停用，24 小时后重新启用。

在未来，随着业务需求的增加，在更小的集群上增加该间隔到 6 小时或是 1 小时将是微不足道的。其他集群基于负责该组件的业务单元的需要独立地在不同间隔启动和关闭。例如，网络抓取工具组件(使用 Bixo，一个基于 Cascading 的网页抓取工具，由 EMI 和 ShareThis 开发)可以连续运行在小集群同 Hypertable 群集上。这种按需模式与 Hadoop 一起十分有效，每个集群都可根据期望处理的工作量进行调整。

14.5.7 小结

Hadoop 是处理和协调各种架构组件上的数据移动的一个非常强大的平台。它唯一缺点是主要的计算模型是 MapReduce。

Cascading 的目的是帮助开发人员通过一个优秀的 API 快速简单地构建强大的应用程序，而不需要仔细深入思考 MapReduce，同时舍弃了繁重的数据分布移动、复

制、分发过程管理以及 Hadoop 活跃度。

想要阅读更多关于 Cascading 的内容，可加入网上社区并访问项目网站 *http://www.cascading.org/*下载示例应用程序。

(作者Chris K. Wensel)

14.6 Apache Hadoop 的 1 TB 排序

这篇文章由 http://sortbenchmark.org/YahooHadoop.pdf 再次写成的，它写于 2008 年 5 月。Jim Gray 和他的继承人每年定义一组标准来发现最快的排序程序。1 TB 排序和其他排序标准作为全年冠军被列在 http://sortbenchmark.org/。在 2009 年 4 月，Arun Murthy 和我获得了分排序的冠军(目的是在一分钟内尽可能多地排序)。我们在 1406 个 Hadoop 节点上在 59 秒内排序了 500 GB 的数据。我们也在相同的集群上在 62 秒内排序了 1 TB 的数据。我们在 2009 年使用的集群与下列的硬件很相似，但是它的网络在机架间使用 2-1 超额申请来代替先前的 5-1，性能有了明显改善。我们也在节点间的中间数据上使用了 LZO 压缩。同时，我们还在 3658 个节点上，用 975 秒的时间排序了 1 PB 的数据(即 10^{15} 个字节)，平均速率为 1.03 TB/分钟。关于 2009 年的详细结果，请参见 http://developer.yahoo.net/blogs/hadoop/2009/05/hadoop_sorts_a_petabyte_in_162.html。

Apache Hadoop 是一个开源的软件结构，它大大地简化了写分布式密集型数据的应用。它提供了一个分布式文件系统，后者效仿了 Google File System[①]及一个管理分布式计算的 MapReduce[②]应用。由于 MapReduce 的基本原型是一个分布式排序，所以大多数自定义代码都能够获得想要的行为。

我写了三个 Hadoop 应用来运行 TB 排序。

1. TeraGen 是用来生成数据的 MapReduce 程序。
2. TeraSort 抽样输入数据，并使用 MapReduce 将数据排成总的顺序。
3. TeraValidate 是确认输出被排序的 MapReduce 程序。

Java 代码总数大约是 1000 行，它们将被注册到 Hadoop example 库中。

---

① S. Ghemawat, H. Gobioff 和 S.-T. Leung. “The Google File System.” 第十九届操作系统原理研讨会(2003 年 10 月)Lake George, NY: ACM。

② J. Dean 和 S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters.” 第十六届操作系统设计和执行研讨会(2004 年 12 月), San Francisco, CA。

TeraGen 生成的输出数据与 C 版本字节与字节相当，包括换行符和特定键。它用所需任务数目去除所需行数目，并将行的范围分配给每个 map。map 将跳过随机数生成器，直接到第一行正确的值，再生成接下去的行。在运行的最后，我将 TeraGen 配置成使用 1800 个任务，在 HDFS 中生成了总共 100 亿的行，每个块大小为 512 MB。

TeraSort 是一个标准的 MapReduce 排序，除了一个自定义的 partitioner，它使用了拥有 N-1 样本键的排序列表，这些键为每一个 reduce 定义了键的范围。具体说来，所有的例如 sample[i−1] <= key < sample[i]的键被发送到 reduce i。这样保证了 reduce i 的输入比 reduce i+1 的输出要少。为了加快划分，partitioner 建立了一个两层查找树，它快速索引了基于键中前两个字节的样本键的列表。TeraSort 通过在作业被提交前抽样以及将键的列表写入 HDFS 来生成样品键。我写了一个输入和输出格式。reduce 的输出被设置为 1 次复制，取代了默认的 3 次，因为竞争不需要输出数据被复制到不同的节点上。我让作业拥有 1800 个 map、1800 个 reduce 以及 io.sort.mb, io.sort.factor, fs.inmemory.size.mb，同时将任务堆大小配置得充足，使在 map 尾部的瞬态数据不会溢出到其他磁盘上。采样器使用 100 000 个键来确定 reduce 边界，尽管如图 14-21 所示，在 reduce 间的分布不会因为更多的样本而变得更完美或从中获益更多。可以在图 14-22 看到在整个作业运行上运行的各个任务的分布。

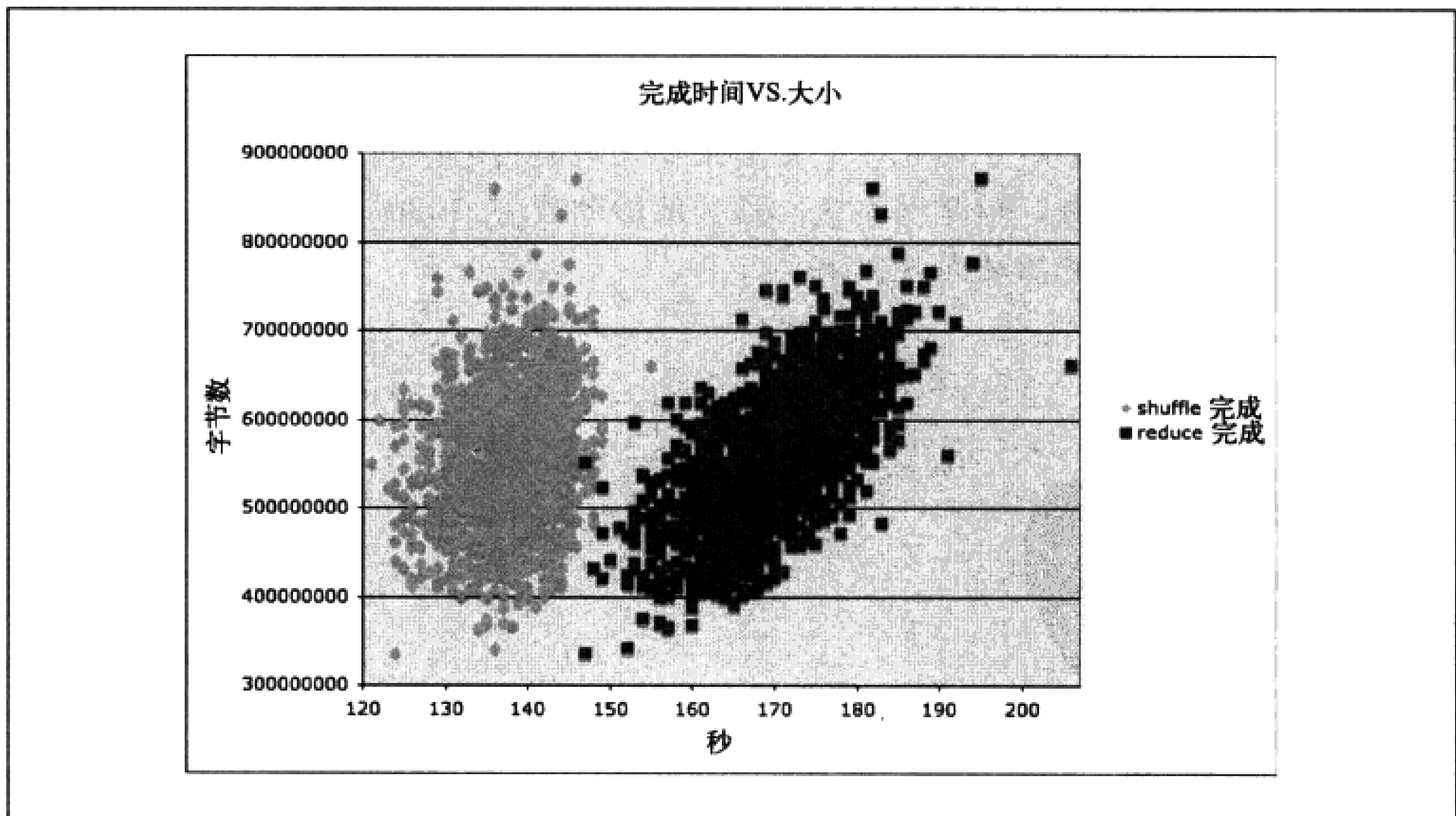


图 14-21：reduce 输出大小绘制 VS 完成时间

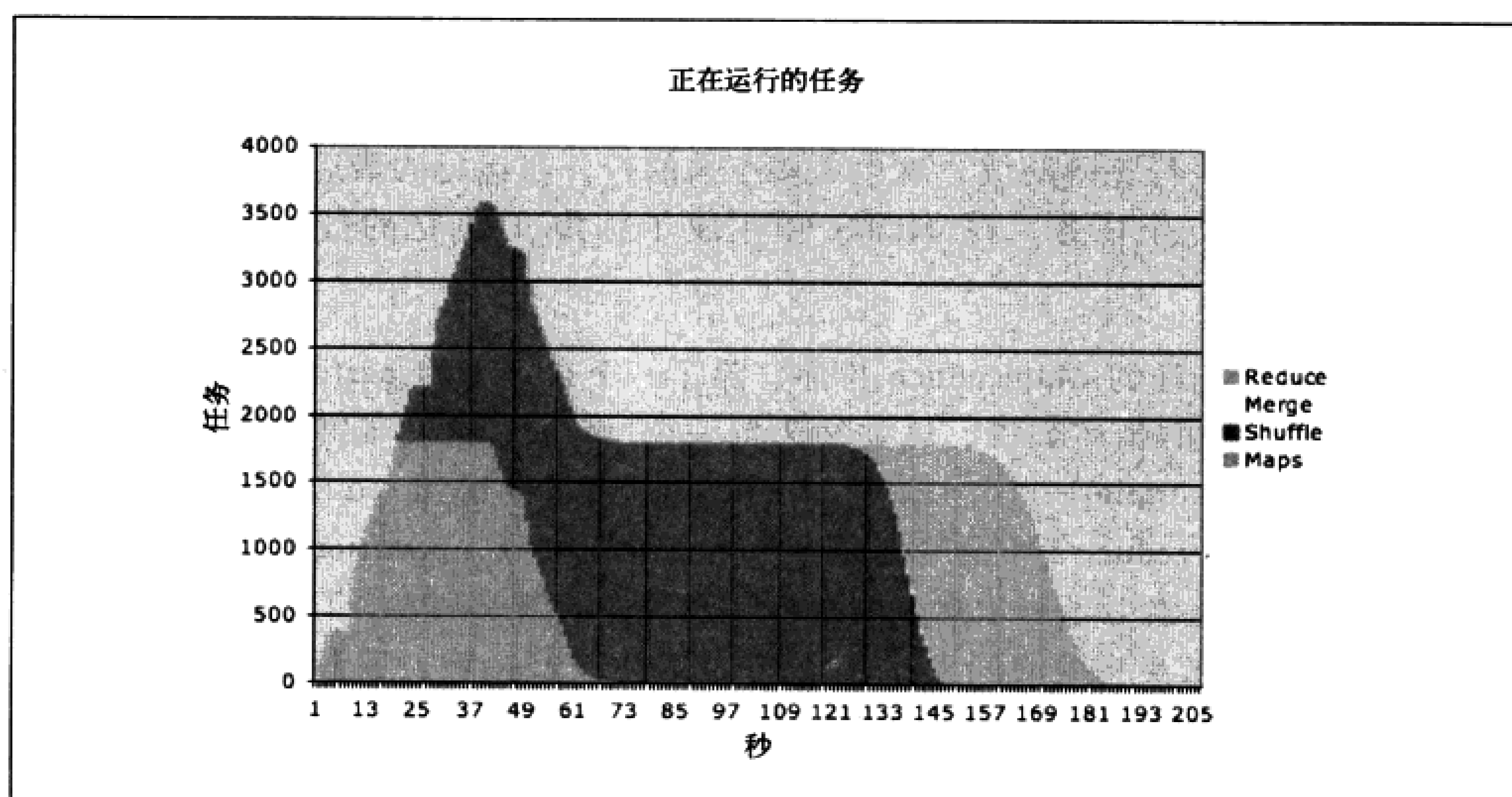


图 14-22：每次每个阶段任务数

TeraValidate 保证了输出是全局性排序的。它为在输出库中的每个文件创建一个 map，而且每个 map 确保了每个键小于或等于之前的一个。map 同时生成了带有文件第一和最后一个键的记录，而且 reduce 保证了文件 i 的第一个键大于文件 i-1 的最后一个键。任何问题以及有错误的键将会作为 reduce 输出被报告。

排序在 209 秒(3.48 分钟)内完成。我运行了带有 HADOOP-3443 and HADOOP-3446 补丁的 Hadoop trunk(前 0.18.0)，它被要求移除对磁盘的中间写操作。尽管我拥有 910 个节点中的大部分，但是网络核心被其他活动着的 2000 个节点集群分享着，因此倍数会根据其他的活动而改变很大。

(作者 Owen O'Malley, Yahoo!)

附录 A

Apache Hadoop 的安装

在一台机器上安装 Hadoop 是非常简单的。(用于群集安装，请参阅第 9 章。)最快的安装方法是从 Apache 软件基金会镜像站下载并运行二进制版本。在附录中，我们介绍如何安装 Hadoop Core，HDFS 和 MapReduce。Pig，HBase 和 ZooKeeper 的安装说明在相关章节(第 11 章、第 12 章和第 13 章)中有介绍。

A.1 先决条件

Hadoop 是用 Java 编写的，所以机器上需要安装 Java 6 或更新版本。Hadoop 最常用的是 Sun 公司的 JDK，不过其他的 JDK 也可以。

Hadoop 可以运行在 Unix 和 Windows 上。Linux 是唯一支持的生产平台，但其他版本的 Unix(包括 Mac OS X)可用于 Hadoop 开发。Windows 只支持作为开发平台，而且需要 Cygwin 才能运行。在 Cygwin 的安装过程中，如果你计划在伪分布模式下运行 Hadoop，就应该包括 Openssh 包。(详见后文的说明)

A.2 安装

首先需要决定你想要运行的 Hadoop 的用户。如果尝试应用 Hadoop 或者开发 Hadoop 程序，最简单的无疑是在单机上运用自己的用户账号运行 Hadoop。

从 Apache Hadoop 的发布页面(*http://hadoop.apache.org/core/releases.html*)下载一个稳定的版本，它会是一个打包成 gzipped tar 的文件，可以将它解压到文件系统的某个地方：

```
% tar xzf hadoop-x.y.z.tar.gz
```

在运行 Hadoop 之前，需要告诉它 Java 在系统上的位置。如果已设置一个 JAVA_HOME 的环境变量，它指向合适的 Java 安装程序，就不必再进行任何配置。否则，可以设置 Hadoop 在编辑 *conf/hadoop-env.sh* 时使用的 Java 安装程序，并指定 JAVA_HOME 变量。举个例子，在我的 Mac 中，我把一行改成如下形式：

```
export JAVA_HOME=/System/Library/Frameworks/JavaVM.framework/
Versions/1.6.0/Home
```

这是为了指向 Java 的 1.6.0 版本。在 Ubuntu 上，相同的一行代码是：

```
export JAVA_HOME=/usr/lib/jvm/java-6-sun
```

创建一个指向 Hadoop 的安装目录的环境变量是非常方便的(HADOOP_INSTALL)，将 Hadoop 的二进制目录放到命令行路径上即可。例如：

```
% export HADOOP_INSTALL=/home/tom/hadoop-x.y.z
% export PATH=$PATH:$HADOOP_INSTALL/bin
```

要检查 Hadoop 的运行，可以键入以下命令：

```
% hadoop version
Hadoop 0.20.0
Subversion https://svn.apache.org/repos/asf/hadoop/core/
branches/branch-0.20 -r 763504
Compiled by ndaley on Thu Apr 9 05:18:40 UTC 2009
```

A.3 配置

Hadoop 中的每个组件都使用一个 XML 文件配置。核心属性在 *core-site.xml* 中，HDFS 属性在 *hdfs-site.xml* 中，MapReduce 属性在 *mapred - site.xml*。这些文件都在 conf 子目录。

注意：在 Hadoop 的早期版本中，有一个专门给 Core、HDFS 和 MapReduce 组件的配置文件，称为 hadoop_site.xml。从版本 0.20.0 起，这个文件已被分成三个：每个组件有

一个。该属性的名称没有改变，只是它们要进入不同的配置文件。你可以看到所有属性的默认设置，这些配置文件通过在 *docs* 目录下访问 Hadoop 安装程序中名为 *core-default.html*、*hdfs-default.html* 和 *mapred-default.html* 的 HTML 文件来管理所有属性。

Hadoop 可以在以下三个模式中的任意一个运行。

本地模式

没有守护进程会运行而且一切都运行在单个 JVM 上。独立模式适用于在开发过程中运行 MapReduce 程序，因为它很容易测试和调试。

伪分布模式

Hadoop 的守护程序在本地计算机上运行，因此是在小范围内模拟集群。

完全分布模式

Hadoop 的守护程序在一群机器上运行。这个设置见第 9 章的描述。

在一个特定的模式运行 Hadoop 时，需要做两件事：设置适当的属性，并启动 Hadoop 的守护进程。表 A-1 显示配置每个模式所需的最少的属性。在独立模式中，会用到本地文件系统和本地 MapReduce 任务运行器，而在分布模式，HDFS 和 Map-Reduce 守护进程会启动。

表 A-1：不同模式下的关键配置属性

组件名称	属性名称	本地模式	伪分布模式	完全分布模式
Core	Fs.defalut.name	File:///(default)	Hdfs://localhost/	Hdfs://namenode/
HDFS	Dfs.replication	N/A	1	3(default)
MapReduce	Mapred.job.tracker	Local(default)	Localhost:8021	Jobtracker:8021

更多配置相关信息，可参见 9.4 节。

A.3.1 独立模式

在独立模式中，我们不会采取进一步行动，因为默认属性被设置为独立模式，而且没有守护进程运行。

A.3.2 伪分布模式

创建配置文件应该要有以下内容，并放置在 conf 目录下(虽然只要开启守护进程时

附带--config 选项，就可以将配置文件放在任何地方)。

```
<?xml version="1.0"?>
<!-- core-site.xml -->
<configuration>
  <property>
    <name>fs.default.name</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>mapred.job.tracker</name>
    <value>localhost:8021</value>
  </property>
</configuration>
```

SSH 的配置

在伪分布模式中，必须开启守护进程，为此，我们需要 SSH 的安装程序。Hadoop 实际上无法区分伪分布模式和完全分布模式：SSH-ing 只是在启动集群主机上的保护进程时才会在每个主机上启动一个守护进程的程序。主机是本地计算机的时候，伪分布模式只不过是完全分布模式的一个特例，因此我们需要确保我们能 SSH 到本地主机而且无需输入密码即可登录。

首先，确保 SSH 已被安装并且服务器正在运行。例如，在 Ubuntu，可以进行如下设置：

```
% sudo apt-get install ssh
```

注意：在带有 Cygwin 的窗口下，可以通过运行 ssh-host-config -y 开启一个 SSH 服务器(在安装 openssh 包之后)。

为了自动登录，可以用空字符生成一个新 SSH 密钥：

```
% ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
% cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

用以下这个测试：

```
% ssh localhost
```

不输入密码就可以登录。

HDFS 文件系统格式化

在可以使用之前，一个全新的 HDFS 安装需要进行格式化。格式化过程通过创建存储目录以及名称节点的永久数据结构的最初版本来创建一个空的文件系统。由于名称节点管理了所有文件系统的元数据，最初的格式化过程并不涉及数据节点，名称节点可以动态的加入或者离开集群的。同样的原因；你不需要知道你需要创建多大的文件系统，因为它是由集群中数据节点的数量决定的，它可以在文件系统格式化很久之后按照需要增加。

格式化 HDFS 是很快的。只需输入以下内容：

```
% hadoop namenode -format
```

开始和结束守护进程

要启动 HDFS 和 MapReduce 守护进程，输入以下内容：

```
% start-dfs.sh
% start-mapred.sh
```

注意：如果在默认的 conf 目录之外，可以使用--config 选项开始守护进程，这样便可以获得配置目录的绝对路径。

```
% start-dfs.sh --config path-to-config-directory
% start-mapred.sh --config path-to-config-directory
```

三个守护进程会在本地机器上启动：一个名称节点，一个二级名称节点和一个数据节点。可以通过查阅日志目录的 logfiles 或者查阅有关 jobtracker 的 *http://localhost:50030/*或者有关 namenode 的 *http://localhost:50070/*上的 UI 来检查

守护进程是否启动成功。也可以使用Java的jps命令来查看它们是否运行。

停止守护进程的方法很直观：

```
% stop-dfs.sh
% stop-mapred.sh
```

A.3.3 完全分布模式

启动一个机器集群会带来其他麻烦，所以在第9章中对这种模式的启动方法有更详细的介绍。

附录 B

Cloudera 的 Hadoop 分发包

(作者 *Matt Massie 和 Todd Lipcon(Cloudera)*)

Cloudera 的 Hadoop 分发包基于最新、最稳定的 Apache Hadoop 版本，它由大量的补丁，修改(backpoint)[①]以及更新组成。Cloudera 通过很多不同的格式分享了这种分布状态：tar 压缩文件、RPM、Debian 包以及 Amazon EC2 AMI。Hadoop 的 Cloudera 的分布是免费的，在 Apache 2.0 的许可下发行，可以在 *http://www.cloudera.com/hadoop/*可以找到。

Cloudera 在 *http://www.cloudera.com/configurator* 有一个在线配置器，可用于轻松建立一个 Hadoop 集群(图 B-1)。配置器有一个简单的向导型界面询问关于集群的固定问题。完成后，配置器会生成用户化 Hadoop 的包并将其放置在你的包库中。可以管理任何数量的集群，并且为了更新动态配置，可以稍后返回。

为了简化对包的管理，Cloudera 在一个 yum 库中分享了 RPM 并且在一个 apt 库中分享了 Debian 包。Hadoop 中的 Cloudera 分发包可供你通过运行一个单一、简单的命令在集群每一台机器上安装并配置 Hadoop。新手用户可以受益更多，因为可以自动委托整个 Hadoop 集群而不用任何一点手动干涉。

① 译注：backpoints 很难翻译。在维基百科的定义是“Backporting is the action of taking a certoin software modification(patch) and applying it to an old version of the software than it was initially created for.”由于是把新版本的补丁应用到老版本上，所以有 back 的含义。

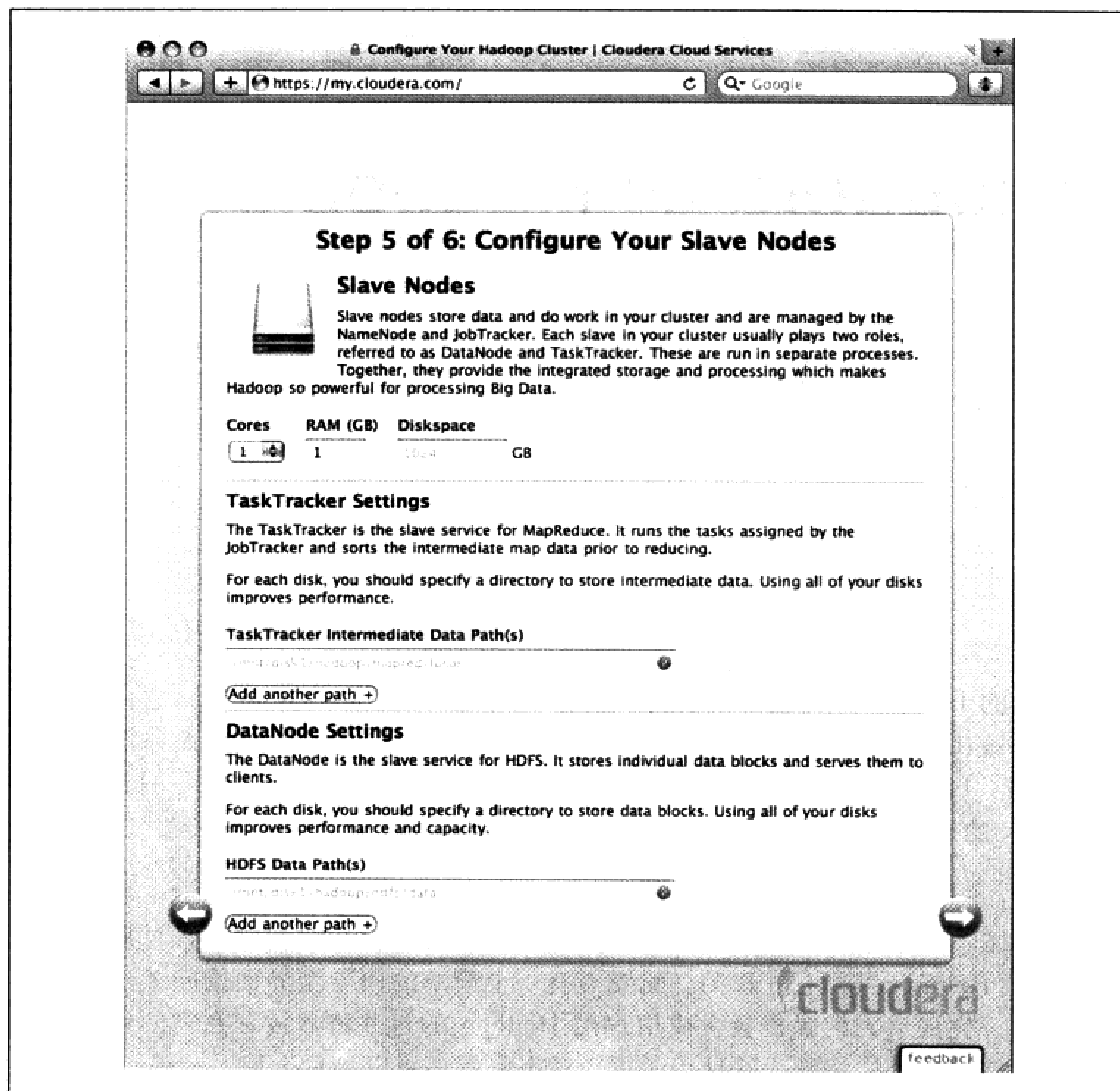


图 B-1：Cloudera 的在线配置器简化了 Hadoop 集群设置

B.1 先决条件

Hadoop 的 Cloudera 分发需要安装 Sun Java 6 或更新版本。Sun Java Debian 和 RPM 包要求要在使用之前同意 Sun 许可。对于基于 Debian 的系统，你会希望使用付费的 `apt` 库，因为它包含 `sun-java6d-`条款的包。对于基于 Red Hat 的系统，可以从 *http://java.sun.com/javase/downloads/*下载 Sun Java RPM 包。

在可以使用自己最喜欢的包管理器中(如 `yum`，`apt-get` 和 `aptitude`)安装 Cloudera

分发包之前，需要在 yum 和/或 apt 源列表中加入 Cloudera 的库。

要想用最简单的方法满足这些先决条件，请访问 *http://www.cloudera.com/hadoop/*，了解最新指导。

B.2 独立模式

要在独立模式安装 Hadoop，需要在基于 Red Hat 的系统上运行以下命令：

```
% yum install hadoop
```

若基于 Debian 的系统，则运行以下命令：

```
% apt-get install hadoop
```

Hadoop 分发包包括一个说明页。要阅读说明页，运行以下命令：

```
% man hadoop
```

如果想在一台机器上安装完整的 Hadoop 的文件，则安装 hadoopdocs 包。在基于 Red Hat 的系统上，运行以下命令：

```
% yum install hadoop-docs
```

要在 Debian 系统上安装该文件，运行以下命令：

```
% apt-get install hadoop-doc
```

B.3 伪分布模式

要在伪分布模式安装 Hadoop，在基于 Red Hat 的系统上运行以下命令：

```
% yum install hadoop-conf-pseudo
```

若基于 Debian 的系统，则运行以下命令：

```
% apt-get install hadoop-conf-pseudo
```

安装 Hadoop 伪分布式配置包后，若想开启 Hadoop 服务，需要在基于 Red Hat 和基于 Debian 的系统上运行相同的命令：

```
% for x in namenode secondarynamenode datanode jobtracker
tasktracker ;
```

```
do /etc/init.d/hadoop-$x start ; done
```

无需担心创建 Hadoop 的用户或格式化的 HDFS，因为这由 hadoop-conf-pseudo 包自动处理。在安装完包并开启服务之后，可以立即使用 Hadoop。Hadoop-conf-pseudo 包也会保证你的 Hadoop 服务能在系统引导时开启。

B.4 完全分布式模式

有关部署一个完全分布式 Hadoop 集群的详情，可以访问 Cloudera 的 Hadoop 分发包的网页：*http://www.cloudera.com/hadoop/*。

运行 Cloudera 的在线配置器时，它会创建一个个性化的 apt 或 yum 库，如此一来，你管理的每个集群都将拥有配置包。举个例子，假设你为你的一个集群命名为 mycluster。要想看到 mycluster 的所有配置包的列表，则在基于 Red Hat 的系统上运行以下命令：

```
% yum search hadoop-conf-mycluster
```

若基于 Debian 的系统，则运行以下命令：

```
% apt-cache search hadoop-conf-mycluster
```

这些命令将把配置包列表返回给 mycluster 集群。配置包的数量和类型取决于你是如何回答 Cloudera 配置器所提出的问题的。这其中的一些包是给集群中的特定主机的，其他的则给予集群的一组或一类机器。对于特别主机的配置，完全合格的主机名将被添加到包名中。例如，在 mycluster 集群中，可能有一个名为 myhost.mydommain 的配置。在基于 Red Hat 的系统上的 myhost.mydomain 上安装 Hadoop，运行以下命令：

```
% yum install hadoop-conf-mycluster-myhost.mydomain
```

若基于 Debian 的系统，则运行以下命令：

```
% apt-get install hadoop-conf-mycluster-myhost.mydomain
```

在 Hadoop 的配置包将确保你的服务在系统启动时开启。

B.5 Hadoop 分发包

Hadoop 中的 Cloudera 的 Hadoop 分发包可用于轻松配置那些建立在 Hadoop 最高层

上的 Hive 和 Pig 等工具。Hive 是一个数据仓库基础架构，它允许你在 Hadoop 中使用基于 SQL 的查询语言来查询数据。若要了解关于 Hive 的详细信息，请参见 14.2 节。Pig 是一个使用高层语言分析大型数据集的平台，在第 11 章中有详细介绍。

在基于 Red Hat 的系统上安装 Hive 和 pig，可运行以下命令：

```
% yum install hadoop-hive hadoop-pig
```

在基于 Debian 的系统上安装 Hive 和 pig，可运行以下命令：

```
% apt-get install hadoop-hive hadoop-pig
```

随着时间的推移，更多 Hadoop 分发包将被加到 Cloudera 的 Hadoop 分发包中。

附录 C

预备 NCDC 气象资料

这里为准备天气的原数据文件给出详细的步骤，所以它们是一种可以用 Hadoop 分析的形式。如果想要得到一份数据副本并用 Hadoop 处理，可以按照网页 *http://hadoopbook.com/*的指示进行操作，该网页还附带了本书。下面将解释未加工的天气数据文件是如何处理的。数据以 tar 文件集的形式提供原始文件集，被压缩为 bzip2 格式。每年的读数放在单独的文件里。文件的部分目录列表如下：

```
1901.tar.bz2
1902.tar.bz2
1903.tar.bz2
...
2000.tar.bz2
```

每一个 tar 文件都包含一个每个气象站年度读数的文件，此文件用 gzip 压缩。(即在存档中的文件被压缩的事实使存档中的 bzip2 压缩变得冗余。)例如：

```
% tar jxf 1901.tar.bz2
% ls -l 1901 | head
011990-99999-1950.gz
011990-99999-1950.gz
...
011990-99999-1950.gz
```

由于世界上有数万个气象站，所以整个数据集是由许多相关的小文件组成的。一般，在 Hadoop 中处理小数量的相对更大文件会比较简单、迅速，所以在这个案例中，我解压缩了整个年度的文件，将它们连成一个单独的文件，以年份每年命名。我利用 MapReduce 程序的并行处理能力做到这点，让我们仔细看一下这个程序。

程序只有一个 map 函数：由于 map 并行处理了所有的文件，无需合并阶段，所以并不需要 reduce 函数。处理过程可以使用 Unix 脚本，所以 MapReduce 的 Streaming 接口在这个案例中是适用的。

见例 C-1。

例 C-1：Bash 脚本处理原始 NCDC 数据文件并存储到 HDFS

```
#!/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_INSTALL/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 | $HADOOP_INSTALL/bin/hadoop fs -put -
gz/$target.gz
```

输入是一个很小的文本文件(ncdc_files.txt)，它列出了所有要处理的文件(文件以 S3 开头，所以被引用的文件都是 Hadoop 能识别的 S3 URI)。下面是一个示例：

```
s3n://hadoopbook/ncdc/raw/isd-1901.tar.bz2
s3n://hadoopbook/ncdc/raw/isd-1902.tar.bz2
...
s3n://hadoopbook/ncdc/raw/isd-2000.tar.bz2
```

通过指定输入格式 NLineInputFormat，每个 mapper 接收输入数据中的一行，其中包含必须处理的文件。处理是用脚本解释的，但非常简单，它将 bzip2 的文件解压缩，然后把一整年中每个站文件连接起来成为一个单独文件。最后，将文件打包并复制到 HDFS 中去。注意，hadoop fs -put - to 的使用必须是基于标准的输入。

状态消息将在标准错误报告中显示：状态前缀随 MapReduce 状态更新而被中断。这告诉 Hadoop 的该脚本正在进行，并且不是挂起。

该脚本运行 Streaming 作业如下：

```
% hadoop jar $HADOOP_INSTALL/contrib/streaming/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
```

由于它是一道只含 map 的作业，所以我将 reduce 任务的数量设为零。我也关闭了推测式执行，所以重叠的任务并不会写入同一个文件。任务超时时间被设定的很高，因此 Hadoop 并不会杀死占用长时间的任务(例如，当文件解压缩或在复制到 HDFS 时，没有进展报告)。

最后，文件用 distcp 将它们从 HDFS 中复制出并存档到 S3。