



华章IT
Apress®

世界级大数据专家亲笔撰写，深入剖析Hadoop系统实现原理及其工程实践应用，包含大量真实案例和数据

从Hadoop使用者角度深入浅出地讲解Hadoop各组件的运行机理，深入到系统实现源码，帮助你从架构、开发、应用和运维等多方面全面掌握Hadoop系统



技术丛书

Pro Apache Hadoop

Second Edition

深入理解Hadoop

(原书第2版)

萨米尔·瓦德卡 (Sameer Wadkar)
[美] 马杜·西德林埃 (Madhu Siddalingaiah) 著
杰森·文纳 (Jason Venner)

于博 冯傲风 译



机械工业出版社
China Machine Press

大数据技术丛书

深入理解 Hadoop

(原书第 2 版)

Pro Apache Hadoop, Second Edition

[美] 萨米尔·瓦德卡 (Sameer Wadkar)

马杜·西德林埃 (Madhu Siddalingaiah)

杰森·文纳 (Jason Venner)

于 博 冯傲风 译

HZ BOOKS

华章图书



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

深入理解 Hadoop (原书第 2 版) / (美) 瓦德卡 (Wadkar, S.), (美) 西德林埃 (Siddalingaiah, M.), (美) 文纳 (Venner, J.) 著; 于博, 冯傲风译. —北京: 机械工业出版社, 2015.10 (大数据技术丛书)

书名原文: Pro Apache Hadoop, Second Edition

ISBN 978-7-111-51565-4

I. 深… II. ①瓦… ②西… ③文… ④于… ⑤冯… III. 数据处理软件 IV. TP274

中国版本图书馆 CIP 数据核字 (2015) 第 227389 号

本书版权登记号: 图字: 01-2015-4154

Sameer Wadkar, Madhu Siddalingaiah and Jason Venner: Pro Apache Hadoop, Second Edition (ISBN : 978-1-4302-4863-7).

Original English language edition published by Apress L.P., 2560 Ninth Street, Suite 219, Berkeley, CA 94710 USA. Copyright © 2014 by Sameer Wadkar and Madhu Siddalingaiah. Simplified Chinese-language edition copyright © 2016 by China Machine Press. All rights reserved.

This edition is licensed for distribution and sale in the People's Republic of China only, excluding Hong Kong, Taiwan and Macao and may not be distributed and sold elsewhere.

本书原版由 Apress 出版社出版。

本书简体字中文版由 Apress 出版社授权机械工业出版社独家出版。未经出版者预先书面许可, 不得以任何方式复制或抄袭本书的任何部分。

此版本仅限在中华人民共和国境内 (不包括中国香港、台湾、澳门地区) 销售发行, 未经授权的本书出口将被视为违反版权法的行为。

HZ BOOKS
华章图书

深入理解 Hadoop (原书第 2 版)

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 吴 怡

责任校对: 殷 虹

印 刷:

版 次: 2016 年 1 月第 1 版第 1 次印刷

开 本: 186mm × 240mm 1/16

印 张: 25

书 号: ISBN 978-7-111-51565-4

定 价: 79.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有 · 侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

The Translator's Words 译者序

我们正处在一个数据量爆炸增长的时代，也是一个越来越依靠数据进行决策的时代，对海量数据的挖掘分析也从抽样数据分析变成了全量数据分析。为此，业界涌现出了许多新技术，Apache Hadoop 是其中当之无愧的王者。当前 Hadoop 已然发展为拥有 HDFS、YARN、MapReduce、Pig、HCatalog、HBase 等系统的较为完整的大数据系统生态圈，为大数据的开发利用提供了比较全面的解决方案。毫无疑问，Hadoop 已成为大数据行业发展背后强劲的驱动力。

本书详细地讲述了 Hadoop 生态圈中最为重要的几个组件。不仅介绍了 Hadoop 涉及的分布式理论基础知识，还着重讲解 Hadoop 系统的工程实践应用。为了深入浅出地讲述 Hadoop 各个组件的运行机理，作者使用了贴切的实战用例贯穿全书，同时，又深入到系统实现源码，使读者做到知其然，更知其所以然。通过对本书的学习，读者可以从架构、开发、应用和运维等多方面全面地掌握 Hadoop 系统，从而成为名副其实的 Hadoop 专家。

我曾经拜读过本书英文版的第 1 版，机缘巧合，华章给了我翻译该书第 2 版的机会，感谢我的小伙伴百度公司的冯傲风同学与我并肩奋战，完成了本书的翻译；感谢吴怡编辑对本书认真负责的审校和中肯的修改建议，使得本书能够顺利地出版，她的认真和敬业令人钦佩！

最后还要感谢家人，只有他们的理解和支持，我才能每天抽出那么多时间来完成本书的翻译工作。

书中概念和术语颇多，有一些目前尚无固定中文译法，加上译者水平有限，译文中的不当之处在所难免，我们真诚地希望同行和读者们不吝赐教。

于博

2015 年 9 月于北京

作者简介 *About the Author*



萨米尔·瓦德卡 (Sameer Wadkar) 在软件架构与开发方面有多达 16 年的工作经验。他在孟买大学取得了电子工程方向的硕士学位，及金融专业的 MBA 学位，在 National Center of Software Technology (现为 Center for Development of Advanced Computing) 取得了软件工程专业的硕士学位，在约翰霍普金斯大学获得了应用与计算数学专业的博士学位。他还实现了分布式系统及高访问量的 Web 网站，从政府机构到银行投资公司多种领域都有客户使用这个 Web 网站。他从 2011 年就积极地参与到了 Hadoop/HBase 的开发实现当中，也是开源软件的贡献者。他的 GitHub 的主页是：<https://github.com/sameerwadkar>。

他在开源领域的贡献包括 Latent Dirichlet Allocation 文本挖掘算法，该算法能够在单台机器上处理数百万级别的文档。他还是一位活跃的国际象棋玩家，经常在网站 www.chessclub.com 上下国际象棋。



马杜·西德林埃 (Madhu Siddalingaiah) 是一名拥有 25 年工作经验的技术顾问，其工作经验涉及多个领域，包括航空航天、医疗保险、金融、能源、科学研究等。多年以来，他专门从事电子工程、互联网技术、大数据等领域。最近，他发布了几个著名的大数据系统及其解决方案。

他在马里兰大学取得了物理学的硕士学位。专业之外，Madhu 还是一名私人直升飞机驾驶员，他喜欢旅行，不断参与越来越多的爱好活动。

杰森·文纳 (Jason Venner) 有 20 多年的软件开发经验，涉及软件工程、数据挖掘、架构设计等领域，曾在 IT 企业中担任过副总裁、主管、顾问，近些年来他的钻研方向是 Java、Hadoop、云计算等领域。

技术评审人简介



威米什·奥姆·米塔尔 (Vimlesh Om Mittal) 具有多达 14 年的技术实现经验，是 Apache Hadoop CDH4 的 Cloudera 认证开发者 (Cloudera Certified Developer)。他具有非常广博的技术背景，他的经验主要包括关于现代商业处理的相关项目、客户应用开发、ETL 开发、数字应用开发、商业智能、数据库设计和数据转换策略。

他拥有构建健壮的信息管理系统、解决方案架构设计和基于多种开发框架进行软件开发的综合经验。这些开发框架包括 Hadoop 生态系统中的大数据框架、数据获取、ETL、报告 / 分析、.NET、J2EE 和不同技术供应商提供的脚本语言，这些技术供应商包括 Cloudera、SAS、IBM、Oracle、Microsoft、iOS 以及 BEA。

Vimlesh 还专注于金融服务数据的解决方案、移动应用设计及开发。他对软件开发的整个生命周期中的各个方面都十分熟悉，在项目管理科学方面有宝贵经验。

前言 *Preface*

Hadoop 已经进入 Apache 社区发展五年多了，使用 Hadoop 系统进行开发的工作仍然富于挑战但收获丰厚。本书第 1 版在若干年前就已经出版了，在这期间，Hadoop 系统已经被越来越多的企业使用，自身也得到了飞速发展。Hadoop2.0 基于 YARN 框架做了全新升级，重写了 Hadoop 系统的底层平台。本书从 Hadoop 使用者的角度出发讲解 Hadoop 的实现原理，浓缩了 Hadoop 软件系统的精华。作为作者，我们希望可以深入到源代码级别来理解 Hadoop 的运行原理及其背后的设计目标，渴望与你分享 Hadoop 带给我们的启迪。通过本书，你不仅能够深入学习 Hadoop 系统，还可以对 Java 语言在大数据处理过程中的运用有更深入的认识。

本书总体上介绍了大数据的范畴概念，然后对 Hadoop 进行了详细讲解。因为如果对大数据没有一个总体上的认知，是不可能真正理解 Hadoop 系统的。本书的读者对象具备中级 Java 开发能力的 Hadoop 开发人员。本书可以让你成为企业中的 Hadoop 专家。通过本书的学习，你可以获得大量实用技巧，这些技巧都是我们在从事基于 Hadoop 的分布式数据处理系统实践过程中总结出来的。

本书循序渐进地讲解了大量的实例，会帮助你从 Hadoop 集群初级使用者成长为能胜任运行任何复杂应用程序的专家。下面是本书的一个内容纲要：

- ❑ 第 1 章，讲解大数据系统的作用，介绍各种各样的大数据系统。
- ❑ 第 2 章，高屋建瓴地介绍 Hadoop2.0 和 YARN，讲解 Hadoop 平台的关键概念。
- ❑ 第 3 章，开始 Hadoop 学习，构建你的第一个 MapReduce 程序。
- ❑ 第 4 章，讲解 Hadoop 平台管理的关键概念要素。
- ❑ 第 5 章、第 6 章和第 7 章是本书的重点，深入分析讲解了 MapReduce 框架。你将会学习 MapReduce 框架所有知识点。我们以当前广泛使用的语言 SQL 作为对照，来学习理解 MapReduce 框架的使用。利用 MapReduce 框架来实现 SQL 语言中 SELECT、WHERE、GROUP BY 和 JOIN 这些关键字的功能。Hadoop 系统一个常用用途就是来做数据 ETL。这几章会让你掌握如何使用 MapReduce 框架支持通用的数据处理功能。我们不仅仅学习 MapReduce 框架的 API，还会学习 MapReduce 框架中更复杂的概念

及其设计理念。

- ❑ 第 8 章，介绍一种测试框架，这种测试框架支持 MapReduce 程序的单元测试和集成测试。
- ❑ 第 9 章，讲解 Hadoop 框架的监控和日志。
- ❑ 第 10 章，讲解 Hive 框架，这是一个基于 MapReduce 的数据仓库框架。
- ❑ 第 11 章，讲解 Pig 和 Crunch 框架。这些框架可以帮助使用者在 Hadoop 平台上构建数据处理管道。
- ❑ 第 12 章，讲解 HCatalog 框架。该框架帮助企业用户可以像访问数据库中的数据表一样，来访问存放在 Hadoop 文件系统上的海量数据。
- ❑ 第 13 章，讲解如何使用 Hadoop 来分析处理日志流。
- ❑ 第 14 章，介绍 HBase，这是一个基于 Hadoop 系统的 NoSQL 数据库。你会学习许多 HBase 用法范例。
- ❑ 第 15 章，对数据科学做简要介绍。介绍了 MapReduce 框架在数学科学上的局限和不足之处。同时引入介绍新的框架，比如 Spark、Hama，这些框架突破了 MapReduce 框架中的某些局限。
- ❑ 第 16 章，简要介绍了 Hadoop 系统在云计算中的应用。讲解如何在一个真实的生产环境中的 Hadoop 集群上工作。
- ❑ 第 17 章，粗略地介绍 Hadoop2.0 的一个关键附加功能：在 Hadoop 上开发一个自己的类似于 MapReduce 框架的分布式数据处理框架。讲解了如何基于 Hadoop2.0 开发一个简单的分布式下载服务。

致谢

Hadoop 在过去的 10 年发展过程中，有许多厂商和独立开发者为之贡献代码。Hadoop 系统中也使用了大量的开源函数库。我们要感谢为 Hadoop 系统贡献源码的所有开源作者所付出的辛勤劳动。我们也要感谢那些在论坛上回答了 Hadoop 系统相关问题的博客作者及专家学者。他们的积极参与，使得 Hadoop 这样复杂的系统变得易于使用并成为应用主流。

我们还想感谢 Apress 的全体员工，他们的努力付出使得本书内容清晰易读。

我们最后要感谢我们的家庭、朋友及同事，本书在编写过程中得到他们的一贯支持。

目 录 *Contents*

译者序		第 2 章 Hadoop 中的概念	13
作者简介		2.1 Hadoop 简介	13
前言		2.2 MapReduce 编程模型简介	15
第 1 章 为什么会有大数据	1	2.3 Hadoop 系统的组成	19
1.1 什么是大数据	1	2.3.1 Hadoop 分布式文件系统	20
1.2 大数据技术背后的核心思想	2	2.3.2 辅助名称节点	25
1.2.1 把数据分发到多个节点	2	2.3.3 任务跟踪器	26
1.2.2 把计算逻辑移动到数据附近	3	2.3.4 作业跟踪器	26
1.2.3 计算节点进行本地数据处理	3	2.4 Hadoop 2.0	27
1.2.4 优选顺序读，次之随机读	4	2.4.1 容器	29
1.2.5 一个例子	4	2.4.2 节点管理器	29
1.3 大数据的编程模型	5	2.4.3 资源管理器	30
1.3.1 大规模并行处理数据库系统	5	2.4.4 应用程序管理器	30
1.3.2 内存数据库系统	6	2.4.5 分步详解 YARN 请求	31
1.3.3 MapReduce 系统	6	2.5 HDFS 的高可用性	33
1.3.4 整体同步并行系统	8	2.6 本章小结	33
1.4 大数据和事务性系统	8	第 3 章 初识 Hadoop 框架	34
1.5 我们能处理多大的数据量	9	3.1 安装类型	34
1.5.1 一个计算密集型的例子	10	3.1.1 单机模式	35
1.5.2 Amdhal 定律	10	3.1.2 伪分布式集群模式	35
1.6 大数据商业用例	11	3.1.3 多节点集群安装模式	35
1.7 本章小结	12	3.1.4 基于 Amazon EMR 预安装模式	35

3.2 使用 Cloudera 虚拟机搭建开发环境.....	36	4.5.6 按照 drf 策略来确定优势资源的分配	68
3.3 一个 MapReduce 程序的组成.....	37	4.6 从属文件.....	69
3.4 第一个 Hadoop 程序.....	38	4.7 机架感知.....	69
3.4.1 以本地模式运行程序的必要条件	39	4.8 集群管理工具.....	71
3.4.2 使用旧 API 编写的单词计数程序	39	4.8.1 检查 HDFS	71
3.4.3 构建程序	42	4.8.2 HDFS 管理命令行	73
3.4.4 在集群模式下运行单词计数程序	42	4.8.3 均衡 HDFS 上的数据分布	75
3.4.5 使用新 API 编写的单词计数程序	43	4.8.4 从 HDFS 中复制海量数据	76
3.4.6 构建程序	44	4.9 本章小结.....	76
3.4.7 在集群模式下运行单词计数程序	45		
3.5 Hadoop 作业中的第三方函数库.....	45	第 5 章 MapReduce 开发基础	78
3.6 本章小结.....	50	5.1 Hadoop 和数据处理	78
第 4 章 Hadoop 系统管理	51	5.2 航空公司数据集介绍.....	79
4.1 Hadoop 的配置文件.....	51	5.2.1 准备开发环境	80
4.2 配置 Hadoop 守护进程.....	52	5.2.2 准备 Hadoop 系统.....	81
4.3 Hadoop 配置文件的优先级.....	53	5.3 MapReduce 编程模式	81
4.4 深入探究 Hadoop 配置文件.....	54	5.3.1 只有 Map 阶段的作业 (SELECT 和 WHERE 查询)	82
4.4.1 core-site.xml	54	5.3.2 问题定义——SELECT 子句.....	82
4.4.2 hdfs-*.xml	55	5.3.3 问题定义——WHERE 子句	90
4.4.3 mapred-site.xml	56	5.3.4 Map 和 Reduce 作业 (聚合查询)	93
4.4.4 yarn-site.xml	58	5.3.5 问题定义——GROUP BY 和 SUM 子句.....	93
4.4.5 YARN 中的内存分配.....	60	5.3.6 应用 Combiner 提高 Aggregation 性能	99
4.5 调度器	61	5.3.7 问题定义——优化后的 Aggregators.....	99
4.5.1 计算能力调度器	62	5.3.8 Partitioner 的作用.....	104
4.5.2 公平调度器	65	5.3.9 问题定义——按月分离航空数据	105
4.5.3 公平调度器配置	65	5.4 综合分析.....	108
4.5.4 yarn-site.xml 配置	66	5.5 本章小结.....	110
4.5.5 策略文件的格式和配置	67		

第 6 章 MapReduce 开发进阶	111	作业	147
6.1 MapReduce 编程模式	111	6.6.3 总结探讨 Map-Only 连接时的 Hadoop 关键特性	149
6.2 Hadoop I/O 介绍	111	6.7 在 MR 作业中保存结果到多 输出文件	149
6.3 问题定义——排序	114	6.8 使用计数器收集统计数据	151
6.3.1 主要挑战：全排序	115	6.9 本章小结	153
6.3.2 在 Cluster 中运行 Sorting 作业	125		
6.3.3 仅根据 Writable 键排序	125		
6.3.4 根据排序回顾 Hadoop 的关键 特性	128		
6.4 问题定义——分析连续的记录	128	第 7 章 Hadoop 输入 / 输出	155
6.4.1 支持二次排序的重要组件	129	7.1 压缩方式	155
6.4.2 在没有 Grouping Comparator 的 情况下实现 Secondary Sort	136	7.1.1 压缩内容的选择	156
6.4.3 在 Cluster 中运行 Secondary Sort 作业	137	7.1.2 各种压缩方式	157
6.4.4 利用 Secondary Sort 回顾 Hadoop 的关键特性	137	7.1.3 配置压缩方式	158
6.5 问题定义——使用 MapReducer 进行连接	138	7.2 Hadoop 的 I/O 处理过程内部	159
6.5.1 处理多输入：Multiple- Inputs 类	138	7.2.1 Inputformat	159
6.5.2 具备多个输入的 Mapper 类	139	7.2.2 OutputFormat	161
6.5.3 自定义 Partitioner: Carrier- CodeBasedPartioner	141	7.2.3 自定义 OutputFormat：将文本 转换成 XML	161
6.5.4 在 Reducer 中实现连接	141	7.2.4 自定义 InputFormat：使用 自定义的 XML 文件	165
6.5.5 在集群中运行 MapReduce 连接 作业	143	7.3 Hadoop 文件	173
6.5.6 探讨与 MapReduce 相关的 Hadoop 主要特性	144	7.3.1 SequenceFile	173
6.6 问题定义——使用 Map-Only 作业 进行连接	144	7.3.2 MapFiles	178
6.6.1 基于 DistributeCache 的解决 方案	145	7.3.3 Avro Files	180
6.6.2 在集群中运行 Map-Only 的连接		7.4 本章小结	185
		第 8 章 测试 Hadoop 程序	186
		8.1 回顾一下单词统计的程序	186
		8.2 MRUnit 概述	188
		8.2.1 安装 MRUnit	188
		8.2.2 MRUnit 核心类	188
		8.2.3 编写一个 MRUnit 测试用例	189
		8.2.4 测试计数器	191
		8.2.5 MRUnit 的特性	194
		8.2.6 MRUnit 的局限性	194

8.3 用 LocalJobRunner 测试.....	195	10.1.4 HiveQL 编译基础.....	217
8.3.1 setUp() 方法.....	196	10.1.5 Hive 使用的概念.....	218
8.3.2 LocalJobRunner 的局限性.....	197	10.1.6 HiveQL 编译细节.....	222
8.4 用 MiniMRCluster 测试.....	198	10.1.7 数据定义语言.....	226
8.4.1 配置开发环境.....	198	10.1.8 数据操作语言.....	226
8.4.2 MiniMRCluster 例子.....	199	10.1.9 扩展接口.....	227
8.4.3 MiniMRCluster 的局限性.....	201	10.1.10 Hive 脚本.....	229
8.5 对访问网络资源的 MR 作业进行 测试.....	202	10.1.11 性能表现.....	229
8.6 本章小结.....	202	10.1.12 整合 MapReduce.....	230
第 9 章 Hadoop 的监控	203	10.1.13 创建分区.....	230
9.1 在 Hadoop MapReduce Jobs 中写 日志消息.....	203	10.1.14 用户定义函数.....	232
9.2 在 Hadoop MapReduce Jobs 中查看 日志消息.....	206	10.2 Impala.....	234
9.3 在 Hadoop 2.x 中使用日志管理.....	208	10.2.1 Impala 架构.....	234
9.3.1 Hadoop 2.x 中的日志存储.....	208	10.2.2 Impala 特性.....	235
9.3.2 日志管理提升.....	210	10.2.3 Impala 的局限.....	235
9.3.3 使用基于 Web 的界面查看 日志.....	210	10.3 Shark.....	235
9.3.4 命令行界面.....	211	10.4 本章小结.....	237
9.3.5 日志的保存.....	211	第 11 章 使用 Pig 进行数据处理	238
9.4 Hadoop 集群性能监控.....	211	11.1 Pig 简介.....	238
9.5 使用 YARN REST API.....	212	11.2 运行 Pig.....	240
9.6 使用供应商工具管理 Hadoop 集群.....	213	11.2.1 在 Grunt Shell 中执行.....	241
9.7 本章小结.....	214	11.2.2 执行 Pig 脚本.....	241
第 10 章 使用 Hadoop 构建数据 仓库	215	11.2.3 嵌入式 Java 程序.....	242
10.1 Apache Hive.....	215	11.3 Pig Latin.....	243
10.1.1 安装 Hive.....	216	11.3.1 Pig 脚本中的注释.....	243
10.1.2 Hive 的架构.....	217	11.3.2 Pig 语句的执行.....	243
10.1.3 元数据存储.....	217	11.3.3 Pig 命令.....	244
		11.4 UDF.....	249
		11.4.1 Mapper 中的 Eval 函数 调用.....	249
		11.4.2 Reducer 中的 Eval 函数 调用.....	250
		11.4.3 编写并使用自定义 Filter- Func.....	256

11.5	Pig 与 Hive 对比	258	14.6	本章小结	285
11.6	Crunch API	259	第 14 章 使用 HBase 构建实时系统		286
11.6.1	Crunch 与 Pig 的区别	259	14.1	HBase 是什么	286
11.6.2	Crunch 管道的例子	260	14.2	典型的 HBase 用例场景	287
11.7	本章小结	265	14.3	HBase 数据模型	288
第 12 章 HCatalog 和企业级 Hadoop		266	14.3.1	HBase 逻辑视图和客户端视图	288
12.1	HCatalog 和企业级数据仓库用户	266	14.3.2	HBase 与 RDBMS 的区别	289
12.2	HCatalog 技术背景简介	267	14.3.3	HBase 表	290
12.2.1	HCatalog 命令行接口	269	14.3.4	HBase 单元格	290
12.2.2	WebHCat	269	14.3.5	HBase 列簇	290
12.2.3	HCatalog 的 MapReduce 接口	270	14.4	HBase 命令和 API	291
12.2.4	HCatalog 的 Pig 接口	273	14.4.1	获取命令列表：帮助命令	291
12.2.5	HCatalog 通知接口	274	14.4.2	创建表：create 命令	292
12.3	HCatalog 的安全和认证机制	274	14.4.3	向表中加入行：put 命令	293
12.4	完整的解决方案	275	14.4.4	从表中检索行：get 命令	293
12.5	本章小结	275	14.4.5	读取多行：scan 命令	293
第 13 章 使用 Hadoop 分析日志		277	14.4.6	统计表中的行数：count 命令	293
13.1	日志文件分析应用	277	14.4.7	删除行：delete 命令	294
13.1.1	网络分析	277	14.4.8	清空表：truncate 命令	294
13.1.2	安全规范与法务	278	14.4.9	删除表：drop 命令	294
13.1.3	监控和报警	279	14.4.10	更换表：alter 命令	294
13.1.4	物联网	279	14.5	HBase 架构	295
13.2	分析步骤	280	14.5.1	HBase 组件	295
13.2.1	载入	280	14.5.2	HBase 中的压缩与分区	302
13.2.2	提取	280	14.5.3	压缩	303
13.2.3	可视化	281	14.6	HBase 配置概览	304
13.3	Apache Flume	281	14.7	HBase 应用程序设计	305
13.4	Netflix Suro	283	14.7.1	长表 vs 宽表 vs 窄表	305
13.5	云解决方案	285	14.7.2	行键设计	306
			14.8	使用 Java API 操作 HBase	307
			14.8.1	一切都是字节	307

14.8.2 创建 HBase 表	307	16.2.2 数据保存	337
14.8.3 使用 HBaseAdmin 类管理 HBase	308	16.3 安全性	337
14.8.4 使用 Java API 访问数据	308	16.4 云端应用模型	338
14.9 HBase 与 MapReduce 集成	312	16.5 云服务商	339
14.9.1 使用 MapReduce 任务读取 HBase 表	312	16.5.1 亚马逊网络服务 (AWS)	339
14.9.2 HBase 和 MapReduce 集群	315	16.5.2 谷歌云平台	341
14.10 本章小结	316	16.5.3 微软 Azure	342
第 15 章 Hadoop 与数据科学	317	16.5.4 选择云服务商	342
15.1 Hadoop 中的数据科学方法	318	16.6 案例学习: AWS	342
15.2 Apache Hama	318	16.6.1 EMR	343
15.2.1 整体同步并行计算模型	318	16.6.2 EC2	345
15.2.2 Hama Hello World !	319	16.7 本章小结	348
15.2.3 蒙特卡洛方法	321	第 17 章 构建 YARN 应用程序	349
15.2.4 K-Means 聚类	324	17.1 YARN: 通用分布式系统	349
15.3 Apache Spark	327	17.2 YARN: 快速浏览	351
15.3.1 弹性分布式数据集 (RDD)	327	17.3 创建 YARN 应用程序	353
15.3.2 Spark 与蒙特卡洛算法	328	17.4 DownloadService.java 类	354
15.3.3 Spark 与 KMeans 聚类	330	17.5 Client.java 类	356
15.4 RHadoop	332	17.5.1 从客户端启动应用管理器的 步骤	356
15.5 本章小结	333	17.5.2 创建 YarnClient	357
第 16 章 Hadoop 与云计算	334	17.5.3 配置应用程序	357
16.1 经济性	334	17.5.4 启动应用管理器	360
16.1.1 自有集群	335	17.5.5 监控应用	360
16.1.2 基于云平台的集群	335	17.6 ApplicationMaster.java	362
16.1.3 弹性	336	17.6.1 启动工作任务的步骤	363
16.1.4 按需付费	336	17.6.2 初始化应用管理器协议和 容器管理协议	364
16.1.5 竞价	336	17.6.3 在资源管理器中注册应用 管理器	364
16.1.6 混合集群	336	17.6.4 配置容器参数	364
16.2 后勤保障	337	17.6.5 向资源管理器请求容器	364
16.2.1 导入 / 导出	337	17.6.6 在任务节点上启动容器	364

17.6.7 等待容器结束工作任务.....	365	管理器.....	367
17.6.8 在资源管理器中注销应用 管理器.....	365	17.8 本章小结.....	367
17.7 运行应用管理器.....	367	附录 A 安装 Hadoop.....	369
17.7.1 在非托管模式中启动应用 管理器.....	367	附录 B 使用 Maven 和 Eclipse.....	378
17.7.2 在托管模式中启动应用		附录 C Apache Ambari.....	383



为什么会有大数据

随着近 20 多年来计算技术的不断革新，企业积累了大量数据。数字传感器的进步使得通信系统越来越广泛的使用，尤其是移动平台和移动终端的飞速增长；系统运行产生的大量日志以及越来越多的企业采用无纸化办公的工作方式，这些情况都使得企业积攒起了海量数据资源。并且随着人们对现代科技越来越多的依赖，数据将会以更快的速度增长下去。

摩尔定律告诉我们，大约每隔两年，计算机的性能将历史性地提升一倍。最初，计算资源的提升速度满足了飞速增长的数据的处理需求。好景不长，到 2005 年前后，数据处理需求增长的速度已经快于计算资源处理能力的提升速度。

计算机工业界想到了另外一个更加经济有效的解决办法，就是数据的并行处理。既然单台计算机无法满足大数据量的计算需求，那就用多台计算机来并行处理这些海量数据。Hadoop 就是利用互相联网的多台计算机使用 MapReduce（一种改进的单指令多数据流 [SIMD] 计算技术）来并行地处理计算大量数据。

像亚马逊、谷歌、微软这样的公司为我们提供了较为廉价的云计算服务，使得分布式并行计算的概念更加深入人心，相比于购买计算资源，我们只要花少量的费用就可以租用相应的计算资源。

本书是使用 Hadoop 平台来开发和运行软件的实用指南。Hadoop 项目起初由 Apache Software Foundation 来负责管理，目前像 Cloudera、MapR 和 Hortonworks 这样的公司也加入到维护发展它的阵营中。本章会从整体上介绍大数据的由来以及 Hadoop 项目。

1.1 什么是大数据

就本书而言，大数据（Big Data）姑且定义为无法被符合服务等级协议（service level agreement，

SLA) 的单台计算机处理或(有些时候)存储的任何数据集。后续内容会有更精确的解释。理论上讲,单台计算机可以处理任意规模的数据。对于超过单台计算机存储量的海量数据,可以存放到类似网络附属存储(network attached storage, NAS)这样的共享存储设备中,然后输入到单台计算机去计算处理。但是,这样处理数据所花费的时间往往会大大超过允许的数据处理时间。

举个简单的例子。某个业务单元每次平均需要处理 200GB 的数据,数据的读取速率是每秒钟 50MB。如果按照每秒钟 50MB 的数据读取速率来计算,顺序读取硬盘上的 100MB 数据,需要两秒钟,读取全部的 200GB 数据就大约需要一个小时了。然而这 200GB 的数据要求在 5 分钟之内处理完毕。如果这个业务处理的 200GB 数据可以被平均分布到 100 个计算节点,每个节点处理分配的数据(举一个简单的用例,假设用 SALES_YEAR>2001 这样一条简单的准则来划分数据集),去掉这 100 个计算节点的 CPU 处理时间和数据结果整合的时间,整个数据处理过程会在 1 分钟之内完成。

这个简单的例子说明,大数据是一个相对的概念,是相对于我们的业务需求而言的。



注意 Jeff Dean 博士有一篇关于并行计算的论文,详见 <http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf>。从本地磁盘顺序读取 1MB 数据需要 2 千万纳秒。从速度为 1 Gbps 的网络中读取 1MB 的数据需要 2 亿 5 千万纳秒(假定传输 2KB 的数据需要 25 万纳秒,还要返回传输确认,这样每传输 2KB 数据需要网络往返共 50 万纳秒的时间)。虽然文中的描述稍显过时,数据的传输读取绝对速度已经有了提高,但是这不妨碍我们在本章中拿来作比照,因为这些数据之间的相对值变化不大。

1.2 大数据技术背后的核心思想

上文中的例子我们作了诸多假设,要表明的核心问题是虽然我们可以很快地处理数据,但是从持久性的存储设备中读取的速度受到限制,这是整个数据处理流程上的关键瓶颈所在。相对于读写本地节点存储设备上的数据,通过网络来传输数据会更慢。

下面列出了所有大数据处理方法中的一些共同特征:

- 数据分布在多个节点(网络 I/O 速度 $\ll$ 本地磁盘 I/O 速度)。
- 计算程序离数据更近(集群上的节点),而不是相反。
- 数据的处理尽量在本地完成(网络 I/O 速度 $\ll$ 本地磁盘 I/O 速度)。
- 使用可顺序读取磁盘 I/O 代替随机读取磁盘 I/O(数据交换速度 $\ll$ 数据寻道时间)。

所有大数据计算处理模式都有一个目的,就是使输入/输出(I/O)并行化,从而提高数据处理性能。

1.2.1 把数据分发到多个节点

根据定义,大数据是无法仅靠单台计算机资源来处理的。大数据的一个特点就是使用商

用服务器。一台典型的商用服务器拥有 2TB 至 4TB 的磁盘。因为大数据是指远远超出这个容量的数据集，所以还是要将数据分布在多个节点上。

需要注意的是，我们把要处理的数据分布存放在多个计算节点，并不是因为这些数据的数据量有数十 T 之巨。你会发现大数据处理系统会有条不紊地在各个计算节点上计算处理所分配的数据。把数据分布到各个服务器节点的最终目的是为了大量计算节点同时参与到数据的处理计算过程中来。哪怕仅有 500GB 的数据集，这么大的数据量，集群中的单台计算节点完全可以存储，也会把数据分发到多台计算节点。这样的数据分发有两点好处：

- ❑ 每个数据块会在多个节点上有多份拷贝（Hadoop 默认是一个数据块有 3 份拷贝），这使得系统具备容错性，当一个节点发生故障，其他节点还备份有故障节点上的数据。
- ❑ 为了达到数据并行处理的目的，多个节点可以同时参与数据处理过程。比如，50GB 的数据被分配到 10 台计算节点，每个计算节点仅处理分配给自己的数据集，可以达到数据处理性能 5 ~ 10 倍的提升。读者可能会问，为什么不把这些数据存放到网络文件系统（NFS）中，这样每个节点可以去读取它要处理的部分。答案是本地存储磁盘的数据读取速度要远远高于通过网络来读取数据的速度。当数据分析任务启动的时候，分析任务程序运行所需的函数库被拷贝到各个计算节点，大数据处理系统会尽量在本地处理计算数据。我们会在后面章节中详谈。

1.2.2 把计算逻辑移动到数据附近

对于我们这些精通 J2EE 编程的人来说，三层架构思想深植脑海。在三层编程模型中，所有的数据会通过网络集中到一起，交由应用层来处理。我们由此形成了固有的观念，就是数据应该是分散的，而程序应该是集中的。

大数据系统无法处理网络过载的问题。传输动辄数 T 的数据量给应用层，使得网络带宽耗尽，网络拥挤不堪，导致传输效率大幅下降，甚至有可能导致系统故障。从大数据的观念来看，应该把数据分布存放在各个计算节点，程序也要移动到数据附近。要做到这一点，是一件很不容易的事情。除了程序要移动到存放数据的节点，程序运行所依赖的函数库也要移动到数据处理节点才行。如果大数据系统的集群拥有数百个计算节点，显然那将是程序维护 / 部署人员的噩梦。所以，大数据系统可以让我们集中式地部署程序代码，大数据系统后台会在计算任务启动之前把这些程序移动到各个数据处理节点。

1.2.3 计算节点进行本地数据处理

前文提到的大数据系统的两个特点，决定了分配到计算节点的数据要在计算节点本地处理。所有的大数据编程模型都是基于分布式和并行处理的。网络 I/O 比本地磁盘 I/O 慢了好几个数量级。数据被分发到各个计算节点，程序运行依赖库也移动到了数据所在节点，计算节点就地计算处理数据的条件完备了。

虽然典型的大数据处理系统都希望把数据处理过程放在拥有数据的节点本地完成，但并

不是每次都能实现。大数据系统会把计算任务尽量调度到离数据最近的节点。本章节的后续部分会介绍一些内容，其中大数据系统中某些特定的处理任务需要跨节点获取数据。分布在各个计算节点的计算结果，最终要汇聚到一个计算节点（著名的 MapReduce 框架的 Reduce 阶段或其他海量数据并行化处理编程模型的类似阶段），起码这一步是需要跨节点获取数据的。但是，在大多数的用例场景下，数据结果汇集阶段的数据量，相对于计算节点本地处理的原始数据量来说是微不足道的。此过程的网络开销可以忽略不计（但也不总是这样）。

1.2.4 优选顺序读，次之随机读

首先，你要理解数据是如何从磁盘上读取的。磁盘头首先要移动到所寻找的数据所在的磁盘位置。这个过程称为寻道（seek）操作，是需要花费很多时间的。一旦磁盘头移动到目的位置后，数据便被顺序地读取出来。这叫做数据的传输（transfer）操作。寻道时间大约需要 10 毫秒，数据传输速度量级为 20 毫秒（1MB），这意味着，如果我们从磁盘上彼此不相邻的 1MB 扇区来读取 100MB 数据，会花费 $10 \text{ (寻道时间)} \times 100 \text{ (寻道 100 次)} = 1 \text{ 秒}$ ，再加上 $20 \text{ (每兆数据传输所需时间)} \times 100 = 2 \text{ 秒}$ 。读取 100MB 的数据总共需要 3 秒钟的时间。如果我们从磁盘上彼此相邻的扇区来顺序读取 100MB 数据，将花费 $10 \text{ (寻道时间)} \times 1 \text{ (寻道 100 次)} + 20 \times 100 = 2 \text{ 秒}$ （译者注：此处应为 2.01 秒，疑为作者笔误）。读取 100MB 的数据总共需要 2.01 秒时间。

需要注意的是上文数据读取速度计算是基于 Jeff Dean 博士在 2009 年发表的文章。不得不承认，这个数字现在肯定有变化，实际上，数据的读取速度更快了。但是，所得结果间的比值没有变化，所以我们仍然可以拿来说明问题。

大多数的数据读取密集型的大数据编程模型都利用了这个特征。数据被顺序地从磁盘上读出，然后在内存中过滤数据。与此不同的是典型的关系型数据库管理系统（RDBMS）模型，它们往往是以随机读写数据为主。

1.2.5 一个例子

假设我们要计算 2000 年美国各州的总销售量，并按州排序，销售数据已经随机分发到各个计算节点。利用大数据计算技术，计算这个总量要分成如下步骤：

- 1) 每个计算节点读取分发给自己的全部数据，然后过滤掉不是 2000 年的销售数据。计算节点从本地磁盘顺序读取被随机分发到该节点上的数据。在内存中过滤掉不用的数据，而不是在磁盘上过滤，这样做可以避免磁盘的寻道时间。

- 2) 各个计算节点在处理数据时，每发现一个新的州，就为它建立一个新的分组，把销售数据加到对应的已存在的州分组中。（程序会在各个计算节点运行，分别做本地数据处理。）

- 3) 当所有的节点都完成了本地所有数据的磁盘读取工作，按照州编号分别计算了其销售总额，它们会分别把各自的计算结果发送到一个特定的计算节点（我们称之为汇聚（assembler）节点），这个汇聚节点是各个计算节点在计算任务伊始由所有节点协商出来的。

4) 这个指定的汇聚节点会从所有计算节点按照州编号汇聚全部结果, 把各个州的来自不同计算节点的数据分别相加。

5) 汇聚节点按照州把最终结果排序, 并输出排序结果。

这个过程演示了一个大数据处理系统的典型特征: 关注于使系统吞吐量(单位时间内系统处理的数据量)最大化, 而非处理结果延时(请求响应的快慢, 这个是评价事务性系统好坏的关键指标, 因为我们想尽快获得响应)。

1.3 大数据的编程模型

大数据的编程模型主要有以下几种类型:

- ❑ 大规模并行处理 (Massively Parallel Processing, MPP) 数据库系统: EMC 公司的 Greenplum 系统和 IBM 公司的 Netezza 系统就是这样的系统。
- ❑ 内存数据库系统: Oracle 公司的 Exalytics 和 SAP 公司的 HANA 正是此类系统。
- ❑ MapReduce 系统: 这样的系统包括所有大数据系统中最广泛使用的 Hadoop。
- ❑ 整体同步并行 (Bulk synchronous parallel, BSP) 系统: Apache HAMA 和 Apache Giraph 就是这样的系统。

1.3.1 大规模并行处理数据库系统

大规模并行处理 (MPP) 数据库系统的核心思想是把数据按照某一列或者某一组列的值, 按照某种形式进行划分, 以分别处理。举个例子, 上文例子中计算 2000 年的各州总销售额, 并按州排序, 我们可以按照各个州来划分数据, 这样使用固定的计算节点来处理固定州的相应数据。这样的数据分割方法使得各个计算节点都可以计算出对应州的 2000 年总销售额。

这样的系统有个明显的缺陷。你需要在算法设计的时候就决定数据如何划分。而划分的准则通常由底层的用例来决定。如此一来, 这就不适合临时的数据查询需求。某些数据的查询, 由于数据在各个计算节点的划分合理, 从而执行速度很快。而有些数据查询, 由于数据在各个计算节点的划分不合理 (与数据的访问方式不一致) 导致其执行速度异常缓慢, 为了得到计算结果, 系统需要通过网络来进行大量的数据交换。

为了解决这个缺陷, 大规模并行处理数据库系统经常采用的办法是把数据存储多份, 并按照不同的准则来进行划分。根据不同的查询需求, 选择不同的数据集。

下面列出了 MPP 编程模型的几个要点, 这些要点符合前文中给出的最初的大数据系统定义 (想一下按州排序统计计算销售总额的例子):

- ❑ 数据按州划分, 分配到不同的计算节点。
- ❑ 各个计算节点都拥有程序所需的执行库, 并对分配到该节点的数据进行数据处理。
- ❑ 每个计算节点读取本地数据。一个例外是你未考虑数据的分布情况就进行了数据查

询，这时，计算任务会通过网络从其他节点来获取数据。

- ❑ 每个任务都是顺序读取数据。所需要的所有数据都存放在磁盘上的相邻位置，并且被一次性地读取，并在内存中应用过滤条件（`year = 2000`）。

1.3.2 内存数据库系统

从系统运行的角度来看，内存数据库系统类似于 MPP 系统。它们的不同之处在于，内存数据库系统的每个计算节点拥有巨大容量的内存，并且大部分数据会被预先加载到内存中。SAP 公司的 HANA 系统就是按照这个原则来运行的。另外一些系统，比如 Oracle 公司的 Exalytics 系统，利用特殊的硬件，一个应用程序就可以管理执行多个主机。就本质来说，内存数据库系统就像是带有 SQL 接口的内存 MPP 数据库系统。

内存数据库系统的商业版本中有个重要的缺点是，其中内置了大量的硬件和软件。诚然，这些系统拥有专用设备和特定硬件，但这通常费用高昂。如果因为这些内存数据库系统准备的商用硬件来扩容内存数据库系统集群是非常方便的。举个例子，假设一个商用服务器有 25GB RAM。我们要搭建 1TB 容量的内存数据库就需要 40 台以上的主机（考虑到还有其他业务需要使用这个服务器）。1TB 也未必够用，但是我们的集群节点数已经达到了 40 个。

下面列出了内存数据库系统编程模型的几个要点特征，这些特征同样符合前文中给出的最初的大数据系统定义：

- ❑ 如前面的例子中所述，数据按州划分。各个节点把数据加载到内存中。
- ❑ 各个计算节点都拥有程序所需的执行库，并对分配到该节点的数据进行数据处理。
- ❑ 每个计算节点读取本地数据。一个例外是你未考虑数据的分配情况就进行数据查询请求，这时，计算任务会从其他节点来获取所需数据。
- ❑ 由于数据是被缓存到内存的，所以除了最初的数据加载入内存的过程外，这里不适用顺序读取数据的特性。

1.3.3 MapReduce 系统

MapReduce 编程范型是本书所要讲述内容的核心基础重点。截至目前，MapReduce 框架已被广泛的用于大数据的处理过程（four methods）。Hadoop 系统对 MapReduce 框架的实现具有如下几个重要特征：

- ❑ 使用商用级别的硬件。需要注意的是这个商用硬件的要求不是指笔记本或者台式机。尽管计算集群是商用级别，但是我们可以使用常用的硬件设备来搭建。
- ❑ 无需事先定义数据划分准则来把数据分配到各个计算节点。
- ❑ 用户仅需要定义两个独立的处理过程：Map 和 Reduce。

本书会重点深入地讲解 MapReduce 框架。总体上说，MapReduce 框架需要用户定义一个 Map 处理和 Reduce 处理。当 Hadoop 系统实现 MapReduce 时，数据常常按照 64 ~ 128MB 数据块大小进行分发，每个数据块会被复制两次（Hadoop 系统数据备份默认参数为 3）。在计算

每个州的 2000 年的总销售额，并按照州排序的例子中，所有的销售数据都会分割为大量的数据块（大小 64 ~ 128MB），加载到 Hadoop 分布式文件系统（Hadoop Distributed File System, HDFS）。MapReduce 程序启动的时候，Hadoop 系统会把程序运行依赖库（包括用户自定义 Map 处理和 Reduce 处理）拷贝到各个计算节点。

各个计算节点按照调度执行 Map 任务，该 Map 任务会读取包含销售数据的数据块。每个 Mapper（各自节点上的）会逐条读取数据块中的数据记录（Record），并过滤掉无用数据，仅保留 2000 年的数据。对应所输入的数据记录，Mapper 会输出一条包含键 / 值对（key/value pair）的数据记录。如果读入的销售数据是 2000 年的，那么其键（key）就是州，其值（value）就是从读入数据记录中获取的销售数字。

最后，Reducer 会从每个 Mapper 的输出中获取那些键 / 值对，运行 Reducer 的数量是可以配置的。各个键会被发送到特定 Reducer 进行处理，确保相同的键仅被相同的 Reducer 处理。各个 Reducer 把接受的键 / 值对中的销售数字相加。Reducer 接受的数据格式是键（州）和这个键对应的销售数字列表（销售记录是 2000 年的）。输出结果写回 HDFS 中。客户端从 HDFS 上读取结果数据后，对结果按州排序。最后这个步骤也可以交给 Reducer 完成，因为 Reducer 在接受它要处理的键时，这些键是已经排序完成的。本例中，为了达到按州排序的目的，我们需要把 Reducer 的运行数量严格限制为 1。可是由于 Mapper 和 Reducer 之间传输数据会引起网络 I/O，这样可能会导致网络拥塞。本书的后面部分会详细地讨论这个问题。

下面列出了 MapReduce 编程模型的几个要点特征，这些特征同样符合前文中给出的大数据系统定义：

- ❑ 数据以较大的数据块的形式存放在 HDFS 上。HDFS 是一个分布式文件系统，数据块分散存储到各个节点，数据块是有冗余的。
- ❑ 程序运行依赖库，包括 Map 和 Reduce 代码被复制发送到所有的任务节点。
- ❑ 每个计算节点仅读取节点本地数据。集群中所有节点运行 Mapper，从节点本地读取数据到 Mapper 中（大多数情况下，哪个节点的 Mapper 读取哪个节点磁盘的数据块，这是由调度程序管理决定的。调度程序可能会分配某个节点的 Mapper 任务来处理远程节点的数据块，以保持集群中的所有节点负载均衡）。
- ❑ 数据被每个节点的任务以数据块的方式一次性顺序读取（数据块大小一般为 64 ~ 128MB）。

MapReduce 编程范型的一个重要不足是它不适合迭代算法。大量的数据科学计算算法很自然就要使用迭代，并最终收敛于一个解。当我们使用这样的算法的时候，MapReduce 编程范型需要我们把每个迭代过程放到相互独立的 MapReduce 任务中去。每次迭代产生的数据输出作为下次迭代计算的数据输入。但是，由于 MapReduce 任务每次都要从持久性存储中重新读取数据，所以每次迭代产生的结果需要存到持久性存储中供下次迭代计算使用。这个过程导致了不必要的 I/O 操作，并对系统吞吐量造成重大影响。这样的问题在下面即将讲到的关于 BSP 类型系统的讲解中，也有论述。

1.3.4 整体同步并行系统

整体同步并行（BSP）系统的运行过程跟 MapReduce 过程非常相似。与 MapReduce 程序在它的处理循环结束后即可终止不同的是，BSP 系统程序执行由一系列的超步（processes）（这个与 Map 处理的处理过程类似）组成，这些超步保持栅栏同步（synchronize on a barrier），向主节点发送数据并进行相关的信息交换（exchange relevant information）。每当一次迭代执行完毕，主节点会通知每个数据处理节点进行下一次迭代。

间隔通信是在并行计算处理中经常提到的概念。间隔通信指的是许多线程在分别执行各自的任務，这些线程在运行之前需要协商出一个检查点。这种模式是非常必要的，所有的处理线程在到达那个检查点之前就要决定是继续执行剩下的计算任务还是终止它们（并行的或者顺序的），以便所有线程确认何时完成数据处理任务。间隔同步（Synchronizing on a barrier）的方法在我们的日常生活中就经常使用。例如，一起拼车的伙伴们会常常商定在某一个特定的地方等车。整个等待过程的长短，取决于那个最晚到达该指定等车位置的人何时到达。

在 BSP 方法执行过程中，允许每个类似 Map 的处理过程缓存上次迭代的结果，如此即可大幅提高整个数据处理过程的吞吐量。我们会在本书第 15 章中讲解 BSP 系统。涉及了一些相关的迭代算法。

1.4 大数据和事务性系统

以大数据作为参照来理解事务概念的发展变化是非常重要的。这个问题的讨论会涉及相应的 NoSQL 数据库。Hadoop 系统使用 HBase 来作为自己的 NoSQL 数据存储。你也可以使用 Cassandra 或者云计算提供商（如 Amazon Dynamo）的 NoSQL 系统来替代。

大多数的 RDBMS 使用者都要求数据库必须要遵守 ACID 准则，但是遵守这些准则是有系统代价的。当数据库后台需要处理峰值为每秒数百万次的事务操作的时候，要求苛刻地遵守 ACID 准则对数据库来说是个巨大的挑战。



注意 ACID 是 atomicity（原子性）、consistency（一致性）、isolation（隔离性）和 durability（持久性）的首字母简写。进一步详细讲解见：<http://en.wikipedia.org/wiki/ACID>。

对苛刻的 ACID 准则做出妥协是必要的。做出妥协的理论依据就是著名的 CAP 理论（又称 Brewer 理论）。CAP 理论是下面三点的首字母缩写：

- ☐ Consistency（一致性）：在分布式系统中的所有数据备份，在同一时刻有同样的值。
- ☐ Availability（可用性）：在合理且明确的时间范围内，保证每个请求都能获得成功或者失败的结果的响应。
- ☐ Partition tolerance（分区容忍性）：在集群中一部分节点故障后，集群整体仍可使用。

这个理论用来证明，任何分布式系统只能同时满足其中的两个特性，而无法三者兼顾。现在，让我们仔细地思考对比下面列出的各种系统：

- ❑ **一致性和可用性：**遵守 ACID 准则的单机 RDBMS 是兼顾一致性和可用性的系统的一个例子。但是不满足分区容忍性。如果这个 RDBMS 当机了，用户将无法访问数据。
- ❑ **一致性和分区容忍性：**一个 RDBMS 集群就是这样的系统。分布式事务保证了所有用户在同一时刻获取到相同的数据（一致性）；数据的分布式存储天然地保证在集群部分节点故障的情况下，集群系统仍然可用。可是，分布式事务在满足系统一致性的同时，也有副作用，导致系统无法做到可用性，在两阶段提交事务执行期间，系统是不可用的。一致性限制了系统可以支持的同时执行的事务数量，反过来也影响到了系统的可用性。
- ❑ **可用性和分区容忍性：**本类别中这样的系统常常被归类为“最终一致性”系统。考虑一下流行的电子商务网站，比如亚马逊。假如你正在浏览商品目录，发现你需要的商品正好有货。在购买流程中很自然就可以想到，在你发现所需的商品有货并发起了购买请求这段时间内，其他人更早地进入并抢购了商品。所以有必要及时显示最新库存变化。库存变化将会广播到为用户提供服务的集群中的其他所有节点。为了给用户提供库存变化后的最新值，当商品库存变化后的数据在各个节点的传递还没有完成的时候，如果系统阻止了用户的访问，就会使网站丧失可用性，从而影响交易量。为了系统可用性和分区容忍性而牺牲系统一致性，我们选择这个折中来使多个节点为用户显示相同的数据（在一个小的时间窗内，用户看到的数据可能是不同的，这个时间窗的大小取决于为用户提供服务的那个节点）。

如何做出折中选择，在大数据系统设计的时候是非常关键的。本书的主题 MapReduce 只是大数据生态环境中的一个组件。它常常与其他类似 HBase 这样的产品搭配使用，在选择不同产品配合使用的时候，在本章讲解的这些权衡中做出选择，是解决方案是否可行的关键。

1.5 我们能处理多大的数据量

本章前述例子中，我们作了若干假设。比如，我们忽略了 CPU 时间。对于大多数的商业程序来说，计算的复杂性并不大。但是，随着计算量的提高，从实现的角度来看，各种情况的资源消耗都要考虑。举个例子，在数据挖掘中，会用到复杂的贝叶斯统计算法。这样的情况是计算密集型的应用。针对这样的问题，我们可以增加集群节点数量来提高性能，或者选用其他算法替代。



注意 类似 MapReduce 这样的大数据计算编程范型可以被扩展到其他大数据计算技术中使用。比如，利用计算机的图形编程单元来进行计算机通用计算的技术（GPGPU）可以实现计算密集型应用程序的大规模并行计算。

我们还忽略了网络 I/O 开销。拥有 50 个节点的计算集群需要使用分布式文件系统，为了整合数据，这 50 台计算节点间的数据通信也是有开销的。在所有的大数据解决方案中，I/O 开销是重中之重。在整个大数据处理流程过程中，这些开销消耗会导致串行依赖（Serial dependency）。

1.5.1 一个计算密集型的例子

假设 50 个计算节点来处理 200GB 的数据，平均每个计算节点需要本地处理 4GB 的数据。每个节点读取这些数据要花费 80 秒（速率为每秒 50MB）。无论我们的计算多快，这 80 秒是不能节省的。假设数据处理之后最终的结果数据集大小为 200MB，每个计算节点平均产生 4MB 的计算结果。这些计算结果在带宽 1Gbps（一个数据包大小为 1M）的网络中传输并汇聚到一个节点以便展示结果。传输这些数据到目的节点需要花费 3 毫秒的时间（网络中传输 1MB 的数据需要 250 微秒，每个数据包的网络传输延时为 500 微秒，按照 Jeff Dean 博士之前发表的文章）。忽略计算耗时，整个数据计算流程耗时不低于 40.003 秒。

如果集群拥有 4000 个计算节点，平均每个计算节点需要本地读取处理 500MB 的数据，每个计算节点产生的结果数据量为 0.1M。读取 50MB 的数据块不会少于 1 秒钟。这意味着，4000 台计算节点的集群达到了数据处理速度的极限。换句话说，对于一些特定类型的数据处理问题，如果我们花 4000 个小时才能完成它，那么无论我们把集群节点数量增加多少，都不会把数据处理完成时间缩短到 1 小时内。4000 台计算节点看起来很多了，但是这也是我们所能达到的速度上限了。在这个简单的例子中，我们做了很多使系统简化的假设。我们同样假设在程序逻辑中没有串行依赖（Serial dependencies），这样的假设往往是不成立的。一旦我们考虑了这些开销，系统的性能表现还可能会大幅下滑。

串行依赖（Serial dependencies）这个所有并行计算算法都存在的问题，严重限制了系统性能的提升。这样的限制已经为众人熟知，并表述为著名的 Amdhal 定律。

1.5.2 Amdhal 定律

就像光速被认为是人类在这世界上能达到的理论上的速度极值，Amdhal 定律揭示了通过往集群增加更多计算节点的方法来提高集群性能的极限值。



注意 Amdhal 定律的完整描述详见：http://en.wikipedia.org/wiki/Amdahl's_law。

这个定律可以简单描述为，假如一个给定的解决方案，其数据计算处理并行化程度的比例达到 P （ P 的取值范围是 0 到 1），在集群计算节点数量无限的条件下（或者说集群中有海量的计算节点），我们能获得的性能提升最大为 $1/(1-P)$ 。这样，如果数据计算处理并行化程度的比例达到了 99%，系统性能会提升 100 倍。所有的程序或多或少都会有串行依赖，再加上磁盘 I/O 和网络 I/O 的时间消耗。无论采用什么样的程序计算算法，我们能获得的系统性能提升都无法突破那个限制。

1.6 大数据商业用例

在商业界，大数据和 Hadoop 都有很多应用案例。再从以下三个方面总结大数据的特征：

- ❑ 数据量 (Volume)
- ❑ 处理速度 (Velocity)
- ❑ 数据多样性 (Variety)

数据量是指系统能够处理的数据量的大小。如果你们部门每天晚上都要在两个小时内获取、加载并传输 2TB 的数据，你就会面对数据量的挑战。

处理速度是指当大量数据来到时，系统的处理数据能力。像 Facebook 和 Twitter 这样的公司就碰到了数据处理速度的问题。这些公司每秒钟都会收到海量的短信息 (tiny messages)，这些数据需要被立即处理，把它们发布到社交站点，推送到相关用户 (家庭成员、朋友或者追随者)，触发各种事件等。

数据多样性是指系统处理的数据格式类型是不停增长的。企业检索系统已经成为各企业的标配了。类似 Apache Solr 这样的开源软件使基于搜索的系统越来越普遍。绝大多数非结构化的数据并非孤立存在的，有大量的结构化的数据与其相关联。我们拿一个简单的电子邮件文档来举个例子。电子邮件有很多元数据与其关联，比如发送者、接收者、接收者的顺序、发送 / 接收时间、发送者 / 接收者的企业信息 (比如，发送时的头衔)，等等。

这些信息是动态变化的。比如，你要分析多年的电子邮件 (一些法律行业的相关部门会经常有此工作内容)，当发送者或者接收者第一次进行电子邮件交流的时候，搞清楚他们当时使用的头衔是很重要的。数据的这种动态变化的特性是普遍存在的，而且也是处理的难点。

利用商用软件和硬件，大数据系统能帮助我们每天的工作，比如大数据量的抽取、传输、加载 (ETL)。需要强调的是，开源 Hadoop 系统，这个运行在多个商用服务器上，并可通过添加更多的计算节点来扩容集群的系统，可以使得 ETL (或者是 ELT，这个是大数据领域的通用叫法) 性能在合理的代价上获得大幅提升。

为了支持更快的数据处理速度和更加多样的数据类型，围绕着 Hadoop 系统和 HDFS 系统发展出了多个开源的大数据处理系统。新的数据格式要求我们在处理大量数据的时候对系统 I/O 性能有更好的管理。本书将会讲解这样的开发需求存在的目的和相关用例。

Storm (由 Twitter 公司发起) 和 Apache Flume (用来分析海量日志数据信息) 是专注于数据处理速度的系统。如何选择这样的系统，取决于你需要的数据处理的“实时”程度。Storm 处理数据的实时性会比 Flume 更好。

要明确的是，大数据是一个由多个系统组成的生态系统，这些系统都是用来处理各种复杂的业务问题。Hadoop 系统是处理这些问题的解决方案的核心。理解了 Hadoop 系统，你就能更深刻地理解大数据生态圈中其他大数据系统的使用。

1.7 本章小结

大数据渐成主流，其背后的两大推动力就是开源的 Hadoop 系统和云计算时代的到来。两者的发展使得以较低成本引入大规模大数据处理方法来解决业务问题成为可能。Hadoop 系统是所有大数据解决方案的核心。尽管其他如 MPP 或者 BSP 这样的编程模型，可处理一些特殊的问题，但是它们都还是要以某种形式依赖于 Hadoop 系统，尤其在数据量达到数 T 之巨时。掌握 Hadoop 系统，会让开发者更加高效地使用其他编程模型。本书的目的也正在于此。

下面的章节会讲解 Hadoop 系统的使用，以及使用 Hadoop 系统来解决问题的方法与技巧。



Hadoop 中的概念

价格实惠的普通计算机经常满足不了应用程序的计算资源需求，很多企业的业务应用程序已经不再适合在单台廉价的计算机上运行。这时，一个简单昂贵的解决方案就是购买一些具有多 CPU 的高端服务器，这通常需要巨额资金。只要能买到最高端的服务器，这个解决方案就能够达到理想的效果，但预算往往是个大问题。另一个替代方案，就是搭建一个高可用的集群，这个集群经过专业的安装和精心的管理服务，使用起来就像一台计算机一样。很多高可用的集群都是企业专有的而且价格也十分贵。

为了获取所需的计算资源，一个更经济实惠的解决方案是云计算。面对大数据量的处理需求，一个常用的方法就是把它们分割成互不依赖的小份数据来分别计算处理，这就是所谓的单指令多数据流（single-instruction, multiple-data, SIMD）的数据计算模式。Hadoop 为这样的云计算需求提供了一套开源的计算框架和一套分布式文件系统。

本书是一本利用 Hadoop 进行开发工作的实用指南。Hadoop 项目由 Apache Software Foundation 来负责管理。本章会为大家讲解 Hadoop 的核心概念思想。为了后续章节更有效地学习，你需要安装并运行 Hadoop。

2.1 Hadoop 简介

Hadoop 是以谷歌在 2004 年发表的一篇关于 MapReduce 的论文为基础开发的。Hadoop 项目开始于 2005 年，那个时候，Hadoop 项目是开源的网页搜索引擎 Nutch 项目下的一个子项目。后来，Hadoop 从 Nutch 项目中分离出来，并最终成为了 Apache 基金会的顶级项目。

现在，MapReduce 已经成为业内家喻户晓的 Hadoop 框架。同时，围绕 Hadoop 项目诞生

了好几家公司来提供 Hadoop 系统的支持、咨询和培训。

就其自身来讲，Hadoop 是一个基于 Java 语言的 MapReduce 框架。随着 Hadoop 平台被越来越多的企业采用，Hadoop 需要加入对非 Java 语言的支持。下面列出了 Hadoop 的改进及其子项目，这些都促使 Hadoop 在企业中的应用越来越广：

- ❑ Hadoop Streaming——任何命令行脚本都可以通过 Streaming 调用 MapReduce 框架。UNIX 脚本程序员、Python 程序员等都可以使用 MapReduce 来开发一些临时任务。
- ❑ Hadoop Hive——使用 MapReduce 平台的用户发现，开发一个 MapReduce 程序往往需要大量的编码工作，而且易于出错且难于测试。这就需要一个更有效的类似 SQL 的语言，这样一来，用户就可以关注于业务问题而不必关心类似 SQL 命令功能的底层实现（比如 WHERE 子句、GROUP BY 子句、JOIN 子句等）。Apache Hive 可以把海量数据集放入数据仓库中，用户可以编写类似 SQL 语句的 Hive 查询语句来查找数据。Hive 引擎把 Hive 查询语句透明地转换为底层 MapReduce 任务来执行。高级用户可以使用 Java 语言来编写用户自定义函数（UDF）。Hive 也支持标准的 ODBC、JDBC 数据库驱动。Hive 还可以用来开发商业智能（Business Intelligence, BI）分析程序去处理分析存放在 Hadoop 中的数据。
- ❑ Hadoop Pig——使用 Pig 的目的与使用 Hive 的目的是一样的，但是 Hive 是一个类 SQL 语言，属于陈述性的语言；而 Pig 是一种过程性语言，非常适合数据管道应用场景。Pig 对数据处理管道程序的程序员非常有吸引力（比如，SAS 程序员）。它也非常适合开发抽取、加载和传输（ETL）类型的程序。
- ❑ Hadoop HBase——所有之前介绍的项目，包括 MapReduce 程序，都是批处理程序。现在有个强烈的需求就是在 Hadoop 中实时查询数据。Hadoop 并不是一个原生的键/值存储系统。拿类似 Facebook 的社交网站做例子，如果你想查看一个用户的个人资料，你希望能够立即得到结果（而不是经过一个长时间的批处理任务执行）。这样的场景用例就是开发 HBase 平台的目的所在。

我们刚刚粗略地介绍了 Hadoop 的由来和它的子项目。前面的例子也说明了 Hadoop 系统是因何发展而来。Hadoop 系统最早是作为用于索引海量文本数据的 MapReduce 引擎而启动的项目。后来慢慢地发展成为一个通用的大数据处理模型。这个大数据模型可以支持标准的企业级用例，比如 DW、BI、ELT 或者实时缓存查询。MapReduce 是一个非常实用的编程模型，它在前文刚刚提到的标准企业级用例（ETL、DW）中得到了广泛应用，使其在大数据计算处理领域成为主流。同样重要的是，现在各个企业都在积极应对海量数据处理带来的挑战。

在很长的一段时间内，Hadoop 系统只能做到用户提交任务后，以先进先出（FIFO）的模式在集群上执行任务。这样的话，一些运行耗时、重要性低的任务会占用大量的集群资源，而运行时间短、重要性高的任务迟迟无法运行。为了解决这样的问题，Hadoop 中出现了更加复杂的任务调度器，比如公平调度器（Fair Scheduler）和计算能力调度器（Capacity Scheduler）。但是在 Hadoop 1.x 系列（版本号小于 0.23）中，一些系统本身设计之初的决策导致了集群在

可扩展性方面的限制。

Yahoo 公司的工程师们发现，当 Hadoop 系统集群的计算节点数量达到数千台量级的时候，集群就会存在一些可扩展性问题（详细的情况请参见链接：<http://developer.yahoo.com/blogs/hadoop/scaling-hadoop-4000-nodes-yahoo-410.html>）。当人们对 Hadoop 系统的这些可扩展性问题越来越清晰之后，Hadoop 的工程师们讨论思考，并重新评估了 Hadoop 系统原型设计中的一些基础假设。最终，Hadoop 系统平台的核心架构被重新设计。由此，Hadoop 2.x（版本号大于 0.23 的 Hadoop 系统）诞生了。

本书在全面介绍 Hadoop 2.x 的同时，也会适当地提到 Hadoop 1.x。这样你会深刻地理解 Hadoop 2.x 发生这些变化的原因。

2.2 MapReduce 编程模型简介

Hadoop 系统支持 MapReduce 编程模型，这个编程模型由谷歌公司发明，该模型可以利用由大量商用服务器构成的大规模集群来解决处理千兆级数据量的问题。MapReduce 模型有两个彼此独立的步骤，这两个步骤都是可以配置并需要用户在程序中自定义：

- ❑ Map：数据初始读取和转换步骤，在这个步骤中，每个独立的输入数据记录都进行并行处理。
- ❑ Reduce：一个数据整合或者加和的步骤。在这个步骤中，相关联的所有数据记录要放在一个计算节点来处理。

Hadoop 系统中的 MapReduce 核心思路是，将输入的数据在逻辑上分割成多个数据块，每个逻辑数据块被 Map 任务单独地处理。数据块处理后所得结果会被划分到不同的数据集，且将数据集排序完成。每个经过排序的数据集传输到 Reduce 任务进行处理。图 2-1 展示了 MapReduce 模型是如何工作的。

一个 Map 任务可以在集群的任何计算节点上运行，多个 Map 任务可以并行地运行在集群上。Map 任务的主要作用就是把输入的数据记录（input records）转换为一个个的键值对。所有 Map 任务的输出数据都会进行分区，并且将每个分区的数据排序。每个分区对应一个 Reduce 任务。每个分区内已排好序的键和与该键对应的值会由一个 Reduce 任务处理。有多个 Reduce 任务在集群上并行地运行。

一般情况下，应用程序开发者根据 Hadoop 系统的框架要求，仅需要关注以下四个类：一个类是用来读取输入的数据记录，并将每条数据记录转换成一个键值对；一个 Mapper 类；一个 Reducer 类；一个类是将 Reduce 方法输出的键值对转换成输出记录进行输出。

让我们来使用 MapReduce 编程模型的“Hello-World”程序：计数程序（the word-count application），来示例讲解 MapReduce 的编程思想。

假设你有海量文本文档。大家现在分析非结构化数据的兴趣越来越浓，这样的情况是比较常见的，这些文本文档可以从维基百科下载页面下载获得，详见页面 <http://dumps.wikimedia.org>，

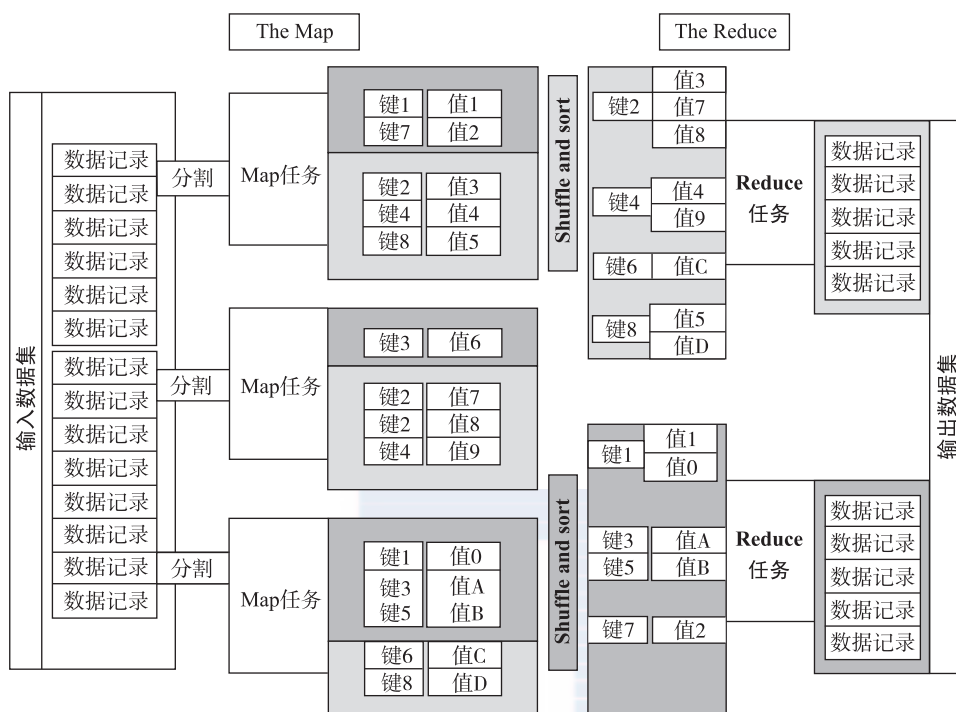


图 2-1 MapReduce 模型

也可以是用于法律分析的一个大型组织的邮件数据存档（例如，安然公司电子邮件数据集：<http://www.cs.cmu.edu/~enron/>）。利用这些文本数据，可以做很多有意思的分析（比如，信息提取，基于文本内容的文档聚类，基于语义的文档分类）。大多数的文本分析最开始要做的就是统计文档语料库中每个单词的数量（文档的集合通常被称为一个语料库，corpus）。另一个用途是计算一个词 / 文档对应的“词频 / 逆向文件频率”（TF/IDF）。



注意 关于 TF/IDF 的讲解和相关程序示例，详见链接：<http://en.wikipedia.org/wiki/Tf-idf>。

从直觉上来说，计算文档中每个词出现的个数并不是一件难事。我们简单地假定文档中的每个词是使用空格分隔的。一个清晰明了的解决方案是：

1) 维护一个哈希表（hashmap），该哈希表的键为文本中的每个词，该键对应的值是该词的个数。

2) 把每篇文档加载到内存。

3) 把文档分隔成一个个的词。

4) 对于文档中的每个词，更新其在哈希表中的计数。

5) 当所有的文档都处理完成，我们就得到了所有单词的计数。

对于大多数的语料库来说，每个不同的单词的计数不过几百万，所以上面介绍的解决方

案在逻辑上是可行的。可是这个解决方案的最大隐患是处理的数据量有限（毕竟，我们这本书是关于大数据的）。当语料库的数据量增加至 T 量级，按照这个方案，一个计算节点需要花费数小时，甚至数天才能完成整个数据的处理过程。

现在，当数据量非常大的时候，我们可以尝试使用 MapReduce 来解决这个计数问题。要记住，这个计数问题将是你们经常碰到的使用场景，解决这个问题是很简单的但却不能使用单台计算机来做，应该使用 MapReduce。

按照上面的解决方案，我们使用 MapReduce 来实现，步骤如下：

1) 有一个多台服务器组成的集群供我们使用。我们假设该集群的计算节点数量为 50，这是生产环境中的典型场景。

2) 每台服务器上都会运行大量的 Map 处理。一个合理的假设是有多少个文件正在被处理，就有多少个 Map 处理。后面的章节会发现，这个假设并不严格（当我们在讲解压缩模式下的文件或者其他格式的序列文件时，该假设并不成立）。但是，现在我们先认为这个假设是成立的。假设现在有一千万个文件，这样会有一千万个 Map 处理这些文件。在给定的时间内，我们假设有多少个 CPU 核，就有多少个 Map 处理可以同时运行。集群的服务器是 8 核的 CPU，所以 8 个 Mapper 可以同时运行，这样一来，每台服务器负责运行 20 万个 Map 处理。整个数据处理过程中，每个计算节点会同时运行 8 个 Mapper，共 25 000 次迭代（每次迭代过程可以运行 8 个 Mapper，一个 CPU 运行一个 Mapper）。

3) 每个 Mapper 处理一个文件，抽取文件中的单词，并输出如下形式的键值对：<{WORD}, 1>。Mapper 的输出如下：

- ❑ <the,1>
- ❑ <the,1>
- ❑ <test,1>

4) 假设我们只有一个 Reducer。这个假设并不是必须的，这只是默认的设置。默认的设置值在实际的应用场景中是常常需要改变的，在本例中，这个默认的设置值是合适的。

5) Reducer 会接受如下格式的键值对：<{WORD}, [1,...1]>。换句话说，Reducer 任务接受的键值对的构成是，其键为任意一个 Mapper 输出的单词（<WORD>），其值为任意一个 Mapper 输出的与键对应的一组值 ([1,...1])。Reducer 的输入键值对示例如下所示：

- ❑ <the,[1,1,1,...,1]>
- ❑ <test,[1,1]>

6) Reducer 每处理一个相同的单词，就简单地将该单词的计数加 1，最终得到了 <单词> 的总数；然后将结果按照以下键值对格式输出：<{WORD}, { 单词总数 }>。Reducer 任务的输出键值对示例如下所示：

- ❑ <the,1000101>
- ❑ <test,2>

从一个键对应一个值变换成了在 Reduce 阶段接受的一个键对应一组值，这个过程

在 MapReduce 中称为排序 / 混洗 (sort/shuffle) 阶段。由 Mapper 任务输出的所有键值对在 Reducer 任务中都按键排序了。如果配置了多个 Reducer, 那么每个 Reducer 将会处理键值对集合中的某个子集。键值对在由某个 Reducer 处理之前, 就已经按键排序完毕, 这确保了相同的键对应的值会由同一个 Reducer 接受并处理。



注意 在 Reducer 阶段, 在 Reduce 任务执行之前, 并不是真正地为一个要处理的键创建一个与该键对应的值的列表。否则, 对于英语中典型的连接词, 会造成大量的内存占用。假设在我们的例子中, 1000 万个文本文档, 每个文本文档中存在 the 单词 20 次。这样一来, 对于 the 这个键, 其对应的值列表将会是含有 2 亿个 1 的列表。这会轻易地占满运行 Reducer 的 Java 虚拟机 (JVM) 的内存。在 sort/shuffle 阶段, 键 the 对应的所有的值 1 会被一起传送到相应的运行 Reducer 的计算节点的本地文件系统中。当 Reducer 开始处理键 the 的时候, 该键对应的值 1 会通过 Java 迭代接口流式地读入。

图 2-2 展示了上述数据处理方案的逻辑过程。

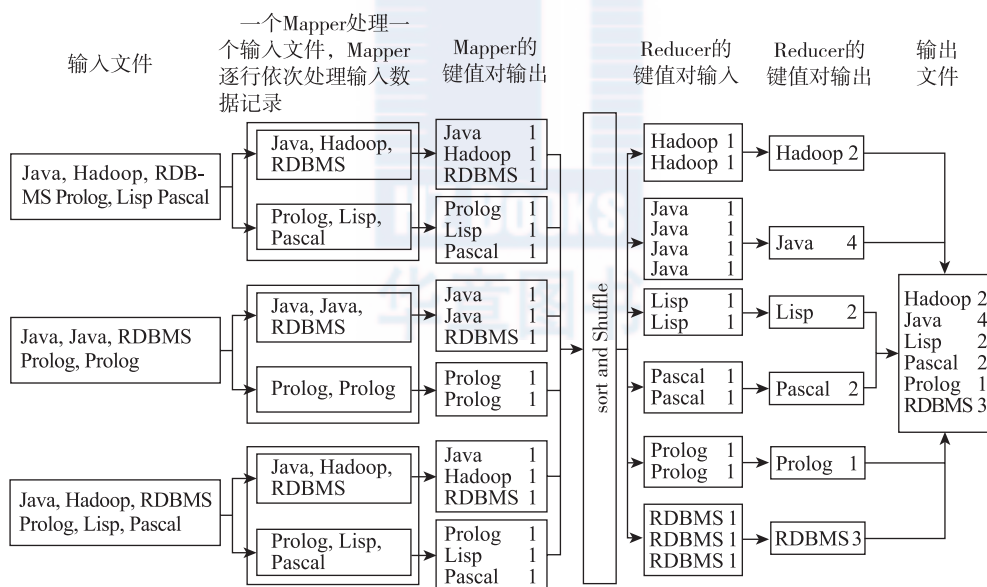


图 2-2 MapReduce 计数程序

讲到这里, 你可能想知道每个 Mapper 是如何访问文件的。这些文件存储在哪里? 每个 Mapper 是从网络文件系统 (NFS) 来获取这些文件的吗? 显然不是。回想一下第 1 章中我们讲到的, 从网络中读取数据的速度要比从本地文件系统中读取慢一个数量级。所以, Hadoop 系统从系统结构设计上就确保了大多数的 Mapper 都从本地磁盘读取文件。这意味着, 我们例子中的整个语料库的所有文档都分布存储在 50 个计算节点中。尽管 HDFS 的总体设计使得文

件对网络传输交换是敏感的，以此来确保计算被调度到文件所在的位置来本地执行，但是从 MapReduce 系统来看这是一个统一一致的文件系统。这个文件系统就是著名的 Hadoop 分布式文件系统 (HDFS)。后面的章节我们会更细致地讲解 HDFS 系统。

2.3 Hadoop 系统的组成

本节内容会细致深入地讲解 Hadoop 系统的各个组成部分。我们先介绍构成 Hadoop 1.x 系统的组件，再介绍构成 Hadoop 2.x 系统的新组件。宏观上说，Hadoop 1.x 系统有以下几个守护进程：

- ❑ 名称节点 (NameNode)：维护着存储在 HDFS 上的所有文件的元数据信息。这些元数据信息包括组成文件的数据块信息，及这些数据块在数据节点上的位置。正如你将看到的，这个组件正是使用 Hadoop 1.x 搭建大型计算集群系统的瓶颈所在。
- ❑ 辅助名称节点 (Secondary NameNode)：这不是名称节点的热备份。实际上，它作为 Hadoop 平台的一个组件，其命名不是很恰当。它为名称节点组件执行一些内务处理功能 (housekeeping functions)。
- ❑ 数据节点 (DataNode)：把真正的数据块存放在本地硬盘上，这些数据块组成了保存在 HDFS 上的每个文件。
- ❑ 作业跟踪器 (JobTracker)：这是 Hadoop 系统的主要组件之一，它负责一个任务的整个执行过程。它的具体功能包括：调度各个子任务 (Mapper 任务和 Reducer 任务各自的子任务) 到各自的计算节点运行，时刻监控任务运行和计算节点的健康状况，对失败的子任务重新调度执行。正如我们即将演示的，就像名称节点一样，这个组件是使用 Hadoop 1.x 搭建大型计算集群系统的瓶颈所在。
- ❑ 任务跟踪器 (TaskTracker)：运行在各个数据节点上，用来启动和管理各个 Map/Reduce 任务。与作业跟踪器进行通信。

Hadoop 1.x 集群有两种类型的节点：主节点 (master nodes) 和从节点 (slave nodes)。

主节点负责执行如下几个守护进程：

- ❑ 名称节点进程
- ❑ 辅助名称节点进程
- ❑ 作业跟踪器进程

从节点分布在整个集群上并执行如下几个守护进程：

- ❑ 数据节点进程
- ❑ 任务跟踪器进程

在整个集群上，虽然每种主节点守护进程只有一个运行实例，但是其数据节点进程和任务跟踪器进程有多个运行实例。在一个规模较小的或者用于开发 / 测试的集群上，三个主节点进程全部运行在一台服务器上。对于生产环境系统或者规模较大的集群，三个主节点进程分

别运行在不同的服务器上明智的选择。

2.3.1 Hadoop 分布式文件系统

Hadoop 分布式文件系统 (HDFS) 用于支持数据处理程序要处理的大文件。这样的程序在处理数据的时候，有一次写、多次读的特点。

HDFS 文件系统由以下几个守护进程协调地运行来提供服务：

- ❑ 名称节点进程
- ❑ 辅助名称节点进程
- ❑ 数据节点进程

HDFS 系统也是主从架构的。运行名称节点进程的服务器为主节点，运行数据节点进程的服务器为从节点。一般情况下，集群中的每个从节点都会运行一个数据节点进程。这个数据节点进程管理着挂载到这个数据节点上的存储设备。HDFS 系统提供一个统一的文件系统命名空间，用户就像使用一个文件系统一样来存取集群节点上的数据。名称节点进程负责管理这些存储在集群上的文件的元数据。

1. Hadoop 文件的本质是块存储

首先，你应该理解集群中文件的物理存储方式。在 Hadoop 系统中，每个文件被分隔为多个数据块。一个典型的数据块大小为 64MB，也可以将其配置为 32MB 或者 128MB。在 HDFS 中，文件块的大小是按照文件的大小来配置的。如果一个文件的大小不是一个文件块大小的整数倍，空间也不会浪费很多，因为只是最后一个文件块未被完全占满。一个大文件会被分隔为多个数据块来存储。

每个数据块存储在数据节点上。为了防止节点故障，这些数据块是有备份的。在 Hadoop 系统中，备份数量默认配置为 3。具备机架感知功能的 Hadoop 系统把文件的一个数据块存放在本地机架上的一台计算节点上（假设 Hadoop 客户端运行在本地机架中的一台数据节点上，否则，Hadoop 客户端会随机选择一个机架）。这个数据块的第二个备份会被存放在另外一个远程机架上的计算节点中。这个数据块的第三个备份会被存放在第二个数据块备份机架上的另一台计算节点中。Hadoop 系统借助一个单独配置的网络拓扑文件实现机架感知功能，这个网络拓扑文件配置了机架到计算节点的域名系统（DNS）名称之间的映射，该网络拓扑文件的路径配置在 Hadoop 配置文件中。



注意 许多 Hadoop 系统会把文件备份因子降为 2。运行在 EMC 公司的 Isilon 硬件之上的 Hadoop 系统就是这样一个例子。这样做的原因是这套硬件系统使用了 RAID 5 的技术，这个技术本身就内置了数据的冗余备份，从而可以降低 Hadoop 系统的文件备份因子。降低文件备份因子的一个很明显的好处就是提高了系统 I/O 性能（少写一个备份）。这个链接中的白皮书讲解了这样的系统架构设计：<http://www.emc.com/collateral/software/white-papers/h10528-wp-hadoop-on-isilon.pdf>。

为什么不把三个文件数据块的备份分别存放到不同的机架上呢？毕竟，这样做可以增加数据冗余度。这样做的话，抵御机架故障的能力更强，而且还可以增加机架的数据吞吐量。但是，机架故障的概率远低于计算节点故障的概率，而且把数据块备份到不同的机架可导致 Hadoop 系统写性能的大幅下降。所以，一个折中的方案就是把两个数据块的备份存放在同一个远程机架上的不同计算机点中，以此换取 Hadoop 系统写性能的提高。这种基于 Hadoop 系统性能限制的巧妙设计在 Hadoop 系统中是很常见的。

2. 文件元数据和名称节点

当客户端向 HDFS 请求读取或者存储一个文件的时候，它需要知道要访问的数据节点是哪一个。知道这个信息之后，客户端可以直接访问那个数据节点。文件的元数据信息由名称节点来负责维护。

HDFS 系统提供一个统一的文件系统命名空间，用户就像使用一个文件系统一样来存取集群节点上的数据。HDFS 把存储在目录中的文件按照一定的层级展示出来，目录是可以嵌套的。所有文件和目录的元数据信息都由名称节点来负责管理。

名称节点管理所有的文件操作，包括文件 / 目录的打开、关闭、重命名、移动等等。数据节点就负责存储实际的文件数据。这是一个非常重要的区别。当客户端请求或者发送文件数据，文件的数据在物理上是不经过名称节点传输的。否则这将是一个极其严重的瓶颈。客户端仅仅是简单地从名称节点获取文件的元数据，然后根据其元数据信息直接从数据节点获取文件的数据块。

名称节点存储的一些元数据包括：

- ❑ 文件 / 目录名称及相对于其父目录的位置。
- ❑ 文件和目录的所有权及权限。
- ❑ 各个数据块的文件名。所有数据块以文件的形式存放在数据节点的本地文件系统的目录中，该目录可以被 Hadoop 系统管理员配置。

需要注意的是名称节点并不存储每个数据块的位置（数据节点的身份信息）。数据节点的身份信息（DataNode identity）在集群启动的时候从每个数据节点获取。名称节点维护的信息是，HDFS 上的文件由哪些数据块（数据节点上每个数据块的文件名）组成。

元数据存储在名称节点的本地磁盘上，但是为了快速访问，在集群操作的时候会把这些元信息加载到内存。这个策略对于提高 Hadoop 系统的操作性能是很关键的，但是同时也成为了 Hadoop 系统的一个瓶颈，由此催生出了 Hadoop 2.x 系统。

每个元数据信息大约占用 200 字节的 RAM。假设一个 1GB 大小的文件，数据块的大小为 64MB。这么大的一个文件需要 16×3 （包括文件备份数量）= 48 个数据块的存储量。现在假设有 1000 个文件，每个文件的大小为 1MB。这么大的一个文件需要 1000×3 （包括文件备份数量）= 3000 个数据块的存储量（每个文件只需要占用数据块 1MB 的存储空间即可，但是多个文件不能存储在同一个数据块中）。这样的话，元数据的数据量将会大幅增长。这将会导致名称节点更大量的内存占用。这个例子也解释了为什么 Hadoop 系统更适合存储大文件，而

不适合存储大量的小文件。海量的小文件会轻易地拖垮名称节点服务器。

名称节点服务器上存放元数据的文件为 `fsimage`。Hadoop 系统运行期间任何涉及对元数据修改的操作都保存在内存中，并且被持久存储到另外一个名为 `edits` 的文件。辅助名称节点会周期性地把 `edits` 文件中的信息合并到 `fsimage` 文件中。（当我们在讲解辅助名称节点的时候，会深入细致地分析这个合并过程。）保存到 HDFS 上的实际文件数据并不会存放在 `fsimage` 和 `edits` 文件中，实际的数据存放在运行着数据节点守护进程的从节点的数据块中。正如前文讲到的，这些数据块以文件的形式存放在集群的从节点中。这些数据块中存放的是文件的原始数据，不存放文件元数据。但是，如果名称节点中保存的元数据丢失了，会导致整个集群不可用。名称节点中的元数据为客户端提供了数据块的信息，这些存放在从节点中的数据块存有文件的真实数据。

数据节点守护进程周期性的发送心跳信息给名称节点。这使得名称节点可以感知所有数据节点的健康状况，从而，客户端的请求不会被发送到故障数据节点。

3. HDFS 系统写文件的机制

一个 HDFS 文件系统的写操作涉及文件的创建。从客户端的角度来看，HDFS 系统不支持文件的更新。（这样说并不十分准确，因为在 HBase 中已经使用了 HDFS 系统的文件追加特性。但是，不建议客户端使用这样的功能。）在下文的讲述中，我们假定 HDFS 文件系统的默认备份因子为 3。

图 2-3 用图表的形式描述了 HDFS 文件系统写操作过程，通过这张图表我们对整个操作过程有了一个大致认识。

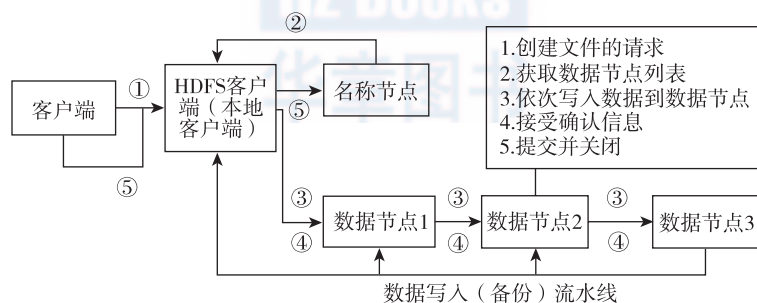


图 2-3 HDFS 文件系统写操作过程

客户端把一个文件写入到 HDFS 文件系统需要经过以下几个步骤：

- 1) 客户端在联系名称节点之前，会把文件数据流式地读入（streaming the file contents）到客户端本地文件系统中的一个临时文件中。
- 2) 当文件数据的大小达到一个数据块的大小时，客户端就联系名称节点。
- 3) 名称节点会在 HDFS 文件系统的层级结构中创建一个文件，然后把数据块的标识符和数据节点上的位置信息发送给客户端。这个数据节点数据块信息列表里面还包括了其备份节点的数据块信息列表。

4) 进行完上述的步骤之后, 客户端就会根据上一步的数据块信息把本地临时文件中的数据刷新 (flush) 到集群上的数据块中 (只写入到第一个数据节点中)。这样, 真实的文件数据就放在了集群数据节点本地文件存储系统中。

5) 当文件 (客户端可以访问的 HDFS 文件) 被关闭时, 名称节点会执行一个提交操作, 从而使得该文件在集群中为可见状态。如果在提交操作完成之前名称节点挂掉了, 这个文件就丢失了。

其中步骤 4 我们要拿出来详细讲解一下, 这个步骤中的刷新操作过程如下:

1) 第一个数据节点以数据包 (其大小一般为 4KB) 的形式从客户端接收数据。当这一小部分数据写入到第一个数据节点的本地存储中时, 它还会被传送到第二个数据节点中保存。

2) 当第二个数据节点开始把数据块写入到本地磁盘时, 这个包含文件数据块的数据包会被同时传送到第三个数据节点。

3) 最终, 这个包含文件数据的数据包写入到第三个数据节点中, 此时, 这部分的文件数据及其备份就以数据管道的方式存放到了 HDFS 文件系统中。

4) 保存包含文件数据的数据包成功之后, 其确认包 (acknowledgment packet) 会从保存了该数据包的计算节点通过数据管道返回给前一个保存了该数据包的计算节点。最后, 第一个数据节点会发送确认包到客户端。

5) 当客户端接收到了数据块保存成功的确认, 数据块就被认为是持久地存储到了集群中的计算节点上。客户端会发送最终的确认信息给名称节点。

6) 在数据管道中的任何数据节点发生故障, 数据管道就会关闭。文件数据将会被写入到其他数据节点。名称节点会感知到文件数据没有备份完成, 就会执行重新备份的过程, 把数据备份到状态良好的节点, 以确保达到要求的文件备份水平。

7) 每个数据块都会有一个校验和, 这个校验和用来检测数据块的完整性。这些校验和存放在 HDFS 中另外一个隐藏的文件中, 当读取数据块的时候用来校验数据块的完整性。

4. HDFS 系统读文件的机制

现在我们来讲解如何从 HDFS 中读取一个文件。图 2-4 展示了 HDFS 系统读文件的过程。

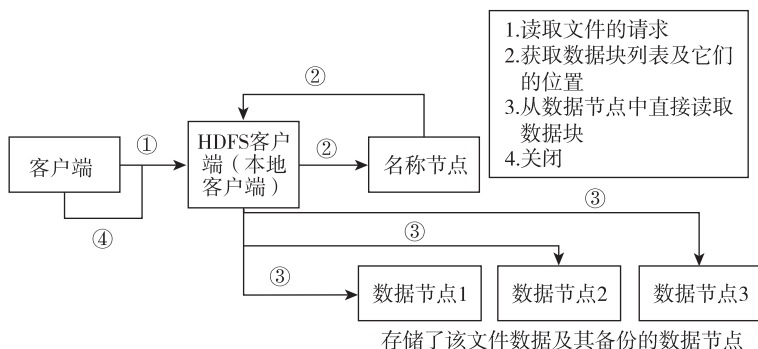


图 2-4 HDFS 文件系统读操作过程

客户端从 HDFS 中读取文件的步骤为：

- 1) 客户端访问名称节点，名称节点返回组成文件的数据块列表及数据块的位置（包括备份数据块的位置）。
- 2) 客户端会直接访问数据节点以获取数据块中的数据。如果此时其访问的数据节点出现故障，就会访问存放备份数据块的数据节点。
- 3) 读取数据块的时候会计算该数据块的校验和，并将该校验和与写入文件时的校验和作比较。如果检验失败，则从其他数据节点获取备份数据块。

5. HDFS 系统删除文件的机制

在 HDFS 系统上删除一个文件，遵从以下步骤：

- 1) 名称节点仅仅重命名了文件路径，使其移动到了 /trash 目录。需要注意的是，这个操作过程是链接到重命名文件路径的元数据的更新操作。这个执行过程非常迅速。/trash 目录中的文件会保存一段时间，这个保存时间是预先确定的（当前设定为 6 小时而且当前不可配置）。在这段时间内，把删除的文件从 /trash 目录中移动出来即可迅速地恢复该文件。
- 2) 当 /trash 目录中的文件超过了保存时间时，名称节点就会将该文件从 HDFS 命名空间中删除。
- 3) 删除文件会使得该文件相关的数据块被释放，HDFS 系统随后会显示增加了一些空闲空间。

HDFS 系统中的文件备份因子是可变的，它可以减小。通过一次心跳信息，这个文件备份因子的变化信息就可以由名称节点发出，数据节点就会动态地从本地存储系统中删除相应的数据块，这样一来，集群就会有更多的空闲存储空间了。所以，名称节点可以动态维护文件的备份数量。

6. 确保 HDFS 系统的可靠性

Hadoop 系统和 HDFS 系统都有很强的抗故障能力。在以下两种情况下会造成数据丢失：

- ❑ 数据节点故障：每个数据节点周期性地发送心跳信息到名称节点（默认 3 秒钟一次）。如果名称节点在预先设定的时间段内没有收到心跳信息，它就会认为数据节点有故障了。此刻，名称节点就会主动地对存储在故障数据节点上的数据块重新备份，把相关数据块（从其他健康节点的备份数据块获取）转移到状态健康的数据节点。这样就可以确保数据块备份数量不变。
- ❑ 位腐（bit rot）现象导致的数据损坏：这是指当电荷（在传输过程中）发生衰减而导致的数据丢失的情况。在 HDFS 系统读取数据块的操作过程中会进行数据校验和核对，这种情况的数据损坏就会被发现。如果数据块的校验失败了，便认为这个数据块已经损坏，就会读取该数据块的备份，然后名称节点会重新备份这个损坏了数据块，使得数据块的备份数量达到预先设定的数据块备份数量。

2.3.2 辅助名称节点

现在我们来讨论辅助名称节点在 Hadoop 系统中的角色。这个组件因为它不合适的名字而导致很多错误的理解。辅助名称节点不是用于进行故障切换的节点。

通过前文，我们知道名称节点在内存中维护着所有的元数据。名称节点最先从存放在本地文件系统中的 fsimage 文件将元数据加载到内存中。在 Hadoop 系统中的一系列操作运行过程中，元数据信息在内存中不断更新。为了防止数据丢失，这些数据操作记录还会被持久存储到另外一个名为 edits 的本地文件中。



注意 fsimage 文件并不实际存储各个数据块的位置。这些信息是 Hadoop 系统在每个数据节点启动的时候，名称节点从每个数据节点获取的，并被存放在名称节点的内存中。

edits 文件是用来在 Hadoop 系统运行期间收集元数据的变化。如果 Hadoop 系统重启了，在 Hadoop 重启的过程中，edits 文件中的内容会与 fsimage 文件中的内容合并。显然，这会拖慢 Hadoop 系统的重启时间。辅助名称节点就是来处理这个问题的。

辅助名称节点的作用就是周期性地把 edits 文件中的内容与 fsimage 文件中的内容合并。为此目的，辅助名称节点会周期性地顺序执行下列步骤：

- 1) 辅助名称节点会请求名称节点来结转 (roll over) edits 文件，确保新的更新保存到一个新的文件。这个新的文件名字叫做 edits.new。
- 2) 辅助名称节点向名称节点请求获取 fsimage 文件和 edits 文件。
- 3) 辅助名称节点把 edits 文件和 fsimage 文件合并，生成一个新的 fsimage 文件。
- 4) 名称节点从辅助名称节点接收到新生成的 fsimage 文件，并替代旧的 fsimage 文件。同时将 edits 文件中的内容替换成步骤 1 中创建的 edit.new 文件的内容。
- 5) 更新 fstime 文件来记录发生的检查点操作。

现在应该很清楚为什么名称节点可以导致 Hadoop 1.x 系统单点故障。如果 edits 文件和 fsimage 文件数据损坏了，存放在 HDFS 系统上的所有数据都会丢失。所以，集群中的数据节点可以是安装了 JBOD（不使用 RAID 技术的数组硬盘）的商用服务器，而名称节点和辅助名称节点必须安装了更为可靠的存储系统（基于 RAID 技术的存储系统），以确保数据不丢失。前面提到的那两个文件还必须要定期备份。如果这两个文件需要从其备份文件恢复，那么从当前到备份时刻所有的数据操作都会丢失。表 2-1 总结了 HDFS 系统中名称节点上的关键文件。

表 2-1 名称节点上的关键文件

文 件 名	功 能 描 述
fsimage	保存了最近的一次检查点的 HDFS 系统元数据的持久化状态信息
edits	保存了最近的一次检查点之后到现在的 HDFS 系统元数据的状态变化信息
fstime	保存了最近一次检查点的时间戳

2) 大多数的 MapReduce 作业都需要一个或多个输入文件目录。任务跟踪器向名称节点

发出请求，获得一个数据节点的列表，这个列表上的数据节点中存储了组成输入文件数据的数据块。

3) 作业跟踪器为作业的执行做准备工作。在这个步骤中，任务跟踪器确定执行该作业需要的任务（Map 任务和 Reduce 任务）数量。作业跟踪器尽量把这些任务都调度到离数据块最近的位置上运行。

4) 作业跟踪器把任务提交到每个任务跟踪器节点去执行。任务跟踪器节点监控任务执行情况。任务跟踪器以预先设定的时间间隔发送心跳信息到作业跟踪器。如果作业跟踪器在预先设定的时间间隔之后，没有收到任务跟踪器发来的心跳信息，那么就认为该任务跟踪器节点出现故障，任务就会被调度到另外一个节点去运行。

5) 一旦所有的任务都执行完毕，作业跟踪器就会更新作业状态为成功。如果任务反复失败达到一定的数量（这个数值可以通过 Hadoop 系统的配置文件来指定），作业跟踪器就会宣布作业运行失败。

6) 客户端会轮询作业跟踪器及时地获得作业运行状态。

到目前为止对 Hadoop 1.x 系统组件的讲解，可以很清楚地知道，作业跟踪器的故障会导致 Hadoop 系统的单点故障。如果作业跟踪器节点挂掉，集群上所有的任务都将无法正确运行。同样，由于仅有一个作业跟踪器节点运行，在多任务同时运行的系统环境中，会增加该作业跟踪器节点的工作负载。

2.4 Hadoop 2.0

MapReduce 已经进行了全新升级，即 Hadoop 2.0，升级后的版本经常被称为 MapReduce 2.0（MR v2）或者 YARN。本书中常常提到其版本号 2.x，虽然发行版本小数点后面的数字有变化，但是系统架构或者其运行方式并不会发生根本的变化。

MR v2 是一套应用编程接口（API），该接口兼容 MR v1，根据 MR v1 接口编写的程序仅需重新编译即可。Hadoop 2.x 系统的底层架构已经完全改变了，Hadoop 1.x 中的作业调度器承担两个主要功能：

- ❑ 资源管理（Resource management）
- ❑ 作业调度 / 作业监控（Job scheduling/job monitoring）

YARN 把这两个功能分为两个守护进程来分别承担。这样的设计使得系统有一个全局的资源管理器以及每个程序有一个应用程序管理器（Application Master）。注意，我们这里提到了程序（application），而不是作业（job）。在新的 Hadoop 2.x 系统中，一个程序（application）既可以指传统概念上的一个单独的 MapReduce 作业，也可以指一系列作业组成的有向无环图（Directed Acyclic Graph, DAG）。DAG 是一个由许多节点相连构成的图，图中没有循环。也就是说，无论使用什么样的方法来遍历这张图，你都不可能遇到已经在遍历过程中经过的节点。用简单明了的话说就是，多个作业组成了有向无环图就意味着这些作业之间存在着层属

关系 (hierarchical relationship)。

YARN 还使得 Hadoop 的功用不仅仅局限于 MapReduce。在后续章节中，我们会发现 MapReduce 框架存在着诸多局限。新的框架已经开始突破这些局限。举个例子，Apache Hive 给 Hadoop 系统带来了 SQL 特性，Apache PIG 可以让用户使用基于脚本的数据流方式在 Hadoop 系统中处理数据。像 HAMA 这样的新框架使得 Hadoop 系统更加适合迭代计算，这在机器学习这样的应用场景中非常有用。

来自 Berkley 的 Spark/Shark 框架是 Hive 和 HAMA 的结合，提供了低延时的 SQL 访问和一些内存计算能力。这些框架都是运行在 HDFS 之上的，Hadoop 系统不能仅仅只支持其中的某一种。对框架重新设计是非常必需的，以使得基于海量数据批处理计算框架（不仅限于 MapReduce 模型），基于 HAMA 的数据整体同步并行计算（BSP）框架或者基于 Shark/Spark 的内存缓存和计算框架都能运行在统一的 Hadoop 系统之上。

经过重新设计的新框架可以使得整个 Hadoop 系统对其提供一致的原生支持。这会使得关于安全和资源管理的系统级策略能以一致的方式应用，所有系统都共享相同的底层 HDFS。

YARN 系统由以下几个组成部分：

- ❑ 全局资源管理器 (Global Resource Manager)
- ❑ 节点管理器 (Node Manager)
- ❑ 针对每种应用程序的应用程序管理器 (Application-specific Application Master)
- ❑ 调度器 (Scheduler)
- ❑ 容器 (Container)

一部分 CPU 内核和一部分内存构成了一个容器。一个应用程序 (application) 运行在一组容器中。应用程序管理器的一个实例会向全局资源管理器请求获取资源。调度器会通过每个节点的节点管理器 (Node Manager) 来分配资源 (容器)。节点管理器会向全局资源管理器汇报每个容器的使用情况。

全局资源管理器和每个节点的节点管理器构成了新 MapReduce 框架的管理系统。全局资源管理器全权负责系统资源的分配。每种应用程序都有一个应用程序管理器。(比如，MapReduce 是一种应用程序，每个 MapReduce 作业是 MapReduce 类型程序的一个实例，就像面向对象编程中的类和对象之间的关系一样)。针对同一应用程序类型的所有应用程序，一个应用程序管理器实例被初始化。应用程序管理器实例 (instance) 向全局资源管理器协商获得容器来运行作业。全局资源管理器利用调度器 (全局组件) 与每个节点的节点管理器的沟通结果来分配资源。从系统角度来看，应用程序管理器也是运行在一个容器之中的。

YARN 全局架构如图 2-6 所示。

为了确保对已经存在的 MapReduce 程序的向后兼容，对 MapReduce v1 框架进行了重用，并没有做任何重大的修改。

我们来更详细地讲解 YARN 中的每个组件。从整体上来说，我们在一群商用服务器上搭建了 Hadoop 集群，每台服务器称为一个节点。

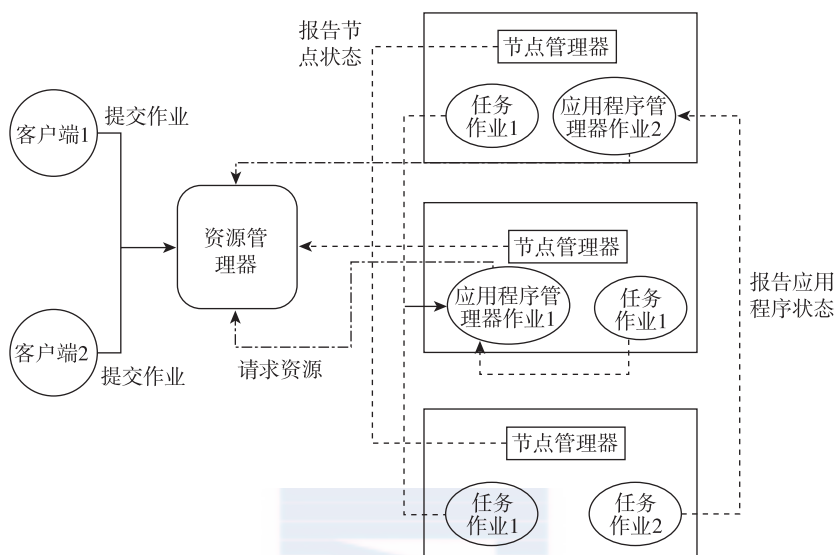


图 2-6 YARN 全局架构图

2.4.1 容器

容器（Container）是 YARN 框架中的计算单元。它是一个任务进行工作的单元子系统。也可以这么认为，YARN 框架中的容器相当于 MapReduce v1 中的一个任务（task）执行器。集群节点与容器之间的关系是：一个节点可以运行多个容器，但一个容器只能运行在一个节点之内。

一个容器就是已分配的一组系统资源。目前支持两种类型的系统资源：

- ❑ 中央处理器内核（CPU core）
- ❑ 内存（单位为 MB）

拥有系统资源的容器在某一节点上执行，所以一个容器中隐含了“资源名称”的概念，这个“资源名称”就是容器所在的机架和节点的名称。请求一个容器的时候，就会向一个节点发出请求。容器使得程序可以在某个节点上获得指定数量的 CPU 内核和一定数量的内存。

实际上，任何任务或者程序（单个任务或者多个任务组成的有向无环图）都运行在一个或者多个容器中。在 YARN 框架中全权负责分配容器的组件叫做节点管理器。

2.4.2 节点管理器

节点管理器（Node Manager）运行在集群中的一个节点上，集群中每个节点都会运行一个自己的节点管理器。它是作为一个从属服务（slave service）：它接受来自另外一个称为资源管理器的组件的请求，然后分配容器给应用程序。它还负责监控和汇报资源使用情况给资源管理器。在 Hadoop 集群中，节点管理器与资源管理器一起协同工作，负责管理分配 Hadoop 系统资源。资源管理器是一个 Hadoop 集群的全局组件，节点管理器作为各个节点的代理来负责管理集群中每个节点的健康情况。节点管理器的任务如下：

- ❑ 接受来自资源管理器的请求，为作业的执行分配容器。
- ❑ 与资源管理器交换信息，确保整个集群的稳定运行。资源管理器依靠各个节点管理器的汇报来跟踪整个集群的健康状况，节点管理器作为代理任务来监控和管理本节点的健康状况。
- ❑ 管理每个已启动的容器的整个生命周期。
- ❑ 每个节点的日志管理。
- ❑ 运行各种 YARN 应用程序使用的辅助服务（auxiliary service）。举个例子，在目前的 Hadoop 系统实现中，MapReduce 程序中的 Shuffle 服务就是一个辅助服务。

当一个节点启动，它会向资源管理器注册，并且告诉资源管理器有多少资源（最终可分配给容器使用的资源）可用。在节点运行期间，节点管理器和资源管理器会协同工作，不停地更新系统资源状态，保证集群资源有效地、优化地利用。

节点管理器仅对抽象出来的容器进行管理，而对单个应用程序或者应用程序类型的情况一无所知。负责管理这部分内容的是一个被称为应用程序管理器（Application Master）的组件。在我们介绍应用程序管理器之前，先简单介绍一下资源管理器。

2.4.3 资源管理器

资源管理器的核心是一个调度器：当多个应用程序竞争使用集群资源的时候，它来负责资源的分配调度，确保集群资源的优化合理使用。资源管理器有一个插件化的调度器，该调度器按照程序队列和集群的处理能力，负责为正在运行的多个应用程序分配其所需的集群资源。Hadoop 自带了计算能力调度器和公平调度器，在后面的章节中我们会详细地介绍这两种调度器。

一个任务的启动、配置及其资源的监控都由计算节点上的节点管理器（Node Manager）来负责。这种职责的分离使得资源管理器相比传统的作业调度器（JobScheduler）具备更好的系统扩展性。

2.4.4 应用程序管理器

应用程序管理器（Application Master）是老的 MapReduce v1 框架和 YARN 之间的关键区别之处。应用程序管理器是一个特定的框架函数库（framework-specific library）实例。它同资源管理器协调沟通资源，并通过节点管理器来获取这些系统资源，然后执行任务。应用程序管理器就是与资源管理器沟通以获取拥有系统资源的容器的组件。

应用程序管理器为 YARN 框架主要带来了以下好处：

- ❑ 提高了可扩展性。
- ❑ 框架更加通用。

在 MapReduce v1 中，负责处理任务失败的责任由作业跟踪器承担。作业跟踪器还要负责给要执行的任务分配系统资源。资源管理器（作业跟踪器的替代者）现在仅仅负责系统资源的调配，使得 MapReduce v1 系统框架的可扩展性获得了提升。管理作业或应用程序的工作就交

给了应用程序管理器。如果一个任务失败了，应用程序管理器会同资源管理器协调沟通以获取系统资源，并尝试着重新执行该任务。

在 MapReduce v1 中，Hadoop 系统框架仅仅支持 MapReduce 类型的作业，所以它并不是一个通用的框架。其主要原因就是因为在系统中比如作业跟踪器和任务跟踪器这样的关键组件，它们的设计开发过于局限于 Map 和 Reduce 概念。随着 MapReduce 使用的越来越广泛，人们发现某些类型的数据计算是不适用 MapReduce 框架的。由此新框架被开发出来，比如基于 Apache HAMA 和 Apache Giraph 的 BSP 框架。这些新框架适合图计算，在 HDFS 系统上运行得很好。本书成稿之际，像 Shark/Spark 这样的内存框架越来越引起人们的注意。虽然它们在 HDFS 上运行得很好，但是却无法运行在 Hadoop 1.x 之中，因为他们对计算框架的设计思路完全不同。

在 Hadoop 2.x 中把应用程序管理器作为 YARN 的一部分引入进来，改变了这一切。使用不同的应用程序管理器来管理基于不同设计思想的计算框架，使得多种计算框架可以共存于一个统一的管理系统之中。在 Hadoop 1.x 中，Hadoop/HAMA/Shark 虽然运行于相同的 HDFS 系统，但是只能分别运行在不同的管理系统之中，这样就造成了系统不一致和系统资源冲突，现在他们可以运行在同一个 Hadoop2.x 系统之中了。它们都通过资源管理器来公平地申请使用系统资源。YARN 使得 Hadoop 系统的使用变得更加广泛。Hadoop 系统现在不仅支持 MapReduce 类型的计算，它变得更加插件化：如果系统要增加某种类型的计算框架，就开发一个对应的应用程序管理器，并把这个程序管理器以插件的形式整合到 Hadoop 系统之中。应用程序管理器的概念使得 Hadoop 系统突破了 MapReduce 的限制，使得 MapReduce 可以与其他类型的计算框架共存并相互协作。

2.4.5 分步详解 YARN 请求

当一个用户向 Hadoop2.x 框架提交了一份作业，YARN 框架后台处理该请求（如图 2-7 所示）。步骤如下：

- 1) 一个客户端提交作业程序。该应用程序的类型确定了，就决定了使用何种应用程序管理器。
- 2) 资源管理器协调资源，在一个节点上获取一个用于运行应用程序管理器实例的容器。
- 3) 应用程序管理器在资源管理器中注册。注册之后，客户端就可以向资源管理器查询该应用程序管理器的详细信息了。客户端将会与应用程序管理器通信，应用程序管理器已经通过自己的资源管理器启动了。
- 4) 在这个操作过程中，应用程序管理器通过资源请求与资源管理器协商资源。除了其他内容，一个资源请求包括被请求的容器所在的节点和该容器的详细说明（CPU 核数量和内存大小）。
- 5) 在启动的容器中运行的应用程序会通过该应用程序特定的协议向应用程序管理器汇报它的执行进度（有可能是远程的）。

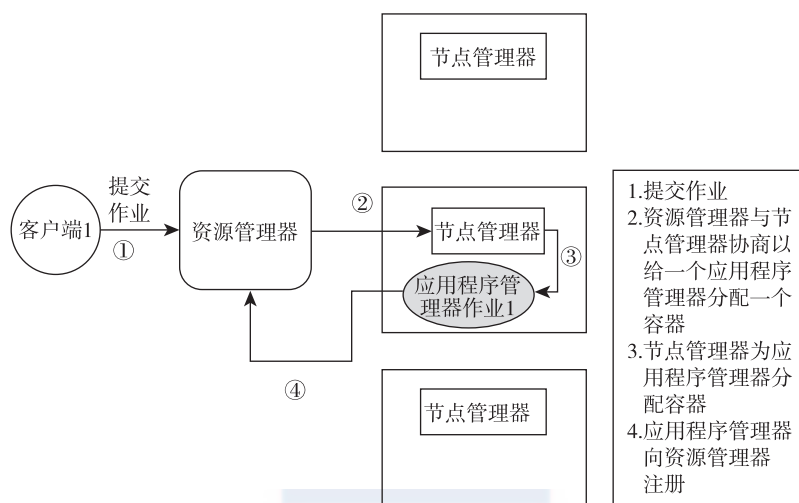


图 2-7 应用程序管理器启动

6) 通过应用程序特定的协议，客户端与应用程序管理器通信。客户端通过查询在步骤 3 中注册的资源管理器的信息就可以找到对应的应用程序管理器。

上面的步骤如图 2-8 所示：

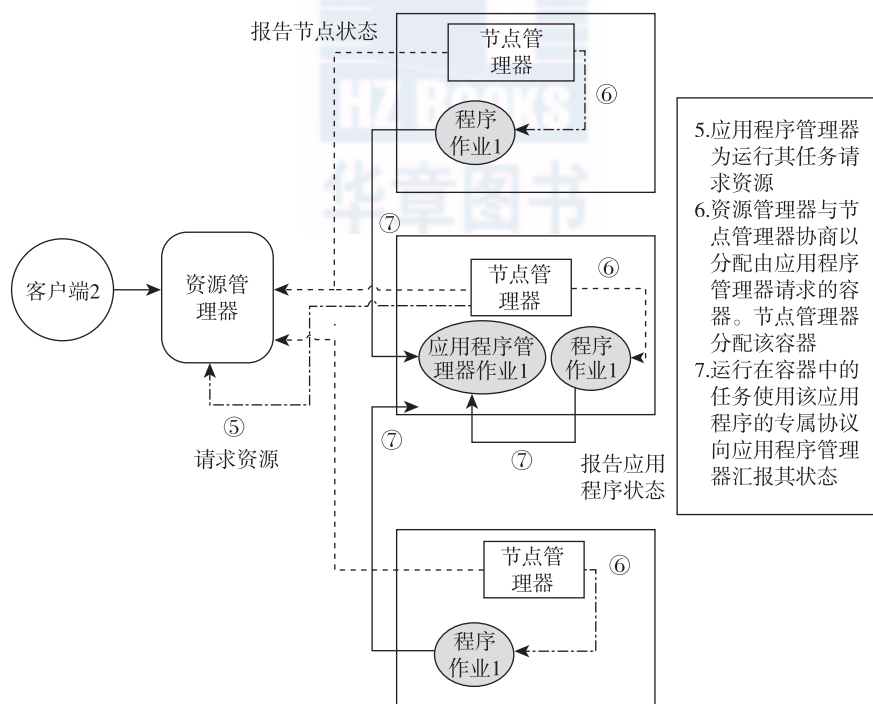


图 2-8 给作业分配资源并执行作业

当应用程序执行完毕，应用程序管理器就会从资源管理器中取消注册，作业占用的容器将会被释放回系统中。

2.5 HDFS 的高可用性

通过前面对 HDFS 的讲解，我们很清楚地知道在 Hadoop1.x 系统中，名称节点会引发系统单点故障。Hadoop1.x 系统只有一个名称节点，如果运行名称节点服务的节点机器出现故障，那么整个集群将会处于不可用的状态，除非名称节点服务在另外一台机器上重新启动。除了名称节点服务偶尔出现故障，从维护的角度来看，同样存在隐患。如果运行名称节点服务的机器需要重启，那么在名称节点服务无法运行的这段时间内，整个集群是不可用的。

Hadoop2.x 引入了高可用名称节点（High Availability NameNode）的概念，在这里，我们只是从概念上做个讲解。访问 Hadoop 网站可以更详细地了解高可用名称节点的实现细节。

高可用名称节点背后的核心思想是使用两个相同的名称节点：一个处在活动模式（active mode），另一个处在待机模式（standby mode）。处于活动模式的名称节点为系统提供服务，处在待机模式的名称节点需要实时同步活动名称节点的数据，一旦活动名称节点当机，系统可快速的进行故障切换。在当前设计中，为了达到这个目的，两个名称节点必须共享同一个存储设备（通过 NFS）。活动名称节点的任何修改都会记录到共享存储设备中的 edits 日志文件中。待机名称节点将这些修改应用到自己的名称空间中。一旦活动名称节点发生故障，待机名称节点会确保 edits 文件中的所有数据都被应用，并接管活动名称节点的职责。

我们知道名称节点保存的元数据中不包括数据块的存储位置。这些信息是在数据节点启动过程中，名称节点请求数据节点获得。为了保证待机名称节点的迅速启动，数据节点要知道两个名称节点的位置，并在启动的时候向两个名称节点都发送信息。数据节点也同时向两个名称节点交换心跳信息。

2.6 本章小结

本章介绍了 Hadoop 系统的各种概念。从一个典型的单词计数程序示例开始，一步步地介绍了 Hadoop 系统的各种特性。介绍了 Hadoop 分布式文件系统（HDFS），介绍了 Hadoop1.x 系统中是如何通过作业跟踪器守护进程和任务跟踪器守护进程来管理作业的。认识到了这些守护进程限制了系统的可扩展性之后，我们引入了 YARN 的讲解。YARN 是 Hadoop2.x 系统的重要特性，解决了系统扩展性的问题。最后，我们讲解了高可用名称节点。

下面的章节，我们会介绍 Hadoop 系统的安装，还会教你编写和执行自己的第一个 MapReduce 程序。

初识 Hadoop 框架

前面的章节介绍了大数据的价值，然后对 Hadoop 系统进行了总体介绍，Hadoop 系统是业界最重要的大数据框架。本章我们会真正地使用 Hadoop 系统框架。本章将指导你如何搭建 Hadoop 系统开发环境，以及如何在选择的操作系统上安装 Hadoop 系统。你将能写出第一个简单的 Hadoop 程序，这会帮助你更深刻地理解 Hadoop 系统架构中的各种概念。

3.1 安装类型

尽管 Hadoop 系统的安装常常是富有经验的系统管理员的工作，并且 Hadoop 系统的 Apache 站点上也有详尽的安装说明，但是对于在多种平台上安装 Hadoop 系统有个基本的了解还是非常重要的，这有以下两个原因：

- 要想对 Hadoop 程序进行单元测试，Hadoop 系统需要安装在单机模式下。这个过程在 Linux 系统中操作是相对简单的，但在 Windows 系统中非常繁琐。
- 为了能够模拟 Hadoop 程序在集群中的运行情况，Hadoop 系统提供了伪分布式集群模式下的操作。

本章内容涵盖了 Hadoop 系统的各种使用模式。对 Hadoop 开发环境配置的讲解是在提供商已经为虚拟机配置了开发环境的前提下进行的。我们会演示在 Windows 和 Linux 环境中以单机模式安装 Hadoop 系统（还会讲解 Linux 环境中以伪分布式模式安装 Hadoop 系统）。Hadoop 系统尚不十分完善，它的安装还是非常复杂的。附录 A 讲解了 Hadoop 系统在 Windows 平台和 Linux 平台下的安装步骤。这些步骤仅是一个通用的安装指南，在具体的安装过程中你可能会遇到各种不同的情况。我们建议你在虚拟机中安装一个开发环境来进行 Hadoop2.x 平台的开发，本章将会讲述如何安装。

3.1.1 单机模式

单机模式（Stand-Alone Mode）是最简单的安装类型模式，这种模式最适合调试。在这样的模式中，Hadoop 系统的所有程序运行在一个单独的 JVM 中。从系统性能的角度看，这种模式显然是性能最差的，但是在开发过程中是最高效的方式。

3.1.2 伪分布式集群模式

在伪分布式集群模式（Pseudo-Distributed Cluster）下，Hadoop 系统以伪分布式的方式运行在一个服务器节点中，所有的守护进程分别运行在不同的 Java 进程中。这样的运行模式经常用来模拟一个集群环境。

3.1.3 多节点集群安装模式

在多节点集群安装模式（Multinode Node Cluster Installation）中，Hadoop 系统安装在由若干机器组成的集群中。其安装过程比较复杂，一般由富有经验的 Linux 系统管理员来进行操作。从系统逻辑上看，其系统运行情况与伪分布式模式是一致的。

3.1.4 基于 Amazon EMR 预安装模式

另外一种快速搭建并使用一个真正的 Hadoop 系统集群的方法是使用亚马逊公司的 EMR（Elastic MapReduce）服务。目前这个服务既支持 Hadoop1.x 版本，也支持 Hadoop2.x 版本。同时，这个服务还支持 Hadoop 系统的多种发行版本，比如 Apache 版本或 MapR 公司发布的版本。

EMR 使得用户只要在 Web 页面上简单地点击几下就可以启动一个 Hadoop 集群。EMR 背后的主要思想如下：

- 1) 用户使用亚马逊公司的 S3 服务加载数据，S3 是一项简单的存储服务。亚马逊的 S3 是由 AWS（Amazon Web Services）提供的一个分布式文件存储系统。它通过 Web Services 接口来提供存储服务。通过配置，Hadoop 系统可以把 S3 当做分布式文件系统使用。在此模式下，S3 服务扮演了 HDFS 的角色。

- 2) 用户也可以使用亚马逊 S3 服务加载应用程序函数库。

- 3) 用户通过指定函数库和输入文件在 S3 上的位置就可以启动一个 EMR 作业，还要指定在 S3 上的输出目录供作业执行完毕写入输出结果。

- 4) 在亚马逊云上启动一个 Hadoop 集群，执行作业，结果输出到前面步骤中指定的目录中。

在默认操作中，集群会被自动关闭，用户也不必继续支付费用。用户可以通过一个选项（现在在启动 EMR 的 Web 页面上就有这个选项）来控制集群是否一直处于激活状态：自动终止选项（Auto-terminate option）。当在这个选项中选择 No，任务结束后，集群就不会被关闭了。

你可以使用一个 SSH（Secure Shell）客户端登录集群中的任何一个计算节点。当用户通

过一个 SSH 客户端连接到一台物理节点之后,就能够继续使用 Hadoop 系统集群的全部功能,甚至 HDFS 也是可用的。

用户可以在集群中使用示例程序执行数据量小的任务,这样可以保证集群一直处于运行状态。用户可以连接到集群中的一台节点执行更多的任务。使用一个普通的由两台服务器构成的集群每小时需要花费 1 美元(这依赖于你选择的服务器类型,如果选择的是顶级服务器,其花费会高达到每小时 14 美元)。当作业执行完毕,用户可以关闭集群,也不必继续支付费用了。用户只需花费少量的钱,就可以在真实的企业级 Hadoop 集群中运行我们实际需求中的作业(第 16 章会讲解云计算中的 Hadoop 系统)。



警告 就算是每小时 1 美元,时间积攒起来也是个不小的花费。留意亚马逊云提供的服务所处的状态,可以使我们的花费更加经济。不使用的云服务要及时关闭,并且还要很清楚自己使用了那些服务。据我们所知,确实有人不使用集群之后却忘记关掉它,结果一个月就损失了数百美元。不要让这种事发生在你的身上。

3.2 使用 Cloudera 虚拟机搭建开发环境

本书着重讲解基于 Hadoop 系统的开发。Hadoop 系统的安装是一个比较复杂的操作过程,服务提供商会提供安装工具使得该安装操作易如反掌。比如,Cloudera 公司开发的 Cloudera Manager 就大大简化了 Hadoop 系统的安装。作为一个开发人员,我们希望有一个可快速安装且易于搭建的可靠的开发环境。Cloudera 公司发布了基于 VMware 和 VirtualBox 的 CDH5.0。如果你没有安装这些虚拟机的播放器,请先下载安装其最新版本。然后,从下面这个链接下载 Cloudera 5 QuickStart VM:

www.cloudera.com/content/support/en/downloads/download-components/download-products.html?productID=F6m0278Rvo

需要注意的是,Cloudera 5 虚拟机需要 8GB 的内存。请确保你的机器有足够的内存来运行这个虚拟机。然后,一步步地执行剩下的操作步骤,自己的开发环境最终会安装成功。

当你启动虚拟机之后,你会看到如图 3-1 的界面。箭头指向的是虚拟机中桌面上的 Eclipse 图标。因为在 Eclipse 环境中已经将 Hadoop 程序配置为以本地模式的方式执行,所以,打开 Eclipse 即可开发 Hadoop 程序代码。

你要使用 Hadoop 2.0 系统作为开发环境的工作就完成了。该开发环境也允许用户以伪分布式模式的方式来运行作业,以此来模拟测试该作业代码在真实集群环境下的运行情况。这是一套集程序开发、单元测试和集成测试于一体的完整全面的开发环境。在该开发环境中,只要稍加配置就可以使用 Cloudera Manager,这是一个监控和管理作业的用户友好的 GUI(图形用户界面)。你最好能够熟练使用这个工具,它会极大地帮你简化作业的管理和跟踪工作。

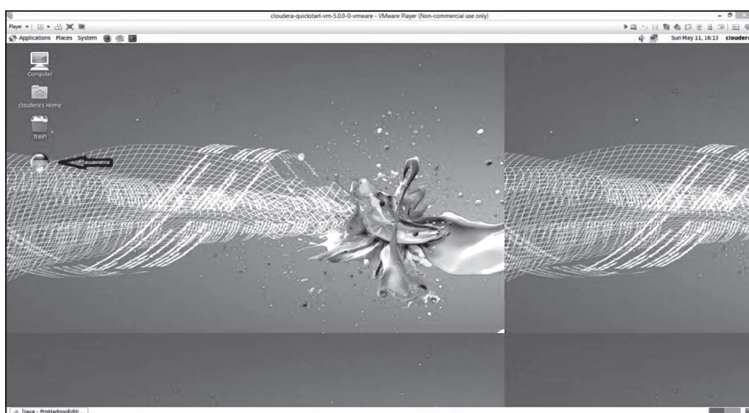


图 3-1 Cloudera 5 虚拟机

我们强烈推荐你使用上述方法来快速地搭建 Hadoop2.0 开发环境。



如果要使用本节讲到的 Cloudera 虚拟机来搭建 Hadoop 开发环境，你就无须了解 Hadoop 系统安装步骤了。但是我们仍然在附录 A 中介绍了 Hadoop 系统在 Windows 和 Linux 平台上的安装过程。你可以按照附录 A 中介绍的步骤，安装一个在伪分布式模式下运行的 Hadoop 系统。

3.3 一个 MapReduce 程序的组成

本节介绍基于 Java 语言的 MapReduce 程序由哪些部分构成。下文逐一讲解了各个组成部分：

- ❑ **Java 程序客户机 (Client Java program)**：一个 Java 程序，由集群中的一个客户端节点（又被称为边缘节点）提交运行。这个客户端节点可以访问 Hadoop 集群，它经常（并不总是）由集群中的一个数据节点来充当。该节点仅是集群中的一台机器，并且有权访问 Hadoop。
- ❑ **自定义 Mapper 类 (Custom Mapper class)**：除非在最简单的应用场景下，MapReduce 程序中的这个 Mapper 类通常是一个用户自定义类。如果不是在伪集群模式下运行 MapReduce 程序作业，这个类的实例会在远程任务节点上执行。这些任务节点往往与用来提交作业程序的客户端节点不同。
- ❑ **自定义 Reducer 类 (Custom Reducer class)**：除非在最简单的应用场景下，MapReduce 程序中的这个 Reducer 类通常是一个用户自定义类。与 Mapper 类一样，如果不是在伪集群模式下运行 MapReduce 程序作业，这个类的实例会在远程任务节点上执行。这些任务节点往往与用来提交作业程序的客户端节点不同。
- ❑ **客户端函数库 (Client-side libraries)**：客户端函数库不同于 Hadoop 系统的标准函数库，这个函数库是在客户端运行期间使用的。客户端需要使用的 Hadoop 系统的标

准函数库已经安装，并且使用通过 Hadoop 的 Client 命令（这与客户端程序不同）配置到 CLASSPATH 中。我们在文件夹 \$HADOOP_HOME/bin/ 中可以找到它，其名称为 hadoop。就像 Java 命令用来执行一个 Java 程序，hadoop 命令用来执行客户端程序，该程序会启动一个 Hadoop 作业。这些函数库都被配置到了环境变量 HADOOP_CLASSPATH 中。这个变量与 CLASSPATH 变量一样，是一个冒号分隔的函数列表。

- ❑ 远程函数库（Remote libraries）：这个函数库是用户自定义 Mapper 类和 Reducer 类所需要的。这个远程函数库不包括 Hadoop 系统自带的函数库，因为 Hadoop 系统自带的函数库已经在每个数据节点都配置好了。举个例子，如果 Mapper 类用到了一个特殊的 XML 解析器，包含这个解析器的函数库就必须要被传输到执行这个 Mapper 类实例的远程数据节点。
- ❑ Java 程序档案文件（Java Application Archive (JAR) files）：Java 程序以 JAR 文件的形式打包，这个 JAR 文件中包括了客户端 Java 类，以及用户自定义 Mapper 和 Reducer 类。还包括了客户端 Java 类、Mapper 类和 Reducer 类用到的其他自定义依赖类。

3.4 第一个 Hadoop 程序

本节中，你将开发你的第一个 Hadoop 程序。开发此程序使用的开发环境是带有 Maven 插件的 Eclipse。如果你使用 Cloudera 公司的虚拟机，开发环境已经预先安装于其中了。附录 B 讲解了如何在 Eclipse 中创建一个 Maven 工程。

首先创建一个空的 Maven 工程，并将所需依赖库添加其中。项目对象模型（Project Object Model, POM）如代码清单 3-1 所示。创建一个文件名为 pom.xml 的文件，把代码清单 3-1 中的代码拷贝到该文件中。

代码清单 3-1 pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>prohadoop</groupId>
  <artifactId>prohadoop</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>ProHadoop</name>
  <dependencies>
    <dependency>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-client</artifactId>
      <version>2.2.0</version>
    </dependency>
  </dependencies>
</project>
```

现在可以开发你的第一个 MapReduce 程序，并在本地运行它。但是，让人迷惑的是有两

套 MapReduce API：新 API 和旧 API。旧 API 已经被废弃了，但之前已经被广泛地使用。我们会使用两套 API 来运行作业程序，以便你熟悉了解它们。



提示 你无须自己动手编写上面的代码。在 Apress.com 站点上找到关于本书的目录页，你会在该目录页上发现一个标签，点击该标签就可以下载一个 ZIP 压缩文档，该文档中包含了所有的代码文件。

3.4.1 以本地模式运行程序的必要条件

能够以本地模式运行 Hadoop 程序是非常重要的，这样我们就可以对程序做单元测试以便快速地进行开发工作。为了这个目的，要做的唯一工作就是正确地配置 HADOOP_HOME 环境变量，而且 {HADOOP_HOME}/bin 要包含在 PATH 变量中。对于 Linux 系统来说，这两个变量配置正确就足够了。

但是，在 Windows 环境中，Hadoop 程序的整个构建过程中所需要应用的动态链接库（DLL）都要在本机配置好。这些动态链接库文件并不包含在从 Apache 网站下载的 tar 压缩文件中。在 Windows 平台上安装 Hadoop 系统的本地模式或者伪分布式模式的具体步骤，详见附录 A 的讲解。

如果你使用上文中提到的虚拟机，以本地模式运行 Hadoop 程序所必需的东西都已经预先安装配置好了。

假设准备工作都已经做好，下面介绍的示例程序会像你使用的集成开发环境（IDE）中的任何其他 Java 程序一样运行。

本书中，以本地模式运行的所有作业的输入数据都在文件夹 \${project.basedir}/src/main/resources/input 中。\${project.basedir} 目录是项目的根目录。作为 WordCount 程序的文本示例输入数据，存放在 \${project.basedir}/src/main/resources/input 目录中的 wordcount 子目录下。

3.4.2 使用旧 API 编写的单词计数程序

代码清单 3-2 中，WordCountOldAPI.java 文件展示的是使用已经被废弃的旧 API 编写的单词计数 WordCount 程序。但是旧 API 已经被广泛使用，很多以前写的程序仍在运行。旧 API 并不会完全消失，所以，知道如何使用它们还是很重要的，读者在维护老代码的时候会经常碰到它们。

代码清单 3-2 WordCountOldAPI.java

```
package org.apress.prohadoop.c3;

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

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
```

```

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;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.TextOutputFormat;

public class WordCountOldAPI {

    public static class MyMapper extends MapReduceBase implements
        Mapper<LongWritable, Text, Text, IntWritable> {
        public void map(LongWritable key, Text value,
            OutputCollector<Text, IntWritable> output, Reporter reporter
            throws IOException {
            output.collect(new Text(value.toString()), new IntWritable(1));
        }
    }

    public static class MyReducer 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 sum = 0;
            while (values.hasNext()) {
                sum += values.next().get();
            }
            output.collect(key, new IntWritable(sum));
        }
    }

    public static void main(String[] args) throws Exception {
        JobConf conf = new JobConf(WordCountOldAPI.class);
        conf.setJobName("wordcount");

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

        conf.setMapperClass(MyMapper.class);
        conf.setCombinerClass(MyReducer.class);
        conf.setReducerClass(MyReducer.class);
        conf.setNumReduceTasks(1);
        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);
        FileInputFormat.setInputPaths(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));

        JobClient.runJob(conf);
    }
}

```

我们从分析 main() 函数开始。下面列出了必须要理解的重要概念：

- JobConf 是 Hadoop 框架中配置 MapReduce 任务的主要接口。框架会按照 JobConf 对象中的配置信息来执行作业。

- ❑ `TextInputFormat` 向 Hadoop 框架声明其输入文件为文本格式。它是 `InputFormat` 的子类。(各种 `InputFormat` 类会在第 7 章中进行更细致地讲解。)现在,我们知道了 `TextInputFormat` 类会读取输入文件中的每行作为一个记录。
- ❑ `TextOutputFormat` 指定了该 `MapReduce` 作业的输出情况。举个例子,它会检测其输出目录是否已经存在。如果输出目录已经存在, Hadoop 系统会拒绝该作业的执行。Hadoop 作业会在海量数据上运行,它通常会花费几分钟或者几小时才行运行完毕。执行的作业偶尔会出现故障,导致其所有工作丢失,这种情况下,该作业会重新运行 (rerun)。各种特定的 `OutputFormat` 类会在第 7 章中讲解,但是,现在我们可以知道, `TextOutputFormat` 类用来声明其 `MapReduce` 作业的输出是文本格式文件。
- ❑ `FileInputFormat.setInputPaths(conf, new Path(args[0]))` 向 Hadoop 框架声明了要读取的文件所在的目录。该输入目录中可以包含一个或多个文件,并且每个文件中每行含有一个单词。注意在 `setInputPaths` 中使用的是复数。这个方法可以给 Hadoop 程序配置一个文件目录数组作为其输入数据路径。这些输入目录中的所有文件构成了该作业的输入数据源。
- ❑ `FileOutputFormat.setOutputPath(conf, new Path(args[1]))` 指定了作业的输出目录。作业最终的结果输出会保存到该目录下。
- ❑ `conf.setOutputKeyClass` 和 `conf.setOutputValueClass` 指定了输出的键 (Key) 类和值 (Value) 类,它们应该与 `Reducer` 类相匹配。这里似乎有些多余,确实是这样。键 (Key) 类和值 (Value) 类必须要同指定的 `Reducer` 类一致,否则在作业执行的时候将会抛出 `RuntimeException` 异常。
- ❑ `Reducer` 实例的执行数量默认为 1。该值是可以改变的,并且常常可以用来提高程序的运行效率。调用 `JobConf.class` 实例中的 `setNumReduceTasks(int n)` 方法就可以配置该参数。

现在你理解了整个程序是如何在 Hadoop 框架中运行的。当程序开始的时候, `Mapper` 函数会从输入文件夹中的输入文件中读取其数据块。一个文件的字节流会被转换成一个记录 (键 / 值对格式),然后作为 `Mapper` 的输入。键是当前行字节偏移量 (`LongWritable.class` 实例),值 (`Text.class` 实例)是从文件中读取的一行文本数据。在这个简化的例子中,使用的是本书源代码中的示例文件,每行一个单词。

`Mapper` 发送每行的单词和整数 1。要注意的是, Hadoop 作业程序需要使用与 `Integer.class` 对应的系统自带的 `IntWritable.class` 类。其原因是 Hadoop 系统底层 I/O 传输设计 (这些会在接下来的章节介绍)。现在我们只需知道, Hadoop 作业程序的 `Mappers` 和 `Reducers` 的输入和输出必须是 `Writable.class` 类型的实例。

紧接着是 Hadoop 系统的 `Shuffle` 阶段,这个过程在上面的列表中并不明显。所有的键 (在我们的例子中是单词) 在 `Shuffle/Sort` 阶段被排序,然后发送给 `Reducer`。逻辑上, `Reducer`

接受到的是一个 `IntWritable.class` 类实例的迭代器。实际上，在 `Reducer` 中使用的是相同的 `IntWritable.class` 类实例。当 `Reducer` 迭代地取出键对应的一系列值（在我们的例子中是整数 1 的一个列表），相同的 `IntWritable.class` 类实例被复用。从其内部运行来看，Hadoop 系统框架在其迭代器中使用 `values.next()` 之后，调用了 `IntWritable.set`（译者注，此处应为 `IntWritable.get`，疑为笔误）方法。当你在获取了这些值，并且作为引用来复用它们的时候要格外注意。因为 Hadoop 的这个特性（节约内存使用的一种优化），复用 `values.next()` 实例的引用会导致意想不到的结果。

最后，当 `Reducer` 输出结果的时候，结果被写入到由客户端 `WordCountOldAPI` 程序的 `FileOutputFormat.setOutputPath(conf, new Path(args[1]))` 指定的输出文件目录中。`Reducer` 会向输出文件中写入结果键和值的实例。输出文件中的键和值实例的默认分隔符是 `TAB`，这个分隔符可以通过配置参数：`mapreduce.textoutputformat.separator` 来修改。举个例子，在 `main(...)` 方法中调用 `conf.set("mapreduce.textoutputformat.separator",",")` 会把输出中键 / 值实例之间的分隔符变换为一个“`,`”。

在一个真实集群上运行作业的话，配置了多少个 `Reducers` 实例运行，就会生成多少个结果文件。本例中，结果文件包含的结果数据是一个单词，后跟该单词的计数。

在集群上提交执行上面这个作业程序的命令行与下面一节中使用的命令行是相似的。我们会在下一节中开发新 `API` 的程序时讲解。

3.4.3 构建程序

接下来，你将使用 `Maven` 来构建你的程序。附录 B 描述了这个过程，所以如果你对 `Maven` 不熟悉，请先阅读附录 B 中的内容。

构建过程生成了文件 `prohadoop-0.0.1-SNAPSHOT.jar`，表 3-1 描述了该应用程序中的各个组件。

表 3-1 使用旧 API 编写的 WordCount 程序中的各个组件

组 件	名 称
Client 类	WordCountOldAPI.java
Mapper 类	WordCountOldAPI.MyMapper.java
Reducer 类	WordCountOldAPI.MyReducer.java
Application JAR 文件	prohadoop-0.0.1-SNAPSHOT.jar
客户端函数库	在这个简单的程序中没有使用
远程函数库	在这个简单的程序中没有使用

3.4.4 在集群模式下运行单词计数程序

你可以在集群模式（或者伪分布式模式）下运行 `MapReduce` 作业。在集群模式下运行作

业与在本地模式下运行有以下两点不同：

- ❑ 运行 Map 和 Reduce 任务的节点往往与提交作业的客户端节点不是同一个节点。
- ❑ Map 和 Reduce 任务都在各自的 JVM 中运行。

在集群模式下运行 WordCount 程序的第一步是把程序的类文件打包成一个 JAR 文件。程序用以下方法提交到集群执行：

```
export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:<DEPENDENT_JARS_USED_BY_CLIENT_CLASS>

hadoop jar prohadoop-0.0.1-SNAPSHOT.jar \
org.aspress.prohadoop.c3.WordCountOldAPI \
<INPUT_PATH> <OUTPUT_PATH>
```

尽管在我们的客户端程序中并没有使用一些需要特殊指定的函数库，但是我们还是使用了下面的方法，该方法可以使程序访问那些指定的函数库：

```
export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:<DEPENDENT_JARS_USED_BY_CLIENT_CLASS>
```

Hadoop 命令使用 HADOOP_CLASSPATH 环境变量，用于在客户端程序提交运行之前配置客户端节点的 CLASSPATH 环境变量。

Hadoop 命令还负责把 prohadoop-0.0.1-SNAPSHOT.jar 文件发送到远程节点，并确保准备运行 Mapper 和 Reducer 实例的节点中已经配置好 CLASSPATH 环境变量。这个正是第 1 章中提到的把程序移动到数据附近的例子。

<INPUT_PATH> 和 <OUTPUT_PATH> 指向 HDFS 上的文件夹。前者指向的文件夹中包含了 Mapper 实例要读取的输入文件。后者指向的文件夹是 Reducer 实例执行结果输出的文件夹。

3.4.5 使用新 API 编写的单词计数程序

本节介绍新 API，使用新 API 可以使程序得到更好的控制。新 API 使用 Job.class 代替了 JobConf.class。利用它用户可以配置作业、提交作业、控制作业运行和监控作业执行情况。Job.class 中的各种 set() 方法与 JobConf.class 中的 set() 方法类似，它只有在作业被提交的时候才会被调用。如果在作业已经提交完毕之后，再调用那些方法，作业就会抛出 IllegalStateException 异常。用户使用 Job.class 中的 getStatus() 方法可以监控作业的运行状况。程序清单 3-3 显示了利用新 API 编写的单词计数程序。请注意在 main() 函数中使用 Job.class 代替了 JobConf.class。

代码清单3-3 WordCountNewAPI.java

```
package org.apress.prohadoop.c3;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
```

```

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordCountNewAPI {
    public static class MyMapper extends
        Mapper<LongWritable, Text, Text, IntWritable> {
        public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
            String w = value.toString();
            context.write(new Text(w), new IntWritable(1));
        }
    }

    public static class MyReducer extends
        Reducer<Text, IntWritable, Text, IntWritable> {

        public void reduce(Text key, Iterable<IntWritable> values,
            Context context) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            context.write(key, new IntWritable(sum));
        }
    }

    public static void main(String[] args) throws Exception {
        Job job = Job.getInstance(new Configuration());
        job.setJarByClass(WordCountNewAPI.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setMapperClass(MyMapper.class);
        job.setReducerClass(MyReducer.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        boolean status = job.waitForCompletion(true);
        if (status) {
            System.exit(0);
        } else {
            System.exit(1);
        }
    }
}

```

3.4.6 构建程序

再次构建该应用程序生成 prohadoop-0.0.1-SNAPSHOT.jar 文件。表 3-2 中描述了该应用程序中的各个组件。

表 3-2 使用新 API 编写的 WordCount 程序中的各个组件

组 件	名 称
Client 类	WordCountNewAPI.java
Mapper 类	WordCountNewAPI.MyMapper.java
Reducer 类	WordCountNewAPI.MyReducer.java
Application JAR 文件	prohadoop-0.0.1-SNAPSHOT.jar
客户端函数库	在这个简单的程序中没有使用
远程节点使用的函数库	在这个简单的程序中没有使用

3.4.7 在集群模式下运行单词计数程序

我们运行这个 MapReduce 程序的方法就像上节内容中如何运行使用旧 API 编写的 MapReduce 程序一样。在集群模式下运行该 MapReduce 程序的命令如下：

```
hadoop jar prohadoop-0.0.1-SNAPSHOT.jar \
org.aspress.prohadoop.c3.WordCountNewAPI \
<INPUT_PATH> <OUTPUT_PATH>
```

因为无须 HADOOP_CLASSPATH 环境变量，所以命令行中就没有涉及。如前所述，JAR 文件会发送到所有的数据节点，这些节点的本地 classpath 中已经配置为包含这个 JAR 文件。<INPUT_PATH> 和 <OUTPUT_PATH> 指向 HDFS 上的路径。Hadoop 命令负责执行这些预备步骤，然后在集群中以分布式的模式运行这个作业。

3.5 Hadoop 作业中的第三方函数库

到目前为止，在 Mapper 和 Reducer 类中只使用了标准 Java 函数库和 Hadoop 函数库。这些标准函数库包括了 Hadoop 发行版中的类库和标准 Java 类库（比如 String.class）。

可是，仅仅使用这些标准函数库不能够开发复杂的 Hadoop 作业程序，有时我们需要第三方库的支持。如前所述，Mapper 和 Reducer 类中使用的这些库需要被发送并配置到集群中运行 Mapper 和 Reducer 实例的所有节点。

首先，你要编写一个跟介绍过的代码不同的使用第三方库的程序。代码清单 3-4 中是一个使用了第三方库的 Word Count 程序。这个例子是故意如此编写的，以便更好地演示说明。

假设你有一个包含一系列单词的文件，其中有些单词由 Unicode 字符和数字（0～9）组成。其他都是由 Unicode 字符组成。由数字和字母组成的单词常常用在合同文件中，合同文件中各种各样的标示符就是由数字和字母组成的。简便起见，像“-”和“\$”这样的字符是不允许出现在单词中的。

假设我们只关心由字母和数字组成的单词。为了判断单词中是否只包含有字母和数字，我们需要用到 commons-lang 函数库中的 StringUtils.class。使用下面的逻辑来调用相应的方法

以做出正确的判断:

```
if(StringUtils.isAlphanumeric(key.toString()) && !StringUtils.isAlphaSpace(key.toString()) )
```

代码清单 3-4 中展示了全部源代码。

代码清单3-4 WordCountUsingToolRunner.java

```
package org.aspress.prohadoop.c3;

import java.io.IOException;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.util.GenericOptionsParser;
import org.apache.commons.lang.StringUtils;;

public class WordCountUsingToolRunner extends Configured implements Tool{

    public static class MyMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);

        public void map(LongWritable key, Text value, Context context) throws IOException,
            InterruptedException {
            String w = value.toString();
            context.write(new Text(w), one);
        }
    }

    public static class MyReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
            if(StringUtils.isAlphanumeric(key.toString()) && !StringUtils.isAlphaSpace(key.toString()) )
            {
                int sum = 0;
                for (IntWritable val : values) {
                    sum += val.get();
                }
                context.write(key, new IntWritable(sum));
            }
        }
    }

    public int run(String[] allArgs) throws Exception {
        Job job = Job.getInstance(getConf());
        job.setJarByClass(WordCountUsingToolRunner.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setMapperClass(MyMapper.class);
        job.setReducerClass(MyReducer.class);
        job.setInputFormatClass(TextInputFormat.class);
```

```

        job.setOutputFormatClass(TextOutputFormat.class);
        String[] args = new GenericOptionsParser(getConf(), allArgs).getRemainingArgs();
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.submit();
        return 0;
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        ToolRunner.run(new WordCountUsingToolRunner (), args);
    }
}

```

稍后讨论代码清单 3-4。首先，你要理解为了使用第三方函数库，POM 文件所作出的修改。代码清单 3-5 展示了 POM 文件中的有关部分。

代码清单3-5 POM文件中的相关变化

```

<dependencies>
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-client</artifactId>
    <version>2.2.0</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>commons-lang</groupId>
    <artifactId>commons-lang</artifactId>
    <version>2.3</version>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-assembly-plugin</artifactId>
      <version>2.4</version>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
      <executions>
        <execution>
          <id>assemble-all</id>
          <phase>package</phase>
          <goals>
            <goal>single</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>

```

在这个 POM 文件中，添加了 commons-lang 2.3 版本的函数库。注意添加 hadoop-client 函数库时使用的 <scope> 标签中的内容为 provided。到目前为止，<build> 标签中提供的配置项会使程序在编译之后生成两个独立的 JAR 文件。

- ❑ prohadoop-0.0.1-SNAPSHOT.jar 仅包含 org.apache 包及其子包中的用户自定义的类。
- ❑ prohadoop-0.0.1-SNAPSHOT-jar-with-dependencies.jar 中不仅仅包含了用户自己编写的类，还包含了 commons-lang-2.3.jar 文件中的类。由于 POM 文件中，添加 hadoop-client 函数库依赖时配置的 scope 值为 provided，所以生成的 JAR 文件中不包括这个函数依赖库。这个配置项的含义是，在程序编译时使用这个函数库，但是不把它打包到最终的 JAR 文件中。

为什么 hadoop-client 及其他的依赖函数库不包含在 prohadoop-0.0.1-SNAPSHOT-jar-with-dependencies.jar 文件中呢？我们知道，Maven 不但解析声明的函数库，还会解析该函数库（带有正确版本号）所依赖的其他函数库。如果 hadoop-client 函数库及其他依赖的函数库也被打包，会导致 JAR 文件的臃肿不堪。hadoop-client 函数库已经配置在了集群中的各个节点上，所以 hadoop-client 中的类都不需要打包到作业程序的 JAR 文件中。这是一个非常重要的优化。由于程序打包排除掉了 hadoop-client 函数库，整合依赖之后的 JAR 文件体积大大减小（比如 200KB 与 20MB）。回想一下，这个 JAR 文件需要从客户端节点传输到远程节点。JAR 文件的大小会影响到作业程序的启动耗时，因为只有当该文件传输完毕，作业才能够启动运行。试着去掉 <scope> 标签，并重新编译。

我们来讲解前文中代码清单 3-4 中的程序。需要注意的是有一个附加的函数库包含在程序之中。这个函数库是 commons-lang-2.3.jar，远程的 Mapper 和 Reducer 类会使用这个函数库。WordCountNewAPIV2.class 扩展并实现了新的类，运行这个程序需要把它依赖的函数库 JAR 文件分发到执行该程序的计算节点。

使用下面的命令来执行该程序：

```
export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:<DEPENDENT_JARS_USED_BY_CLIENT_CLASS>

export LIBJARS=$MYLIB/commons-lang-2.3.jar,<OTHER_JARS_USED_BY_REMOTE_COMPONENTS>

hadoop jar prohadoop-0.0.1-SNAPSHOT.jar \
org.aspress.prohadoop.c3.WordCountUsingToolRunner \
-libjars $LIBJARS \
<INPUT_PATH> <OUTPUT_PATH>
```

上面的命令、参数及其环境变量配置的解释如下：

- ❑ HADOOP_CLASSPATH 中的配置确保了位于 \$HADOOP_HOME/bin 文件夹中的 hadoop 命令能够访问依赖 JAR 文件，这个依赖 JAR 文件是启动 MapReduce 作业的客户端程序要使用的。在本示例中，这个变量不是必须的，因为示例中的 MapReduce 作业的客户程序没有使用任何第三方函数库。
- ❑ \$LIBJARS 变量是一个逗号分隔的函数库文件路径列表，这些函数库文件是集群中数

据节点执行 Mapper 和 Reducer 类的时候要使用的。需要注意的是这个列表的分隔符不同于在 HADOOP_CLASSPATH 变量指定函数库的时候用到的分隔符。

- ❑ 作业程序 JAR 文件和 \$LIBJARS 变量中设定的 JAR 文件会被发送到集群上所有执行该 Map/Reduce 任务的节点上。这是一个移动代码到数据附近的例子（第 1 章中介绍的）。这些 JAR 文件会被配置到远程节点的 CLASSPATH 变量中。main() 函数中的 ToolRunner 类负责这个工作。
- ❑ <INPUT_PATH> 和 <OUTPUT_PATH> 中配置的路径指向 HDFS 上的相关位置。
- ❑ 运行作业程序的时候要确保 Reducers 运行过程中不会抛出由于没有找到包含 StringUtils.class 的函数库 JAR 文件而导致的 ClassNotFoundException 异常。

有另一种不需要配置 -libjars 选项的方法来运行这个作业，其命令如下：

```
hadoop jar prohadoop-0.0.1-SNAPSHOT-jar-with-dependencies.jar \
org.aspress.prohadoop.c3.WordCountUsingToolRunner
<INPUT_PATH> <OUTPUT_PATH>
```

注意两种方法的关键区别。如果依赖的函数库已经被打包进了作业程序的 JAR 文件中，我们就不需要 -libjars 这个选项了。每当作业执行的时候，作业程序 JAR 文件就会被发送到远程节点上进行配置。实际上，新旧 API 写成的作业程序都可以使用这样的方法，而且在不使用代码清单 3-4 中介绍的 ToolRunner 类的情况下也可以使用这个方法。使用这个方法唯一的缺点就是当依赖函数库数量很大的时候，这种情况在实际情况中会经常碰到，把所有的依赖函数库都打包到 JAR 文件中，会导致 JAR 文件变得很大，这样的话，在程序构建 / 测试周期内会增加程序的编译时间。如果你的测试环境与开发环境不在同一个集群环境中（比如你有另外一个配置在云中的测试集群），当你需要经常修改你的作业程序代码的时候，而包含所有依赖函数库的 JAR 文件体积又比较大，经常移动这样的文件是比较耗费时间的。但请记住，这是执行作业程序的最通用的方法。

在这本书的后续讲解过程中，程序作业的开发均基于 MapReduce 框架的新 API，并使用 ToolRunner 类来执行它。

最后，我们来讨论一下代码清单 3-4 中的内容，它与代码清单 3-2 和代码清单 3-3 之间有什么不同呢？其主要不同在于它们使用不同的命令行来执行作业程序；以及为了确保在远程节点上执行的 Map 和 Reduce 任务可以访问第三方函数库，在作业程序开发中使用了不同的方法来编写类：

- ❑ 代码清单 3-4 更多地使用 GenericOptionsParser.class 类来获取程序运行的参数，比如输入文件路径和输出文件路径。使用方法如下所示：

```
new GenericOptionsParser(getConf(),
    allArgs).getRemainingArgs()
```

- ❑ 在提交执行 Hadoop 作业的命令行中，删除了 -libjars 和 \$LIBJARS_PATH 参数，仅仅返回程序所必需的参数。
- ❑ 在代码清单 3-4 中最重要的一个方面是，在 main() 方法中调用 ToolRunner.run() 时

所使用的 Configuration 实例必须与 run() 方法中用来配置作业的 Configuration 实例是同一个实例。为了确保这一点，run() 方法中一般都会使用 getConf() 方法来获取 Configuration 实例，getConf() 是在 Configurable 接口中定义的，并在该接口的继承类 Configured 类中实现的。如果使用的不是同一个 Configuration 实例，作业就会配置错误，远程节点上运行的 Mapper 和 Reducer 任务也无法访问第三方 JAR 文件。

- ❑ ToolRunner.run() 方法负责解析 -libjars 参数。它将解析这个参数的任务委托给 GenericOptionsParser.class 执行。把 -libjars 参数的值添加到 Configuration 对象，用这样的方法来配置远程任务。
- ❑ 代码清单 3-2 和代码清单 3-3 没有方法来传输第三方函数库。如果你想在代码清单 3-2 和代码清单 3-3 的程序中使用第三方函数库，需要打包到应用程序 JAR 文件中。这个打包后的文件就是使用 Maven 构建生成的 prohadoop-0.0.1-SNAPSHOT-jar-with-dependencies。（构建这个 JAR 文件的过程和目的，我们已经在前面介绍过了。）

表 3-3 中列出了代码清单 3-4 中 MapReduce 程序的各个组件。

表 3-3 使用新 API 和 ToolRunner 编写的 WordCount 程序中的各个组件

组 件	名 称
Client 类	WordCountUsingToolRunner.java
Mapper 类	WordCountUsingToolRunner.MyMapper.java
Reducer 类	WordCountUsingToolRunner.MyReducer.java
Application JAR 文件	prohadoop-0.0.1-SNAPSHOT.jar
另一种 Application JAR 文件	prohadoop-0.0.1-SNAPSHOT-jar-with-dependencies.jar
客户端函数库	无
远程节点使用的函数库	commons-lang-2.3.jar (仅在作业程序 JAR 文件是 prohadoop-0.0.1-SNAPSHOT.jar 的时候使用)

3.6 本章小结

在这一章中，我们开始使用 Hadoop 平台。学习了如何使用虚拟机来安装 Hadoop 开发环境，并且开发了第一个 Hadoop 程序。诚然，程序很简单，但是通过这些简单的程序学习了 Hadoop 平台上各种复杂的概念。

我们学习了 Hadoop 平台上的各种配置选项，以及配置一个 Hadoop 作业的方法。熟悉了 Hadoop 平台中用来配置作业的一组类和可高度自定义的 I/O 格式。最后，本章通过一个例子展示了如何将一组函数库文件从客户端传输到 Hadoop 集群中不同的远程节点上。