华章科技

资深Hadoop技术专家撰写，从开发者角度系统深入地讲解了Hadoop分布式文件系统、Hadoop文件I/O、Hive、HBase、Mahout，以及MapReduce的工作原理、编程方法和高级应用

内容细致，包含大量基于实际生产环境的案例，实战性强

大数据技术丛书

Programming Hadoop

Hadoop应用开发技术详解

刘 刚◎著

机械工业出版社
China Machine Press

014013343

TP274
236

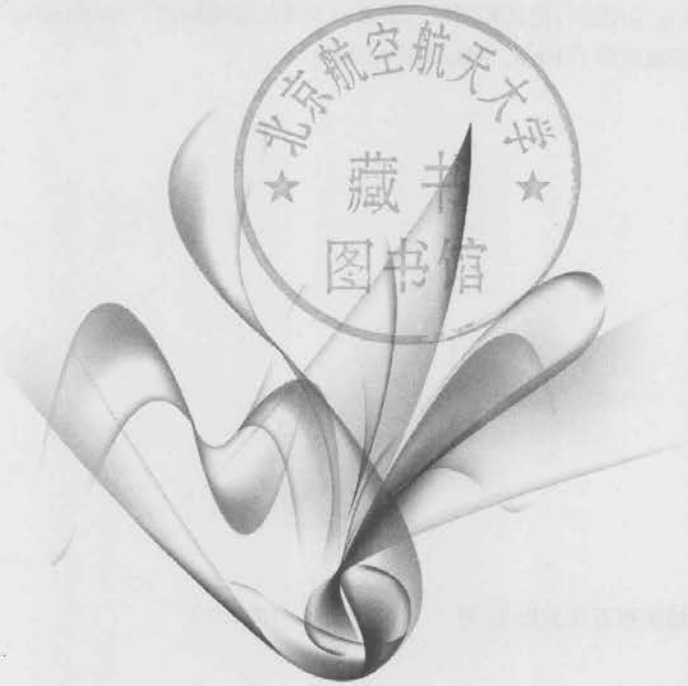
技术丛书

Programming Hadoop

Hadoop应用开发技术详解

刘 刚◎著

北京航空航天大学
★ 藏书 ★
图书馆



TP274
236

机械工业出版社

北航 C1700385

图书在版编目（CIP）数据

Hadoop 应用开发技术详解 / 刘刚著. —北京：机械工业出版社，2014.1
（大数据技术丛书）

ISBN 978-7-111-45244-7

I. H… II. 刘… III. 数据处理软件—程序设计 IV. TP274

中国版本图书馆 CIP 数据核字（2013）第 309446 号

版权所有 · 侵权必究
封底无防伪标均为盗版
本书法律顾问 北京市展达律师事务所

本书由资深 Hadoop 技术专家撰写，系统、全面、深入地讲解了 Hadoop 开发者需要掌握的技术和知识，包括 HDFS 的原理和应用、Hadoop 文件 I/O 的原理和应用、MapReduce 的原理和高级应用、MapReduce 的编程方法和技巧，以及 Hive、HBase 和 Mahout 等技术和工具的使用。并且提供大量基于实际生产环境的案例，实战性非常强。

全书一共 12 章。第 1 ～ 2 章详细地介绍了 Hadoop 的生态系统、关键技术以及安装和配置；第 3 章是 MapReduce 的使用入门，让读者了解整个开发过程；第 4 ～ 5 章详细讲解了分布式文件系统 HDFS 和 Hadoop 的文件 I/O；第 6 章分析了 MapReduce 的工作原理；第 7 章讲解了如何利用 Eclipse 来编译 Hadoop 的源代码，以及如何对 Hadoop 应用进行测试和调试；第 8 ～ 9 章细致地讲解了 MapReduce 的开发方法和高级应用；第 10 ～ 12 章系统地讲解了 Hive、HBase 和 Mahout。

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：白 宇
三河市杨庄长鸣印刷装订厂印刷
2014 年 1 月第 1 版第 1 次印刷
186mm × 240 mm · 26.25 印张
标准书号：ISBN 978-7-111-45244-7
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）88378991 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzjsj@hzbook.com

华章计算机
HZBOOKS | Computer Science and Technology

前　言

为什么要写这本书

三年前接触 Hadoop 的时候，Hadoop 在国内还没有普及，学习的资料也是少之又少，只能看英文文档一步一步操作，出现错误只能自己解决或者查看英文资料，有时候一个问题好几天才解决，学习起来非常的麻烦。

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，他们不曾想过培育更强壮的牛。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。简单说，数据量越来越大，我们的大型机负担不了，因此集群就产生了，而 Hadoop 就是为分布式计算和分布式存储而诞生的。最近两年，随着互联网的快速发展，公司的日志量成数量级增长，从而导致使用 Hadoop 的公司也越来越多，Hadoop 人才也就越来越供不应求，尤其是有丰富工作经验的人才。

在开源的 Hadoop 社区里，很多人问我怎么学习 Hadoop，学习 Hadoop 需要什么样的基础，如何才能成为一个 Hadoop 的技术大牛。为了帮助更多的人，我决定写本书，将我对 Hadoop 的理解和实战经验与大家分享。希望帮助读者在学习 Hadoop 的道路上少走弯路，避免一些不必要的时间付出。

读者对象

本书适合以下读者阅读。

- ❑ 大数据爱好者
- ❑ Hadoop 入门者
- ❑ 海量数据处理的从业人员
- ❑ 有 Java 基础，想学习 Hadoop 的开发者
- ❑ Hive、HBase 和 Mahout 的爱好者
- ❑ 具有 1 ～ 3 年使用 Hadoop 经验，想进一步提高能力的使用者
- ❑ 开设相关课程的大专院校

如何阅读本书

本书共 12 章。

第 1 章是 Hadoop 概述，互联网时代什么最重要——数据！大数据时代什么工具最流行——Hadoop！本章主要从大体上介绍 Hadoop，让读者对 Hadoop 有个大体的印象。

第 2 章主要是对 Hadoop 安装进行一步一步地详解，包括 Hadoop 的单机模式、伪分布式模式以及分布式模式，让读者了解 Hadoop 的安装过程。

第 3 章是 MapReduce 快速入门，通过一个实例介绍 MapReduce 开发的整个过程。包括环境的准备、Mapper 和 Reducer 代码的编写、打包、部署和运行的一整套流程。这套流程对以后 MapReduce 的开发奠定了基础。此外，介绍了 MapReduce 开发遇到的常见错误，以及解决方法。

第 4 章本章详细讲解了 Hadoop 分布式文件系统（HDFS），让读者对 HDFS 有一个全面的了解。HDFS 是 Hadoop 里面的一个核心组件，任何基于 Hadoop 应用的工具都要用到 HDFS，学习本章对 Hadoop 系统的维护有很大的帮助。

第 5 章本章主要讲 Hadoop I/O 底层的知识，让读者了解 Hadoop 底层的一些原理，比如数据在 Hadoop 上是怎么传输的。学完这章对 Hadoop 的输入和输出、序列化和反序列化、Writable 类型等有进一步的理解。

第 6 章主要介绍 MapReduce 的工作原理，讲解了 JobTracker、TaskTracker、任务的调度、任务工作的原理以及 MapReduce 的 Shuffle 和 Sort 机制等，此外还介绍第二代 MapReduce（YARN）。通过本章内容的学习，将会对日后 MapReduce 编程开发奠定基础。

第 7 章主要介绍三部分内容，Eclipse 的 MapReduce 插件安装、利用插件调试 MapReduce 代码，以及 MapReduce 单元测试。目的是提高读者编写 MapReduce 的调试、测试的效率。选择和使用一个好的工具是项目成功的一半。

第 8 章详细讲解了 MapReduce 编程开发用到的知识点，如 Combiner、Partitioner、DistributedCache 等。全面介绍了 Mapper 和 Reducer 两个抽象类，最后介绍了如何使用 Hadoop Streaming 接口开发 MapReduce 程序。

第 9 章介绍了 MapReduce 的一些高级应用、算法等知识点，算是对第 8 章学的知识进行应用，本章内容是 MapReduce 开发和 Hadoop 系统调优常用的知识点，希望读者能够掌握，为 MapReduce 的算法开发和调优打下基础。

第 10 章对 Hive 的知识点和使用做了介绍。Hive 是基于 Hadoop 的一个数据仓库，支持类似 SQL 语句来替代 MapReduce，这使不会写代码的数据分析师也能利用 Hadoop 平台对海量数据进行分析，如 DBA。Hive 工具现在已经非常成熟，使用的人也越来越多。

第 11 章介绍了 HBase 的安装和 Shell 操作、MapReduce 操作 HBase 等。HBase 是一个基于列存储的 NoSQL 数据库，不同于一般的 NoSQL 数据库，HBase 是一个适合于非结构化数据存储的数据库。

第 12 章介绍了 Mahout 算法框架的安装和使用。本章介绍的常用知识是入门的最佳实战。Mahout 是非常优秀的基于 MapReduce 的算法框架。Mahout 工具在商品推荐、电影推荐、广告推荐等领域应用非常广泛。

附录 A 为 Hive 内置操作符与函数。附录 B 为 HBase 默认配置解释。附录 C 为 Hadoop

三个配置文件的参数含义说明。

其中第 8 章和第 9 章侧重于 Java 开发 MapReduce 的用户，如果你是一名有经验的 Hadoop 用户，并且想学习 MapReduce 开发，可以直接阅读这两章的内容。但是如果你是一名初学者，请一定从第 1 章的基础理论知识开始学习。

勘误和支持

除封面署名外，参加本书编写工作的还有：向磊、艾男、曾泽、胡畔、郭洁、童小军、王虎、李晋博、林斌、闫越等。由于作者的水平有限，编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。书中的全部源文件可以从华章网站[⊖]下载，我也会定期将相应的功能更新及时更正出来。可以将书中的错误和遇到的任何问题，发邮件到 jaylg2010@gmail.com。如果你有更多的宝贵意见，也欢迎发送邮件，期待能够得到你们的真挚反馈。

致谢

首先要感谢 Doug Cutting，他开创了一款影响整个互联网领域的开源平台——Hadoop。

感谢机械工业出版社华章公司的杨福川老师，在这一年多的时间始终支持我的写作，你的鼓励和帮助引导我顺利完成全部书稿。感谢编辑白宇对稿件的认真修改，你提出的一些写书注意事项，让我受益匪浅。

感谢胡畔老师对我的支持和鼓励。

最后感谢我的爸爸、妈妈，感谢你们将我培养成人，并时时刻刻为我灌输着信心和力量！

谨以此书献给我最亲爱的家人，以及众多热爱 Hadoop 的朋友们！

⊖ 参见华章网站 www.hzbook.com。——编辑注

推荐阅读

Hadoop实战
第2版
Hadoop实战

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

Hadoop技术内幕
深入解析MapReduce架构设计与实现原理

Hadoop技术内幕
深入解析YARN架构设计与实现原理

Linux软件管理平台
设计与实现

高性能Linux服务器构建实战
运维监控、性能调优与集群应用

构建高可用Linux服务器
第2版
构建高可用
Linux服务器

Ganglia
系统监控
O'REILLY

推荐阅读

北航 C1700385

目　录

前　言

第 1 章　Hadoop 概述 / 1

1.1　Hadoop 起源 / 1

1.1.1　Google 与 Hadoop 模块 / 1

1.1.2　为什么会有 Hadoop / 1

1.1.3　Hadoop 版本介绍 / 2

1.2　Hadoop 生态系统 / 3

1.3　Hadoop 常用项目介绍 / 4

1.4　Hadoop 在国内的应用 / 6

1.5　本章小结 / 7

第 2 章　Hadoop 安装 / 8

2.1　Hadoop 环境安装配置 / 8

2.1.1　安装 VMware / 8

2.1.2　安装 Ubuntu / 8

2.1.3　安装 VMware Tools / 15

2.1.4　安装 JDK / 15

2.2　Hadoop 安装模式 / 16

2.2.1　单机安装 / 17

2.2.2　伪分布式安装 / 18

2.2.3　分布式安装 / 20

2.3　如何使用 Hadoop / 27

2.3.1　Hadoop 的启动与停止 / 27

2.3.2　Hadoop 配置文件 / 28

2.4　本章小结 / 28

第 3 章　MapReduce 快速入门 / 30

3.1　WordCount 实例准备开发环境 / 30

3.1.1　使用 Eclipse 创建一个 Java 工程 / 30

3.1.2　导入 Hadoop 的 JAR 文件 / 31

3.2　MapReduce 代码的实现 / 32

3.2.1　编写 WordMapper 类 / 32

3.2.2　编写 WordReducer 类 / 33

3.2.3　编写 WordMain 驱动类 / 34

3.3　打包、部署和运行 / 35

3.3.1　打包成 JAR 文件 / 35

3.3.2　部署和运行 / 36

3.3.3　测试结果 / 38

3.4　本章小结 / 39

第 4 章　Hadoop 分布式文件系统详解 / 40

4.1　认识 HDFS / 40

4.1.1　HDFS 的特点 / 40

4.1.2　Hadoop 文件系统的接口 / 45

4.1.3　HDFS 的 Web 服务 / 46

4.2　HDFS 架构 / 46

4.2.1　机架 / 47

4.2.2　数据块 / 47

4.2.3　元数据节点 / 48

4.2.4　数据节点 / 50

4.2.5　辅助元数据节点 / 50

4.2.6　名字空间 / 52

4.2.7　数据复制 / 53

4.2.8　块备份原理 / 53

4.2.9　机架感知 / 54

4.3 Hadoop 的 RPC 机制 / 55
4.3.1 RPC 的实现流程 / 56
4.3.2 RPC 的实体模型 / 56
4.3.3 文件的读取 / 57
4.3.4 文件的写入 / 58
4.3.5 文件的一致模型 / 59
4.4 HDFS 的 HA 机制 / 59
4.4.1 HA 集群 / 59
4.4.2 HA 架构 / 60
4.4.3 为什么会有 HA 机制 / 61
4.5 HDFS 的 Federation 机制 / 62
4.5.1 单个 NameNode 的 HDFS 架构的局限性 / 62
4.5.2 为什么引入 Federation 机制 / 63
4.5.3 Federation 架构 / 64
4.5.4 多个名字空间的管理问题 / 65
4.6 Hadoop 文件系统的访问 / 66
4.6.1 安全模式 / 66
4.6.2 HDFS 的 Shell 访问 / 67
4.6.3 HDFS 处理文件的命令 / 67
4.7 Java API 接口 / 72
4.7.1 Hadoop URL 读取数据 / 73
4.7.2 FileSystem 类 / 73
4.7.3 FileStatus 类 / 75
4.7.4 FSDataInputStream 类 / 77
4.7.5 FSDataOutputStream 类 / 81
4.7.6 列出 HDFS 下所有的文件 / 83
4.7.7 文件的匹配 / 84
4.7.8 PathFilter 对象 / 84
4.8 维护 HDFS / 86
4.8.1 追加数据 / 86
4.8.2 并行复制 / 88
4.8.3 升级与回滚 / 88
4.8.4 添加节点 / 90
4.8.5 删除节点 / 91
4.9 HDFS 权限管理 / 92
4.9.1 用户身份 / 92
4.9.2 权限管理的原理 / 93
4.9.3 设置权限的 Shell 命令 / 93
4.9.4 超级用户 / 93
4.9.5 HDFS 权限配置参数 / 94
4.10 本章小结 / 94

第 5 章 Hadoop 文件 I/O 详解 / 95

5.1 Hadoop 文件的数据结构 / 95
5.1.1 SequenceFile 存储 / 95
5.1.2 MapFile 存储 / 99
5.1.3 SequenceFile 转换为 MapFile / 101
5.2 HDFS 数据完整性 / 103
5.2.1 校验和 / 103
5.2.2 数据块检测程序 / 104
5.3 文件序列化 / 106
5.3.1 进程间通信对序列化的要求 / 106
5.3.2 Hadoop 文件的序列化 / 107
5.3.3 Writable 接口 / 107
5.3.4 WritableComparable 接口 / 108
5.3.5 自定义 Writable 接口 / 109
5.3.6 序列化框架 / 113
5.3.7 数据序列化系统 Avro / 114
5.4 Hadoop 的 Writable 类型 / 115
5.4.1 Writable 类的层次结构 / 115
5.4.2 Text 类型 / 116
5.4.3 NullWritable 类型 / 117
5.4.4 ObjectWritable 类型 / 117
5.4.5 GenericWritable 类型 / 117
5.5 文件压缩 / 117
5.5.1 Hadoop 支持的压缩格式 / 118

5.5.2 Hadoop 中的编码器和解码器 / 118
5.5.3 本地库 / 121
5.5.4 可分割压缩 LZO / 122
5.5.5 压缩文件性能比较 / 122
5.5.6 Snappy 压缩 / 124
5.5.7 gzip、LZO 和 Snappy 比较 / 124
5.6 本章小结 / 125

第 6 章 MapReduce 工作原理 / 126

6.1 MapReduce 的函数式编程概念 / 126
6.1.1 列表处理 / 126
6.1.2 Mapping 数据列表 / 127
6.1.3 Reducing 数据列表 / 127
6.1.4 Mapper 和 Reducer 如何工作 / 128
6.1.5 应用实例：词频统计 / 129
6.2 MapReduce 框架结构 / 129
6.2.1 MapReduce 模型 / 130
6.2.2 MapReduce 框架组成 / 130
6.3 MapReduce 运行原理 / 132
6.3.1 作业的提交 / 132
6.3.2 作业初始化 / 134
6.3.3 任务的分配 / 136
6.3.4 任务的执行 / 136
6.3.5 进度和状态的更新 / 136
6.3.6 MapReduce 的进度组成 / 137
6.3.7 任务完成 / 137
6.4 MapReduce 容错 / 137
6.4.1 任务失败 / 138
6.4.2 TaskTracker 失败 / 138
6.4.3 JobTracker 失败 / 138
6.4.4 子任务失败 / 138
6.4.5 任务失败反复次数的处理方法 / 139
6.5 Shuffle 阶段和 Sort 阶段 / 139
6.5.1 Map 端的 Shuffle / 140
6.5.2 Reduce 端的 Shuffle / 142
6.5.3 Shuffle 过程参数调优 / 143
6.6 任务的执行 / 144
6.6.1 推测执行 / 144
6.6.2 任务 JVM 重用 / 145
6.6.3 跳过坏的记录 / 145
6.6.4 任务执行的环境 / 146
6.7 作业调度器 / 146
6.7.1 先进先出调度器 / 146
6.7.2 容量调度器 / 146
6.7.3 公平调度器 / 149
6.8 自定义 Hadoop 调度器 / 153
6.8.1 Hadoop 调度器框架 / 153
6.8.2 编写 Hadoop 调度器 / 155
6.9 YARN 介绍 / 157
6.9.1 异步编程模型 / 157
6.9.2 YARN 支持的计算框架 / 158
6.9.3 YARN 架构 / 158
6.9.4 YARN 工作流程 / 159
6.10 本章小结 / 160

第 7 章 Eclipse 插件的应用 / 161

7.1 编译 Hadoop 源码 / 161
7.1.1 下载 Hadoop 源码 / 161
7.1.2 准备编译环境 / 161
7.1.3 编译 common 组件 / 162
7.2 Eclipse 安装 MapReduce 插件 / 166
7.2.1 查找 MapReduce 插件 / 166
7.2.2 新建一个 Hadoop location / 167
7.2.3 Hadoop 插件操作 HDFS / 168
7.2.4 运行 MapReduce 的驱动类 / 170
7.3 MapReduce 的 Debug 调试 / 171

7.3.1 进入 Debug 运行模式 / 171
7.3.2 Debug 调试具体操作 / 172
7.4 单元测试框架 MRUnit / 174
7.4.1 认识 MRUnit 框架 / 174
7.4.2 准备测试案例 / 174
7.4.3 Mapper 单元测试 / 176
7.4.4 Reducer 单元测试 / 177
7.4.5 MapReduce 单元测试 / 178
7.5 本章小结 / 179

第 8 章 MapReduce 编程开发 / 180

8.1 WordCount 案例分析 / 180
8.1.1 MapReduce 工作流程 / 180
8.1.2 WordCount 的 Map 过程 / 181
8.1.3 WordCount 的 Reduce 过程 / 182
8.1.4 每个过程产生的结果 / 182
8.1.5 Mapper 抽象类 / 184
8.1.6 Reducer 抽象类 / 186
8.1.7 MapReduce 驱动 / 188
8.1.8 MapReduce 最小驱动 / 189
8.2 输入格式 / 193
8.2.1 InputFormat 接口 / 193
8.2.2 InputSplit 类 / 195
8.2.3 RecordReader 类 / 197
8.2.4 应用实例：随机生成 100 个小数并求最大值 / 198
8.3 输出格式 / 205
8.3.1 OutputFormat 接口 / 205
8.3.2 RecordWriter 类 / 206
8.3.3 应用实例：把首字母相同的单词放到一个文件里 / 206
8.4 压缩格式 / 211
8.4.1 如何在 MapReduce 中使用压缩 / 211
8.4.2 Map 作业输出结果的压缩 / 212
8.5 MapReduce 优化 / 212
8.5.1 Combiner 类 / 212
8.5.2 Partitioner 类 / 213
8.5.3 分布式缓存 / 217
8.6 辅助类 / 218
8.6.1 读取 Hadoop 配置文件 / 218
8.6.2 设置 Hadoop 的配置文件属性 / 219
8.6.3 GenericOptionsParser 选项 / 220
8.7 Streaming 接口 / 221
8.7.1 Streaming 工作原理 / 221
8.7.2 Streaming 编程接口参数 / 221
8.7.3 作业配置属性 / 222
8.7.4 应用实例：抓取网页的标题 / 223
8.8 本章小结 / 225

第 9 章 MapReduce 高级应用 / 226

9.1 计数器 / 226
9.1.1 默认计数器 / 226
9.1.2 自定义计数器 / 229
9.1.3 获取计数器 / 231
9.2 MapReduce 二次排序 / 232
9.2.1 二次排序原理 / 232
9.2.2 二次排序的算法流程 / 233
9.2.3 代码实现 / 235
9.3 MapReduce 中的 Join 算法 / 240
9.3.1 Reduce 端 Join / 240
9.3.2 Map 端 Join / 242
9.3.3 半连接 Semi Join / 244
9.4 MapReduce 从 MySQL 读写数据 / 244
9.4.1 读数据 / 245

9.4.2 写数据 / 248
9.5 Hadoop 系统调优 / 248
9.5.1 小文件优化 / 249
9.5.2 Map 和 Reduce 个数设置 / 249
9.6 本章小结 / 250

第 10 章 数据仓库工具 Hive / 251

10.1 认识 Hive / 251
10.1.1 Hive 工作原理 / 251
10.1.2 Hive 数据类型 / 252
10.1.3 Hive 的特点 / 253
10.1.4 Hive 下载与安装 / 255
10.2 Hive 架构 / 256
10.2.1 Hive 用户接口 / 257
10.2.2 Hive 元数据库 / 259
10.2.3 Hive 的数据存储 / 262
10.2.4 Hive 解释器 / 263
10.3 Hive 文件格式 / 264
10.3.1 TextFile 格式 / 265
10.3.2 SequenceFile 格式 / 265
10.3.3 RCFile 文件格式 / 265
10.3.4 自定义文件格式 / 269
10.4 Hive 操作 / 270
10.4.1 表操作 / 270
10.4.2 视图操作 / 278
10.4.3 索引操作 / 280
10.4.4 分区操作 / 283
10.4.5 桶操作 / 289
10.5 Hive 复合类型 / 290
10.5.1 Struct 类型 / 291
10.5.2 Array 类型 / 292
10.5.3 Map 类型 / 293
10.6 Hive 的 JOIN 详解 / 294
10.6.1 JOIN 操作语法 / 294
10.6.2 JOIN 原理 / 294
10.6.3 外部 JOIN / 295
10.6.4 Map 端 JOIN / 296
10.6.5 JOIN 中处理 NULL 值的语义区别 / 296
10.7 Hive 优化策略 / 297
10.7.1 列裁剪 / 297
10.7.2 Map Join 操作 / 297
10.7.3 Group By 操作 / 298
10.7.4 合并小文件 / 298
10.8 Hive 内置操作符与函数 / 298
10.8.1 字符串函数 / 299
10.8.2 集合统计函数 / 299
10.8.3 复合类型操作 / 301
10.9 Hive 用户自定义函数接口 / 302
10.9.1 用户自定义函数 UDF / 302
10.9.2 用户自定义聚合函数 UDAF / 304
10.10 Hive 的权限控制 / 306
10.10.1 角色的创建和删除 / 307
10.10.2 角色的授权和撤销 / 307
10.10.3 超级管理员权限 / 309
10.11 应用实例：使用 JDBC 开发 Hive 程序 / 311
10.11.1 准备测试数据 / 311
10.11.2 代码实现 / 311
10.12 本章小结 / 313

第 11 章 开源数据库 HBase / 314

11.1 认识 HBase / 314
11.1.1 HBase 的特点 / 314
11.1.2 HBase 访问接口 / 314
11.1.3 HBase 存储结构 / 315
11.1.4 HBase 存储格式 / 317
11.2 HBase 设计 / 319
11.2.1 逻辑视图 / 320

11.2.2 框架结构及流程 / 321
11.2.3 Table 和 Region 的关系 / 323
11.2.4 -ROOT- 表和 .META. 表 / 323
11.3 关键算法和流程 / 324
11.3.1 Region 定位 / 324
11.3.2 读写过程 / 325
11.3.3 Region 分配 / 327
11.3.4 Region Server 上线和下线 / 327
11.3.5 Master 上线和下线 / 327
11.4 HBase 安装 / 328
11.4.1 HBase 单机安装 / 328
11.4.2 HBase 分布式安装 / 330
11.5 HBase 的 Shell 操作 / 334
11.5.1 一般操作 / 334
11.5.2 DDL 操作 / 335
11.5.3 DML 操作 / 337
11.5.4 HBase Shell 脚本 / 339
11.6 HBase 客户端 / 340
11.6.1 Java API 交互 / 340
11.6.2 MapReduce 操作 HBase / 344
11.6.3 向 HBase 中写入数据 / 348
11.6.4 读取 HBase 中的数据 / 350
11.6.5 Avro、REST 和 Thrift 接口 / 352
11.7 本章小结 / 353

第 12 章 Mahout 算法 / 354

12.1 Mahout 的使用 / 354
12.1.1 安装 Mahout / 354
12.1.2 运行一个 Mahout 案例 / 354
12.2 Mahout 数据表示 / 356
12.2.1 偏好 Perference 类 / 356
12.2.2 数据模型 DataModel 类 / 357
12.2.3 Mahout 链接 MySQL 数据库 / 358
12.3 认识 Taste 框架 / 360
12.4 Mahout 推荐器 / 361
12.4.1 基于用户的推荐器 / 361
12.4.2 基于项目的推荐器 / 362
12.4.3 Slope One 推荐策略 / 363
12.5 推荐系统 / 365
12.5.1 个性化推荐 / 365
12.5.2 商品推荐系统案例 / 366
12.6 本章小结 / 370

附录 A Hive 内置操作符与函数 / 371

附录 B HBase 默认配置解释 / 392

附录 C Hadoop 三个配置文件的参数含义说明 / 398

第1章　Hadoop 概述

Hadoop 是一个开发和运行处理大规模数据的软件平台，是 Apache 的一个用 Java 语言实现开源软件框架，实现在大量计算机组成的集群中对海量数据进行分布式计算。

1.1　Hadoop 起源

Hadoop 框架中最核心设计就是：HDFS 和 MapReduce。HDFS 提供了海量数据的存储，MapReduce 提供了对数据的计算。

1.1.1　Google 与 Hadoop 模块

Google 的数据中心使用廉价的 Linux PC 机组成集群，在上面运行各种应用。即使是分布式开发的新手也可以迅速使用 Google 的基础设施。Hadoop 核心组件与 Google 对应的组件对应关系如表 1-1 所示。

表 1-1　Google 与 Hadoop 对应模块

Google	功能描述	对应 Hadoop 模块
GFS	分布式文件系统（Google File System），隐藏下层负载均衡、冗余复制等细节，对上层程序提供一个统一的文件系统 API 接口。Google 根据自己的需求对它进行了特别优化，包括超大文件的访问、读操作比例远超过写操作、PC 机极易发生故障造成节点失效等。GFS 把文件分成 64MB 的块，分布在集群的机器上，使用 Linux 的文件系统存放，同时每块文件至少有 3 份以上的冗余。中心是一个 Master 节点，根据文件索引找寻文件块	HDFS
MapReduce	Google 发现大多数分布式运算可以抽象为 MapReduce 操作。Map 是把输入 Input 分解成中间的 Key-Value 对，Reduce 把 Key-Value 合成最终输出 Output。这两个函数由程序员提供给系统，下层设施把 Map 和 Reduce 操作分布在集群上运行，并把结果存储在 GFS 上	MapReduce
BigTable	一个大型的分布式数据库，这个数据库不是关系式的数据库。像它的名字一样，就是一个巨大的表格，用来存储结构化的数据	HBase

1.1.2　为什么会有 Hadoop

随着互联网快速的发展，产生的日志数量级的增加，大量的日志给公司带来了很大的挑战。如日志存储问题、海量日志分析的效率问题、成本问题等。

下面我们来分析个问题。

一般网站把用户的访问行为记录以 Apache 日志的形式记录下来，这些日志中包含了用

户访问网站的所有信息，下面列举关键的信息，关键字段如下。

客服端 IP	用户标示	访问时间	访问 URL	关联的 URL	状态	流量	代理
client_ip	user_ip	access_time	url	refrence	status	page_size	agent

因为需要统一对数据进行离线计算，所以常常把它们全部移到同一个地方，如 Oracle 数据库等，每天产生的日志量大概计算一下，如下所示。

- 网站请求数：1000 万条 / 天
- 每天日志大小：450 字节 / 行 × 1000 万条 = 4.2 GB
- 日志存储周期：2 年

一天产生 4.5 GB 日志，2 年需要 4.2 GB × 2 × 365 = 3.0 TB。

怎么来解决 3.0 TB 的数据备份和容错的问题？解决方案如下

1）为了方便系统命令查看日志，不压缩，总共需要 3.0 TB 空间，刚好有一些 2U 的服务器，每台 1 TB 的磁盘空间。

2）为了避免系统盘坏掉影响服务器使用，对系统盘做 Raid1。

3）为了避免其他存放数据的盘坏掉导致数据无法恢复，对剩下的盘做 Raid5。

所有的数据都汇聚到这几台 LogBackUp 服务器上了。有了 LogBackUp 服务器，离线统计就可以全部在这些服务器上进行了。在这套架构上，用 wc、grep、sort、uniq、awk、sed 等系统命令，完成了很多的统计需求，比如统计访问频率较高的 client_ip，某个新上线的页面的 reference 主要是哪些网站。

当公司业务迅猛发展，网站流量爆发增长，日志量也就成指数级增长，日志对于一个互联网公司来说是非常重要的，互联网公司可以通过分析日志来获取用户的行为，如推荐系统、淘宝的用户买卖行为分析等。假如现在的日志量计算如下所示。

- 日志总行数：10 亿 / 天
- 每天日志大小：450 字节 / 行 × 10 亿 = 420 GB
- 日志种类：5 种

1 天产生 420 GB 日志，2 年需要 420 GB × 2 × 365 = 300 TB，这么大的数据量怎么来存储和分析？

1.1.3 Hadoop 版本介绍

最新发布的 Apache Hadoop 版本如图 1-1 所示。一些 Hadoop 入门者会问：这个版本的功能有哪些？基于哪个版本？后续的版本是什么？要解释这一点，我们应该从 Apache 项目发布的一些基本知识开始分析。一般来说，Apache 项目的新功能在主干（trunk）代码上开发。有时候，很大的特性也会有自己的开发分支（branch），它们期望后续会并入 trunk。新功能通常是在 trunk 发布之前就有的，一般质量或稳定性没有太大保证。候选的分支会定期从主干分支上分离出来发布。一旦一个候选分支发布，它通常停止获得新的功能。如果有 bug 修复，经过投票后，会针对这个特定的分支再发布一个新版本。社区的任何成员可以创建一个版本分支，并可随意命名。

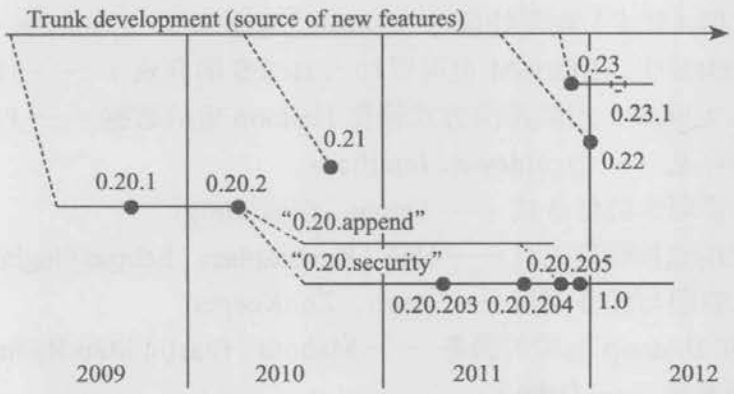


图 1-1 Hadoop 版本图[1]

注意 目前使用比较多的版本是 v-0.20.2、v-1.0.3 或 v-1.0.4。Hadoop 2.0 版本还处于测试阶段。目前 Hadoop 只能在 Linux 环境下运行。JDK 版本要在 1.5 以上。

1.2 Hadoop 生态系统

当下 Hadoop 已经成长为一个庞大的体系，只要和海量数据相关的领域，都有 Hadoop 的身影。图 1-2 是一个 Hadoop 生态系统的图谱，详细列举了在 Hadoop 这个生态系统中出现的各种数据工具。

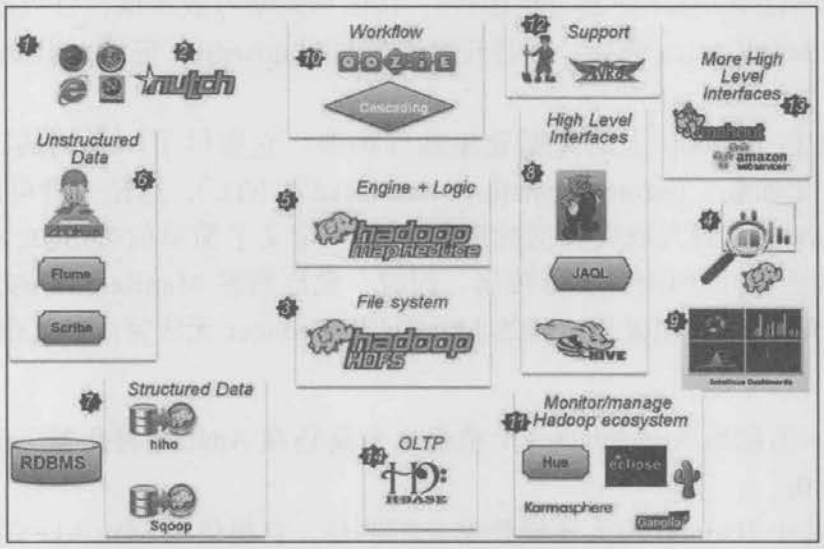


图 1-2 Hadoop 生态系统

这一切，都起源自 Web 数据爆炸时代的来临。Hadoop 生态系统的功能以及对应的开源工具说明如下。

1）海量数据怎么存，当然是用分布式文件系统 —— HDFS。

2）数据怎么用呢，分析、处理 MapReduce 框架，让你通过编写代码来实现对大数据的分析工作。

3）非结构化数据（日志）收集处理 —— Fuse、WebDAV、Chukwa、Flume 和 Scribe。
4）数据导入 HDFS 中，RDBSM 也可以加入 HDFS 的狂欢了 —— HIHO、Sqoop。
5）MapReduce 太麻烦，用熟悉的方式操作 Hadoop 里的数据 —— Pig、Hive、Jaql。
6）让你的数据可见 —— Drilldown、Intellicus。
7）用高级语言管理你的任务流 —— Oozie、Cascading。
8）Hadoop 自己的监控管理工具 —— Hue、Karmasphere、Eclipse Plugin、Cacti、Ganglia。
9）数据序列化处理与任务调度 —— Avro、ZooKeeper。
10）更多构建在 Hadoop 上层的服务 —— Mahout、Elastic Map Reduce。
11）OLTP 存储系统 —— HBase。
12）基于 Hadoop 的实时分析—— Impala。

1.3 Hadoop 常用项目介绍

随着 Hadoop 的使用越来越多，基于 Hadoop 开发的工具也越来越多，下面介绍常用的一些工具。

1. Hive

Hive 是 Facebook 捐献给 Apache 的一个项目，Hive 是基于 Hadoop 的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供完整的 SQL 查询功能，可以将 SQL 语句转换为 MapReduce 任务进行运行。 其优点是学习成本低，可以通过类 SQL 语句快速实现简单的 MapReduce 统计，不必开发专门的 MapReduce 应用，十分适合数据仓库的统计分析。

Hive 是建立在 Hadoop 上的数据仓库基础构架。它提供了一系列的工具，可以用来进行数据提取转化加载（Extract Transform and Load ，ETL），这是一种可以存储、查询和分析存储在 Hadoop 中的大规模数据的机制。Hive 定义了简单的类 SQL 查询语言，称为 HQL，它允许熟悉 SQL 的用户查询数据。同时，允许熟悉 MapReduce 的开发者开发自定义的 Mapper 和 Reducer，以处理内建的 Mapper 和 Reducer 无法完成的复杂的分析工作。

2. Pig

Pig 是 Yahoo 捐献给 Apache 的一个项目，目前还在 Apache 孵化器（incubator）阶段，目前版本是 v0.5.0。

Pig 是一个基于 Hadoop 的大规模数据分析平台，它提供的 SQL-like 语言叫 Pig Latin，该语言的编译器会把类 SQL 的数据分析请求转换为一系列经过优化处理的 MapReduce 运算。Pig 为复杂的海量数据并行计算提供了一个简易的操作和编程接口。

3. Mahout

Mahout 是 ASF（Apache Software Foundation）旗下的一个开源项目，提供一些可扩展的机器学习领域经典算法的实现，旨在帮助开发人员更加方便快捷地创建智能应用程序。Apache Mahout 项目已经发展第四个年头，目前已经有了三个公共发行版本。Mahout 包含

许多实现，包括集群、分类、推荐过滤、频繁子项挖掘。此外，通过使用 Apache Hadoop 库，Mahout 可以有效地扩展到云中。

Mahout 算法库提供了以下一些功能。

- 支持 MapReduce 的集群实现包括 K-means、模糊 K-means、Canopy、Dirichlet 和 Mean-Shift；
- Distributed Naive Bayes 和 Complementary Naive Bayes 分类实现；
- 针对进化编程的分布式适用性功能；
- Matrix 和矢量库等。

4. Flume

Flume 是 Cloudera 提供的一个高可用的、高可靠的、分布式的海量日志采集、聚合和传输的系统。Flume 支持在日志系统中定制各类数据发送方，用于收集数据；同时，Flume 提供对数据进行简单处理，并写到各种数据接受方（可定制）的能力。

Flume 提供了从 console（控制台）、RPC（Thrift-RPC）、text（文件）、tail（UNIX tail）、syslog（syslog 日志系统，支持 TCP 和 UDP 等两种模式）、exec（命令执行）等数据源上收集数据的能力。

Flume 采用多 Master 的方式。为了保证配置数据的一致性，Flume 引入 ZooKeeper，用于保存配置数据。ZooKeeper 本身可保证配置数据的一致性和高可用，另外，在配置数据发生变化时，ZooKeeper 可以通知 Flume Master 节点。Flume Master 间使用 gossip 协议同步数据。

Apache 孵化的 Flume 又称 FlumeNG。

5. Sqoop

Sqoop 是一个用来将 Hadoop 和关系型数据库中的数据相互转移的工具，可以将一个关系型数据库（如 MySQL、Oracle、Postgres 等）中的数据导入 Hadoop 的 HDFS 中，也可以将 HDFS 的数据导入关系型数据库中。

6. Oozie

Oozie 是一种 Java Web 应用程序，它运行在 Java Servlet 容器（即 Tomcat）中，并使用数据库来存储以下内容：

- 工作流定义；
- 当前运行的工作流实例，包括实例的状态和变量。

Oozie 工作流是放置在控制依赖 DAG（Direct Acyclic Graph，有向无环图）中的一组动作（如 Hadoop 的 Map/Reduce 作业、Pig 作业等），其中指定了动作执行的顺序。我们会使用 hPDL（一种 XML 流程定义语言）来描述这个图。

hPDL 是一种很简洁的语言，只会使用少数流程控制和动作节点。控制节点会定义执行的流程，并包含工作流的起点和终点（start、end 和 fail 节点）以及控制工作流执行路径的机制（decision、fork 和 join 节点）。动作节点是一些机制，通过它们工作流会触发执行计算或者处理任务。Oozie 为以下类型的动作提供支持：Hadoop MapReduce、Hadoop 文件系统、

Pig、Java 和 Oozie 的子工作流（SSH 动作已经从 Oozie Schema 0.2 之后的版本中移除了）。

所有由动作节点触发的计算和处理任务都不在 Oozie 之中——它们是由 Hadoop 的 Map/Reduce 框架执行的。这种方法让 Oozie 可以支持现存的 Hadoop 用于负载平衡、灾难恢复的机制。这些任务主要是异步执行的（只有文件系统动作例外，它是同步处理的）。这意味着对于大多数工作流动作触发的计算或处理任务的类型来说，在工作流操作转换到工作流的下一个节点之前都需要等待，直到计算或处理任务结束了之后才能够继续。Oozie 可以通过两种不同的方式来检测计算或处理任务是否完成，也就是回调和轮询。当 Oozie 启动了计算或处理任务的时候，它会为任务提供唯一的回调 URL，然后任务会在完成的时候发送通知给特定的 URL。在任务无法触发回调 URL 的情况下（可能是因为任何原因，比方说网络闪断），或者当任务的类型无法在完成时触发回调 URL 的时候，Oozie 有一种机制，可以对计算或处理任务进行轮询，从而保证能够完成任务。

Oozie 工作流可以参数化（在工作流定义中使用像 ${inputDir} 之类的变量）。在提交工作流操作的时候，我们必须提供参数值。如果经过合适地参数化（比方说，使用不同的输出目录），那么多个同样的工作流操作可以并发。

一些工作流是根据需要触发的，但是大多数情况下，我们有必要基于一定的时间段和（或）数据可用性和（或）外部事件来运行它们。Oozie 协调系统（Coordinator System）让用户可以基于这些参数来定义工作流执行计划。Oozie 协调程序让我们可以以谓词的方式对工作流执行触发器进行建模，那可以指向数据、事件和（或）外部事件。工作流作业会在谓词得到满足的时候启动。

有时我们还需要连接定时运行但时间间隔不同的工作流操作。多个随后运行的工作流的输出会成为下一个工作流的输入。把这些工作流连接在一起，会让系统把它作为数据应用的管道来引用。Oozie 协调程序支持创建这样的数据应用管道。

7．ZooKeeper

ZooKeeper 是 Hadoop 的正式子项目，它是一个针对大型分布式系统的可靠协调系统。提供的功能包括：配置维护、名字服务、分布式同步、组服务等。ZooKeeper 的目标就是封装好复杂易出错的关键服务，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

8．Impala

Impala 采用与 Hive 相同的元数据、SQL 语法、ODBC 驱动程序和用户接口（Hue Beeswax），这样在使用 CDH 产品时，批处理和实时查询的平台是统一的。目前支持的文件格式是文本文件和 Sequence Files（可以压缩为 Snappy、GZIP 和 BZIP，前者性能最好）。其他格式如 Avro、RCFile、LZO 文本和 Doug Cutting 的 Trevni 将在正式版中支持，官方测试速度是 Hive 的 3 ～ 90 倍。

1.4 Hadoop 在国内的应用

图 1-3 由 Hadoop 技术论坛提供，数据的绝对值参考意义不大，主要是看各城市间的相

对数据。北京、深圳和杭州位列前三甲，分析主要原因是：北京有淘宝和百度，深圳有腾讯，杭州有网易等。互联网公司是 Hadoop 在国内的主要使用力量。淘宝是在国内最先使用 Hadoop 的公司之一，而百度赞助了 HyperTable 的开发，另外北京研究 Hadoop 的高校多，所以北京是 Hadoop 方面研究和应用需求最高的城市。

1.	Beijing	656	30.51%
2.	Shenzhen	332	15.44%
3.	Hangzhou	241	11.21%
4.	Shanghai	144	6.70%
5.	Guangzhou	144	6.70
6.	Nanjing	67	3.12%
7.	Chendu	64	2.98%
8.	Changsha	59	2.74%
9.	Shenyang	48	2.23%
10.	Wuhan	40	1.86%

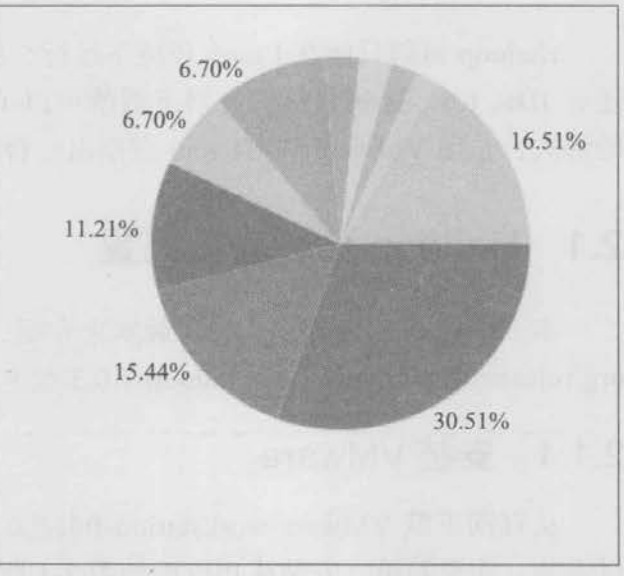


图 1-3 Hadoop 使用统计图

目前，Hadoop 在国内深得互联网企业的青睐，很多互联网公司都在使用 Hadoop 来实现公司的核心业务，如搜索、推荐等。

- 奇虎 360：Hadoop 存储软件管家中的软件，使用 CDN 技术将用户请求引到最近的 Hadoop 集群并进行下载。
- 京东、百度：存储、分析日志、数据挖掘和机器学习（主要是推荐系统）。
- 广告类公司：存储日志，通过协调过滤算法为客户推荐广告。
- 华为：云计算平台。
- 学校：学生上网与社会行为分析。
- 淘宝、阿里巴巴：国内使用 Hadoop 最深入的公司，整个淘宝和阿里巴巴都是数据驱动的。

1.5 本章小结

互联网时代什么最重要——数据！大数据时代什么工具最流行——Hadoop！本章主要从大体上介绍 Hadoop，让读者对 Hadoop 有个大体的印象。接下来我们开始深入介绍 Hadoop 的相关知识，以及 Hadoop 的安装、使用和开发。

第2章　Hadoop 安装

Hadoop 目前只能在 Linux 环境下运行，所以我们要准备 VMware 8.0.2 和 Ubuntu 10.04 还有 JDK 1.6，这些直接去官网下载就可以了。本书中，Hadoop 分布式安装使用了虚拟机的形式，采用 VMware 和 Ubuntu 虚拟出三台设备，然后在这三台上面搭建 Hadoop 集群。

2.1　Hadoop 环境安装配置

本书内容基于 Hadoop-1.0.3 版本来介绍，首先，从 Apache 官网（http://hadoop.apache.org/releases.html#Ne）下载 Hadoop-1.0.3 版本。

2.1.1　安装 VMware

从官网下载 VMware-workstation-full-8.0.2-591240.exe 文件，一直点击“下一步”就可以安装，非常简单。安装成功以后如图 2-1 所示。

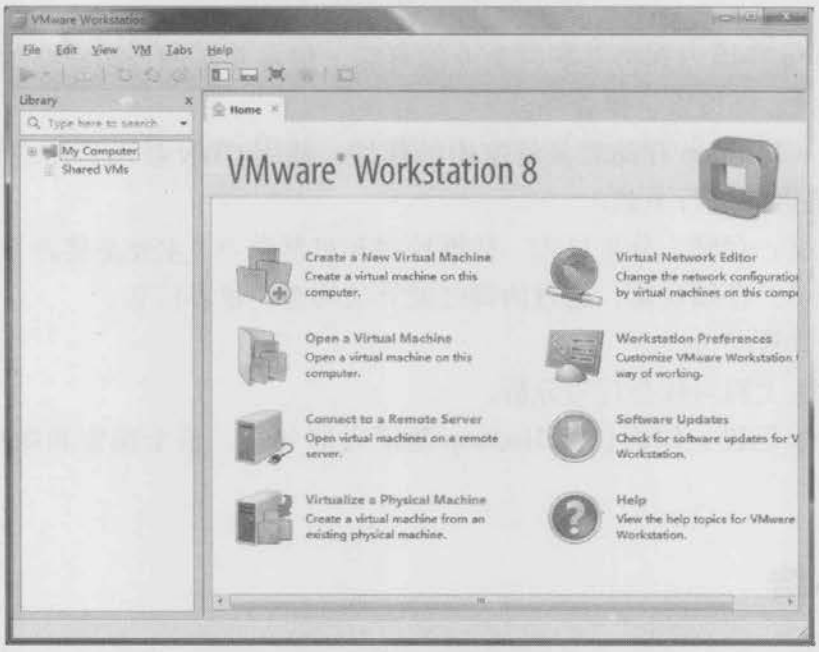


图 2-1　VMware 运行图

2.1.2　安装 Ubuntu

Ubuntu（友帮拓）是一个以桌面应用为主的 Linux 操作系统，其名称来自非洲的

“ubuntu”一词，意思是“人性”、“我的存在是因为大家的存在”，类似“仁爱”思想。

1. 获取 Ubuntu 发行版本

Ubuntu 基于 Debian 发行版和 GNOME 桌面环境，与 Debian 的不同在于它每 6 个月会发布一个新版本。Ubuntu 的目标在于为一般用户提供一个最新的、同时又相当稳定的、主要由自由软件构建而成的操作系统。Ubuntu 具有庞大的社区力量，用户可以方便地从社区获得帮助。2013 年 1 月 3 日，Ubuntu 正式发布面向智能手机的移动操作系统。

本书中为什么使用 Ubuntu 系统来安装 Hadoop 集群？原因在于 Ubuntu 系统的完全免费，而且使用非常多。安装 Ubuntu 系统的步骤如下。

1）从镜像站点上下载 ISO 的镜像文件。

- 中文官网：http://www.ubuntu.com.cn
- 英文官网：http://www.ubuntu.com

2）有时我们在网上下载的压缩包会有损坏，可以使用 md5 来验证下载的 ISO 镜像文件的完整性，使用如下代码。

```
#md5sum ubuntu-10.04-desktop-i386.iso
```

2. 创建虚拟机

启动 VMware 以后，会进入如图 2-2 所示的对话框，用户可打开一个已有的虚拟机或创建一个新的虚拟机等操作。

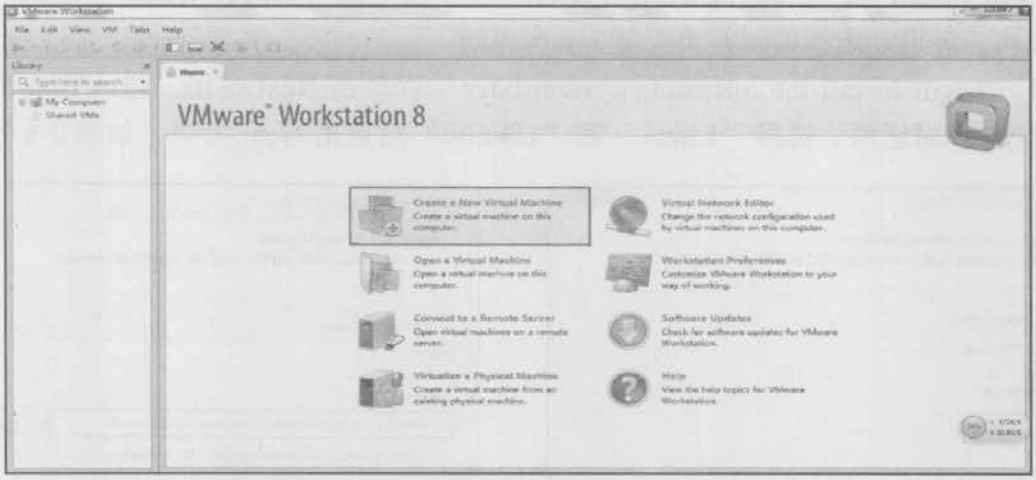


图 2-2 VMware 创建虚拟机对话框

双击 Create a New Virtual Machine 以后，会看到图 2-3 所示的对话框。

还有一种方式来创建虚拟机，步骤是 VMware 的工具栏 File New → Virtual Machine，选中以后会直接出现图 2-3 的对话框。Machine Wizard 有两种安装模式。

- Typical 安装：默认的安装，如果对 VMware 不熟悉的同学可以使用这种安装模式，简单而且不容易出错。
- Custom 安装：用户自动安装，这种安装可以根据自己的需求来定制化，但要求比较高。

选择默认的 Typical 安装后单击 Next 按钮，将进入加载 ISO 文件对话框，如图 2-4 所示。

图 2-3 虚拟机安装方式

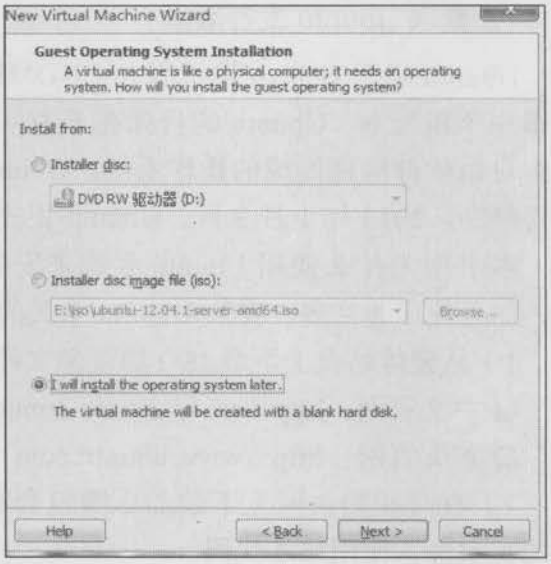


图 2-4 加载 ISO 文件对话框

在加载 ISO 对话框里有三种来源。

- Installer disk：这需要有光盘才能安装，选择光盘的位置即可。
- Installer disc image file(iso)：这就是 ISO 镜像文件的安装，找到 ISO 文件存放的位置。
- I will install the operating system later：选中这个以后就代表先创建一个空的磁盘虚拟机，等启动虚拟的时候再安装 Linux 系统。

选择“I will install the operating system later”后单击 Next 按钮，进入图 2-5 所示的 Linux 系统选择对话框。选择“Linux”和“Ubuntu”后点击 Nent 按钮，如图 2-6 所示。

图 2-5 Linux 版本选择

图 2-6 Ubuntu 名称设置

由于是在 VMware 里面安装的 Linux 系统，所以 Guest operating system 应该选择 Linux，Version 表示要安装哪个 Linux 版本，有 Redhat、Centos、Ubuntu 等，选择 Ubuntu。

选择完成后单击 Next 按钮，如图 2-6 所示，进入虚拟机名称设置对话框。Ubuntu 名称设置对话框有两个属性。

- Virtual machine name：创建虚拟机的名称，这里设置为 Ubuntu 1。
- Location：选择虚拟机安装的路径。Hadoop 文件夹所在的磁盘空间一定要够用，大概需要 20 GB。

设置好以后选择 Next 按钮，进入虚拟机的磁盘设置对话框，如图 2-7 所示。虚拟机磁盘设置有三个选项。

- Maximum disk size：最大磁盘空间大小，可以根据实际的磁盘空间来设置，一般设置为 20 GB。
- Store virtual disk as a single file：设置虚拟机的磁盘只有一个文件，相当于 Windows 系统只有 C 盘。
- Split virtual disk in to multiple file：设置虚拟机的磁盘有多个文件，相当于 Windows 系统有 C、D、E 盘等。

设置完以后单击 Next 按钮，将进入如图 2-8 所示的确认对话框。

图 2-7 虚拟机磁盘设置

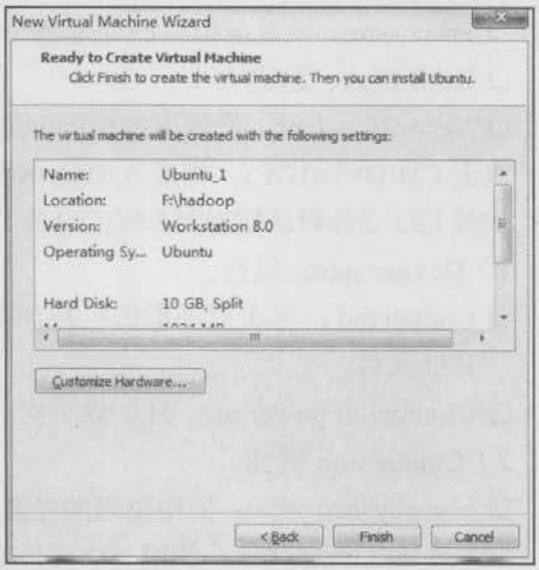


图 2-8 虚拟机设置确认

这个对话框显示了我们设置虚拟机的一些信息，如虚拟机名称、安装路径、虚拟机版本、磁盘空间等信息，如果不对还可以选择 Back 重新设置。

确认以后，选择单击 Finish 按钮，到此我们创建一个名为 Ubuntu_1 的虚拟机就成功了，这相当于一个没有操作系统的机器。

3. 安装 Ubuntu 操作系统

接下来启动刚才创建的 Ubuntu_1 虚拟机，如图 2-9 所示，安装 Ubuntu 操作系统。

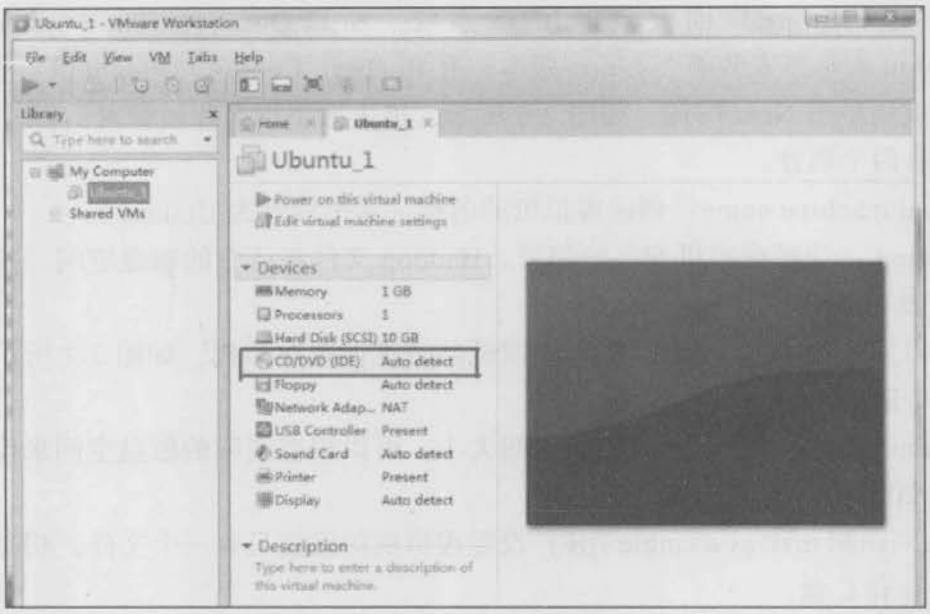


图 2-9 启动虚拟机 Ubuntu_1

启动虚拟机之前，我们可以做一些设置。

- Memory：设置虚拟机的内存大小。
- Processor：设置虚拟机的处理器个数。
- Hard Disk：磁盘大小。
- Network Adapt：设置虚拟机的网络连接方式。

双击 CD/DVD(IDE)，将进入加载 ISO 镜像文件对话框，如图 2-10 所示。

加载 ISO 文件对话框有两个属性设置。

1）Device status 属性。

- Connected：每次启动虚拟机都加载 ISO 文件。
- Connect at power on：只加载一次。

2）Connection 属性。

- Use physical drive：使用物理的光盘；
- Use ISO image file：使用 ISO 文件安装，选择 ISO 文件所在的路径。

选择单击 OK 按钮，回到虚拟机启动的界面，如图 2-11 所示。

单击 Power On this Virtual machine 进入安装操作系统界面，如图 2-12 所示。大概 1 分钟，会进入 Ubuntu 操作系统语言配置对话框。

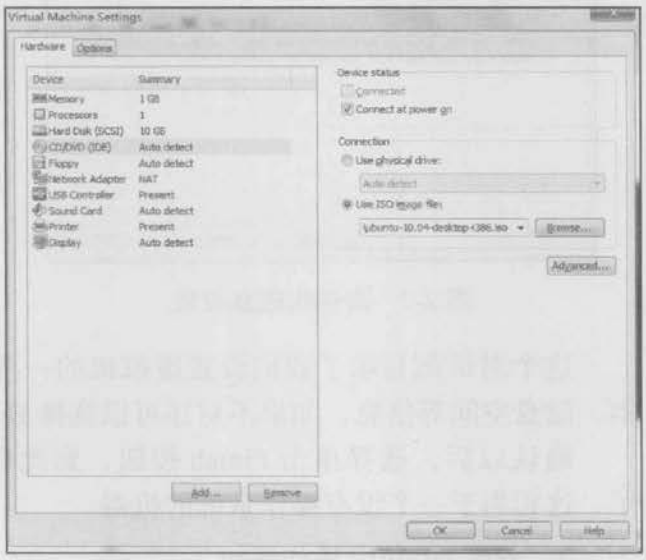


图 2-10 加载 ISO 镜像文件

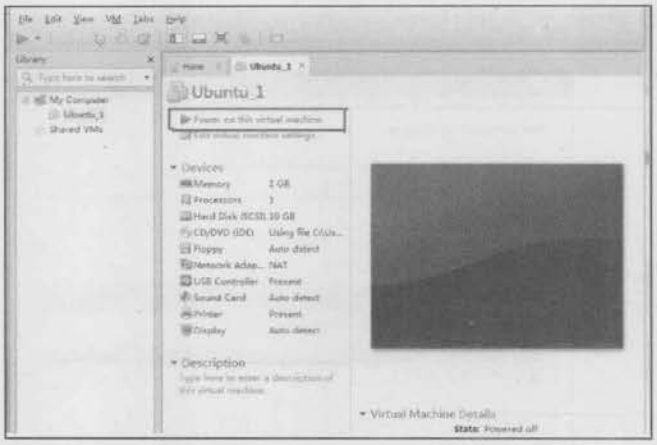


图 2-11 虚拟机启动界面

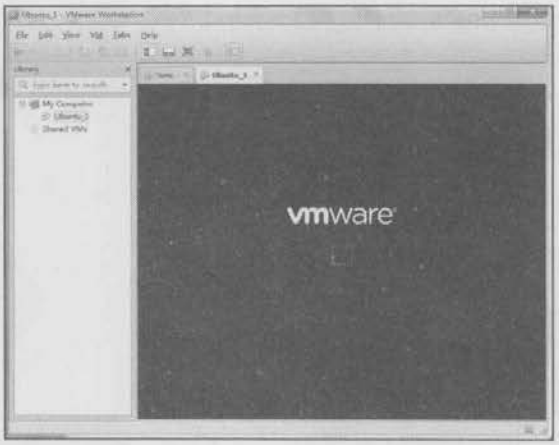


图 2-12 操作系统的安装

如图 2-13 所示，选择 English，当然也有中文的，建议使用英文，因为这样对学习有帮助，这个大家可以根据自己的喜好来选择。

选择好以后，双击 Install Ubuntu 10.04 LTS，进入时区设置对话框，如图 2-14 所示。

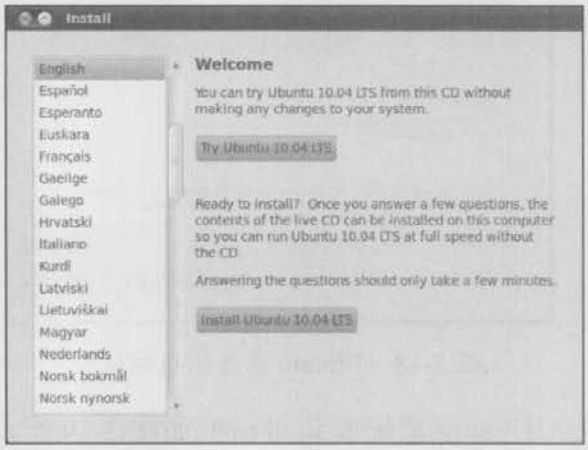


图 2-13 Ubuntu 系统语言设置

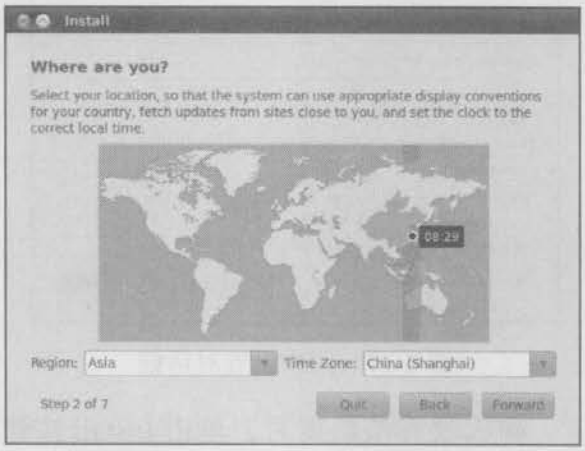


图 2-14 Ubuntu 时区设置

时区设置有两个属性。

- Region：地理区域，这里选择“Asia”。
- Time Zone：时区，这里选择“China Shanghai”。

选择好以后，单击 Forward 按钮，进入输入法设置对话框，如图 2-15 所示。

如果选择安装的操作系统为英文环境，在这里我们就选择 USA，美国的输入法，没有中文输入法；如果选择的是中文操作系统，这里就有中文的输入法可以选择了。

选择好以后单击 Forward 按钮，进入磁盘设置对话框，如图 2-16 所示。这里用默认设置就可以了，单击 Forward 按钮，进入 Ubuntu 系统用户设置对话框。

如图 2-17 所示，创建一个登录 Ubuntu 系统的用户。注意，这里不是 root 用户，因为 root 用户默认没有密码。用户名、密码和机器名设置完成以后单击 Forward 按钮，进入

Ubuntu 系统信息设置确认对话框，如图 2-18 所示。

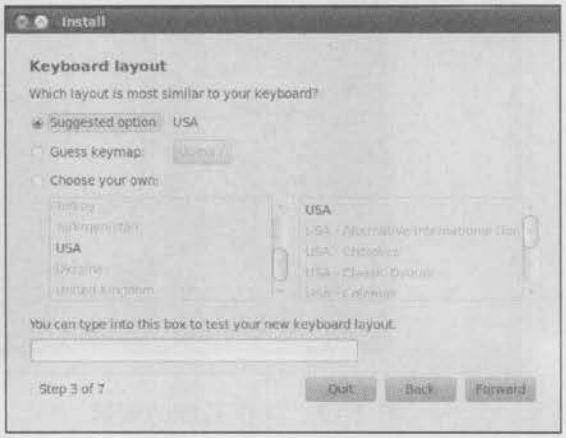


图 2-15 Ubuntu 输入法设置

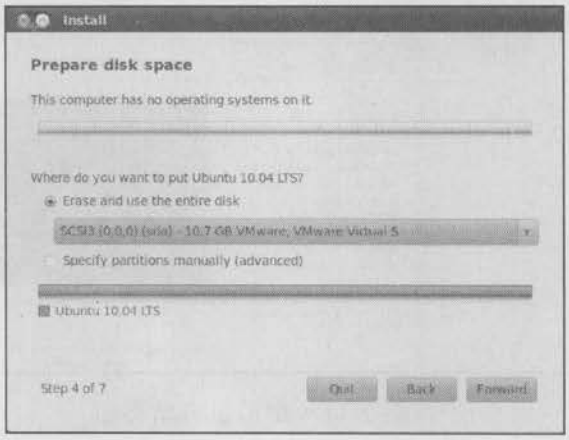


图 2-16 磁盘设置对话框

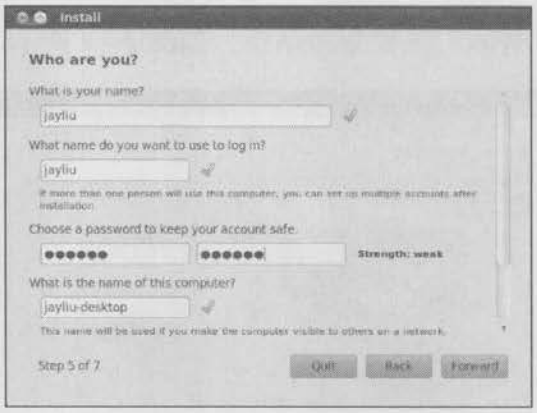


图 2-17 用户设置对话框

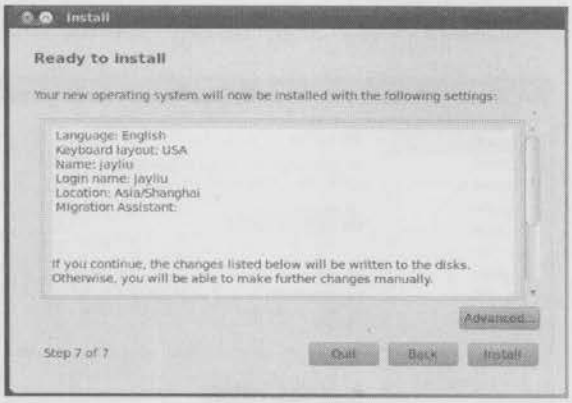


图 2-18 Ubuntu 系统信息确认

确认没有错误以后，单击 Install 按钮，进入 Ubuntu 系统安装对话框如图 2-19 所示。这步根据机器的配置不同需要的时间也不一样，快的大概需要 15 分钟。安装完成以后就会进入 Ubuntu 系统的登录对话框，如图 2-20 所示。

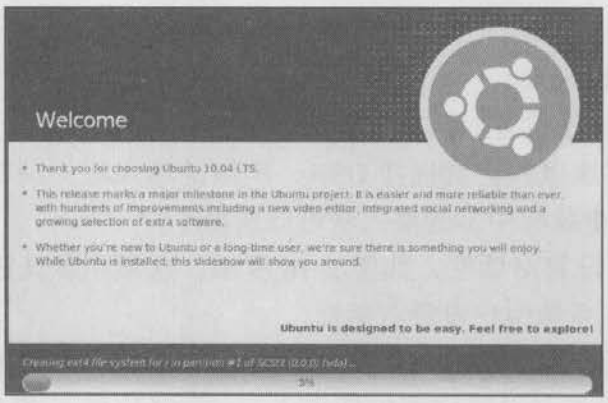


图 2-19 Ubuntu 系统安装界面

图 2-20 Ubuntu 系统登录

这里登录用户就是前面设置的用户名，输入刚才设置的密码就可以登录了。到这里Ubuntu 系统的安装就成功了。按照同样的方式再创建两个虚拟机，分别为 Ubuntu_2 和Ubuntu_3，安装的方法和 Ubuntu_1 一样。

2.1.3 安装 VMware Tools

VMware Tools 是 VMware 虚拟机中自带的一种增强工具，相当于 VirtualBox 中的增强功能（Sun VirtualBox Guest Additions），是 VMware 提供的增强虚拟显卡和硬盘性能，以及同步虚拟机与主机时钟的驱动程序。

只有在 VMware 虚拟机中安装了 VMware Tools，才能实现主机与虚拟机之间的文件共享，同时支持自由拖曳的功能，鼠标也可以在虚拟机与主机之前自由移动（不再需要按 Ctrl + Alt），且虚拟机屏幕也可以实现全屏化。

VMware Tools 的安装步骤如下。

1）启动并进入 Linux 系统。

2）选择虚拟机“ vm”菜单中的“虚拟机 / 安装 VMware-Tools(install VMware-tolls)”，此时就会有把 VMware-tools 文件映像到 CD-ROM 中。

3）把“VMwareTools-6.0.2-59824.tar.gz”文件复制到自己需要的位置。

4）解压，先要进入“VMwareTools-6.0.2-59824.tar.gz”存放目录，执行以下命令。

```
# tar -zxvf VmwareTools-6.0.2-59824.tar.gz
```

5）进入 /home/tsm/Tools/vmware-tools-distrib 目录。

```
cd /home/tsm/Tools/vmware-tools-distrib
```

6）输入“ ./vmware-install.pl”进行安装，在安装过程中根据提示进行选择，其实一直按回车键即可完成安装。

2.1.4 安装 JDK

JDK（Java Development Kit）是 Sun Microsystems 针对 Java 开发的产品。自从 Java 推出以来，JDK 已经成为使用最广泛的 Java SDK。JDK 是整个 Java 的核心，包括 Java 运行环境、Java 工具和 Java 基础类库，掌握 JDK 是学好 Java 的第一步。而专门运行在 x86 平台的 JRocket 在服务器端运行效率比 Sun JDK 好很多。从 Sun JDK 5.0 开始，提供了泛型等非常实用的功能，其版本也不断更新，运行效率得到了非常大的提高。

JDK 只要是 1.5 版本以上就可以，本书使用的是 JDK 1.6。安装过程如下。

1）下载 JDK 1.6，这里存放的目录是 /opt/tools/jdk-6u25-linux-i586.bin。

2）进入 /opt/tools 目录，给文件赋值权限，命令如下。

```
cd /opt/tools
chmod u+x jdk-6u25-linux-i586.bin
```

3）执行下面的命令，执行完命令后，会在 /opt/tools 目录下生成 jdk1.6.0_25 文件夹。

```
./jdk-6u25-linux-i586.bin
```

4）配置 JDK 的环境命令如下。

```
sudo gedit /etc/profile
export JAVA_HOME=/opt/tools/jdk1.6.0_25
export JRE_HOME=$JAVA_HOME/jre
export CLASSPATH=$JAVA_HOME/lib:$JRE_HOME/lib:$CLASSPATH
export PATH=$JAVA_HOME/bin:$JRE_HOME/bin:$PATH
```

5）再执行下面命令使配置文件生效。

```
source /etc/profile
```

或者以下命令：

```
. /etc/profile
```

6）使用以下命令查看 JDK 安装是否成功。

```
jayliu@Ubuntu1:~$ javac
```

如果显示信息如图 2-21 所示，恭喜你，JDK 安装成功了。

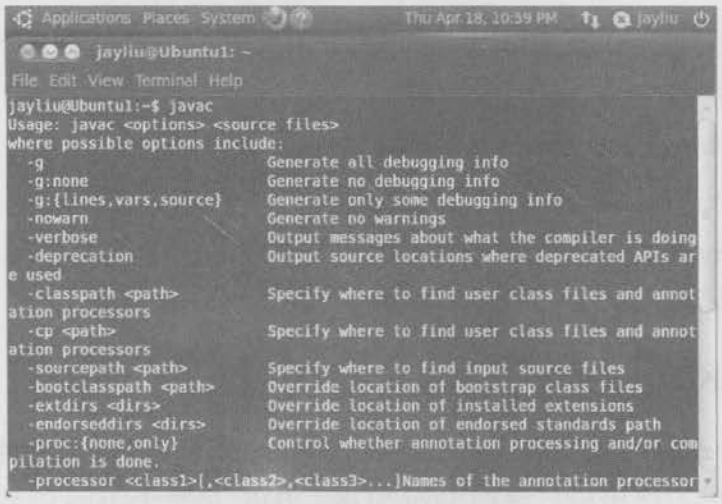


图 2-21 使用 javac 命令查看结果

2.2 Hadoop 安装模式

本节会用到刚才创建的 Ubuntu_1、Ubuntu_2、Ubuntu_3 这三台虚拟机。从 Apache 官网下载 Hadoop-1.0.3 版本。

Hadoop 的安装模式分为三种，分别是：单机模式、伪分布模式及全分布模式。默认安装是单机模式，下面分别演示三种模式的安装方法。准备就绪，下面就开始 Hadoop 的安装旅途吧！

2.2.1 单机安装

单机模式所需的系统资源是最少的，这种安装模式下，Hadoop 的 core-site.xml、mapred-site.xml、hdsf-site.xml 配置文件均为空。因此 Hadoop 保守地采取这种模式为默认安装模式。

当配置文件为空时，Hadoop 完全运行在本地，不与其他节点交互，也不使用 Hadoop 文件系统，不加载任何守护进程，该模式主要用于开发调试 MapReduce 应用程序的逻辑，不与任何守护进程交互进而避免复杂性。安装过程如下。

1）解压 Hadoop 的 TAR 文件。

```
tar -zxvf hadoop-1.0.3.tar.gz
```

2）配置 Hadoop 的环境变量。

```
sudo vi /etc/profile
export HADOOP_HOME=/opt/hadoop-1.0.3
export PATH=$HADOOP_HOME/bin:$PATH
```

3）进入解压的文件夹下的 bin 目录执行 start-all.sh 命令。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3/bin$ ./start-all.sh
```

4）使用 jps 命令查看进程是否启动成功，如图 2-22 所示。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3/conf$ jps
31088 Jps
```

图 2-22 使用 jps 命令查看

因为是单机模式，NameNode 和 JobTracker 等都没有启动，怎么知道安装是否成功呢？

5）查看 HDFS 系统，如图 2-23 所示。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3/conf$ hadoop fs -ls /
Warning: $HADOOP_HOME is deprecated.

Found 25 items
drwxr-xr-x   - root root       4096 2012-03-12 15:59 /home
drwx------   - root root      16384 2012-03-12 15:54 /lost+found
drwxr-xr-x   - root root       4096 2012-03-12 15:58 /cdrom
drwxr-xr-x   - root root       4096 2010-04-29 20:26 /var
drwxr-xr-x   - root root      12288 2013-01-07 11:31 /lib
-rw-r--r--   1 root root    7993796 2012-03-13 09:32 /initrd.img
drwxrwxrwx   - root root       4096 2013-04-18 23:20 /tmp
drwxr-xr-x   - root root       4140 2013-03-30 23:45 /dev
drwxr-xr-x   - root root       4096 2012-03-13 09:31 /doc
dr-xr-xr-x   - root root          0 2013-02-22 19:14 /proc
-rw-r--r--   1 root root    4029792 2010-04-16 21:01 /vmlinuz
drwxr-xr-x   - root root          0 2013-02-22 19:14 /sys
drwxr-xr-x   - root root       4096 2012-03-13 09:32 /mnt
-rw-r--r--   1 root root          5 2013-01-14 18:42 /zookeeper_server.pi
d
drwxr-xr-x   - root root       4096 2012-04-05 10:04 /usr
drwx------   - root root       4096 2012-10-16 08:31 /root
```

图 2-23 HDFS 单机版目录结构

大家发现什么？怎么像 Linux 的文件系统目录呢？没错，显示的就是 Linux 文件系统目录。如果出现图 2-23 所示的结果，说明你的 Hadoop 单机版安装成功了！

到目前为止我们没有对 Hadoop 的配置文件做任何修改，全是默认配置，即配置文件全是空的，如下所示。

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

<!-- Put site-specific property overrides in this file. -->
<configuration>

</configuration>
```

2.2.2 伪分布式安装

伪分布模式即“单节点集群”模式，所有的守护进程都运行在同一台机器上。这种模式下增加了代码调试功能，可以查看内存、HDFS 文件系统的输入 / 输出，以及与其他守护进程交互。core-site.xml、mapred-site.xml、hdsf-site.xml 配置文件如下。

1）修改地址解析文件 /etc/hosts，加入以下代码。

```
192.168.195.140  Ubuntu1
```

2）修改 hadoop 目录下 conf 下的 core-sitexml 文件，如下所示。

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

<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
   <name>hadoop.tmp.dir</name>
  <value>/hadoop</value>
  <description>A base for other temporary directories.</description>
</property>

<property>
 <!-- 用于 dfs 命令模块中指定默认的文件系统协议 -->
  <name>fs.default.name</name>
  <value>hdfs:// Ubuntu1:9000</value>
  <description>The name of the default file system.  A URI whose
  scheme and authority determine the FileSystem implementation.  The
  uri's scheme determines the config property (fs.SCHEME.impl) naming
   the FileSystem implementation class.  The uri's authority is used to
  determine the host, port, etc. for a filesystem.</description>
</property>
<property>

<name>dfs.name.dir</name>
<value>/hadoop/name</value>
  <description>Determines where on the local filesystem the DFS name node should store
the name table. If this is a comma-delimited list of directories then the name table is
replicated in all of the directories, for redundancy. </description>

</property>

</configuration>
```

3）修改 hadoop 目录下的 conf/ hdfs-site.xml 文件，如下所示。

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

<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
  <name>dfs.data.dir</name>
  <value>/hadoop/data</value>
   <description>Determines where on the local filesystem an DFS data node should
store its blocks. If this is a comma-delimited list of directories, then data will be
stored in all named directories, typically on different devices. Directories that do
not exist are ignored.</description>
</property>
<property>
  <name>dfs.replication</name>     // 默认 Block 副本数

   <value>1</value>
   <description>Default block replication. The actual number of replications can
be specified when the file is created. The default is used if replication is not
specified in create time.</description>
</property>
</configuration>
```

4）修改 hadoop 目录下的 conf/ mapred-site.xml 文件，如下所示。

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

<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
<name>mapred.job.tracker</name>
  <value> Ubuntu1:9001</value>
  <description>The host and port that the MapReduce job tracker runs at. If
"local", then jobs are run in-process as a single map and reduce task.</description>
</property>
</configuration>
```

在 core-site.xml 和 mapred-site.xml 文件中分别指定 NameNode 和 JobTracker 的主机名和端口号，而在 hdfs-site.xml 中指定 HDFS 的默认副本数，由于伪分布式运行在单机上，因此只有一个副本。同时在 masters 和 slavers 文件中指定 SecondaryNameNode 和从节点的主机名为 Ubuntu1，即在文件中写入“Ubuntu1”。

配置好以后，启动步骤如下。

1）第一次启动 Hadoop 要格式化，使用如下命令格式化 NameNode。

```
hadoop namenode -format
```

2）执行 start-all.sh 命令启动 Hadoop 集群。

3）使用 jps 命令查看结果，如图 2-24 所示。

```
jayliu@Ubuntu1:/opt$ jps
6919 TaskTracker
7516 Jps
6553 DataNode
6775 JobTracker
6704 SecondaryNameNode
7074 NameNode
```

图 2-24 使用 jps 查看结果

4）使用 hadoop fs–ls/ 命令查看 HDFS 系统，如图 2-25 所示。Hadoop 的伪分布式安装成功。

```
jayliu@Ubuntu1:/opt$ hadoop fs -ls /
Found 1 items
drwxr-xr-x   - jayliu supergroup          0 2013-04-19 00:59 /hadoop
```

图 2-25 HDFS 伪分布式目录结构

以上操作需要注意两点。

- 启动 Hadoop 之前必须执行 hadoop namenode –format 操作，否则报错。
- 设置的 hadoop.tmp.dir 路径必须存在，不存在会报错。

扩展阅读

Linux 系统命令拓展

1）查看 IP 地址。

```
jayliu@Ubuntu1:/opt$ ifconfig
eth1      Link encap:Ethernet  HWaddr 00:0c:29:d7:dc:52
          inet addr:192.168.195.140  Bcast:192.168.195.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fed7:dc52/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1207000 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2216586 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:88270718 (88.2 MB)  TX bytes:3325334709 (3.3 GB)
          Interrupt:19 Base address:0x2024
```

2）查看主机名。

```
jayliu@Ubuntu1:/opt$ hostname
Ubuntu1
```

2.2.3 分布式安装

我们在 Ubuntu1 虚拟机上安装了 Hadoop 的一个节点，按照上一节的步骤把 Hadoop 分别安装在 Ubuntu2 和 Ubuntu3 上。安装成功以后再接着学习看下面的内容。

1. 集群的构建和拓扑图

使用三台虚拟机来搭建 Hadoop 分布式环境，三台虚拟机的拓扑图如图 2-26 所示。

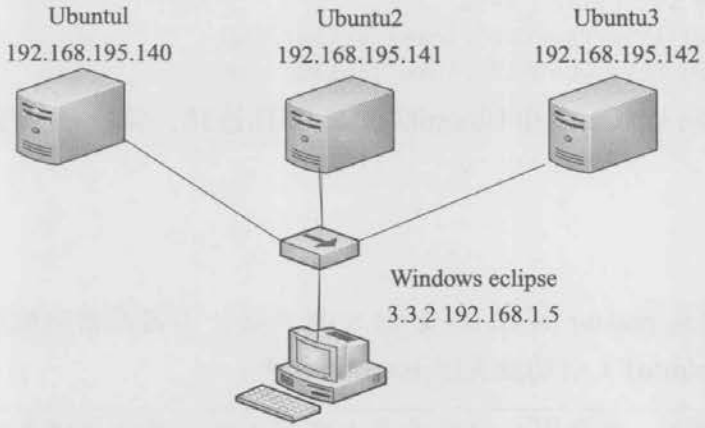


图 2-26 Hadoop 分布式拓扑图

Hadoop 集群中每个节点的角色如表 2-1 所示。

表 2-1 Hadoop 集群节点角色

主机名	Hadoop 角色	IP	Hadoop jps 命令结果	Hadoop 用户	Hadoop 安装目录
Ubuntu1	master slaves	192.168.195.140	NameNode DataNode JobTracker TaskTracker SecondaryNameNode	创建相同的用户组名：jayliu 安装 hadoop-1.0.3 时使用 jayliu 用户，并且 jayliu 的文件夹归属也是“jayliu：jayliu”	/opt/hadoop
Ubuntu2	slaves	192.168.195.141	DataNode TaskTracker		
Ubuntu3	slaves	192.168.195.142	DataNode TaskTracker		
Windows	开发测试环境	安装 JDK 与 Eclipse 本身不需要安装 Hadoop，但是需要 Hadoop 安装包下面的 JAR 包			

注意 Ubuntu1 既是 NameNode 又是 DataNode，同时也是 JobTracker。

2. SSH 配置无密码验证配置

1）登录 master 节点（即 Ubuntu1 节点）上面，切换到 jayliu 用户，在 jayliu 的 home 目录下面创建“.ssh”目录。

```
$ cd
$ mkdir .ssh
```

2）在 Master 节点（即主机 Ubuntu1）上生成密钥对。

```
$ ssh-keygen -t rsa
```

3）然后一直按 Enter 键，按默认的选项生成密钥对保存在 .ssh/id_rsa.pub 文件中。把生成的 id_rsa.pub 复制一份，命名为 authorized_keys，然后分别复制到 Ubuntu2 和 Ubuntu3 两个节点上。执行如下命令。

```
$ cd .ssh
```

```
$ cp id_rsa.pub authorized_keys
$ scp authorized_keys Ubuntu2:/home/jayliu/.ssh
$ scp authorized_keys Ubuntu3:/home/jayliu/.ssh
```

4）从 Ubuntu1 向 Ubuntu2 和 Ubuntu3 发起 SSH 连接，第一次登录时需要输入密码，以后就不需要了。

```
$ ssh Ubuntu2
$ ssh Ubuntu3
```

我们只需要配置从 master 向 slaves 发起 SSH 连接，不需要密码就可以，但这样只能在 master（即在主机 Ubuntu1）启动或关闭 Hadoop 服务。

注意 SSH 安装过程中，很多刚入门的学者会遇到麻烦，虽然按照老师的方法一步一步做下去，但是最后还是连不上。注意：SSH 的安装对“.ssh”文件夹和里面文件的权限是非常严格的，大了或小了都连不上。

1）“.ssh”是一个隐藏的文件夹，用“ll –a”命令可以看到隐藏的文件夹，“.ssh”文件夹权限必须是 700。

2）“.ssh”里面的文件权限最好是 600。

---

3. 修改 hosts 文件

需要把 Ubuntu1、Ubuntu2、Ubuntu3 的主机名和 IP 添加到三台虚拟机的 hosts 文件里面，Ubuntu1 的 hosts 文件信息如图 2-27 所示。

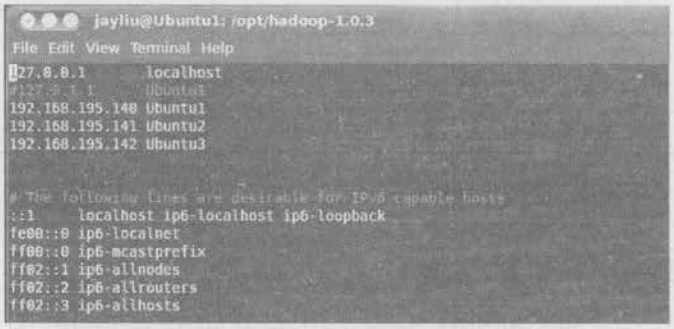


图 2-27 Ubuntu1 的 hosts 文件配置

接下来把 Ubuntu1 的 hosts 文件里的配置分别复制到 Ubuntu2 和 Ubuntu3 上。

4. Hadoop 配置文件修改

这里只需要修改 conf 下面的 master 和 slaves 文件就可以，Ubuntu1 下的 master 和 slaves 配置信息如图 2-28 和图 2-29 所示。

图 2-28 master 配置信息

图 2-29 slaves 配置信息

Ubuntu2 和 Ubuntu3 下的 master 和 slaves 文件按照 Ubuntu1 的配置就可以了！ Hadoop 的分布式安装就完成了。

5. Hadoop 启动和验证

使用 jayliu 用户启动。首先说明，Hadoop 命令和参数都是大小写敏感的，该用大写时用大写，用小写时用小写，否则会执行错误。在 master 节点上进行如下操作。

1）格式化分布式文件系统。

```
$ hadoop namenode -format
```

2）在 master 上启动 Hadoop 守护进行。

```
$ start-all.sh
```

启动日志如图 2-30 所示。

```
jayliu@Ubuntu1:/hadoop/dfs$ start-all.sh
starting namenode, logging to /opt/hadoop-0.20.2/bin/../logs/hadoop-jayliu-na
menode-Ubuntu1.out
Ubuntu2: starting datanode, logging to /opt/hadoop-0.20.2/bin/../logs/hadoop-
jayliu-datanode-Ubuntu2.out
Ubuntu1: starting datanode, logging to /opt/hadoop-0.20.2/bin/../logs/hadoop-
jayliu-datanode-Ubuntu1.out
Ubuntu3: starting datanode, logging to /opt/hadoop-0.20.2/bin/../logs/hadoop-
jayliu-datanode-Ubuntu3.out
Ubuntu1: starting secondarynamenode, logging to /opt/hadoop-0.20.2/bin/../log
s/hadoop-jayliu-secondarynamenode-Ubuntu1.out
starting jobtracker, logging to /opt/hadoop-0.20.2/bin/../logs/hadoop-jayliu-
jobtracker-Ubuntu1.out
Ubuntu1: starting tasktracker, logging to /opt/hadoop-0.20.2/bin/../logs/hado
op-jayliu-tasktracker-Ubuntu1.out
Ubuntu2: starting tasktracker, logging to /opt/hadoop-0.20.2/bin/../logs/hado
op-jayliu-tasktracker-Ubuntu2.out
Ubuntu3: starting tasktracker, logging to /opt/hadoop-0.20.2/bin/../logs/hado
op-jayliu-tasktracker-Ubuntu3.out
```

图 2-30 Hadoop 启动日志

停止 Hadoop 守护进程命令如下。

```
$ stop-all.sh
```

在 Ubuntu1 上查看运行的进程结果如图 2-31 所示。

```
jayliu@Ubuntu1:/hadoop/dfs$ jps
16481 Jps
16418 TaskTracker
15880 NameNode
16269 JobTracker
16195 SecondaryNameNode
16041 DataNode
```

图 2-31 Ubuntu1 进程

在 Ubuntu2 上查看运行的进程结果如图 2-32 所示。

在 Ubuntu3 上查看运行的进程结果如图 2-33 所示。

```
jayliu@Ubuntu2:/hadoop$ jps
6538 DataNode
6723 Jps
6667 TaskTracker
```

图 2-32 Ubuntu2 进程

```
jayliu@Ubuntu3:/hadoop$ jps
4930 Jps
4861 TaskTracker
4732 DataNode
```

图 2-33 Ubuntu3 进程

6. Hadoop WebUI 访问

访问 http://192.168.195.140:50070 如图 2-34 所示，可以查看 Hadoop 集群的节点数、NameNode 及整个分布式文件系统的状态等。

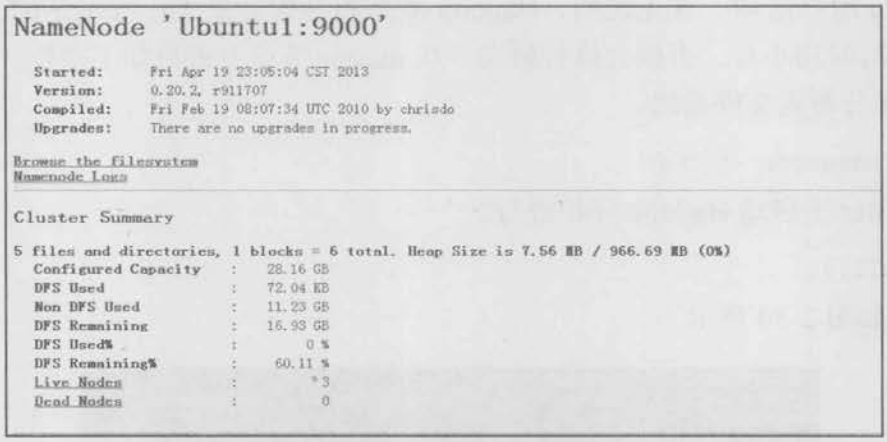


图 2-34 NameNode WebUI 访问

NameNode 的 WebUI 属性解释如下。

- Started：Hadoop 系统启动的时间。
- Version：Hadoop 的版本号。
- Compiled：Hadoop 源码编译的时间。
- Upgrades：是否有升级进程没有结束。在 Hadoop 系统升级确保成功以后需要执行 Upgrades 命令，否则下次没法升级。
- Configured Capacity：HDFS 的容量大小。
- DFS Used：HDFS 使用的空间。
- Non DFS Used：HDFS 预留的空间。
- DFS Remaining：HDFS 系统剩余空间。
- DFS Used%：HDFS 空间使用的百分比。
- DFS Remaining%：HDFS 空间剩余的百分比。
- Live Nodes：Hadoop 集群活着的节点数。
- Dead Nodes：Hadoop 集群宕机的节点数。

访问 http:// 192.168.195.140:50030 如图 2-35 所示，可以查看 JobTracker 的运行状态，如 Job 运行的进度、Map 个数、Reduce 个数等。

JobTracker WebUI 的属性解释如下。

- Maps：Job 运行使用的 Map 个数。
- Reduces：Job 运行使用的 Reduce 个数。
- Map Task Capacity：Hadoop 集群中 Map 的总个数。
- Reduce Task Capacity：Hadoop 集群中 Reduce 的总个数。
- Queue Name：队列的名称，默认为 default。

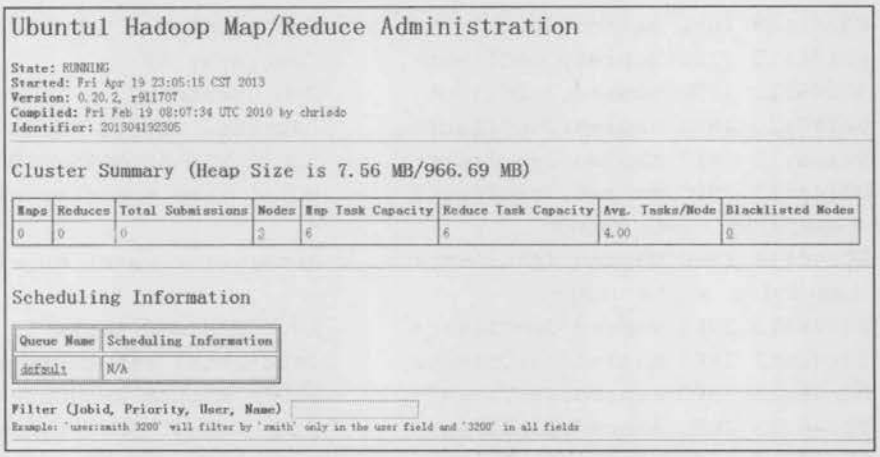


图 2-35 JobTracker WebUI 访问

7. Hadoop 集群测试

我们来运行 hadoop-example.jar 里面自带的 WordCount 程序，作用是统计单词的个数。

1）在 Ubuntu1 的 Hadoop 的 home 目录下创建一个 test.txt 文件，内容如下。

```
Hello world
Hello world
Hello world
Hello world
```

2）在 HDFS 系统里创建一个 input 文件夹，使用命令如下。

```
$ hadoop fs -mkdir /user/hadoop/input
```

3）把创建好的 test.txt 文件上传到 HDFS 系统的 input 文件夹下，命令如下。

```
$ hadoop fs -put /opt/hadoop-0.20.2/test.txt /user/hadoop/input/
```

4）查看文件是否上传成功，结果如图 2-36 所示。

图 2-36 上传文件查询

5）运行 hadoop-1.0.3-examples.jar 下的单词统计案例，执行命令如下。

```
$ cd /opt/hadoop-1.0.3
$ hadoop jar hadoop-examples-1.0.3.jar wordcount /user/hadoop/input/test.txt
/user/hadoop/output
13/04/20 00:47:07 INFO input.FileInputFormat: Total input paths to process : 1
13/04/20 00:47:07 INFO util.NativeCodeLoader: Loaded the native-hadoop library
13/04/20 00:47:07 WARN snappy.LoadSnappy:     Snappy native library not loaded
13/04/20 00:47:08 INFO mapred.JobClient:      Running job: job_201304200039_0001
13/04/20 00:47:09 INFO mapred.JobClient:      map 0% reduce 0%
13/04/20 00:47:45 INFO mapred.JobClient:      map 100% reduce 0%
13/04/20 00:48:08 INFO mapred.JobClient:      map 100% reduce 100%
```

```
    13/04/20 00:48:13 INFO mapred.JobClient:     Job complete: job_201304200039_0001
    13/04/20 00:48:13 INFO mapred.JobClient:     Counters: 29
    13/04/20 00:48:13 INFO mapred.JobClient:     Job Counters
    13/04/20 00:48:13 INFO mapred.JobClient:     Launched reduce tasks=1
    13/04/20 00:48:13 INFO mapred.JobClient:     SLOTS_MILLIS_MAPS=28822
    13/04/20 00:48:13 INFO mapred.JobClient:     Total time spent by all reduces
waiting after reserving slots (ms)=0
    13/04/20 00:48:13 INFO mapred.JobClient:     Total time spent by all maps
waiting after reserving slots (ms)=0
    13/04/20 00:48:13 INFO mapred.JobClient:     Launched map tasks=1
    13/04/20 00:48:13 INFO mapred.JobClient:     Data-local map tasks=1
    13/04/20 00:48:13 INFO mapred.JobClient:     SLOTS_MILLIS_REDUCES=18236
    13/04/20 00:48:13 INFO mapred.JobClient:     File Output Format Counters
    13/04/20 00:48:13 INFO mapred.JobClient:     Bytes Written=16
    13/04/20 00:48:13 INFO mapred.JobClient:     FileSystemCounters
    13/04/20 00:48:13 INFO mapred.JobClient:     FILE_BYTES_READ=30
    13/04/20 00:48:13 INFO mapred.JobClient:     HDFS_BYTES_READ=159
    13/04/20 00:48:13 INFO mapred.JobClient:     FILE_BYTES_WRITTEN=43053
    13/04/20 00:48:13 INFO mapred.JobClient:     HDFS_BYTES_WRITTEN=16
    13/04/20 00:48:13 INFO mapred.JobClient:     File Input Format Counters
    13/04/20 00:48:13 INFO mapred.JobClient:     Bytes Read=48
    13/04/20 00:48:13 INFO mapred.JobClient:     Map-Reduce Framework
    13/04/20 00:48:13 INFO mapred.JobClient:     Map output materialized bytes=30
    13/04/20 00:48:13 INFO mapred.JobClient:     Map input records=4
    13/04/20 00:48:13 INFO mapred.JobClient:     Reduce shuffle bytes=30
    13/04/20 00:48:13 INFO mapred.JobClient:     Spilled Records=4
    13/04/20 00:48:13 INFO mapred.JobClient:     Map output bytes=80
    13/04/20 00:48:13 INFO mapred.JobClient:     CPU time spent (ms)=2870
    13/04/20 00:48:13 INFO mapred.JobClient:     Total committed heap usage
(bytes)=210698240
    13/04/20 00:48:13 INFO mapred.JobClient:     Combine input records=8
    13/04/20 00:48:13 INFO mapred.JobClient:     SPLIT_RAW_BYTES=111
    13/04/20 00:48:13 INFO mapred.JobClient:     Reduce input records=2
    13/04/20 00:48:13 INFO mapred.JobClient:     Reduce input groups=2
    13/04/20 00:48:13 INFO mapred.JobClient:     Combine output records=2
    13/04/20 00:48:13 INFO mapred.JobClient:     Physical memory (bytes)
snapshot= 180101120
    13/04/20 00:48:13 INFO mapred.JobClient:     Reduce output records=2
    13/04/20 00:48:13 INFO mapred.JobClient:     Virtual memory (bytes)
snapshot= 749068288
    13/04/20 00:48:13 INFO mapred.JobClient:     Map output records=8
```

6）查看运行结果如图 3-37 所示。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs -text /user/hadoop/output/part-r-
00000
Warning: $HADOOP_HOME is deprecated.

Hello   4
world   4
```

图 2-37 WordCount 结果

OK！到这里 Hadoop 三个节点的集群就安装结束并且测试成功了。

2.3 如何使用 Hadoop

启动之前，要格式化 NameNode，先进入 /opt/hadoop 目录（如果在 /etc/profile 文件中配置了 Hadoop 的环境变量，就不用进入 $HADOOP_HOME/bin 了），执行下面的命令。

```
[jayliu@Ubuntu1]$bin/hadoop  namenode  -format
```

不出意外，应该会提示格式化成功。如果不成功，就去 hadoop/logs/ 目录下查看日志文件。

2.3.1 Hadoop 的启动与停止

下面就正式启动 Hadoop 啦，bin/ 目录下有很多启动脚本，可以根据自己的需要来启动 Hadoop 的守护进程。启动和停止的脚本和说明如表 2-2 所示。

表 2-2 Hadoop 的启动和停止说明

启动脚本	脚本说明
start-all.sh	启动所有的 Hadoop 守护进程。包括 NameNode、Secondary NameNode、DataNode、JobTracker、TaskTrack
stop-all.sh	停止所有的 Hadoop 守护进程。包括 NameNode、Secondary NameNode、DataNode、JobTracker、TaskTrack
start-dfs.sh	启动 Hadoop HDFS 守护进程 NameNode、SecondaryNameNode 和 DataNode
stop-dfs.sh	停止 Hadoop HDFS 守护进程 NameNode、SecondaryNameNode 和 DataNode
hadoop-daemons.sh start namenode	单独启动 NameNode 守护进程
hadoop-daemons.sh stop namenode	单独停止 NameNode 守护进程
hadoop-daemons.sh start datanode	单独启动 DataNode 守护进程
hadoop-daemons.sh stop datanode	单独停止 DataNode 守护进程
hadoop-daemons.sh start secondarynamenode	单独启动 SecondaryNameNode 守护进程
hadoop-daemons.sh stop secondarynamenode	单独停止 SecondaryNameNode 守护进程
start-mapred.sh	启动 Hadoop MapReduce 守护进程 JobTracker 和 TaskTracker
stop-mapred.sh	停止 Hadoop MapReduce 守护进程 JobTracker 和 TaskTracker
hadoop-daemons.sh start jobtracker	单独启动 JobTracker 守护进程
hadoop-daemons.sh stop jobtracker	单独停止 JobTracker 守护进程
hadoop-daemons.sh start tasktracker	单独启动 TaskTracker 守护进程
hadoop-daemons.sh stop tasktracker	单独启动 TaskTracker 守护进程

如果 Hadoop 集群是第一次启动，可以用 start-all.sh。比较常用的启动方式是一个一个守护进程来启动，启动的步骤如下。

1. 启动 Hadoop 的 HDFS 模块里的守护进程

HDFS 里面的守护进程启动也有顺序，即：

1）启动 NameNode 守护进程；

2）启动 DataNode 守护进程；

3）启动 SecondaryNameNode 守护进程。

2. 启动 MapReduce 模块里面的守护进程

MapReduce 的守护进程启动也是有顺序的，即：

1）启动 JobTracker 守护进程；

2）启动 TaskTracker 守护进程。

关闭的步骤正好相反，在这里就不描述了，读者可以自己试一下。

注意 正常情况下，我们是不使用 start-all.sh 和 stop-all.sh 来启动和停止 Hadoop 集群的。这样出错了不好找原因。建议读者一个一个守护进程来启动，哪个启动失败就去看相应的 log 日志，这样就缩小了找错的范围。

2.3.2 Hadoop 配置文件

Hadoop 安装的配置文件有很多，重要的几个文件如表 2-3 所示。

表 2-3 Hadoop 配置文件

文件名称	格 式	功能描述
hadoop-env.sh	Bash 脚本	记录脚本要用的环境变量，以运行 Hadoop
core-site.xml	Hadoop 配置 XML	Hadoop 核心的配置文件，例如 MapReduce、HDFS 和常用的 I/O 设置等
ddfs-site.xml	Hadoop 配置 XML	Hadoop 守护进程的配置项，包括 NameNode、Secondary NameNode 和 DataNode 等
mapred-site.xml	Hadoop 配置 XML	MapReduce 守护进程的配置项，包含 JobTracker 和 Task Tracker
masters	纯文本	运行 SecondaryNameNode 的机器列表（每行一个）
slaves	纯文本	运行 DataNode 和 TaskTracker 的机器列表（每行一个）
hadoop-metrics.properties	Java 属性	控制 metrics 在 Hadoop 上如何发布，一般对 Hadoop 监控会用到这个配置件
log4j.properties	Java 属性	系统的日志文件、NameNode 审计日志、TaskTracker 子进程任务的日志属性

以上文件全部放在 Hadoop 分发包的 conf 目录下面，这些配置文件也可以重新放在文件系统的其他地方（Hadoop 安装的外面，以便于升级），但是守护进程启动需要用 -config 选项来指定。

2.4 本章小结

本章主要介绍 Hadoop 的安装，包括 Hadoop 单机模式、伪分布式模式以及分布式模式，让读者了解 Hadoop 的安装过程。

Master 和 Slave 上的几个 conf 配置文件不需要全部同步，如果确定都是通过 Master 启动和关闭，Slave 机器上的配置不需要去维护。但如果希望在任意一台机器都可以启动和关闭 Hadoop，那么就需要全部保持一致了。

Master 和 Slave 机器上的 hosts 文件中必须把集群中的机器都配置上去。这方面吃过不少苦头，原来以为如果配成 IP 就不需要去配置 hosts，结果发现在执行 Reduce 的时候总是卡，在复制的时候就无法继续下去，不断重试。另外，如果集群中有两台机器的机器名重复，也会出现问题。

第3章　MapReduce 快速入门

为了让大家快速地认识 Hadoop 的 MapReduce，本章给出一个实际的例子。Hello World 是学一门技术或语言最简单的案例，也是最直观的案例，在这里笔者觉得 Hello World 太简单了，本章用一个比较有难度，并且能体现 MapReduce 特性的案例 WordCount，即统计单词出现的次数，使大家快速入门。

3.1　WordCount 实例准备开发环境

现在有两个文件 file1.txt 和 file2.txt。文件 file1.txt 的内容为：

```
Hello, i love coding
are you ok?
Hello, i love hadoop
are you ok?
```

文件 file2.txt 的内容为：

```
Hello i love coding
are you ok ?
Hello i love hadoop
are you ok ?
```

现在就要实现在 file1.txt 和 file2.txt 两个文件中，按照空格做分隔符，来统计每个单词出现的次数。

虽然 Hadoop 支持很多种语言开发 MapReduce 程序，但是 Java 语言支持得比较完美，提供了原生态的 Java API 接口。C++、Python、Shell 等语言 Hadoop 也是支持的，但是要用到 Hadoop 封装好的一个 Streaming 接口。下面使用 Eclipse 工具开发 MapReduce 程序。

3.1.1　使用 Eclipse 创建一个 Java 工程

Eclipse 的安装很简单，从官网下载一个 Eclipse 版本（这里使用 Eclipse JEE 版本），下载 Eclipse 以后，直接解压到某个文件夹下就可以了。必须要安装 JDK 1.6 以上的版本。打开 Eclipse IDE 工具，鼠标单击 Eclipse 工具栏左上角文件（File）→新建（New）→ Java 项目（Java Project）出现创建工程对话框。Eclipse 创建工程对话框设置属性如下。

1）Project name：创建工程的名字。

2）Use default location：默认是选中的，存放路径可以随便改，但是建议不要改。因为它默认存在 Eclipse 的工作路径下面。

3）JRE 包括以下两个选项。

- Use an execution environment JRE：选择 JRE 的路径。
- Use default JRE：表示使用 Eclipse 工具自带的 JRE，如果没有安装 JDK，可以使用自带的 JRE，如果自己安装了 JDK，最好使用自己安装的 JDK。

4）Project layout：这里选择“Create separate folder for sources and class files”，会在工程下面自动创建一个存放源代码的 src 文件夹。

填好以后单击 Finish 按钮，WordCount 工程就创建好了，如图 3-1 所示。

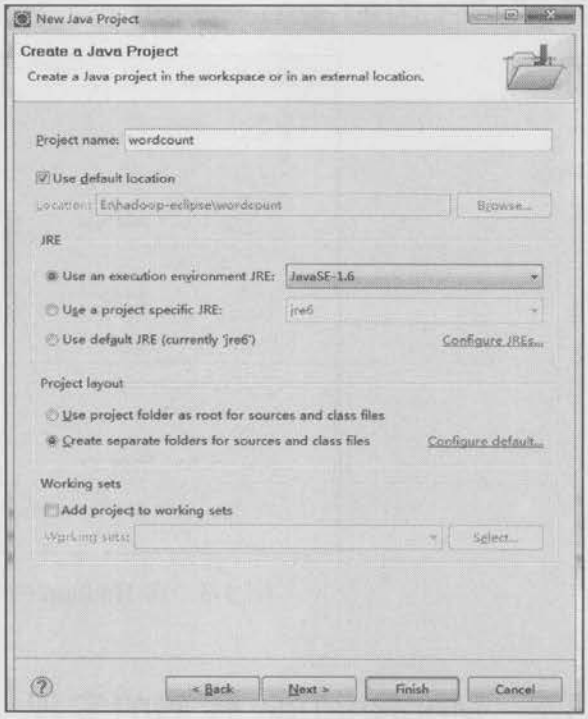


图 3-1 Eclipse 创建工程对话框

3.1.2 导入 Hadoop 的 JAR 文件

导入开发 MapReduce 程序所需要的 Hadoop 依赖的 JAR 文件，如图 3-2 所示，右击项目名称，新建一个 Folder 文件夹，文件夹的名称为 lib，把 Hadoop 的核心包和 lib 下面依赖包全部复制到新建的 lib 目录下面。

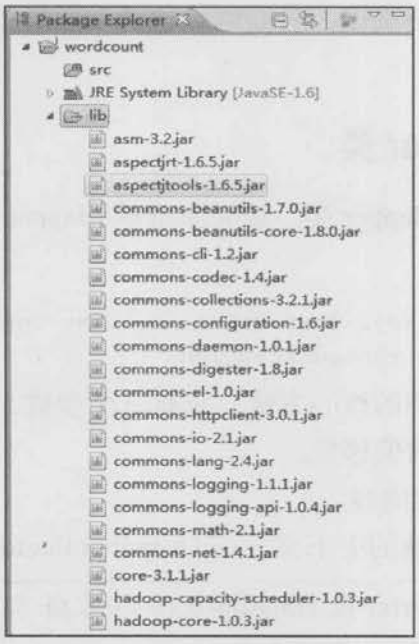


图 3-2 导入 Hadoop 依赖的 JAR 文件

全选 lib 文件夹下的 JAR 文件，鼠标右键单击，选择 Build Path → Add to Build Path，如图 3-3 所示。

OK，到这里一个 Hadoop 工程就准备好了。接下来就可以在这个工程下面开发 MapReduce 程序了。

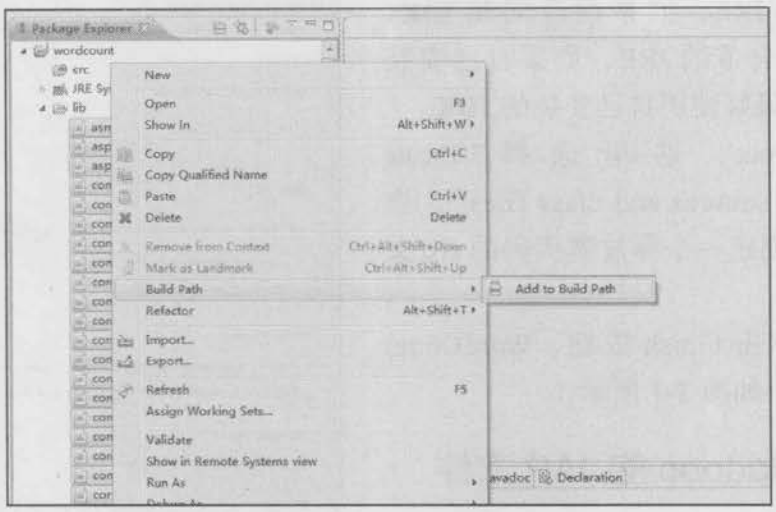


图 3-3 将 Hadoop 的 JAR 文件导入 ClassPath 下面

3.2 MapReduce 代码的实现

使用 Java 语言编写 MapReduce 非常方便，因为 Hadoop 的 API 提供了 Mapper 和 Reducer 抽象类，对于开发人员来说，只要继承这两个抽象类，然后实现抽象类里面的方法就可以了。

3.2.1 编写 WordMapper 类

在工程下创建一个 WordMapper 类，该类要继承 Mapper< Object, Text, Text, IntWritable> 抽象类，并且实现如下方法。

```
public void  map(Object key, Text value, Context context )
throws IOException, InterruptedException
```

这个方法是 Mapper 抽象类的核心方法，它有三个参数。

- Object key：每行文件的偏移量。
- Text value：每行文件的内容。
- Context context：Map 端的上下文，与 OutputCollector 和 Reporter 的功能类似。

注意 OutputCollector 和 Reporter 是 Hadoop-0.19 以前版本里面的 API，在 Hadoop-0.20.2 以后就换成 Context，Context 的功能包含了 OutputCollector 和 Reporter 的功能，此外还添加了一些别的功能。

编写 WordMapper 类如代码清单 3-1 所示。

代码清单 3-1 代码文件 wordcount\WordMapper.java

```
package wordcount;
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
public  class  WordMapper   extends Mapper<Object, Text, Text, IntWritable>{
       private final static IntWritable one = new IntWritable(1);
       private Text word = new Text();
       public void map(Object key, Text value, Context context )
              throws IOException, InterruptedException {
          StringTokenizer itr = new StringTokenizer(value.toString());
          while (itr.hasMoreTokens()) {
              word.set(itr.nextToken());
              context.write(word, one);
       }
  }
}
```

实现 Mapper 抽象类里的 map 方法，map 方法主要就是把字符串解析成 Key-Value 的形式（例如 Key=hello，Value=1），发给 Reduce 端来统计。

3.2.2 编写 WordReducer 类

在工程下创建一个 WordReducer 类，该类要继承 Reducer<Text, IntWritable, Text, IntWritable> 抽象类，并且实现如下方法。

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

这个方法是 Reducer 抽象类的核心方法，它有三个参数。

- Text key：Map 端输出的 Key 值。
- Iterable<IntWritable> values：Map 端输出的 Value 集合（相同 Key 的集合）。
- Context context：Reduce 端的上下文，与 OutputCollector 和 Reporter 的功能类似。

编写 WordReducer 类如代码清单 3-2 所示。

代码清单 3-2 代码文件 wordcount\WordReducer.java

```
package wordcount;
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
public  class WordReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
   private IntWritable result = new IntWritable();
   public void reduce(Text key, Iterable<IntWritable> values, Context context )
                  throws IOException, InterruptedException {
     int sum = 0;
```

```
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
        }
    }
```

reduce 方法的主要功能就是获取 map 方法的 Key-Value 结果，相同的 Key 发送到同一个 reduce 里处理，然后迭代 Key，把 Value 相加，结果写到 HDFS 系统里。

3.2.3 编写 WordMain 驱动类

前面我们实现了 Mapper 和 Reducer 的抽象类，接下来实现一个 MapReduce 的驱动类，驱动类主要用来启动一个 MapReduce 作业。

编写 WordMain 驱动类如代码清单 3-3 所示。

代码清单 3-3 代码文件 wordcount\WordMain.java

```
package wordcount;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class WordMain {
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    /**
     * 这里必须有输入/输出
     */
    if (otherArgs.length != 2) {
     System.err.println("Usage: wordcount <in> <out>");
     System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordMain.class);                    //主类
    job.setMapperClass(WordMapper.class);                 //Mapper
    job.setCombinerClass(WordReducer.class);              //作业合成类
    job.setReducerClass(WordReducer.class);               //Reducer
    job.setOutputKeyClass(Text.class);                    //设置作业输出数据的关键类
    job.setOutputValueClass(IntWritable.class);           //设置作业输出值类
    FileInputFormat.addInputPath(job, new Path(otherArgs[0])); //文件输入
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));       //文件输出
```

```
        System.exit(job.waitForCompletion(true) ? 0 : 1);   //等待完成退出
        }
}
```

WordMain 驱动类解释如下。

- Configuration 类：读取 Hadoop 的配置文件，如 site-core.xml、mapred-site.xml、hdfs-site.xml 等。也可以使用 set 方法进行重新设置，如 conf.set("fs.default.name", "hdfs://xxxxx:9000")。注意，set 方法设置的值会替代配置文件里面配置的值。
- Job 类：表示一个 MapReduce 任务。Job 的构造方法有两个参数，第一个参数为 Configuration，第二个参数为 Job 的名称（等同于任务的名称）。

至此，WordCount 就开发完成了，接下来就是把 WordCount 放在 Hadoop 上运行。

3.3 打包、部署和运行

Hadoop 的部署环境其实指 Linux 的命令行环境，因为现在执行计算的服务器大多是 Linux 文本界面。在集群系统中有一台 Master，这台机器就是我们的部署环境，而且只需在这台机器上部署即可，Hadoop 会自己把任务传送给各个 Slave。

部署是什么意思呢？其实从代码编写完到任务，可以按照我们的想法在服务器上运行的这段时间都叫部署。我们在部署环境下运行项目，一般的做法是：在开发环境下（比如 Eclipse 的图形开发界面），将项目编译好，保存成 JAR 文件，然后传送到 Master 上运行。

注意 传送之前的操作都是 Eclipse 上的操作，对于 Eclipse 开发人员来说，其实是很简单的。所以“部署环境下的运行”只要把编译好的 JAR 文件传送到 Master 并让它运行即可（在 Slave 上也可以运行）。

一般情况下，默认已经在 Master 上获得 JAR 文件，并放置在 Master 的 $HADOOP_HOME 下面。

3.3.1 打包成 JAR 文件

把 3.2 节编写的 MapReduce 工程打包成 JAR 文件，然后发送到 Hadoop 的 Master 节点上运行。如图 3-4 所示，右击 wordcount → Export → JAR fil。

选择 JAR file 以后，单击 Next 按钮，进入 JAR 文件过滤对话框，如图 3-5 所示。

这里只选择 src 文件夹就可以了，因为 lib 下面的 JAR 是 Hadoop 自带的，不需要把它添加到 JAR 文件里面，还要注意的就是，不要把 classpath 和 project 文件添加到 JAR 文件中。在“Select the export destination”下面的 JAR file 选项中，选择 JAR 文件存放的位置和文件名。

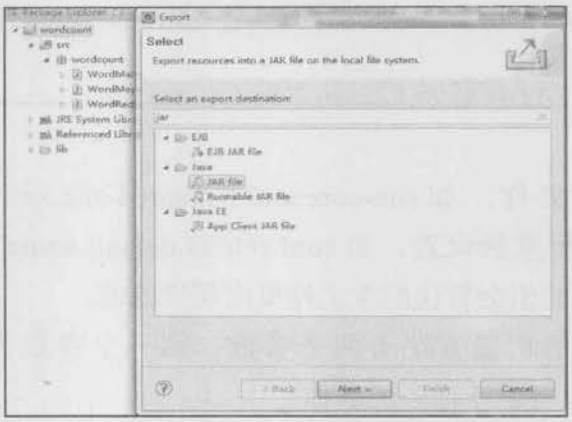


图 3-4 将 WordCount 工程保存成 JAR 文件

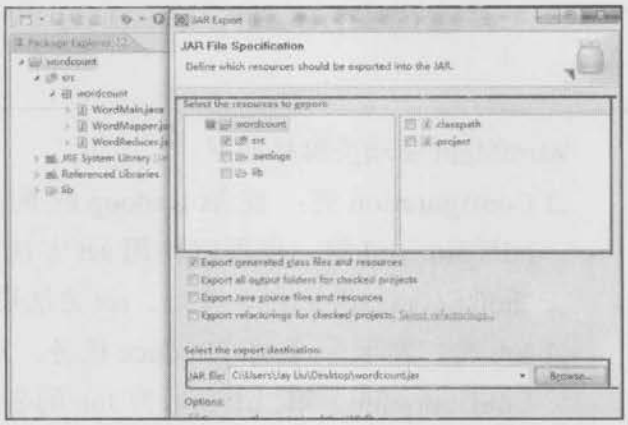


图 3-5 JAR 文件过滤

3.3.2 部署和运行

部署和运行的步骤如下所示。

1）把 wordcount.jar 文件发送到 Hadoop 集群的 Master 节点的 $HADOOP_HOME 下面。

2）在 Master 节点的 /opt/ 下面创建两个文件 file1.txt 和 file2.txt，文件中是我们要统计单词个数的内容。

3）把 file1.txt 和 file2.txt 传到 HDFS 系统下面，命令如下所示。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs -put /opt/file* /user/hadoop/input/
```

4）查看文件是否上传成功。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs -ls /user/hadoop/input/
Warning: $HADOOP_HOME is deprecated.
Found 2 items
-rw-r--r--   1 jayliu supergroup      66 2013-04-20 18:29
/user/hadoop/input/file1.txt
-rw-r--r--   1 jayliu supergroup      66 2013-04-20 18:29
/user/hadoop/input/file2.txt
```

5）运行 WordCount 程序。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop jar wordcount.jar wordcount.WordMain
/user/hadoop/input/file* /user/hadoop/output1
Warning: $HADOOP_HOME is deprecated.

13/04/20 18:35:56 INFO input.FileInputFormat: Total input paths to process : 2
13/04/20 18:35:56 INFO util.NativeCodeLoader: Loaded the native-hadoop library
13/04/20 18:35:56 WARN snappy.LoadSnappy: Snappy native library not loaded
13/04/20 18:36:00 INFO mapred.JobClient: Running job: job_201304200039_0002
13/04/20 18:36:01 INFO mapred.JobClient: map 0% reduce 0%
13/04/20 18:37:56 INFO mapred.JobClient: map 50% reduce 0%
13/04/20 18:38:04 INFO mapred.JobClient: map 100% reduce 0%
13/04/20 18:38:26 INFO mapred.JobClient: map 100% reduce 100%
```

```
13/04/20 18:38:34 INFO mapred.JobClient: Job complete: job_201304200039_0002
13/04/20 18:38:35 INFO mapred.JobClient: Counters: 29
13/04/20 18:38:35 INFO mapred.JobClient: Job Counters
13/04/20 18:38:35 INFO mapred.JobClient: Launched reduce tasks=1
13/04/20 18:38:35 INFO mapred.JobClient: SLOTS_MILLIS_MAPS=214131
13/04/20 18:38:35 INFO mapred.JobClient: Total time spent by all reduces waiting
                                         after reserving slots (ms)=0
13/04/20 18:38:35 INFO mapred.JobClient: Total time spent by all maps waiting
                                         after reserving slots (ms)=0
13/04/20 18:38:35 INFO mapred.JobClient: Launched map tasks=2
13/04/20 18:38:35 INFO mapred.JobClient: Data-local map tasks=2
13/04/20 18:38:35 INFO mapred.JobClient: SLOTS_MILLIS_REDUCES=19234
13/04/20 18:38:35 INFO mapred.JobClient: File Output Format Counters
13/04/20 18:38:35 INFO mapred.JobClient: Bytes Written=73
13/04/20 18:38:35 INFO mapred.JobClient: FileSystemCounters
13/04/20 18:38:35 INFO mapred.JobClient: FILE_BYTES_READ=188
13/04/20 18:38:35 INFO mapred.JobClient: HDFS_BYTES_READ=356
13/04/20 18:38:35 INFO mapred.JobClient: FILE_BYTES_WRITTEN=64611
13/04/20 18:38:35 INFO mapred.JobClient: HDFS_BYTES_WRITTEN=73
13/04/20 18:38:35 INFO mapred.JobClient: File Input Format Counters
13/04/20 18:38:35 INFO mapred.JobClient: Bytes Read=132
13/04/20 18:38:35 INFO mapred.JobClient: Map-Reduce Framework
13/04/20 18:38:35 INFO mapred.JobClient: Map output materialized bytes=194
13/04/20 18:38:35 INFO mapred.JobClient: Map input records=8
13/04/20 18:38:35 INFO mapred.JobClient: Reduce shuffle bytes=94
13/04/20 18:38:35 INFO mapred.JobClient: Spilled Records=34
13/04/20 18:38:35 INFO mapred.JobClient: Map output bytes=252
13/04/20 18:38:35 INFO mapred.JobClient: CPU time spent (ms)=5540
13/04/20 18:38:35 INFO mapred.JobClient: Total committed heap usage (bytes)= 413466624
13/04/20 18:38:35 INFO mapred.JobClient: Combine input records=30
13/04/20 18:38:35 INFO mapred.JobClient: SPLIT_RAW_BYTES=224
13/04/20 18:38:35 INFO mapred.JobClient: Reduce input records=17
13/04/20 18:38:35 INFO mapred.JobClient: Reduce input groups=11
13/04/20 18:38:35 INFO mapred.JobClient: Combine output records=17
13/04/20 18:38:35 INFO mapred.JobClient:  Physical memory (bytes) snapshot=231944192
13/04/20 18:38:35 INFO mapred.JobClient: Reduce output records=11
13/04/20 18:38:35 INFO mapred.JobClient: Virtual memory (bytes) snapshot=1119518720
13/04/20 18:38:35 INFO mapred.JobClient: Map output records=30
```

如果看到上面的结果，说明运行成功！

注意，运行的JAR文件需要放在$HADOOP_HOME下面，否则会报文件打开错误，错误提示如下所示。

```
Caused by: java.util.zip.ZipException: error in opening zip file
  at java.util.zip.ZipFile.open(Native Method)
  at java.util.zip.ZipFile.<init>(ZipFile.java:127)
  at java.util.jar.JarFile.<init>(JarFile.java:135)
  at java.util.jar.JarFile.<init>(JarFile.java:72)
  at org.apache.hadoop.util.RunJar.main(RunJar.java:88)
```

结果输出的目录必须是不存在，否则会报文件已存在错误，错误提示如下所示。

```
    13/04/20 18:39:54 INFO mapred.JobClient: Cleaning up the staging area hdfs://
Ubuntu1:9000/hadoop/mapred/staging/jayliu/.staging/job_201304200039_0003
    13/04/20 18:39:54 ERROR security.UserGroupInformation: PriviledgedActionException
as:jayliu cause:org.apache.hadoop.mapred.FileAlreadyExistsException: Output directory
/user/hadoop/output1 already exists
    Exception in thread "main" org.apache.hadoop.mapred.FileAlreadyExistsException:
Output directory /user/hadoop/output1 already exists
      at org.apache.hadoop.mapreduce.lib.output.FileOutputFormat.checkOutputSpecs
         (FileOutputFormat.java:137)
      at org.apache.hadoop.mapred.JobClient$2.run(JobClient.java:887)
      at org.apache.hadoop.mapred.JobClient$2.run(JobClient.java:850)
      at java.security.AccessController.doPrivileged(Native Method)
      at javax.security.auth.Subject.doAs(Subject.java:396)
      at org.apache.hadoop.security.UserGroupInformation.doAs
         (UserGroupInformation.java:1121)
```

3.3.3 测试结果

接下来就是查看运行的结果，看看是否和我们预测的一样，预测的结果如下。

```
?       2
Hello   2
Hello,  2
are     4
coding  2
hadoop  2
i       4
love    4
ok      2
ok?     2
you     4
```

查看生成的结果文件，如图 3-6 所示。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs -ls /user/hadoop/output1
Warning: $HADOOP_HOME is deprecated.

Found 3 items
-rw-r--r--   1 jayliu supergroup          0 2013-04-20 18:38 /user/hadoop/out
put1/_SUCCESS
drwxr-xr-x   - jayliu supergroup          0 2013-04-20 18:36 /user/hadoop/out
put1/_logs
-rw-r--r--   1 jayliu supergroup         73 2013-04-20 18:38 /user/hadoop/out
put1/part-r-00000
```

图 3-6 查看结果文件目录

结果文件一般由三部分组成。

- _SUCCESS 文件：表示 MapReduce 运行成功。
- _logs 文件夹：存放运行 MapReduce 的日志。
- part-r-00000 文件：存放结果，也是默认生成的结果文件。

使用“hadoop fs –text/user/hadoop/output1/part-r-00000”命令查看 MapReduce 生成的结果文件，如图 3-7 所示。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs -text /user/hadoop/output1/part-r
-00000
Warning: $HADOOP_HOME is deprecated.

?       2
Hello   2
Hello,  2
are     4
coding  2
hadoop  2
i       4
love    4
ok      2
ok?     2
you     4
```

图 3-7 查看结果文件

到这里，整个 MapReduce 的快速入门就结束了。本章使用一个完整的案例，从开发到部署再到查看结果，让大家对 MapReduce 的基本使用有所了解。

3.4 本章小结

本章主要是通过一个实例来介绍 MapReduce 开发的整个过程。整个过程包括环境的准备、Mapper 和 Reducer 代码的编写、打包、部署和运行的一整套流程。学习这套流程对以后 MapReduce 的开发奠定了基础。此外，介绍了 MapReduce 开发遇到的常见错误，在这里希望读者按照本章的步骤一步一步地往下做。争取使程序能跑起来，对以后章节的上机操作打下基础。

第4章　Hadoop 分布式文件系统详解

Hadoop 分布式文件系统（Hadoop Distributed File System，HDFS）是一个用在普通硬件设备上的分布式文件系统。它与现有的分布式文件系统有很多近似的地方，但又和这些文件系统有很多明显的不同。

4.1　认识 HDFS

HDFS 开始是为开源的 Apache Nutch 项目的基础结构而创建，HDFS 是 Hadoop 项目的一部分，而 Hadoop 又是 Lucene 的一部分。本项目的地址在 http://projects.apache.org/projects/hadoop.html。

4.1.1　HDFS 的特点

HDFS 有高容错性（fault-tolerant）的特点，可以用来部署在低廉的硬件上。它提供高吞吐量（high throughput）的特性，用来访问应用程序的数据，适合有超大数据集（large data set）的应用程序。HDFS 放宽了对 POSIX 的要求，这样可以实现通过流的形式访问文件系统中的数据。

1. 硬件故障

硬件故障是常态，而不是异常。整个 HDFS 系统由数百或数千个存储着文件数据片断的服务器组成。实际上它里面有非常多的组成部分，每一个组成部分都会频繁地出现故障，这就意味着 HDFS 里的一些组成部分总是失效的，因此，故障的检测和自动快速恢复是 HDFS 一个核心的结构目标。

2. 流式的数据访问

运行在 HDFS 之上的应用程序必须流式地访问它们的数据集，它不是典型的运行在常规的文件系统之上的常规程序。HDFS 的设计适合批量处理，而不是用户交互式的；重点是数据吞吐量，而不是数据访问的反应时间，POSIX 强制的很多硬性需求对很多应用不是必需的，去掉 POSIX 的很多关键地方的语义，可以获得更好的数据吞吐量。

3. 大数据集

运行在 HDFS 之上的程序有很大量的数据集。这意味着典型的 HDFS 文件是 GB 到 TB 的大小，所以 HDFS 能很好地支持大文件。它应该提供很高的聚合数据带宽，应该一个集群中支持数百个节点，还应该支持一个集群中千万的文件。

4. 简单一致性模型

大部分的HDFS程序对文件操作需要一次写入，多次读取。一个文件一旦创建、写入、关闭之后就不需要修改了。这个假定简单化了数据一致的问题和高吞吐量的数据访问。MapReduce程序或者网络爬虫程序都是非常完美地适合这个模型。计划在将来实现文件的附加写入。

5. 移动计算比移动数据更经济

在靠近要计算的数据存储的位置进行计算是最理想的状态，尤其是在数据集特别巨大的时候。这样消除了网络的拥堵，提高了系统的整体吞吐量。这个假定就是将计算离数据更近比将文件移动到程序运行的位置更好。HDFS提供了接口，让程序将自己移动到离数据存储更近的位置。

6. 轻便访问异构的软硬件平台

HDFS应该设计成这样的一种方式，就是简单轻便地从一个平台到另外一个平台，这将推动需要大数据集的应用更广泛地采用HDFS作为平台。

7. 名字节点和数据节点

HDFS是一个主从结构的体系，一个HDFS集群由以下部分组成。

- 名字节点（NameNode）：一个，用来管理文件的名字空间和调节客户端访问文件的主服务器。
- 数据节点（DataNode）：一个或多个，用来管理存储。

HDFS暴露文件名字空间，并且允许用户数据存储成文件。内部机制是将一个文件分割成一个或多个的块，这些块存储在一组数据节点中。名字节点操作文件名字空间的文件或目录操作，如打开、关闭、命名等。它同时确定块与数据节点的映射。数据节点来负责来自文件系统客户的读写请求。

数据节点同时还要执行块的创建、删除和来自名字节点的块复制指示。

名字节点和数据节点都是运行在普通的机器之上，机器典型的都是Linux，HDFS是用Java写的，任何支持Java的机器都可以运行名字节点或数据节点，利用Java语言的超轻便性，很容易将HDFS部署到大范围的机器上。典型的部署时将有一个专门的机器来运行名字节点软件，机群中的其他机器运行一个数据节点实例。体系结构排斥在一个机器上运行多个数据节点的实例，但是实际的部署不会有这种情况。

集群中只有一个名字节点极大地简单化了系统的体系。名字节点是仲裁者和所有HDFS的元数据的仓库。系统设计，用户的实际数据不经过名字节点。

8. 文件名字空间

HDFS支持传统的继承式的文件组织。一个用户或一个程序可以创建目录，存储文件到很多目录之中。文件系统的名字空间层次和其他的文件系统相似。可以创建文件，也可以将文件从一个目录移动到另外一个，或重命名。HDFS现在还没有实现用户的配额和访问

控制。HDFS 还不支持硬链接和软链接。然而，HDFS 结构不排斥在将来实现这些功能。

名字节点维护文件系统的名字空间，任何文件名字空间的改变都会被名字节点记录下来。应用程序可以指定文件的复制数，文件的复制被称为文件的复制因子，这些信息由名字空间来负责存储。

9. 数据复制

HDFS 能可靠地在集群中的大量机器之间存储非常大量的文件，它以块序列的形式存储每一个文件。文件中除了最后一个块的其他块都是相同的大小。属于文件的块为了故障容错而被复制。块的大小和复制数可以为每个文件配置。HDFS 中的文件都是严格地要求任何时候只有一个写操作。程序可以特别地为某个文件指定。复制数、文件的复制数可以在文件的创建时指定或者以后改变。名字节点来做所有的块复制，它周期性地接受来自集群中数据节点的心跳和块报告。一个心跳的收条表示这个数据节点是健康的，是渴望服务数据的。一个块报告包括该数据节点上的所有块列表。复制块放置位置的选择严重影响 HDFS 的可靠性和性能。这个特征是 HDFS 和其他分布式文件系统的区别。这个特征需要很多的调节和经验。机架的复制布局目的就是提高数据的可靠性、可用性和网络带宽的利用率。

HDFS 运行在跨越很多机架的集群机器之上。两个不同机架上的节点通过交换机进行通信，在大多数情况下，两个相同机架上的节点之间的网络带宽是优于在不同机架之上的两个机器。

在开始的时候，每一个数据节点自检它所属的机架，然后在向名字节点注册的时候告知它的机架 ID。HDFS 提供接口以便很容易地挂载检测机架标示的模块。一个简单但不是最优的方式就是将复制跨越不同的机架，以保证在这个机架出现故障而不丢失数据，还能在读数据的时候充分利用不同机架的带宽。这个方式均匀地将复制分散在集群中以简单化地实现组件实效的负载均衡。然而，这个方式增加了写的成本，因为写的时候需要传输文件块到很多的机架。

在复制数为 3 的情况下，HDFS 放置方式是将第一个放在本地节点，将第二个复制放到本地机架上的另外一个节点，而将第三个复制放到不同机架上的节点。这种方式减少了机架内的写流量，提高了写的性能。机架失效的机会远小于机器失效的机会。这种方式没有影响数据的可靠性和可用性的保证，但是减少了读操作的网络聚合带宽，因为文件块存在。

在复制数为 3 的情况下，是备份在两个不同的机架上，而不是三个机架。文件的复制不是均匀地分布在机架当中。1/3 在同一个节点上，第二个 1/3 复制在同一个机架上，另外 1/3 是均匀地分布在其他的机架上。这种方式提高了写性能，而没有影响数据的可靠性和读性能。

10. 复制的选择

HDFS 尝试满足一个读操作来自离它最近的复制。假如在读节点的同一个机架上有这个复制，则直接读。如果 HDFS 集群是跨越多个数据中心，那么本地数据中心的复制优先于远程的复制。

11. 安全模式

在启动的时候，名字节点进入一个特殊的状态叫做安全模式。安全模式是不发生文件块的复制的。名字节点接受来自数据节点的心跳和块报告。一个块报告包括数据节点向名字节点报告数据块的列表。

每一个块有一个特定的最小复制数。当名字节点检查这个块已经大于最小的复制数，就被认为是安全地复制了，当达到配置的块安全复制比例时（+30s），名字节点就退出安全模式。它将检测数据块的列表，将小于特定复制数的块复制到其他的数据节点。

12. 元数据的持久化

HDFS 的名字空间是由名字节点来存储的。名字节点用事务日志（EditsLog）来持久化每一个对文件系统的元数据的改变。例如，在 HDFS 中创建一个新的文件，名字节点将会插入一条记录到 EditsLog 来标示这个改变。类似地，改变文件的复制因子也会向 EditsLog 中插入一条记录。名字节点在本地文件系统中用一个文件来存储这个 EditsLog。完整的文件系统名字空间、文件块的映射和文件系统的配置都存在一个叫 FsImage 的文件中，FsImage 也是名字节点的本地文件系统中。

名字节点在内存中有一个完整的文件系统名字空间和文件块的映射镜像。这个元数据是设计紧凑的，这样 4GB 内存的名字节点就能很轻松地处理非常大文件数和目录，当名字节点启动，它将从磁盘中读取 FsImage 和 EditsLog 文件，然后将新的元数据刷新到本地磁盘中，生成一个新的 FsImage 文件，至此 EditsLog 文件已经被处理并持久化的 FsImage 中，这个过程叫做**检查点**。检查点发生在名字节点启动的时候。

数据节点存储 HDFS 数据到本地的文件系统中。数据节点没有关于 HDFS 文件的信息。它以单独的文件存储每一个 HDFS 的块到本地文件系统中。数据节点不产生所有的文件到同一个目录中，而是启发式检测最优的每一个目录的文件数。它在适当的时候创建子目录。在本地文件的同一个目录下创建所有的文件不是最优的，因为本地文件系统可能单个目录里有数目巨大的文件，效率较差。当数据节点启动的时候，将扫描它的本地文件系统，根据本地的文件产生一个所有 HDFS 数据块的列表并报告给名字节点，这个报告称为**块报告**。

13. 通信协议

所有的通信协议都是在 TCP/IP 协议之上的。一个客户端和明确的配置端口的名字节点建立连接之后，它和名字节点的协议是 ClientProtocal，数据节点和名字节点之间通信用 DatanodeProtocal。这些协议的详细内容将在后面解释。

RPC 抽象地包装了 ClientProtocol 和 DataNodeProtocol。根据设计，名字节点不会发起一个 RPC，它只是对数据节点和客户端发起的 RPC 做出反馈。

14. 鲁棒性

HDFS 的主要目标就是在存在故障的情况下可靠地存储数据。三个普通的故障是名字节点失效、数据节点失效和网络断开。

15. 磁盘故障、心跳和重新复制

一个数据节点周期性发送一个心跳信息到名字节点。网络断开会造成一个数据节点子集和名字节点失去联系。名字节点发现这种情况是根据有没有心跳信息。名字节点标记这些数据节点死掉了，就不再将新的 I/O 请求转发到这些数据节点上。而这些数据节点上的数据将对 HDFS 不再可用。这将导致一些块的复制因子降低到指定的值。

名字节点检查所有需要复制的块，并开始复制它们到其他的数据节点上。重新复制有很多原因，例如数据节点变得不可用、被破坏的复制、数据节点上的磁盘损坏或增加了文件的复制因子。

16. 集群的重新均衡

HDFS 体系结构兼容数据的重新平衡方案。在数据节点的可用空间降低到一个极限时，数据可能自动从一个数据节点移动到另外一个数据节点。如果一个客户端突然地对一个特殊的文件发生高请求时也会引发额外的复制，这时将集群中的其他数据重新均衡。而这种类型的重新均衡方案还没有实现。

17. 数据正确性

从数据节点上取一个文件块有可能出现损坏的情况，这种情况发生的原因包括存储设备、差劲的网络、软件缺陷等。HDFS 客户端实现了校验去检查 HDFS 的文件内容。当一个客户端创建一个 HDFS 文件，它为每一个文件块计算一个校验码，并存储校验码在同一个 HDFS 名字空间中一个单独的隐藏文件中。当客户端找回这个文件内容时，它再根据这个校验码来验证从数据节点接收到的数据。如果不对，客户端可以从另外一个有该块复制的数据节点取这个块。

18. 元数据磁盘实效

FsImage 和 EditsLog 是 HDFS 的中心数据结构，这些文件的损坏会导致整个集群不工作。因此，名字节点可以配置成多个 FsImage 和 EditsLog 的副本。任何对 FsImage 和 EditsLog 的更新都会同步地更新每一个副本。

同步更新多个 EditsLog 可能降低了名字节点的可支持名字空间的每秒交易数。但是这个降低是可接受的，因为 HDFS 程序都是自然地对数据要求强烈，而不是对元数据的要求强烈。名字节点重新启动时，选择最新的一致的 FsImage 和 EditsLog。

对以 HDFS 集群名字节点是单点名字节点失效的。假如名字节点失效，手工的干涉是必要的，当前，自动的重启和切换到另外的名字节点还不支持。

19. 快照

快照支持在一个特定时间存储一个数据副本，快照的一个用途可以将失效的集群回滚到之前的一个正常时间点上。HDFS 目前还不支持快照，但是将来的版本可能会支持。

20. 数据组织

HDFS 的设计是支持大文件数据块的。程序也和 HDFS 一样处理大数据集。这些程序

写数据仅一次，读数据一次或多次，需要一个比较好的流读取速度。HDFS 支持文件的一次写入多次读取。HDFS 典型的块大小是 64MB，一个 HDFS 文件最多可以切分成 128 个块，每个块分布在不同的数据节点上。

4.1.2 Hadoop 文件系统的接口

Hadoop 是用 Java 开发的，通过 Java API 可以实现与 Hadoop 文件系统的交互操作。例如，文件系统的命令解释器就是一个 Java 应用。它使用 Java 的 FileSystem 类提供文件系统操作，其他一些接口也将在本章中介绍。这些接口通常与 HDFS 一同使用，因为其他的文件系统一般都有访问基本文件系统的工具，如 FTP 的客户端、S3 工具等，下面介绍 HDFS 其他的访问方式。

1. Thrift

因为 Hadoop 文件系统的接口是通过 Java API 提供服务的，所以其他的非 Java 应用程序要访问 Hadoop 文件系统会比较麻烦。Thrift 会定制功能模块中的 Thrift API，通过把 Hadoop 文件系统包装一个 Apache Thrift 服务来弥补不足，从而使任何具有 Thrift 绑定（binding）的语言都能轻松地与 Hadoop 文件系统（如 HDFS）进行交互。

为了使用 Thrift API 需要运行提供 Thrift 服务的 Java 服务器，并以代理的方式访问 Hadoop 文件系统。应用程序访问 Thrift 服务时，实际上两者是运行在同一台机器上。

Thrift API 包含许多其他语言实现远程过程调用的接口，包括 C++、Perl、PHP、Python 以及 Ruby。Thrift 支持不同版本的 Hadoop，因此可以通过同一个客户端访问不同版本的 Hadoop 文件系统。但是，需要针对每个不同的 Hadoop 版本运行相应的代理，才能实现该功能。

2. C 语言

Hadoop 提供一个名为 libhdfs 的 C 语言库，该语言库是 Java FileSystem 接口类的一个镜像。它被编程成访问 HDFS 的 C 语言的库，但它其实可以访问任意 Hadoop 文件系统。它可以使用 Java 原生接口（Java Native Interface，JNI）调用 Java 文件系统客户端。

C 语言 API 与 Java 的 API 非常相似，但它的开发一般滞后于 Java API，因此目前一些新的特性可能还不支持。

3. 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（一般为 HDFS）都可以作为一个标准文件系统进行挂载。 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。

4. WebDAV

WebDAV（Web-based Distributed Authoring and Versioning）是基于 HTTP 1.1 的一个通信协议。它为 HTTP 1.1 添加了一些扩展（就是在 GET、POST、HEAD 等几个 HTTP 标准方法以外添加了一些新的方法），使得应用程序可以直接将文件写到 Web Server 上，并且在写文件时候可以对文件加锁，写完后对文件解锁，还可以支持对文件所做的版本控制。

5. 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）。

6. FTP

对 HDFS 的 FTP 接口，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 FTP ileSystem 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。

4.1.3 HDFS 的 Web 服务

HDFS 里内置了很多 Web 服务，这些服务可以让用户通过浏览器来查看 HDFS 的运行状况，详细信息如表 4-1 所示。

表 4-1 HDFS 的 Web 服务

参数名称	功能描述	默 认	配置文件	例 子 值
fs.default.name	NameNode RPC 交互端口	8020	core-site.xml	hdfs://master:8020/
dfs.http.address	NameNode Web 管理端口	50070	hdfs-site.xml	0.0.0.0:50070
dfs.datanode.address	DataNode 控制端口	50010	hdfs-site.xml	0.0.0.0:50010
dfs.datanode.ipc.address	DataNode 的 RPC 服务器地址和端口	50020	hdfs-site.xml	0.0.0.0:50020
dfs.datanode.http.address	DataNode 的 HTTP 服务器和端口	50075	hdfs-site.xml	0.0.0.0:50075

4.2 HDFS 架构

HDFS 为 Hadoop 这个分布式计算框架提供高性能、高可靠、高可扩展的存储服务。HDFS 架构是一个典型的主从架构，包括一个 NameNode 节点（主节点）和多个 DataNode 节点（从节点），并提供应用程序访问接口。

NameNode 是整个文件系统的管理节点，它负责文件系统名字空间（NameSpace）的管

理与维护，同时负责客户端文件操作的控制以及具体存储任务的管理与分配；DataNode 提供真实文件数据的存储服务。系统架构如图 4-1 所示。

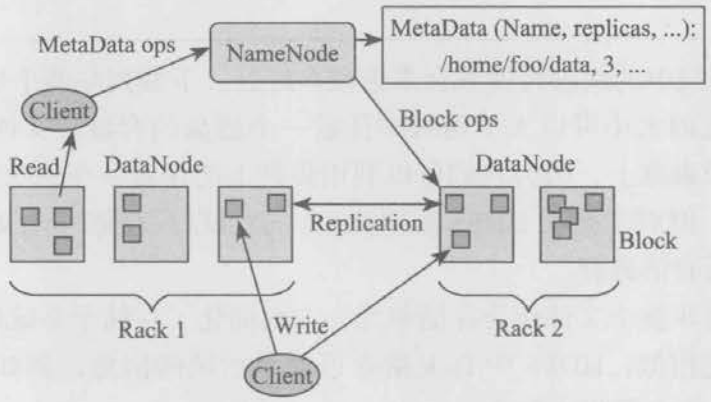


图 4-1 HDFS 架构图

4.2.1 机架

Rack 是机架的意思，一个 Block 的三个副本通常会保存到两个或者两个以上的机架中（当然是机架中的服务器），这样做的目的是防灾容错，因为发生一个机架掉电或者一个机架的交换机挂了的概率还是很高的。

4.2.2 数据块

在 Linux 系统里面，每个磁盘都有默认的数据块大小，这是磁盘进行数据读和写的最小单位。构建于单个磁盘之上的文件系统通过磁盘块来管理该文件系统中的块，该文件系统中的块可以是磁盘块的整数倍。文件系统块一般为几千字节，而磁盘块一般为 512 字节。

HDFS 同样也有块（Block）的概念，但是要大得多，默认为 64MB，与单一磁盘上的文件系统类似，HDFS 上的文件系统也被划分为块大小的多个分块（Chunk）作为独立的存储单元。但与其他文件系统不同的是，HDFS 中小于一个块大小的文件不会占据整个块的空间。如果没有特殊的指出，本书中的块是指 HDFS 中的块。

扩展阅读

为什么 HDFS 默认的 Block 为 64MB？

HDFS 的块之所以这么大，主要是为了把寻道（Seek）时间最小化。如果一个块足够大，从硬盘传输数据的时间将远远大于寻找块的起始位置的时间。这样就使 HDFS 的数据传输速度和硬盘的传输速度更加接近。

让我们进行简单的计算，假设寻道时间为 10ms，传输速度为 100MB/s。为了使得寻道时间仅为传输时间的 1%，就需要设置块的大小为 100MB。尽管很多应用的 HDFS 使用 128MB 的块，但默认的大小是 64MB，这个参数将随着新一代硬盘速度的增长而增长。

虽然大块有一定好处，但也不能太大。MR 中的 Map Task 一般一次只处理一个块，所以如果你的 Task 太少了（少于集群中的节点数），相对于设置较小块的 Job 来说，你的 Job 将会运行得比较慢。

对分布式文件系统中的块进行抽象会来带很多好处。下面列举两个好处。

- 一个文件系统的大小可以大于网络中任意一个磁盘的容量。文件的所有块并不需要存储在同一个磁盘上，因为它们可以利用集群上的任意一个磁盘进行存储。事实上，尽管不常见，但对于整个 HDFS 集群而言，也可以只存储一个文件，该文件的块占满集群中的所有的磁盘。
- 使用块抽象而非整个文件作为存储单元，大大简化了存储子系统的设计。

与磁盘文件系统相似，HDFS 中 fsck 指令可以显示块的信息，例如执行以下命令将列出文件系统中各个文件由那些块构成。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fsck / -files -blocks
FSCK started by jayliu from /192.168.195.140 for path / at Sat Apr 20 21:58:02
CST 2013
/ <dir>
/hadoop <dir>
/hadoop/input <dir>
/hadoop/input/log.rar 1389158877 bytes, 21 block(s):  OK
0.  blk_-7375797407905273137_1019 len=67108864 repl=1
1.  blk_-6315605280944297587_1019 len=67108864 repl=1
2.  blk_5255555707302838680_1019  len=67108864 repl=1
3.  blk_3834812033770488757_1019  len=67108864 repl=1
4.  blk_5007288343118620499_1019  len=67108864 repl=1
5.  blk_2202353214158484157_1019  len=67108864 repl=1
6.  blk_2989169531498368426_1019  len=67108864 repl=1
7.  blk_-1156957313062524605_1019 len=67108864 repl=1
8.  blk_-3111578455959489577_1019 len=67108864 repl=1
9.  blk_7742738890307496909_1019  len=67108864 repl=1
10. blk_7884544365024519326_1019  len=67108864 repl=1
11. blk_7517554983450116569_1019  len=67108864 repl=1
12. blk_7575277603104841598_1019  len=67108864 repl=1
13. blk_-5478001995858009037_1019 len=67108864 repl=1
14. blk_-7869747358547664605_1019 len=67108864 repl=1
15. blk_-8882262405585470331_1019 len=67108864 repl=1
16. blk_8531649117484498718_1019  len=67108864 repl=1
17. blk_560419254426126900_1019   len=67108864 repl=1
18. blk_-8156157960663364905_1019 len=67108864 repl=1
19. blk_-1893860936828878743_1019 len=67108864 repl=1
20. blk_8238692764711284673_1019  len=46981597 repl=1
```

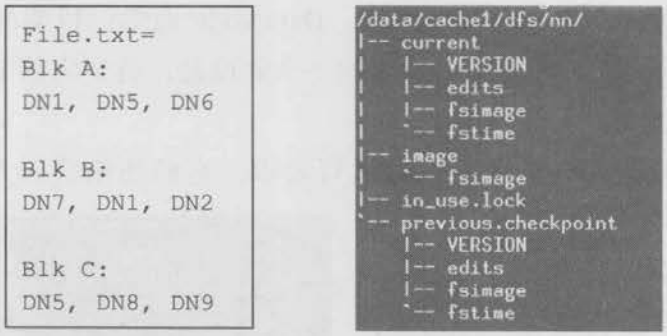
4.2.3 元数据节点

元数据节点 NameNode 是管理者，一个 Hadoop 集群这有一个 NameNode 节点，是一个通常在 HDFS 实例中的单独机器上运行的软件。它负责管理文件系统名字空间和控制外

部客户机的访问。NameNode 决定是否将文件映射到 DataNode 的复制块上。对于最常见的 3 个复制块，第一个复制块存储在同一机架的不同节点上，最后一个复制块存储在不同机架的某个节点上。注意，这里需要了解集群架构。

实际的 I/O 事务并没有经过 NameNode，只有表示 DataNode 和块的文件映射的元数据经过 NameNode。当外部客户机发送请求要求创建文件时，NameNode 会以块标识和该块的第一个副本的 DataNode IP 地址作为响应。这个 NameNode 还会通知其他将要接收该块的副本的 DataNode。

NameNode 在一个 FsImage 文件中存储所有关于文件系统名称空间的信息，如图 4-2 所示。



块存储结构　　metadate 物理

图 4-2　NameNode 的 metadata 信息

这个文件和一个包含所有事务的记录文件（这里是 EditsLog）将存储在 NameNode 的本地文件系统上。FsImage 和 EditsLog 文件也需要复制副本，以防文件损坏或 NameNode 系统丢失。NameNode 是 Master，负责管理多个 DataNode，所以容易出现单点的问题，这里比较常用的解决方案是使用 NFS 对 NameNode 上面的 FsImage 和 EditsLog 做备份。

NameNode 主要功能如下。

1）NameNode 提供名称查询服务，它是一个 Jetty 服务器。

2）NameNode 保存 metadate 信息。具体包括：文件 owership 和 permissions；文件包含哪些块；Block 保存在哪个 DataNode（由 DataNode 启动时上报）。

3）NameNode 的 metadate 信息在启动后会加载到内存。

设置一个 Block 64MB，如果上传文件小于该值，仍然会占用一个 Block 的 NameNode metadata，但是物理存储上不会占用 64MB 的空间。

VERSION 文件是一个属性文件，保存了 HDFS 的版本号等信息。VERSION 文件的信息如图 4-3 所示。

```
jayliu@Ubuntu1:/hadoop/dfs/name/current$ cat VERSION
#Sat Apr 20 22:38:26 CST 2013
namespaceID=1446620736
cTime=0
storageType=NAME_NODE
layoutVersion=-32
```

图 4-3　VERSION 信息

VERSION 的参数含义解释如下。

- namespaceID 是文件系统的唯一标识符，是在文件系统初次格式化时生成的。
- cTime 表示 FsImage 产生的时间。
- storageType 表示此文件夹中保存的是元数据节点的数据结构。
- layoutVersion 是一个负整数，保存 HDFS 持续化在硬盘上的数据结构的格式版本号。

4.2.4 数据节点

数据节点 DataNode 也是一个通常在 HDFS 实例中的单独机器上运行的软件。Hadoop 集群包含一个 NameNode 和大量 DataNode。DataNode 通常以机架的形式组织，机架通过一个交换机将所有系统连接起来。Hadoop 的一个假设是：机架内部节点之间的传输速度快于机架间节点的传输速度。

DataNode 响应来自 HDFS 客户机的读写请求，它们还响应来自 NameNode 的创建、删除和复制块的命令。NameNode 依赖来自每个 DataNode 的定期心跳（heartbeat）消息。每条消息都包含一个块报告，NameNode 可以根据这个报告验证块映射和其他文件系统元数据。如果 DataNode 不能发送心跳消息，NameNode 将采取修复措施，重新复制在该节点上丢失的块。DataNode 上面存放的数据块信息如图 4-4 所示。

```
jayliu@Ubuntu1:/hadoop/data/current$ ll
total 1413900
drwxr-xr-x 2 jayliu jayliu    12288 2013-04-20 18:39 ./
drwxr-xr-x 6 jayliu jayliu     4096 2013-04-20 00:39 ../
-rw-r--r-- 1 jayliu jayliu    20234 2013-04-20 18:36 blk_-10169756614365385l6
-rw-r--r-- 1 jayliu jayliu      167 2013-04-20 18:36 blk_-10169756014365385l6
_1050.meta
-rw-r--r-- 1 jayliu jayliu    13327 2013-04-20 00:48 blk_-1156638513958733838
-rw-r--r-- 1 jayliu jayliu      115 2013-04-20 00:48 blk_-1156638513958733838
_1050.meta
-rw-r--r-- 1 jayliu jayliu 67108864 2012-09-06 14:02 blk_-1156957313862524685
-rw-r--r-- 1 jayliu jayliu   524295 2012-09-06 14:02 blk_-1156957313862524685
_1019.meta
-rw-r--r-- 1 jayliu jayliu    20307 2012-08-22 17:11 blk_1204575313306609608
-rw-r--r-- 1 jayliu jayliu      167 2012-08-22 17:11 blk_1204575313306609608_
1011.meta
-rw-r--r-- 1 jayliu jayliu 46936400 2012-09-16 15:57 blk_-1786906826888174l9
-rw-r--r-- 1 jayliu jayliu   366651 2012-09-16 15:57 blk_-1786906826888174l9_
1021.meta
```

图 4-4 DataNode 存储块信息

DataNode 的功能如下。

- 保存 Block，每个块对应一个元数据信息文件。这个文件主要描述这个块属于那个文件、第几个块等信息；
- 启动 DataNode 线程的时候会向 NameNode 汇报 Block 信息；
- 通过向 NameNode 发送心跳保持与其联系（3 秒一次），如果 NameNode 10 分钟没有收到 DataNode 的心跳，则认为其已经 lost，并将其上的 Block 复制到其他 DataNode。

4.2.5 辅助元数据节点

辅助元数据节点 SecondaryNameNode，会周期性地将 EditsLog 文件中记录对 HDFS 的操作合并到一个 FsImage 中文件中，然后清空 EditsLog 文件。NameNode 的重启就会加载（load）最新的一个 FsImage 文件，并重新创建一个 EditsLog 文件来记录 HDFS 操作，由于 EditsLog 中记录的是从上一次 FsImage 以后到现在的操作列表，所以就会比较小。

如果没有 SecondaryNameNode 这个周期性的合并过程，当每次重启 NameNode 的时候，就会花费很长的时间。而这样周期性地合并就能减少重启的时间。同时也能保证 HDFS 系统的完整性。SecondaryNameNode 合并 EditsLog 文件的过程如图 4-5 所示。

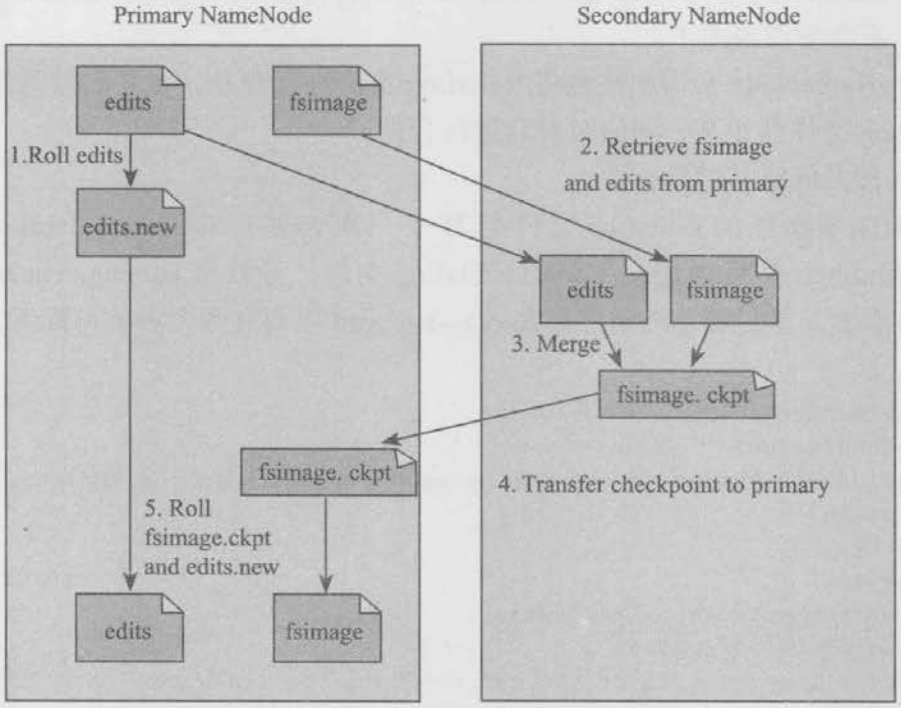


图 4-5 SecondaryNameNode 工作原理

SecondaryNameNode 合并 FsImage 和 EditsLog 文件的过程如下。

1）文件系统客户端（Client）进行写操作时，首先把它记录在修改日志中（EditsLog）。

2）元数据节点在内存中保存了文件系统的元数据信息。在记录修改日志后，元数据节点修改内存中的数据结构。

3）每次的写操作成功之前，修改日志都会同步（Sync）到文件系统。

4）FsImage 文件即名字空间映像文件，是内存中的元数据在硬盘上的 CheckPoint，它是一种序列化的格式，并不能够在硬盘上直接修改。

5）同数据的机制相似，当元数据节点失败时，最新 CheckPoint 的元数据信息从 FsImage 加载到内存中，然后逐一重新执行修改日志中的操作。

6）从元数据节点就是用来帮助元数据节点将内存中的元数据信息 CheckPoint 到硬盘上的。CheckPoint 的过程如下。

1）从元数据节点通知元数据节点生成新的日志文件，以后的日志都写到新的日志文件中。

2）从元数据节点用 HTTP Get 从元数据节点获得 FsImage 文件及旧的日志文件。

3）从元数据节点将 FsImage 文件加载到内存中，并执行日志文件中的操作，然后生成新的 FsImage 文件。

4）从元数据节点将新的 FsImage 文件用 HTTP Post 传回元数据节点。

5）元数据节点可以将旧的 FsImage 文件及旧的日志文件，换为新的 FsImage 文件和新的日志文件（第一步生成的），然后更新 fstime 文件，写入此次 CheckPoint 的时间。

6）这样元数据节点中的 FsImage 文件保存了最新 CheckPoint 的元数据信息，日志文件也重新开始，不会变得很大了。

SecondaryNameNode 会周期性地将 EditsLog 文件进行合并，合并前提条件如下：

- EditsLog 文件到达某一阈值时对其进行合并。
- 每隔一段时间对其进行合并。

将记录 HDFS 操作的 EditsLog 文件与其上一次合并后存在的 FsImage 文件合并到 Fsimage.checkpoint，然后创建一个新的 EditsLog 文件，最后将 Fsimage.checkpoint 复制到 NameNode 节点上。复制触发的条件是在 core-site.xml 里面有两个参数可配置。

```
<property>
   <name>fs.checkpoint.period</name> // 时间
<value>3600</value>
<description>The number of seconds between two periodic checkpoints.
</description>
 </property>
 <property>
    <name>fs.checkpoint.size</name>
    <value>67108864</value>          // 大小
    <description>The size of the current edit log (in bytes) that triggers
a periodic checkpoint even if the fs.checkpoint.period hasn't expired.
</description>
 </property>
```

参数解释如下。

- fs.checkpoint.period：时间间隔，默认为 1 小时合并一次。
- fs.checkpoint.size：文件大小，默认为 64MB，当 EditsLog 文件大小超过 64MB，就会触发 EditsLog 与 FsImage 文件的合并。

如果 NameNode 损坏或丢失之后，无法启动 Hadoop，这时就要人工干预恢复到 SecondaryNameNode 中所照的状态，这就意味着集群的数据会或多或少地丢失一些宕机时间，并且将 SecondaryNameNode 作为重要的 NameNode 来处理。这就要求，尽量避免将 SecondaryNameNode 和 NameNode 放在同一台机器上。

4.2.6 名字空间

HDFS 支持传统的文件组织体系结构。用户或者程序可以创建目录，并在目录中存储文件。名字空间（NameSpace）的结构和许多现有文件系统类似，可以创建、删除文件，也可以把文件从一个目录移动到另一个目录，或者重命名文件。

HDFS 还没有实现用户配额和访问权限控制，也不支持硬连接和软连接。当然体系也不妨碍实现这些特性。名字节点维护系统的名字空间，它将记录名字空间内的任何改动或

者名字空间本身的属性改动。用户可以指定 HDFS 中文件复制的份数，这个份数称为复制因子，由名字节点记录。

4.2.7 数据复制

HDFS 能够在一个大集群里跨机器、可靠地存储非常大的文件。每个文件都存储为一系列的块，同一文件中除最后一块以外的所有块都是一样大的，如图 4-6 所示。

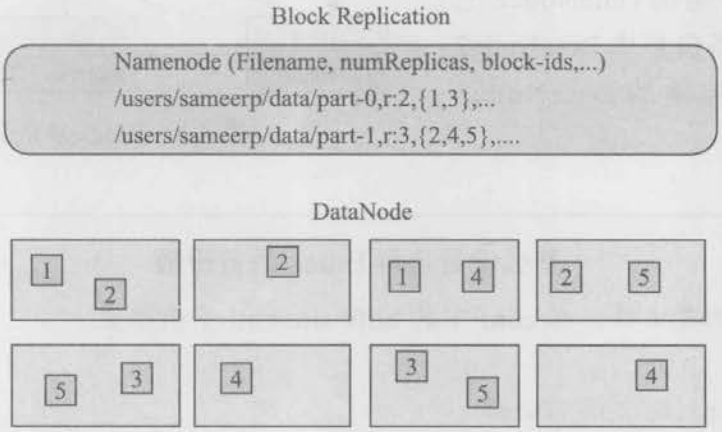


图 4-6 数据块复制

文件的块都通过复制来保证容错。每个文件的块的大小和复制因子都是可以配置的。程序可以指定文件复制的次数，复制因子可以在文件创建时候指定，也可以在以后指定。HDFS 的文件都是一次性写入的，并且严格限制任何时候只有一个写用户。名字节点根据块复制状态做出所有决定，它会周期性地收到来自集群内数据节点的心跳和块报告。能收到心跳证明数据节点工作是正常的。一个块报告应该包括数据节点上所有块的列表。

4.2.8 块备份原理

Block 是 HDFS 文件系统的最小组成单元，它通过一个 Long 整数被唯一标识。每个 Block 会有多个副本（默认为 3）。为了数据的安全和高效，Hadoop 默认对 3 个副本的存放策略如图 4-7 所示。

- 第一块：在本地机器的 HDFS 目录下存储一个 Block。
- 第二块：不同 Rack 的某个 DataNode 上存储一个 Block。
- 第三块：在该机器的同一个 Rack 下的某台机器上存储最后一个 Block。

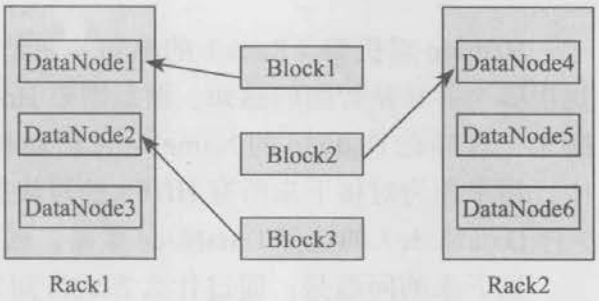


图 4-7 Block 备份规则

这样的策略可以保证对该 Block 所属文件的访问能够优先在本 Rack 下找到，如果整个

Rack 发生了异常，也可以在另外的 Rack 上找到该 Block 的副本。这样足够的高效，并且同时做到了数据的容错。

现在假如 Block 备份设置成三份，那么这三份一样的块是怎么复制到 DataNode 上的呢？Block 块的备份机制如图 4-8 所示。

假设第一个备份传到 DataNode1 上面，那么第二个备份是从 DataNode1 上面以流的形式传输的 DataNode2 上面，同样，第三个备份是从 DataNode2 上面以流的形式传输的 DataNode3 上面。

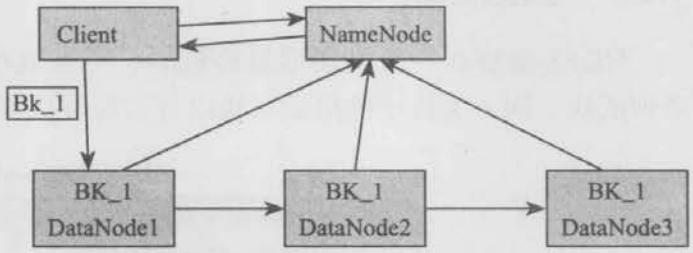


图 4-8 Block 块的备份机制

扩展阅读

怎么设置集群 Block 的备份数

方案一：修改配置文件。在 conf 下的 hdfs-site.xml 下面配置。

```
<property>
  <name>dfs.replication</name>
  <value>1</value>
   <description>Default block replication. The actual number of replications can be specified when the file is created. The default is used if replication is not specified in create time.</description>
</property>
```

默认 dfs.replication 的值为 3。通过这种方案参数其实只在文件被写入 dfs 时起作用，虽然更改了配置文件，但是不会改变之前写入的文件的备份数。

方案二：通过命令更改备份数。

```
bin/hadoop fs -setrep -R 1 /
```

这样可以改变整个 HDFS 里面的备份数，不需要重启 HDFS 系统。方案一需要重启 HDFS 系统才能生效。

4.2.9 机架感知

Hadoop 对机架（Rack）的感知并非是自适应的，即 Hadoop 集群需要手动指定机器是属于哪个并非是智能的感知，而是需要 Hadoop 的管理者人为地告知 Hadoop 哪台机器属于哪个，这样在 Hadoop 的 NameNode 启动初始化时，会将这些机器与对应信息保存在内存中，用来作为对接下来所有 HDFS 的写块操作分配 DataNode 列表时（比如 3 个 Block 对应 3 台 DataNode）的选择 DataNode 策略，做到：尽量将 3 个副本分布到不同的节点上。

接下来的问题是：通过什么方式告知 Hadoop 哪些 Slave 机器属于哪个 Rack？以下是配置步骤。

默认情况下，Hadoop 的机架感知是没有启用的。所以，通常情况下，Hadoop 集群的

HDFS 在选机器的时候是随机选择的。也就是说，很有可能在写数据时，Hadoop 将第一块数据 Block1 写到 Rack1，然后随机选择下将 Block2 写入 Rack2，此时两个之间产生了数据传输的流量。接下来，在随机情况下，将 Block3 重新写回 Rack1，此时，两个之间又产生一次数据流量。在 Job 处理的数据量非常的大，或者往 Hadoop 推送的数据量非常大的时候，这种情况会造成之间的网络流量成倍上升，成为性能的瓶颈，进而影响作业的性能以至于影响整个集群的服务。

启用 Hadoop 机架感知功能，配置非常简单，在 NameNode 所在机器的 hadoop-site.xml 文件中配置一个选项：

```
<property>
 <name>topology.script.file.name</name>
 <value>/path/to/script</value>
</property>
```

这个配置选项的 value 指定为一个可执行程序，通常为一个脚本，该脚本接受一个参数，输出一个值。接受的参数通常为某台 DataNode 机器的地址，而输出的值通常为该地址对应的 DataNode 所在的地址，例如“ /rack1”。NameNode 启动时，会判断该配置选项是否为空，如果非空，表示已经用机架感知的配置，此时 NameNode 会根据配置寻找该脚本，并在接收到每一个 DataNode 的 heartbeat 时，将该 DataNode 的地址作为参数传给该脚本运行，并将得到的输出作为该 DataNode 所属的机架，保存到内存的一个 Map 中。

至于脚本的编写，就需要将真实的网络拓扑和机架信息了解清楚后，通过该脚本能够将机器的地址正确地映射到相应的机架上去。一个简单的实现如下：

```
#!/usr/bin/perl -w
use strict;
my $ip = $ARGV[0];
my $rack_num = 3;
my @ip_items = split /./, $ip;
my $ip_count = 0;
foreach my $i (@ip_items) {
$ip_count += $i;
}
my $rack = "/rack".($ip_count % $rack_num);
print "$rack";
```

4.3 Hadoop 的 RPC 机制

一般我们所了解的 RPC（Remote Procedure Call，远程过程调用）机制都要面对两个问题：1）对象调用方式；2）序列 / 反序列化机制。

RPC 架构如图 4-9 所示，基于此的 RPC 产品与框架也很多，但 Hadoop 却自己实现了简单的 RPC 组件，依赖于 Hadoop Writable 类型的支持。

Hadoop Writable 接口要求每个实现类都要确保将本类的对象正确序列化（writeObject）与反序列化（readObject）。因此，Hadoop RPC 使用 Java 动态代理与反射实现对象调用方式，

客户端到服务器数据的序列化与反序列化由 Hadoop 框架或用户自己来实现，也就是数据组装是定制的。

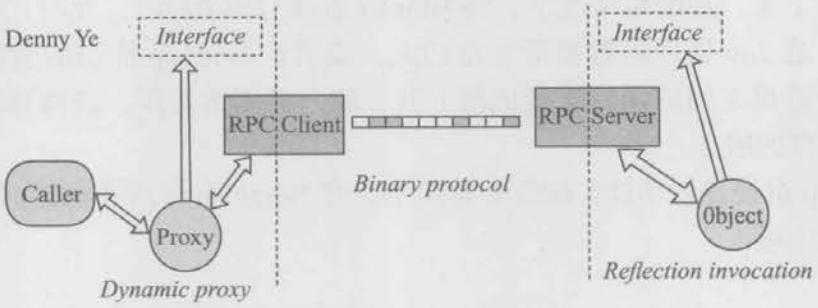


图 4-9 RPC 架构

4.3.1 RPC 的实现流程

简单来说，Hadoop RPC = 动态代理 + 定制的二进制流。如果不关注细节，从用户的角度来看，RPC 的实现流程如下。

远程的对象拥有固定的接口，这个接口用户也是可见的，只是真正的实现（Object）只在服务端。用户如果想使用哪个实现，调用过程是：先根据那个接口动态代理生成一个代理对象，调用这个代理对象的时候，用户的调用请求被 RPC 捕捉到，然后包装成调用请求，序列化成数据流发送到服务端；服务端从数据流中解析出调用请求，然后根据用户所希望调用的接口，调用接口真正的实现对象，再把调用结果返回给客户端。

提示 这种实现很粗犷，它屏蔽了其他框架对 Hadoop 的影响，对风险可控，又不失性能。

4.3.2 RPC 的实体模型

上一部分是站在用户的角度，宏观地观察整个调用过程。这节在细节上分析 RPC 都有哪些实体。为什么要提到这些实体呢？如果把 RPC 流程看做流水线，这些实体就是一个个做具体工作的工人，如果想深入了解流水线的处理，就要知道每个工人工作的职责及概况。

RPC 在客户端的细节不多，只想提一点，就是用户在调用代理对象时 RPC 是怎样拦截这次调用请求的。对动态代理清楚的朋友都知道，创建代理对象时需要为它关联一个 InvocationHandler，对代理对象的每次调用都会进入绑定的 InvocationHandler 中，RPC 就从这里获取用户的请求，这里没有疑点。

RPC 在服务端的模型由一系列实体组成，分别负责调用的整个流程。各个实体分工明确，各司其职。下面一一介绍。

- Listener：监听 RPC Server 的端口，如果客户端有连接请求到达，它就接受连接，然后把连接转发到某个 Reader，让 Reader 读取那个连接的数据。如果有多个 Reader，当有新连接过来时，就在这些 Reader 间顺序分发。这里需要提到的是，

Hadoop 0.21 版本在支持多 Reader 时有个 bug（JIRA），如果有 Reader 在 Server 运行期没被使用，Server 进程不能正常关闭。

- Reader：从某个客户端连接中读取数据流，把它转化成调用对象（call），然后放到调用队列（call queue）里。
- Handler：真正做事的实体。它从调用队列中获取调用信息，然后反射调用真正的对象，得到结果，再把此次调用放到响应队列（response queue）里。
- Responder：不断地检查响应队列中是否有调用信息，如果有，就把调用的结果返回给客户端。

RPC Server 架构如图 4-10 所示。

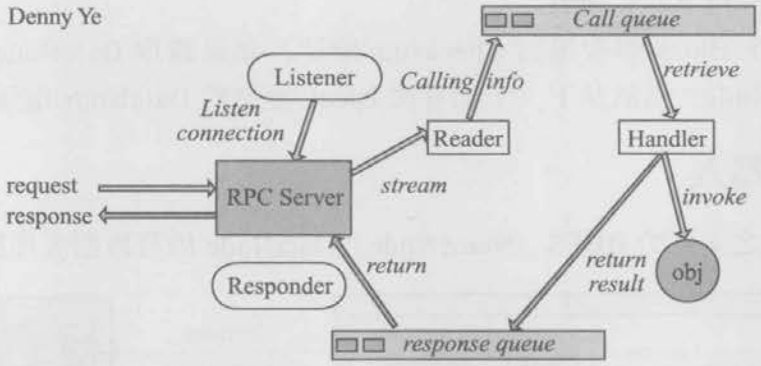


图 4-10 RPC Server 架构

提示 整个调用流程中与网络有关的地方都是用 NIO 来处理的。

4.3.3 文件的读取

客户端以及与之交互的 HDFS、NameNode、DataNode 的读数据流如图 4-11 所示。

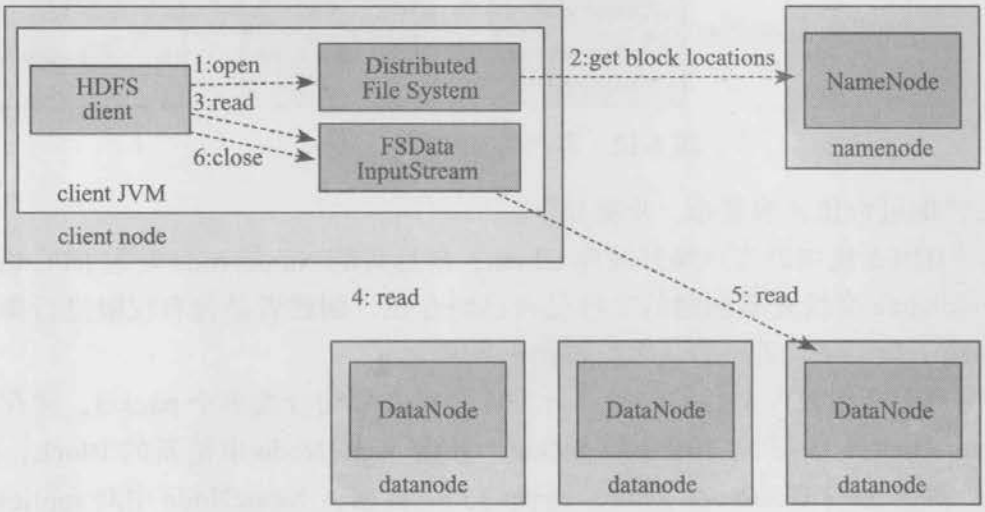


图 4-11 客户端与 HDFS 读交互

文件读取的过程如下。

1）使用 HDFS 提供的客户端开发库 Client，向远程的 NameNode 发起 RPC 请求。

2）NameNode 会视情况返回文件的部分或者全部 Block 列表，对于每个 Block，NameNode 都会返回有该 Block 副本的 DataNode 地址。

3）客户端开发库 Client 会选取离客户端最接近的 DataNode 来读取 Block；如果客户端本身就是 DataNode，将从本地直接获取数据。

4）读取完当前 Block 的数据后，关闭与当前的 DataNode 连接，并为读取下一个 Block 寻找最佳的 DataNode。

5）当读完列表的 Block 后，且文件读取还没有结束，客户端开发库会继续向 NameNode 获取下一批 Block 列表。

6）读取完一个 Block 都会进行 Checksum 验证，如果读取 DataNode 时出现错误，客户端会通知 NameNode，然后从下一个拥有该 Block 复制的 DataNode 继续读。

4.3.4 文件的写入

客户端以及与之交互的 HDFS、NameNode、DataNode 的写数据流如图 4-12 所示。

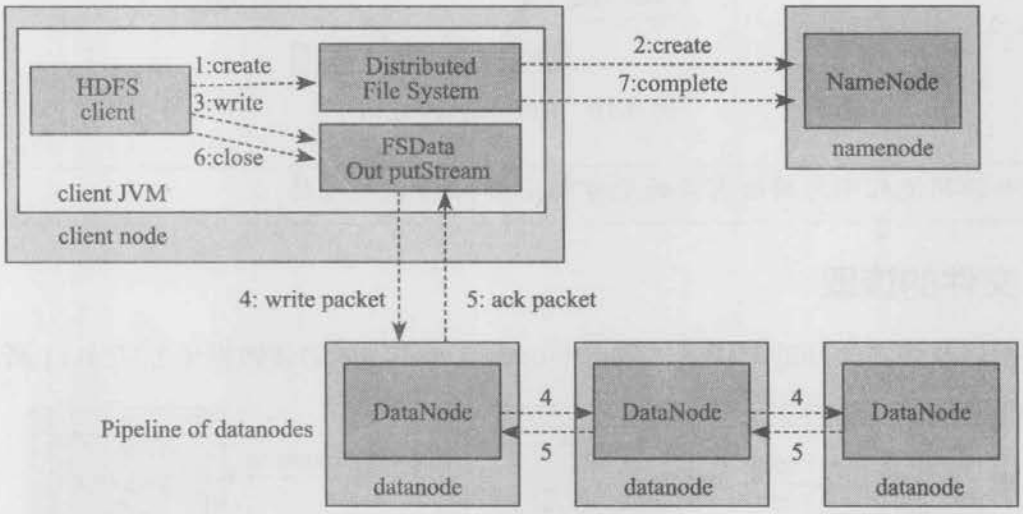


图 4-12 客户端与 HDFS 写交互

写入文件的过程比读取复杂，步骤如下。

1）使用 HDFS 提供的客户端开发库 Client，向远程的 NameNode 发起 RPC 请求。

2）NameNode 会检查要创建的文件是否已经存在，创建者是否有权限进行操作，成功会为文件创建一个记录，否则会让客户端抛出异常。

3）当客户端开始写入文件的时候，开发库会将文件切分成多个 packet，并在内部以数据队列（data queue）的形式管理这些 packet，并向 NameNode 申请新的 Block，获取用来存储 replicas 的合适的 DataNode 列表，列表的大小根据在 NameNode 中对 replication 的设置而定。

4）开始以管道（pipeline）的形式将 packet 写入所有的 replicas 中。开发库把 packet 以流的方式写入第一个 DataNode，该 DataNode 把 packet 存储之后，再将其传递给在此管道中的下一个 DataNode，直到最后一个 DataNode，这种写数据的方式呈流水线的形式。

5）最后一个 DataNode 成功存储之后会返回一个 ack packet，在管道里传递至客户端，在客户端的开发库内部维护着 ack queue，成功收到 DataNode 返回的 ack packet 后会从 ack queue 移除相应的 packet。

6）如果传输过程中，有某个 DataNode 出现了故障，当前的管道会被关闭，出现故障的 DataNode 会从当前的管道中移除，剩余的 Block 会继续剩下的 DataNode 中继续以管道的形式传输，同时 NameNode 会分配一个新的 DataNode，保持 replicas 设定的数量。

4.3.5 文件的一致模型

HDFS 应用需要一个“一次写入，多次读取”的文件访问模型。一个文件经过创建、写入和关闭之后就不需要改变。这一假设简化了数据一致性问题，并且使高吞吐量的数据访问成为可能。MapReduce 应用或者网络爬虫应用都非常适合这个模型。目前计划在将来扩充这个模型，使之支持文件的附加写操作。

4.4 HDFS 的 HA 机制

Hadoop 2.0.0 版本之前，NameNode 是 HDFS 集群的单点故障点，每一个集群只有一个 NameNode，如果这个机器或进程不可用，整个集群就无法使用，直到重启 NameNode 或者新启动一个 NameNode 节点。

影响 HDFS 集群不可用主要包括以下两种情况。

- 类似机器宕机这样的意外情况将导致集群不可用，只有重启 NameNode 之后才可使用。
- 计划内的软件或硬件升级（NameNode 节点），将导致集群在短时间范围内不可用。

HDFS 的高可用性（High Availability，HA）就可以解决上述问题，通过提供选择运行在同一集群中的一个热备用的“主 / 备”两个冗余 NameNode，允许在机器宕机或系统维护的时候，快速转移到另一个 NameNode。

4.4.1 HA 集群

一个典型的 HA 集群，两个单独的机器配置为 NameNode，在任何时候，一个 NameNode 处于活动状态，另一个处于待机状态，活动的 NameNode 负责处理集群中所有客户端的操作，待机时仅仅作为一个 Slave，保持足够的状态，如果有必要提供一个快速的故障转移。

为了保持备用节点与活动节点状态的同步，目前的实现需要两个节点同时访问一个共享存储设备（例如从 NAS、NFS 挂载）到一个目录。将有可能在未来的版本中放宽此限制。

当活动节点对名字空间进行任何修改，它将把修改记录写到共享目录下的一个日志文

件，备用节点会监听这个目录，当发现更改时，它会把修改内容同步到自己的名字空间。备用节点在故障转移时，它将保证已经读取了所有共享目录内的更改记录，保证在发生故障前的状态与活动节点保持完全一致。

为了提供快速的故障转移，必须保证备用节点有最新的集群中块的位置信息，为了达到这一点，DataNode 节点需要配置两个 NameNode 的位置，同时发送块的位置信息和心跳信息到两个 NameNode。

任何时候只有一个 NameNode 处于活动状态，对于 HA 集群的操作是至关重要的，否则两个节点之间的状态就会产生冲突、数据丢失或其他不正确的结果。为了达到这个目的或者所谓的“脑裂场景”出现，管理员必须为共享存储配置至少一个 fencing 方法。在宕机期间，如果不能确定之间的活动节点已经放弃活动状态，fencing 进程负责中断以前的活动节点编辑存储的共享访问。这可以防止任何进一步的修改名字空间，允许新的活动节点安全地进行故障转移。

注意 目前，只有手动故障转移支持。这就意味着 HA NameNode 不能自动检测活动 NameNode 的失败，而是通过手动启动故障转移。自动故障检测和故障转移将在未来的版本中实现。

4.4.2 HA 架构

HA 架构如图 4-13 所示。

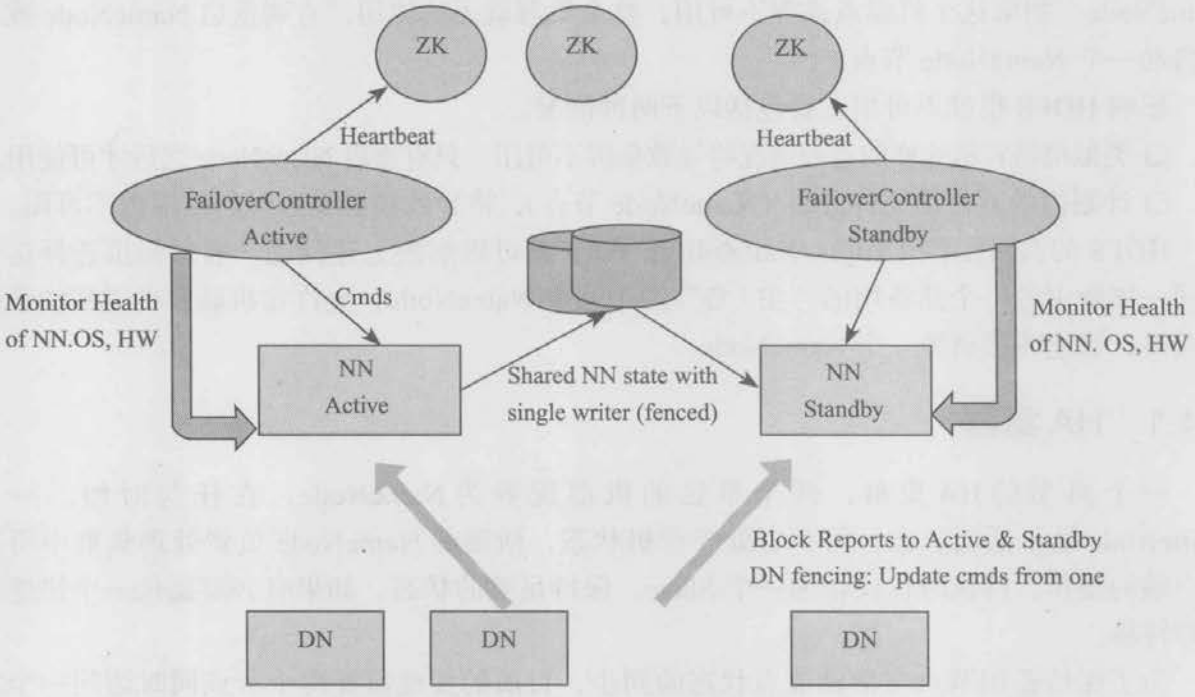


图 4-13 HA 架构

HA 架构解释如下。

- 只有一个 NameNode 是 Active 的，并且只有这个 ActiveNameNode 能提供服务，改变 NameSpace。以后可以考虑让 StandbyNameNode 提供读服务。
- 提供手动 Failover，在升级过程中，Failover 在 NameNode-DataNode 之间写不变的情况下才能生效。
- 在之前的 NameNode 重新恢复之后，不能提供 failback。
- 数据一致性比 Failover 更重要。
- 尽量少用特殊的硬件。
- HA 的设置和 Failover 都应该保证在两者操作错误或者配置错误的时候，不得导致数据损坏。
- NameNode 的短期垃圾回收不应该触发 Failover。
- DataNode 会同时向 NameNode Active 和 NameNode Standby 汇报块的信息。NameNode Active 和 NameNode Standby 通过 NFS 备份 MetaData 信息到一个磁盘上面。

4.4.3 为什么会有 HA 机制

1. 单点故障

在 Hadoop 2.0 之前，也有若干技术试图解决单点故障的问题，在这里做简短的介绍。

- Secondary NameNode：它不是 HA，只是阶段性地合并 Edits 和 FsImage，以缩短集群启动的时间。当 NameNode（以下简称 NN）失效的时候，Secondary NN 无法立刻提供服务，Secondary NN 甚至无法保证数据完整性。如果 NN 数据丢失，在上一次合并后的文件系统的改动会丢失。
- Backup NameNode（HADOOP-4539）：它在内存中复制了 NN 的当前状态，算是 Warm Standby，可仅限于此，并没有 Failover 等。它同样是阶段性地做 CheckPoint，也无法保证数据完整性。
- 手动把 name.dir 指向 NFS：这是安全的 Cold Standby，可以保证元数据不丢失，但集群的恢复则完全靠手动。
- Facebook AvatarNode：Facebook 有强大的运维做后盾，所以 AvatarNode 只是 Hot Standby，并没有自动切换，当主 NN 失效的时候，需要管理员确认，然后手动把对外提供服务的虚拟 IP 映射到 Standby NN，这样做的好处是确保不会发生脑裂的场景。其某些设计思想和 Hadoop 2.0 里的 HA 非常相似，从时间上来看，Hadoop 2.0 应该是借鉴了 Facebook 的做法。

还有若干解决方案，基本都是依赖外部的 HA 机制，如 DRBD、Linux HA、VMware 的 FT 等。

2. 集群容量和集群性能

单 NameNode 的架构使得 HDFS 在集群扩展性和性能上都有潜在的问题，当集群大到

一定程度后，NameNode 进程使用的内存可能会达到上百 GB，常用的估算公式为 1GB 对应 1 百万个块，按默认块大小计算，大概是 64TB（这个估算比例是有比较大的富裕，其实，即使是每个文件只有一个块，所有元数据信息也不会有 1KB/block）。同时，所有的元数据信息的读取和操作都需要与 NN 进行通信，如客户端的 addBlock、getBlockLocations，还有 DataNode 的 blockRecieved、sendHeartbeat、blockReport，在集群规模变大后，NameNode 成为性能的瓶颈。Hadoop 2.0 里的 HDFS Federation 就是为了解决这两个问题而开发的。

4.5 HDFS 的 Federation 机制

HDFS 架构中（Hadoop 2.0.0 之前），在整个 HDFS 集群中只有一个名字空间，并且只有单独的一个 NameNode，这个 NameNode 负责对这单独的一个名字空间进行管理。这也正是单点失效（Single Point Failure）的隐患所在。本文所讲的 HDFS Federation 就是针对当前 HDFS 架构上的缺陷所做的改进。

简单说，HDFS Federation 就是使得 HDFS 支持多个名字空间，并且允许在 HDFS 中同时存在多个 NameNode。

4.5.1 单个 NameNode 的 HDFS 架构的局限性

简单回顾一下目前 HDFS 的架构，如图 4-14 所示。

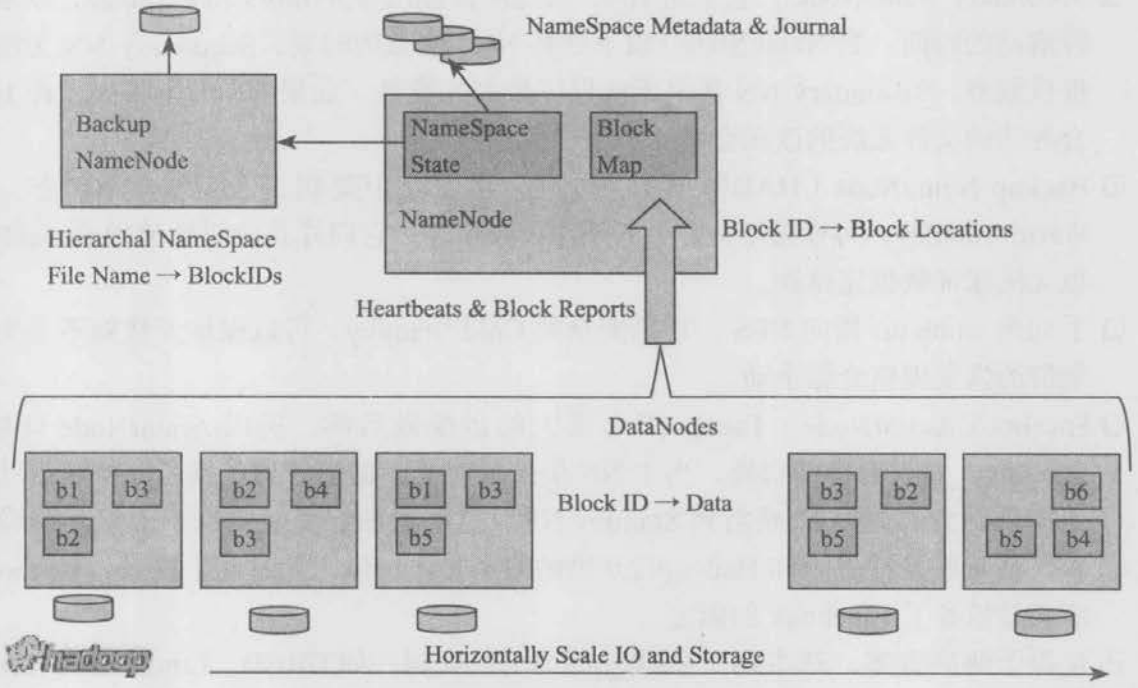


图 4-14 HDFS 架构

在整个 HDFS 集群中只有一个 NameNode，还有一个 Backup NameNode。NameNode 会实时将变化的 HDFS 的信息同步给 Backup NameNode。Backup NameNode 顾名思义是用

来做 NameNode 的备份。NameNode 中名字空间以层次结构组织中存储着文件名和 BlockID 的对应关系、BlockID 和具体 Block 位置的对应关系。这个单独的 NameNode 管理着数个 DataNode，Block 分布在各个 DataNode 中，每个 DataNode 会周期性地向此 NameNode 发送心跳消息，报告自己所在 DataNode 的使用状态。Block 是用来存储数据的最小单元，通常一个文件会存储在一个或者多个 Block 中，默认 Block 大小为 64MB。

由于 NameNode 在内存中存储所有的元数据（MetaData），因此单个 NameNode 所能存储的对象（文件 + 块）数目受到 NameNode 所在 JVM 的 heap size 的限制。50GB 的 heap 能够存储 20 亿个对象，这 20 亿个对象支持 4000 个 DataNode，12PB 的存储（假设文件平均大小为 40MB）。

随着数据的飞速增长，存储的需求也随之增长。单个 DataNode 从 4TB 增长到 36TB，集群的尺寸增长到 8000 个 DataNode。存储的需求从 12PB 增长到大于 100PB。

单个 NameNode 的 HDFS 架构的局限性如下所示。

- **性能的瓶颈**：由于是单个 NameNode 的 HDFS 架构，因此整个 HDFS 文件系统的吞吐量受限于单个 NameNode 的吞吐量。毫无疑问，这将成为下一代 MapReduce 的瓶颈。
- **隔离问题**：由于 HDFS 仅有一个 NameNode，无法隔离各个程序，因此 HDFS 上的一个实验程序就很有可能影响整个 HDFS 上运行的程序。那么在 HDFS Federation 中，可以用不同的 NameSpace 来隔离不同的用户应用程序，使得不同 NameSpace Volume 中的程序相互不影响。
- **集群的可用性**：在只有一个 NameNode 的 HDFS 中，此 NameNode 的宕机无疑会导致整个集群不可用。
- **NameSpace 和 Block Management 的紧密耦合**：当前在 NameNode 中的 NameSpace 和 Block Management 组合的紧密耦合关系会导致如果想要实现另外一套 NameNode 方案比较困难，而且也限制了其他想要直接使用块存储的应用。

4.5.2 为什么引入 Federation 机制

引入 Federation 的最主要原因是对 HDFS 系统中文件的隔离，Federation 能够快速解决大部分单 NameNode HDFS 的问题。

Federation 是简单鲁棒的设计，由于联盟中各个 NameNode 之间是相互独立的，Federation 整个核心设计实现大概用了三个半月。大部分改变是在 DataNode、Config 和 Tools，而 NameNode 本身的改动非常少，这样 NameNode 原先的鲁棒性不会受到影响。

Federation 比分布式的 NameNode 简单，虽然这种实现的扩展性比起真正的分布式 NameNode 要小些，但是可以迅速满足需求。

另外一个原因是 Federation 良好的向后兼容性，已有的单 NameNode 的部署配置不需要任何改变就可以继续工作。

4.5.3 Federation 架构

HDFS Federation 使用了多个独立的 NameNode/NameSpace 使得 HDFS 的命名服务能够水平扩展，如图 4-15 所示。

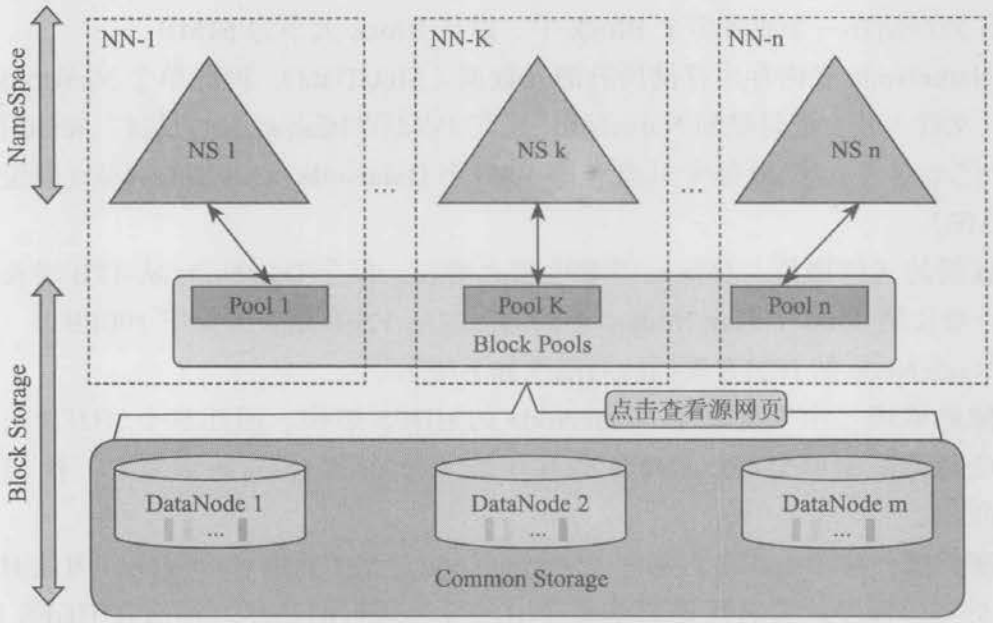


图 4-15 Federation 架构

HDFS Federation 中的 NameNode 之间是联盟关系，它们之间相互独立且不需要相互协调。HDFS Federation 中的 NameNode 提供了名字空间和块管理功能。HDFS Federation 中的 DataNode 被所有的 NameNode 用作公共存储块的地方。每一个 DataNode 都会向所在集群中所有的 NameNode 注册，并且会周期性的发送心跳和块信息报告，同时处理来自 NameNode 的指令。

当前 HDFS 只有一个名字空间（NameSpace），它使用全部的块。而 Federation HDFS 中有多个独立的 NameSpace，并且每一个名字空间使用一个块池（Block Pool）。

当前 HDFS 中只有一组块。而 Federation HDFS 中有多组独立的块。块池就是属于同一个名字空间的一组块。

当前 HDFS 由一个 NameNode 和一组 DataNode 组成。而 Federation HDFS 由多个 NameNode 和一组 DataNode，每一个 DataNode 会为多个块池存储块。

所谓块池就是属于单个名字空间的一组 Block。每一个 DataNode 为所有的块池存储块。DataNode 是一个物理概念，而块池是一个重新将块划分的逻辑概念。同一个 DataNode 中可以存着属于多个块池的多个块。块池允许一个名字空间在不通知其他名字空间的情况下，为一个新的 Block 创建 Block ID。同时，一个 NameNode 失效不会影响其下的 DataNode 为其他 NameNode 的服务。

当 DataNode 与 NameNode 建立联系并开始会话后自动建立块池。每个 Block 都有唯一标识，这个标识称为扩展块 ID（Extended Block ID = BlockID+BlockID）。这个扩展的块 ID 在 HDFS 集群之间都是唯一的，这为以后集群归并创造了条件。

DataNode 中的数据结构都通过块池 ID（BlockPoolID）索引，即 DataNode 中的 BlockMap、Storage 等都通过 BPID 索引。

在 HDFS 中，所有的更新、回滚都是以 NameNode 和 BlockPool 为单元发生的。即同 HDFS Federation 中不同的 NameNode/BlockPool 之间没有什么关系。

在 DataNode 中，对应于每个 NameNode 都有一个相应的线程。每个 DataNode 会去每一个 NameNode 注册，并且周期性地给所有的 NameNode 发送心跳及 DataNode 的使用报告。DataNode 还会给 NameNode 发送其所在的块池的块报告（block report）。由于有多个 NameNode 同时存在，因此任何一个 NameNode 都可以随时动态加入、删除和更新。

4.5.4 多个名字空间的管理问题

在一个集群中需要唯一的名字空间还是多个名字空间，核心问题是名字空间中数据的共享和访问的问题。使用全局唯一的名字空间是解决数据共享和访问的一种方法。在多个名字空间下，还可以使用 Client Side Mount Table 方式做到数据共享和访问。

如图 4-16 所示，每个深色三角形代表一个独立的名字空间，上方浅色的三角形代表从客户角度访问下方的子名字空间。各个深色的名字空间挂载到浅色的表中，客户可以访问不同的挂载点来访问不同的名字空间，这就如同在 Linux 系统中访问不同挂载点一样。这就是 HDFS Federation 中名字空间管理的基本原理：将各个名字空间挂载到全局 mount-table 中，就可以将数据到全局共享；同样，名字空间挂载到个人的 mount-table 中，就成为应用程序可见的名字空间视图。

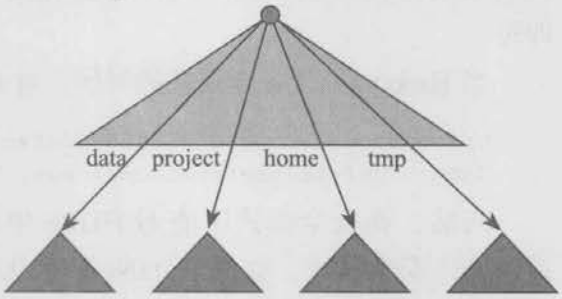


图 4-16 Federation 中名字空间

Federation 中其他方面的改进如下。

- 提供了工具，对于 NameNode 的初始化和退役的监控和管理。
- 允许在 DataNode 级别或者块池级别的负载均衡。
- DataNode 的后台守护进程，为 Federation 所做的磁盘和目录扫描。
- 提供了显示 NameNode 的块池使用状态的 Web UI。
- 提供了对全部集群存储使用状态的 UI 展示。
- 在 Web UI 中列出了所有 NameNode 及其细节，如 NameNode-BlockPoolID 和存储的使用态，失去联系的活的和死的块信息，以及前往各个 NameNode Web UI 的链接。
- DataNode 退役状态的展示。

扩展阅读

关于 HDFS 的 HA 和 Federation

HDFS 的 HA 和 Federation 两个功能是 Hadoop 2.0.0 以后才有的，因为现在 Hadoop 2.0.0 是测试版本，所以目前公司都在调研、测试阶段。HA 解决了 HDFS 的 NameNode 的单点的问题。Federation 解决了整个 HDFS 集群中只有一个名字空间，并且只有单独的一个 NameNode 管理所有的 DataNode 的问题。

4.6 Hadoop 文件系统的访问

在分布式文件系统启动的时候，开始时会有安全模式，当分布式文件系统处于安全模式的情况下，文件系统中的内容不允许修改也不允许删除，直到安全模式结束。

4.6.1 安全模式

安全模式主要是为了系统启动的时候检查各个 DataNode 上数据块的有效性，同时根据策略复制或者删除部分数据块。运行期通过命令也可以进入安全模式。在实践过程中，系统启动的时候修改和删除文件也会有安全模式不允许修改的出错提示，只需要等待一会儿即可。

当 Hadoop 处于安全模式的时候，对 HDFS 进行上传、删除、修改等操作时会报以下错误：

```
org.apache.hadoop.dfs.SafeModeException:
Cannot delete/user/hadoop/input. Name node is in safe mode.
```

当然，在安全模式下查看 HDFS 里面的文件是没有问题的。NameNode 在启动的时候首先进入安全模式，如果 DataNode 丢失的 Block 达到一定比例（fs.safemode.threshold.pct），系统会一直处于安全模式状态（即只读状态）。

dfs.safemode.threshold.pct（默认值 0.999f）表示 HDFS 启动的时候，如果 DataNode 上报的 Block 个数达到了元数据记录的 Block 个数的 0.999 倍才可以离开安全模式，否则一直是这种只读模式。如果设为 1 则 HDFS 永远处于安全模式（SafeMode）。

下面这行内容摘录自 NameNode 启动时的日志（Block 上报比例 1 达到阈值 0.9990）。

```
The ratio of reported blocks 1.0000 has reached the threshold0.9990. Safe mode
will be turned off automatically in 18seconds.
```

安全模式格式如下。

```
Usage: java DFSAdmin [-safemode enter |
leave | get |wait]
```

用户可以通过“hadoop dfsadmin -safemode value”来操作安全模式，安全模式操作如图 4-17 所示。

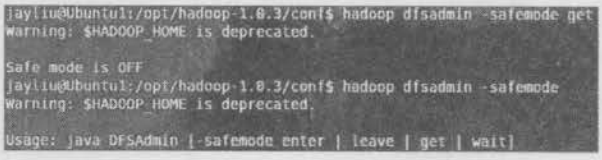


图 4-17 安全模式操作

参数 value 的说明如下。

- enter：进入安全模式。
- leave：强制 NameNode 离开安全模式。
- get ：返回安全模式是否开启的信息。
- wait ：等待，一直到安全模式结束。

4.6.2 HDFS 的 Shell 访问

HDFS 设计的主要目的是对海量数据进行处理，也就是说在其上能够存储很大量文件（可以存储 TB 级的文件）。HDFS 将这些文件分割之后，存储在不同的 DataNode 上，HDFS 提供了一个 Shell 接口，对 HDFS 里面的文件进行操作，具体每个 Block 放在哪台 DataNode 上面，对于开发者来说是透明的。所有的 Hadoop 命令均由 bin/hadoop 脚本引发。不指定参数运行 Hadoop 脚本会打印所有命令的描述。

Hadoop 有一个选项解析框架用于解析一般的选项和运行类。

```
hadoop [--config confdir] [COMMAND] [GENERIC_OPTIONS] [COMMAND_OPTIONS]
```

Shell 命令参数解释如表 4-2 所示。

表 4-2 Shell 参数解释

命令选项	功能描述
--config confdir	覆盖默认配置目录。默认是 ${HADOOP_HOME}/conf
GENERIC_OPTIONS	多个命令都支持的通用选项
COMMAND 命令选项 S	各种各样的命令和它们的选项会在下面提到。这些命令被分为 用户命令和管理命令两组

4.6.3 HDFS 处理文件的命令

HDFS 处理文件的命令和 Linux 命令基本相同，这里区分大小写。下面介绍 HDFS 操作分布式文件系统的命令。

1. fs 命令

键入命令“./bin/hadoop fs”将输出能够支持的命令列表。

```
Usage: java FsShell
       [-ls <path>]
       [-lsr <path>]
       [-df [<path>]]
       [-du [-s] [-h] <path>]
       [-dus <path>]
       [-count[-q] <path>]
       [-mv <src> <dst>]
       [-cp <src> <dst>]
       [-rm [-skipTrash] <path>]
       [-rmr [-skipTrash] <path>]
```

```
        [-expunge]
        [-put <localsrc> ... <dst>]
        [-copyFromLocal <localsrc> ... <dst>]
        [-moveFromLocal <localsrc> ... <dst>]
        [-get [-ignoreCrc] [-crc] <src> <localdst>]
        [-getmerge <src> <localdst> [addnl]]
        [-cat <src>]
        [-text <src>]
        [-copyToLocal [-ignoreCrc] [-crc] <src> <localdst>]
        [-moveToLocal [-crc] <src> <localdst>]
        [-mkdir <path>]
        [-setrep [-R] [-w] <rep> <path/file>]
        [-touchz <path>]
        [-test -[ezd] <path>]
        [-stat [format] <path>]
        [-tail [-f] <file>]
        [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
        [-chown [-R] [OWNER][:[GROUP]] PATH...]
        [-chgrp [-R] GROUP PATH...]
        [-help [cmd]]
```

如果熟悉 Linux，这些命令基本不需要解释。下面列出几个常用场景下的命令。

1）创建文件夹。

HDFS 上的文件目录结构类似 Linux，根目录使用“/”表示。下面的命令将在 /user/hadoop 目录下建立目录 input。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs -mkdir /user/hadoop/input
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs -ls /user/hadoop/
Found 1 items
drwxr-xr-x  - jayliu supergroup          0 2013-04-21 23:38 /user/hadoop/input
```

2）上传文件 test.txt 到 input 下。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs -put test.txt /user/hadoop/input/
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs -ls /user/hadoop/input/
Found 1 items
-rw-r--r--  1 jayliu supergroup         75 2013-04-21 23:45 /user/hadoop/input/test.txt
```

还可以用 -copyFromLocal 参数。

3）查看 test.txt 文件内容。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs -text /user/hadoop/input/test.txt
hello world
hello hadoop
```

还可以用 -cat、-tail 参数查看文件的内容。但是对于压缩的结果文件只能用 -text 参数来查看，否则是乱码。

4）把 test.txt 文件复制到本地。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs -get /user/hadoop/input/test.txt
```

还可以用 -copyToLocal 参数。

5）删除 test.txt 文件。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs -rm /user/hadoop/input/test.txt
```

2. namenode 命令

运行 namenode 进行格式化、升级、回滚等操作。键入“./bin/hadoop namenode”将输出能够支持的命令列表。

```
hadoop namenode [-format] | [-upgrade] | [-rollback] | [-finalize] | [-importCheckpoint]
```

参数解释如表 4-3 所示。

表 4-3 namenode 命令参数说明

命令选项	功能描述
-format	格式化 NameNode。启动 NameNode，格式化 NameNode，之后关闭 NameNode
-upgrade	分发新版本的 Hadoop 后，NameNode 应以 upgrade 选项启动
-rollback	将 NameNode 回滚到前一版本。这个选项要在停止集群、分发 Hadoop 旧版本后使用
-finalize	删除文件系统的前一状态。最近的升级会被持久化，rollback 选项将再不可用升级终结操作之后，它会停掉 NameNode
-importCheckpoint	从检查点目录装载镜像并保存到当前检查点目录，检查点目录由 fs.checkpoint.dir 指定

说明 Hadoop 集群第一次启动时一定要格式化 HDFS 系统，否则 NameNode 会报错。

3. secondarynamenode 命令

运行 secondarynamenode 命令对 EditsLog 进行操作（如获取大小等）。键入“./bin/hadoop secondarynamenode”将输出能够支持的命令列表。

```
hadoop secondarynamenode [-checkpoint [force]] | [-geteditsize]
```

secondarynamenode 参数解释如表 4-4 所示。

表 4-4 secondarynamenode 命令参数说明

命令选项	功能描述
-checkpoint [force]	如果 EditsLog 的大小≥ fs.checkpoint.size，启动 Secondary NameNode 的检查点过程。如果使用了 -force，将不考虑 EditsLog 的大小
-geteditsize	打印 EditsLog 大小

4. dfsadmin 命令

该命令运行一个 HDFS 的 dfsadmin 客户端。键入“./bin/hadoop dfsadmin”将输出能够支持的命令列表。

```
hadoop dfsadmin [GENERIC_OPTIONS] [-report] [-safemode enter | leave | get | wait] [-refreshNodes] [-finalizeUpgrade] [-upgradeProgress status | details | force] [-metasave filename] [-setQuota <quota> <dirname>...<dirname>] [-clrQuota <dirname>...<dirname>] [-help [cmd]]
```

dfsadmin 命令参数解释如表 4-5 所示。

表 4-5 dfsadmin 命令参数说明

命令选项	功能描述
-report	报告文件系统的基本信息和统计信息
-safemode enter | leave | get | wait	安全模式维护命令。安全模式是 NameNode 的一个状态，在这种状态下，NameNode 不接受对名字空间的更改（只读）和不复制或删除块。NameNode 会在启动时自动进入安全模式，当配置的块最小百分比数满足最小的副本数条件时，会自动离开安全模式。安全模式可以手动进入，但是这样的话也必须手动关闭安全模式
-refreshNodes	重新读取 hosts 和 exclude 文件
-finalizeUpgrade	终结 HDFS 的升级操作
-upgradeProgress status | details | force	请求当前系统的升级状态、状态的细节，或者强制升级操作进行
-metasave filename	保存 NameNode 的主要数据结构到 hadoop.log.dir 属性指定目录下的 <filename> 文件。下面的每一项，<filename> 中都有一行内容与之对应： • NameNode 收到的 DataNode 的心跳信号 • 等待被复制的块 • 正在被复制的块 • 等待被删除的块
-setQuota <quota> <dirname>...<dirname>	为每个目录 <dirname> 设定配额 <quota>。目录配额是一个长整型整数，强制限定了目录树下的名字个数。命令会在这个目录上工作良好，以下情况会报错： • N 不是一个正整数 • 用户不是管理员 • 这个目录不存在或是文件 • 目录会马上超出新设定的配额
-clrQuota <dirname>...<dirname>	为每一个目录 <dirname> 清除配额设定。命令会在这个目录上工作良好，以下情况会报错，如果目录原来没有配额不会报错： • 这个目录不存在或是文件 • 用户不是管理员
-help [cmd]	显示给定命令的帮助信息，如果没有给定命令，则显示所有命令的帮助信息

5. job 命令

该命令用于和 Map Reduce 作业交互和命令。键入“./bin/hadoop job”将输出能够支持的命令列表。

```
hadoop job [GENERIC_OPTIONS] [-submit <job-file>] | [-status <job-id>] | [-counter <job-id> <group-name> <counter-name>] | [-kill <job-id>] | [-events <job-id> <from-event-#> <#-of-events>] | [-history [all] <jobOutputDir>] | [-list [all]] | [-kill-task <task-id>] | [-fail-task <task-id>]
```

job 命令参数解释如表 4-6 所示。

表 4-6 job 命令参数说明

命令选项	功能描述
-submit <job-file>	提交作业
-status <job-id>	打印 Map 和 Reduce 完成百分比和所有计数器
-counter <job-id> <group-name> <counter-name>	打印计数器的值
-kill <job-id>	杀死指定作业
-events <job-id> <from-event-#> <#-of-events>	打印给定范围内 JobTracker 接收到的事件细节
-history [all] <jobOutputDir>	-history <jobOutputDir> 用于打印作业的细节，失败及被杀死原因的细节。更多关于一个作业的细节（如成功的任务、做过的任务尝试等信息）可以通过指定 [all] 选项查看
-list [all]	显示所有作业。-list 只显示将要完成的作业
-kill-task <task-id>	杀死任务。被杀死的任务不会不利于失败尝试
-fail-task <task-id>	使任务失败。被失败的任务会对失败尝试不利

6. fsck 命令

该命令用于和 MapReduce 作业交互和命令。键入“./bin/hadoop job”将输出能够支持的命令列表。

```
hadoop fsck [GENERIC_OPTIONS] <path> [-move | -delete | -openforwrite] [-files [-blocks [-locations | -racks]]]
```

fsck 命令参数解释如表 4-7 所示。

表 4-7 fsck 命令参数说明

命令选项	功能描述	命令选项	功能描述
<path>	检查的起始目录	-files	打印正被检查的文件
-move	移动受损文件到 /lost+found	-blocks	打印块信息报告
-delete	删除受损文件	-locations	打印每个块的位置信息
openforwrite	打印写打开的文件	-racks	打印 data-node 的网络拓扑结构

7. pipes 命令

该命令用于运行 pipes 作业。键入“./bin/hadoop fsck”将输出能够支持的命令列表。

```
hadoop pipes [-conf <path>] [-jobconf <key=value>, <key=value>, ...] [-input <path>] [-output <path>] [-jar <jar file>] [-inputformat <class>] [-map <class>] [-partitioner <class>] [-reduce <class>] [-writer <class>] [-program <executable>] [-reduces <num>]
```

pipes 命令参数解释如表 4-8 所示。

表 4-8 pipes 命令参数说明

命令选项	功能描述
-conf <path>	作业的配置
-jobconf <key=value>, <key=value>, ...	增加 / 覆盖作业的配置项

（续）

命令选项	功能描述
-input <path>	输入目录
-output <path>	输出目录
-jar <jar file>	JAR 文件名
-inputformat <class>	InputFormat 类
-map <class>	Java Map 类
-partitioner <class>	Java Partitioner
-reduce <class>	Java Reduce 类
-writer <class>	Java RecordWriter
-program <executable>	可执行程序的 URI
-reduces <num>	Reduce 个数

8. version 命令

该命令用于查看 Hadoop 版本信息。

```
hadoop version
```

获取 Hadoop 版本如图 4-18 所示。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop version
Warning: $HADOOP_HOME is deprecated.

Hadoop 1.0.3
Subversion https://svn.apache.org/repos/asf/hadoop/common/branches/branch-1.0
 -r 1335192
Compiled by hortonfo on Tue May  8 20:31:25 UTC 2012
From source with checksum e6b0c1e23dcf76907c5fecb4b832f3be
```

图 4-18 获取 Hadoop 版本

9. balance 命令

该命令用于运行集群平衡工具。管理员可以通过 Ctrl+C 快捷键来停止平衡过程。

```
hadoop balance [-threshold <threshold>]
```

balance 命令参数如表 4-9 所示。

表 4-9 balance 命令参数说明

命令选项	功能描述
-threshold <threshold>	磁盘容量的百分比。这会覆盖默认的阈值

其实，HDFS 的 Shell 操作和 Linux 下对文件的操作非常相似。这里就不举更多的例子了，有时间可以把没有演示的命令自己操作一下，对 HDFS 的 Shell 命令加深印象。

4.7 Java API 接口

HDFS 提供了 Java API 接口对 HDFS 进行操作。写代码的步骤如下。

```
static FileSystem get(Configuration conf)
operator()
    {
        //step1
       得到 Configuration 对象
        //step2
       得到 FileSystem 对象
        //step3
       进行文件操作
    }
```

4.7.1 Hadoop URL 读取数据

要从 Hadoop 文件系统中读取数据，最简单的方法就是使用 java.net.URL 类打开一个数据流，然后从中读取数据。但是在真正读取数据之前，需要让 Java 能够识别 Hadoop 文件系统的 URL。

可以通过 URL 的 setURLStreamHandlerFactory 方法设置 Hadoop 文件系统的 URLStreamHandlerFactory 的实现类 FsUrlStreamHandlerFactory，这个方法在一个 Java 虚拟机中只能被调用一次，因此该方法一般都是放在一个 static 块中执行。这个限制意味着，如果程序的其他部分设置了 URLStreamHandlerFactory，就不能从 Hadoop 文件系统中读取数据了。

代码清单 4-1 通过 URLStreamHandler 实例以标准输出方式显示 Hadoop 文件系统里面文件的内容。

代码清单 4-1 hdfs\HDFSURLReader.java

```
import java.io.InputStream;
import java.net.URL;
import org.apache.hadoop.fs.FsUrlStreamHandlerFactory;
import org.apache.hadoop.io.IOUtils;
public class HDFSURLReader {
 static {
        URL.setURLStreamHandlerFactory(newFsUrlStreamHandlerFactory());
 }
 public static void main(String[] args) {
       InputStreaminputStream = null;
       try {
            inputStream = newURL(args[0]).openStream();
            IOUtils.copyBytes(inputStream, System.out, 1024, false);
       } catch (Exception e) {
            IOUtils.closeStream(inputStream);
       }
    }
}
```

运行结果如下。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop HDFSURLReader hdfs://192.168.153.100/user/hadoop/input/test.txt
hello world
hello hadoop
```

4.7.2 FileSystem 类

当我们无法在程序中设置 URLStreamHandlerFactory 的时候，需要使用 Hadoop 提供的 FileSystem 类打开一个文件的输入流。

文件在 Hadoop 中表示为一个 Path 对象，可以把路径看做 Hadoop 文件系统的 URI，如 hdfs://192.168.153.100/user/hadoop/input/test.txt。

FileSystem 是 Hadoop 中文件系统的抽象父类，所以首先需要做的就是检索 Hadoop 具体使用的文件系统，比如 HDFS。取得 FileSystem 的实例需要通过 FileSystem 的如下两个静态方法来实现。

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

Configuration 对象封装了客户端或者服务器端的配置信息，这里使用从类路径中读取的 core-site.xml 配置文件来设置。第一个方法返回的是在配置文件中通过 fs.default.name 属性设置的默认文件系统，如果没有设置这个属性，就使用本地默认文件系统。第二个方法使用 URI 指定的文件系统方案来决定使用的文件系统，如果 URI 中没有指定文件系统的方案，则使用默认的文件系统。获得了 FileSystem 实例之后，可以通过调用它的 open 方法来获得输入流。

代码清单 4-2 通过 FileSystem 实例以标准输出方式显示 Hadoop 文件系统里面文件的内容。

代码清单 4-2 hdfs\FileSystemReader.java

```
importjava.io.InputStream;
importjava.net.URI;
importorg.apache.hadoop.conf.Configuration;
importorg.apache.hadoop.fs.FileSystem;
importorg.apache.hadoop.fs.Path;
importorg.apache.hadoop.io.IOUtils;
public class FileSystemReader {
 public static void main(String[] args) throws Exception {
       Configuration configuration = new Configuration();
       String uri = args[0];
       FileSystem fileSystem = FileSystem.get(URI.create(uri), configuration);
       InputStream inputStream = null;
       try {
            inputStream = fileSystem.open(newPath(uri));
            IOUtils.copyBytes(inputStream, System.out, 4096, false);
       } finally {
            IOUtils.closeStream(inputStream);
       }
    }
}
```

运行结果如下。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop FileSystemReader hdfs://192.168.153.100/
user/hadoop/input/test.txt
hello world
hello hadoop
```

4.7.3 FileStatus 类

Hadoop 中的 FileStatus 类可以用来查看 HDFS 中文件或者目录的元信息，任意文件或者目录都可以得到对应的 FileStatus。FileStatus 类封装了文件系统中文件和目录的元数据信息，包括文件长度、块大小、备份、修改时间、所有者以及权限等信息。

FileSystem 的 getFileStatus() 方法用于获取文件或目录的 FileStatus 对象。这里简单演示 FileStatus 的相关 API，如代码清单 4-3 所示。

代码清单 4-3 hdfs\FileStatusMetadata.java

```
import java.net.URI;
import java.sql.Timestamp;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class FileStatusMetadata {
    public static void main(String[] args) throws Exception {
        //读取 hadoop 文件系统的配置
        Configuration conf = new Configuration();
        conf.set("hadoop.job.ugi", "jayliu,jayliu");
        //实验 1：查看 HDFS 中某文件的元信息
        System.out.println("实验 1：查看 HDFS 中某文件的元信息 ");
        String fileUri = args[0];
        FileSystem fileFS = FileSystem.get(URI.create(fileUri) ,conf);
        FileStatus fileStatus = fileFS.getFileStatus(new Path(fileUri));
        //获取这个文件的基本信息
        if(fileStatus.isDir()==false){
            System.out.println("这是个文件 ");
        }
        System.out.println("文件路径 : "+fileStatus.getPath());
        System.out.println("文件长度 : "+fileStatus.getLen());
        System.out.println("文件修改日期: "+new Timestamp (fileStatus.
        getModificationTime()).toString());
        System.out.println("文件上次访问日期: "+new Timestamp(fileStatus.
        getAccessTime()).toString());
        System.out.println("文件备份数: "+fileStatus.getReplication());
        System.out.println("文件的块大小: "+fileStatus.getBlockSize());
        System.out.println("文件所有者:  "+fileStatus.getOwner());
        System.out.println("文件所在的分组: "+fileStatus.getGroup());
        System.out.println("文件的权限: "+fileStatus.getPermission().toString());
        System.out.println();

        //实验 2：查看 HDFS 中某文件的元信息
        System.out.println("实验 2：查看 HDFS 中某目录的元信息 ");
        String dirUri = args[1];
        FileSystem dirFS = FileSystem.get(URI.create(dirUri) ,conf);
```

```
            FileStatus dirStatus = dirFS.getFileStatus(new Path(dirUri));
            //获取这个目录的基本信息
            if(dirStatus.isDir()==true){
                System.out.println("这是个目录");
            }
            System.out.println("目录路径： "+dirStatus.getPath());
            System.out.println("目录长度： "+dirStatus.getLen());
            System.out.println("目录修改日期: "+new Timestamp (dirStatus.
            getModificationTime()).toString());
            System.out.println("目录上次访问日期: "+new Timestamp(dirStatus.
            getAccessTime()).toString());
            System.out.println("目录备份数: "+dirStatus.getReplication());
            System.out.println("目录的块大小: "+dirStatus.getBlockSize());
            System.out.println("目录所有者:  "+dirStatus.getOwner());
            System.out.println("目录所在的分组: "+dirStatus.getGroup());
            System.out.println("目录的权限: "+dirStatus.getPermission().toString());
            System.out.println("这个目录下包含以下文件或目录: ");
            for(FileStatus fs : dirFS.listStatus(new Path(dirUri))){
                System.out.println(fs.getPath());
            }
        }
    }
```

运行结果如下。

```
    jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop FileStatusMetadata hdfs://192.168.153.
100:9000/user/hadoop/test.txt hdfs://192.168.153.100:9000/user/hadoop
    实验 1: 查看 HDFS 中某文件的元信息
    这是个文件
    文件路径： hdfs://192.168.153.100:9000/user/hadoop/test.txt
    文件长度： 93
    文件修改日期: 2013-01-26 12:49:04.285
    文件上次访问日期: 2012-01-26 12:49:02.285
    文件备份数: 3
    文件的块大小: 67108864
    文件所有者: jayliu
    文件所在的分组: supergroup
    文件的权限: rw-r--r--
    实验 2: 查看 HDFS 中某目录的元信息
    这是个目录
    目录路径： hdfs://192.168.153.100:9000/user/hadoop
    目录长度： 0
    目录修改日期: 2012-01-26 13:34:10.743
    目录上次访问日期: 1970-01-01 08:00:00.0
    目录备份数: 0
    目录的块大小: 0
    目录所有者:  jayliu
    目录所在的分组: supergroup
    目录的权限: rwxr-xr-x
    这个目录下包含以下文件或目录:
```

```
hdfs://192.168.153.100:9000/user/hadoop/easyhadoop.txt
hdfs://192.168.153.100:9000/user/hadoop/input
hdfs://192.168.153.100:9000/user/hadoop/output
hdfs://192.168.153.100:9000/user/hadoop/text.txt
```

查找某个文件 Block 在 HDFS 集群的位置如代码清单 4-4 所示。

代码清单 4-4 hdfs\BlockLocation.java

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.BlockLocation;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
public class BlockLocation {
 //查找某个文件在 HDFS 集群的位置
 public static void main(String[] args) throws Exception {
    Configuration conf=new Configuration();
    FileSystem hdfs=FileSystem.get(conf);
    Path fpath=new Path(arg[0]);
    FileStatus filestatus = hdfs.getFileStatus(fpath);
    BlockLocation[] blkLocations = hdfs.getFileBlockLocations(filestatus, 0,
                filestatus.getLen());
    int blockLen = blkLocations.length;
    for(int i=0;i<blockLen;i++){
         String[] hosts = blkLocations[i].getHosts();
         System.out.println("block_"+i+"_location:"+hosts[0]);
    }
  }
}
```

运行结果如下。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop BlockLocation hdfs://192.168.153.100/
user/hadoop/input/test.txt
block_0_location:Ubuntu1
block_1_location:Ubuntu1
block_2_location:Ubuntu3
```

4.7.4 FSDataInputStream 类

从源代码可以看出，FSDataInputStream 并没有亲自实现两个接口的方法，而是通过调用构造函数传递的 InputStream 方法来实现，传递进来的 InputStream 必须也实现了 Seekable 和 PositionedReadable 接口。

正如之前所说，FSDataInputStream 的主要作用就是实现将包装进 DataInputStream，所以接下来详解 FSDataInputStream 类。

FSDataInputStream 类的继承结构如图 4-19 所示。

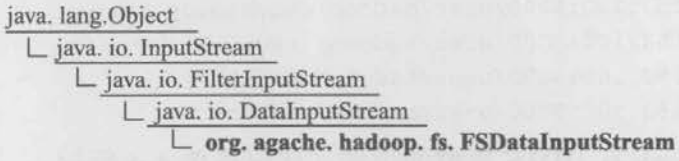


图 4-19 FSDataInputStream 的继承结构图

FSDataInputStream 类的主要作用就是用 DataInputStream 包装一个输入流，并且使用 BufferedInputStream 实现对输入的缓冲。该类继承了 DataInputStream 父类，实现了 Closeable、Seekable 两个接口。

FSDataInputStream 实现了 DataInputStream 抽象类和接口 Closeable，从而使 Hadoop 中的文件输入流具有流式搜索和流式定位读取的功能。下面来看定义的接口。

```
package org.apache.hadoop.fs;
import java.io.*;
/** Stream that permits seeking. */
public interface Seekable {
  /**
    从指定文件的 pos 位置，对文件流向前搜索
  */
  void seek(long pos) throws IOException;
  /**
   * 返回文件流中的当前偏移量
   */
  long getPos() throws IOException;

  /** 在文件的另一个副本中查找 targetPos 指定的位置
   * 如果找到 targetPos 的位置返回 true，否则返回 false
   */
  boolean seekToNewSource(long targetPos) throws IOException;
}
```

Seekable 接口允许通过 seek(long pos) 方法在文件中进行定位，如果 pos 的值大于文件的长度，会出现 IOException。此外，还可以通过 getPos() 方法返回当前位置相对于文件开头的偏移量。应用程序员一般不关心 seekToNewSource(long targetPos) 方法，此方法实现在文件的另一个副本中查找 targetPos 指定的位置，如果找到了 targetPos 的位置返回 true，否则返回 false。HDFS 内部就通过这个方法，当客户节点发生故障时为客户提供可靠的输入流。

FSDataInputStream 也实现了 PositionedReadable 接口，从而实现从指定位置读取一部分数据。该接口的定义如下：

```
public interfacePositionedReadable {
  /**
     该方法从文件的指定位置 position 向后读取最多 length 的字节，并将读取到的字节存储在 buffer
的指定偏移量 offset 位置上，该方法不会改变文件当前的偏移位置，并且是线程安全的。返回实际读取的字节数
   */
  public int read(long position, byte[] buffer, int offset, int length)
    throws IOException;
```

```
    /**
     该方法从文件的指定位置 position 向后读取 length 的字节，并将读取到的字节存储在 buffer 的指
定偏移量 offset 位置上，该方法不会改变文件当前的偏移位置，并且是线程安全的。该方法与上面方法的区别是：
该方法读取的字节数是固定的，就是 length，而上面方法读取的字节数可能小于等于 length。如果到达文件的末
尾，则会抛出 IOException
     */
    public void readFully(long position, byte[] buffer, int offset, int length)
      throws IOException;
      /**
      从文件的 position 位置开始读取长度为 byte[] buffer 长度的字节到字节数组 buffer 中。方法不
会改变文件当前的偏移位置 offset，并且是线程安全的。如果到达文件的末尾，则会抛出 IOException
      */
    public void readFully(long position, byte[] buffer) throws IOException;
}
```

以上方法都是线程安全的，而且保持了文件的当前位置，主要是通过 Seekable 接口来实现的，如：

```
Long oldPos=getPos();
Try{
    Seek(position);
// 读取数据
}finally{
    Seek(oldPos);
}
```

需要牢记，seek() 的开销很大，使用时要慎重。建议使用流数据来构建应用的访问模式（如使用 MapReduce），而不是执行大量的 seek()。

代码清单 4-5 利用 FSDataInputStream 来实现在 HDFS 上创建文件和读取文件的内容。

代码清单 4-5　hdfs\ HDFSDataInputExample.java

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.URI;
import java.net.URL;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.FsUrlStreamHandlerFactory;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
import org.apache.log4j.Logger;
```

```
@SuppressWarnings("deprecation")
public class HDFSDataInputExample {
    static final Logger logger = Logger.getLogger(HDFSSample.class);
    static {
        URL.setURLStreamHandlerFactory(new FsUrlStreamHandlerFactory());
    }
    public static void main(String[] args) throws Exception {
        HDFSSample sample = new HDFSSample();
        String cmd = args[0];
        String localPath = args[1];
        String hdfsPath = args[2];
        if (cmd.equals("create")) {
            sample.createFile(localPath, hdfsPath);
        } else if (cmd.equals("get")) {
            boolean print = Boolean.parseBoolean(args[3]);
            sample.getFile(localPath, hdfsPath, print);
        }
    }
    /**
     * 创建文件
     * @param localPath
     * @param hdfsPath
     * @throws IOException
     */
    @SuppressWarnings("deprecation")
    public void createFile(String localPath, String hdfsPath) throws IOException {
        InputStream in = null;
        try {
            Configuration conf = new Configuration();
            FileSystem fileSystem = FileSystem.get(URI.create(hdfsPath), conf);
            FSDataOutputStream out = fileSystem.create(new Path(hdfsPath));
            in = new BufferedInputStream(new FileInputStream(new
            File(localPath)));
            IOUtils.copyBytes(in, out, 4096, false);
            out.hsync();
            out.close();
            logger.info("create file in hdfs:" + hdfsPath);
        } finally {
            IOUtils.closeStream(in);
        }
    }
    /**
     * 从 HDFS 获取文件
     * @param localPath
     * @param hdfsPath
     * @throws IOException
     */
    public void getFile(String localPath, String hdfsPath, boolean print) throws
    IOException {
        Configuration conf = new Configuration();
        FileSystem fileSystem = FileSystem.get(URI.create(hdfsPath), conf);
```

```
            FSDataInputStream in = null;
            OutputStream out = null;
            try {
                in = fileSystem.open(new Path(hdfsPath));
                out = new BufferedOutputStream(new FileOutputStream(new
                File(localPath)));
                IOUtils.copyBytes(in, out, 4096, !print);
                logger.info("get file form hdfs to local: " + hdfsPath + ", " +
                localPath);
                if (print) {
                    in.seek(0);
                    IOUtils.copyBytes(in, System.out, 4096, true);
                }
            } finally {
                IOUtils.closeStream(out);
            }
        }
    }
```

运行结果如下。

```
    //创建文件
    jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop HDFSDataInputExample create /opt/
hello.txt hdfs://192.168.153.100/user/hadoop/input/test.txt
    //读取文件
    jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop HDFS DataInputExample get
hdfs://192.168.153.100/user/hadoop/input/test.txt /opt/hello.txt
```

4.7.5 FSDataOutputStream 类

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
```

以上方法有重载的版本，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做。如果想在父目录不存在时不执行写入，就必须在调用 exists() 时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，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。

代码清单 4-6 展示了如何将本地文件复制到 Hadoop 文件系统。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”。)

代码清单 4-6 hdfs\ FileCopyWithProgress.java

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.URI;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.util.Progressable;
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);
  }
}
```

运行结果如下。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop FileCopyWithProgress /opt/hello.txt
hdfs://192.168.153.100/user/hadoop/input/
```

提示 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，我们会感到进度对于 MapReduce 应用的重要性。

FileSystem 中的 create() 方法返回一个 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。

4.7.6 列出 HDFS 下所有的文件

如果不是获取文件或目录信息，只是检查是否存在，还是使用 exists() 更方便。要列出目录的内容，使用 FileStatus 的 listStatus() 方法。

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

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

显示 HDFS 文件系统里面给定路径的文件信息如代码清单 4-7 所示。

代码清单 4-7 hdfs\ ListAllFile.java

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.FileUtil;
import org.apache.hadoop.fs.Path;
public class ListAllFile {
 //读取 HDFS 某个目录下的所有文件
    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);
            }
        }
    }
```

运行结果如下。

```
    jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop ListAllFile hdfs://192.168.153.100/
user/hadoop/input/
    hdfs://192.168.153.100/user/hadoop/input/test.txt
    hdfs://192.168.153.100/user/hadoop/input/text.txt
    hdfs://192.168.153.100/user/hadoop/input/easyhadoop/test.txt
    hdfs://192.168.153.100/user/hadoop/input/easyhadoop/text.txt
```

4.7.7 文件的匹配

在操作中处理一批文件，是一个常见的要求。举个例子来说，处理日志的 MapReduce 作业可能需要分析一个月内包含在大量目录中的日志文件。在一个表达式中使用通配符来匹配多个文件是非常方便的，无须列举每个文件或目录来指定输入。Hadoop 为了执行通配符提供了两个 FileStatus 方法。

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

globStatus() 方法返回与路径相匹配的所有文件的 FileStatus 对象数组，并按路径排序。PathFilter 命令作为可选项进一步对匹配进行限制。匹配的通配符有 \c、*、?、[^0-9a-z]、{a,b} 等，类似正则，不多解释。注意，{a,b} 的含义跟正则不一样，表示匹配 a 或者 b。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs -lsr /user/*/*
Warning: $HADOOP_HOME is deprecated.

-rw-r--r--   1 jayliu supergroup         75 2013-04-22 22:43 /user/easyhadoop
/test.txt
-rw-r--r--   1 jayliu supergroup         75 2013-04-21 23:45 /user/hadoop/inp
ut/test.txt
```

图 4-20 正则表达式使用

例如查看 /user/ 目录下的所有文件，操作如图 4-20 所示。

查看 /user/ 目录下以 e 开头的所有文件和文件夹下的文件，操作如图 4-21 所示。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs -lsr /user/e*/*
Warning: $HADOOP_HOME is deprecated.

-rw-r--r--   1 jayliu supergroup         75 2013-04-22 22:45 /user/easy/test.
txt
-rw-r--r--   1 jayliu supergroup         75 2013-04-22 22:43 /user/easyhadoop
/test.txt
```

图 4-21 正则表达式使用

4.7.8 PathFilter 对象

有时候通配符匹配不能满足我们的选择要求，FileStatus 中的 listStatus() 和 globalStatus() 方法提供了可选的 PathFilter 对象，类似 C++、STL 中比较器的函数对象，写一个函数返回 true 或者 false 来确定是否是我们需要的内容。

通过 FilterPath 过滤 CRC 或 TEMP 格式的文件如代码清单 4-8 所示。

代码清单 4-8 hdfs\ FileFilter.java

```
import java.io.IOException;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.fs.PathFilter;
/**
 * 基于正则的文件过滤器实现
 * @author muyannian
 */
public  class FileFilter implements PathFilter {
    private String matcher = "*";
    private FileSystem fs = null;
    public void setFileSystem(FileSystem _fs) {
        fs = _fs;
 }
 public void setMatcher(String sMatcher) {
        matcher = sMatcher;
 }
 public boolean accept(String filePath) {
        // 文件路径过滤
        boolean isMatcherAll = matcher.equals("*");
        boolean isStrIndexOf = (filePath.indexOf(matcher) >= 0);
        if (isMatcherAll || isStrIndexOf) {
               return true;
        }
        try {
             if (filePath.matches(matcher)) {
                    return true;
             }
        } catch (Exception e) {
             System.err.println(FileFilter.class.getName() + "err for"+ matcher);
             return true;
        }
        return false;
 }
 public boolean accept(Path path) {
        // 过滤隐藏文件或隐藏文件夹
        String fileName = path.getName();
        if (fileName.endsWith(".temp") || fileName.endsWith(".crc")) {
               return false;
        }
        // 如果是目录，则跳过，不过滤
        try {
             if(HadoopUtil.exists(path.toString(), fs)) {
                  if (fs.getFileStatus(path).isDir()) {
                         return true;
                  }
             }
```

```
            } catch (IOException e1) {
                e1.printStackTrace();
                return true;
            }
            return accept( path.toString());
        }
    }
```

4.8 维护 HDFS

HDFS 对于 Hadoop 集群的维护者来说是非常重要的，与 DBA（DataBase Administrator）的身份比较类似。Hadoop 集群需要不停的维护，如添加和删除 Hadoop 节点、Hadoop 版本的升级等操作。

4.8.1 追加数据

append 操作是向 HDFS 里追加数据，这也 HDFS 提供的一个新功能。代码如下。

```
FSDataOutputStream stm = fs.create(path, true,
            conf.getInt("io.file.buffer.size", 4096),
            (short)3, blocksize);
String a = make(1000);
stm.write(a.getBytes());
stm.sync();
```

可以看到，append 过程分两步：先 write 操作，然后执行 sync。如果不执行 sync，理论上会存在丢失数据的风险。append 操作的流程如图 4-22 所示。

从 Client 角度来看，追加一个文件首先需要调用 DistributedFileSystem 的 append 操作，该操作会返回一个 FSDataOutputStream 的流对象 out，Client 要向该文件追加数据，可以调用 out.write 写入数据，最后调用 out.close 进行关闭。但是其内部的交互流程比较复杂，下面从上述三个 Client 端的调用来进行深入分析。

1. Client 调用 fs 的 append 操作

首先调用 NameNode 的 getFileInfo RPC 调用，获取该文件的信息。

NameNode 的 appendFile RPC 调用，进行 lease 的检查（是否可以追加操作，如果别的 Client 正在调用 append 操作，就不能再进行操作了），将该文件从 INodeFile 转换从 INodeFileUnderConstruction，并且将该文件的最后一个 Block 以 LocatedBlock 的对象返回给 Client。如果没有则返回 null。注意，上述流程图表示的是要追加的文件之前已经成功关闭并且是一个 INodeFile 类型文件，否则需要判断拥有该文件追加权限的 Client 的 lease 是否过期，如果过期则进行 lease recover 操作并由 NameNode 发起 block recover 操作来关闭文件。

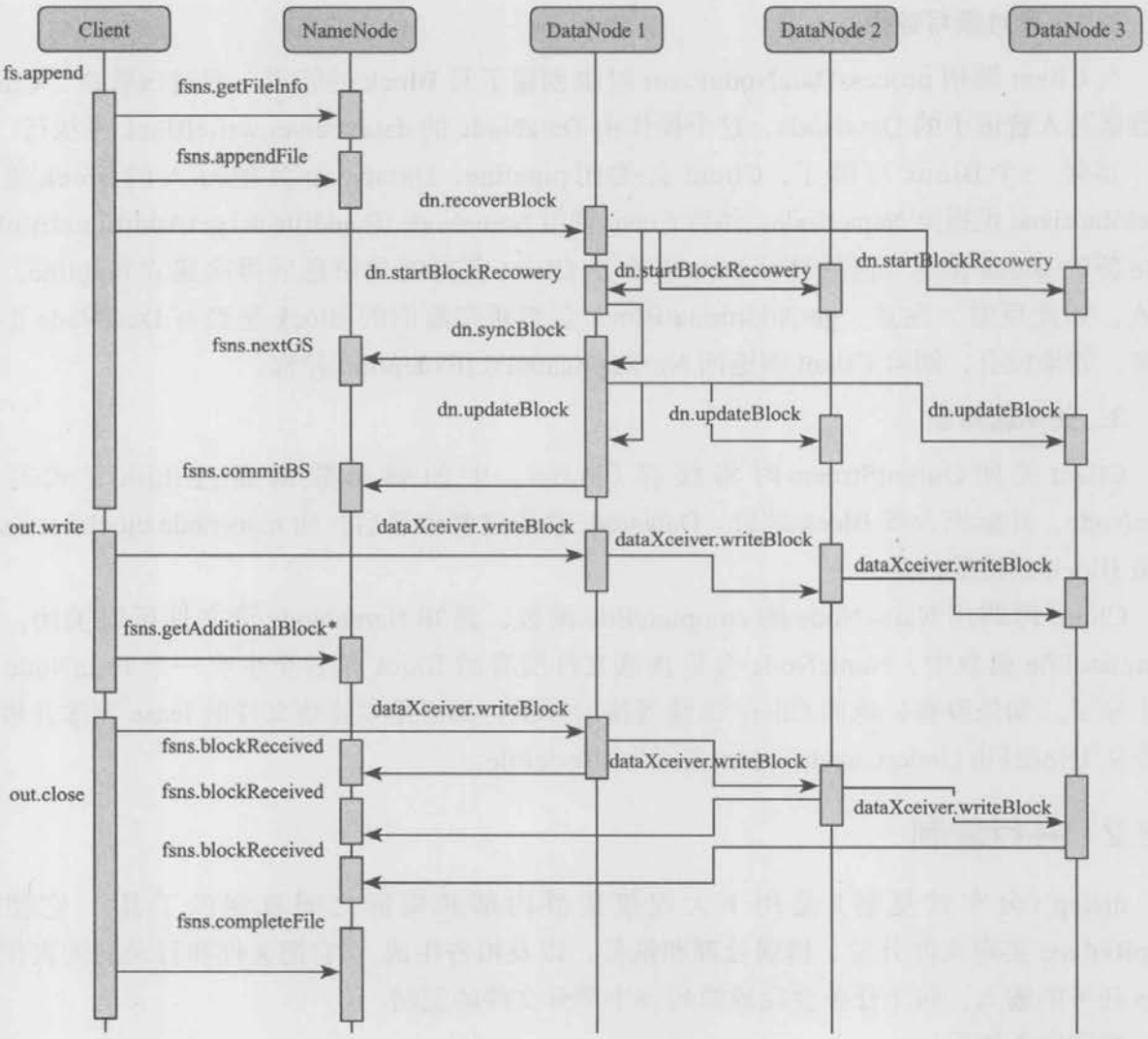


图 4-22 append 操作流程图

如果上面的调用返回的 LocatedBlock 不为 null，则在 Client 中创建 Outputstream 时调用 processDatanodeError 方法进行 block recover 操作。具体是从 LocatedBlock 中获取该 Block 位于哪些 DataNode 上（这里假设是 DataNode1、DataNode2 和 DataNode3），然后从这些 DataNode 中选取一个 primaryDatanode，由该 DataNode（这里假设是 DataNode1）进行 Block recover 操作。在进行 block recover 操作中，首先 DataNode1 协调其他两个 DataNode 进行 startBlockRecovery 操作，目的是强制终止该 block file 的写线程，然后调用 syncBlock 操作，在 syncBlock 操作中由 DataNode1 调用 NameNode 的 nextGenerationStampForBlock 操作获取新的 GS，DataNode1 再协调相关 DataNode 进行 updateBlock 操作，即将本地的 block file 更新长度和 GS，最后 DataNode1 向 NameNode 汇报 commitBlockSynchronization，至此该 block recover 完毕。另外，上面提到由 NameNode 发起的 block recover 操作流程，与由 primary DataNode 发起的类似，只是参数有些区别。

2. 向流对象写数据

在 Client 调用 processDataNodeError 时也创建了写 Block 的管道。通过该管道，Client 将数据写入管道中的 DataNode，这个操作由 DataNode 的 dataxceiver.writeBlock 来执行。

如果一个 Block 写满了，Client 会关闭 pipeline，DataNode 会把写入的 Block 通过 blockReceived 汇报给 NameNode。然后 Client 调用 NameNode 的 addBlock（getAdditionalBlock）获得新的 Block 信息（包括目标 DataNode），Client 获得这些信息后再次建立 pipeline，并写入，如此反复。注意，getAdditionalBlock 会判断它前面的 Block 是否有 DataNode 汇报上来，如果没有，则给 Client 端返回 NotReplicatedYetException 异常。

3. 关闭流对象

Client 关闭 OutputStream 时将缓存（buffer）中的剩余数据通过 flush 方式写入 DataNode，并标识为该 Block 结束。DataNode 获取这些信息后，向 namenode.blockReceived 汇报 Block 的接受信息。

Client 再调用 NameNode 的 completeFile 函数，通知 NameNode 该文件可以关闭。在 completeFile 函数中，NameNode 会检查该文件所有的 Block 是否至少有一个 DataNode 汇报上来了。如果没有，返回 Client 继续等待的标识，如果是则将该文件的 lease 删除并将该文件从 INodeFileUnderConstruction 变回为 INodeFile。

4.8.2 并行复制

distcp（分布式复制）是用于大规模集群内部和集群之间复制的工具。它使用 MapReduce 实现文件分发、错误处理和恢复，以及报告生成。它把文件和目录的列表作为 Map 任务的输入，每个任务会完成源列表中部分文件的复制。

操作命令如下。

```
hadoop  distcp -overwrite -delete -i dir1 dir2
```

参数的含义如下。

- -delete：删除已经存在的目标文件，不会删除源文件。这个删除是通过 FS Shell 实现的。所以如果垃圾回收机制启动，删除的目标文件会进入 trash。
- -i：忽略失败。这个选项会比默认情况提供关于复制的更精确的统计，同时它还将保留失败复制操作的日志，这些日志信息可以用于调试。最后，如果一个 Map 失败了，但并没完成所有分块任务的尝试，这不会导致整个作业的失败。
- -overwrite：覆盖目标。如果一个 Map 失败并且没有使用 -i 选项，那些复制失败的文件，以及这个分块任务中的所有文件都会被重新复制。因此要使用 -i 参数。

4.8.3 升级与回滚

Hadoop 升级分为两种。一种是集群布局不发生任何变化的，这种升级非常简单，类似

安装一次新的 Hadoop 程序；另外一种是集群布局发生变化的，这种升级要非常小心，因为涉及数据安全，要让 HDFS 内的数据完整迁移到新版本，所以要仔细。

两种升级都简单分为以下几步。

1）在执行新一轮的升级前，要确保前一次升级已经完成，用以下命令查看。

```
$ hadoop dfsadmin -upgradeProgress status
There are no upgrades in progress.
```

出现上边的提示表示可以进行下次升级。

2）进行数据的备份，以方便升级后对照，如果有问题可发现然后回滚版本。

```
$ hadoop fs -lsr / > ~/namenode.log
$ hadoop fsck / >> ~/namenode.log
$ cp -r ${dfs.name.dir} ~/namenode_backup
```

3）使用 stop-all.sh 关闭 hadoop 集群。

下载 1.0.3 版本的程序进行配置，虽然 conf 里边多了几个新文件，但是不用管它，就按之前的配置就可以，或者直接把旧版本的 conf 里边的 core、hdfs、mapred、master、slaves 几个配置文件复制过来就行。别忘了配置 hadoop-env.sh，不确定这个是否有变化，还是手动配置 JAVA_HOME 好了。

4）把 ${dfs.name.dir} 目录下的所有内容复制到新配置的路径下，这是集群布局有变化的操作，如果没有新配置布局就不用这个操作了。单独启动新的主节点 NameNode 的 hdfs 进行更新操作，命令如下。

```
$ ${NEW_HADOOP}/bin/start-dfs.sh -upgrade
```

使用如下命令进行监控，查看是否升级完成。

```
$ ${NEW_HADOOP}/bin/hadoop dfsadmin -upgradeProgress status
```

出现“Upgrade for version -* has been completed.”的提示即完成升级。

5）把新版本的 Hadoop 程序和配置文件一起分发给集群里所有的机器，根据情况看是不是需要进行个别修改，没有特别的就不用改了。修改 /etc/profile 中的 Hadoop 环境变量，改成新版本 Hadoop 的指向。现在使用启动集群的 start-all.sh 命令来启动集群就行了。

6）使用 fs -lsr 和 fsck 与 namenode.log 文件进行核对前后信息变化，如果没有问题，数据块体检也通过，就顺利升级完成了。这时需要执行如下命令来最终确定升级完成和清理前版本数据。

```
$ hadoop dfsadmin -finalizeUpgrade
$ rm -r ~/namenode_backup ~/namenode.log
```

阈值至此升级完成，不能再进行回滚操作，如果升级后发现数据不符，可以使用如下命令回滚版本。

```
$ stop-all.sh
$ start-dfs.sh -rollback
```

注意 升级的时候一点要仔细。确保你的 HDFS 可用空间大于 15% 以上才能升级。笔者在公司把 Hadoop 从 0.20.2 版本升级到 1.0.3 版本的时候，由于 HDFS 的可用空间只有 4%，升级完以后，NameNode 成功了，但是 DataNode 启动全部失败，查看 DataNode 的日志发现“No Space Disk”，意思是 DataNode 上面的磁盘全部满了。当时急的直冒汗，各种方法也没有找到解决方案，也不能回滚。最终删除每台 DataNode 上的 Block 文件，然后分别启动 DataNode 就可以了。因为 Block 有多份，所以最后的日志基本上没有丢失。

4.8.4 添加节点

在 Hadoop 集群里面添加和删除节点很常见，比如随着业务的不断增加，日志量越来越大，导致原来的集群不够用。下面来看怎么动态地添加和删除 Hadoop 集群的节点。

准备好要添加的设备，设备的配置、挂载的盘数最好和原来集群一样，如果不一样要手动作相应的修改。从原有集群的任意一台 DataNode 上复制一份 Hadoop 的安装包到要添加的节点上。

1）修改 host。和普通的 DataNode 一样，添加 NameNode 的 IP 。

2）修改 NameNode 的配置文件 conf/slaves 。添加新增节点的 IP 或主机名。

3）在新节点的机器上启动服务。操作命令如下。

```
[jayliu@Ubuntu4 hadoop]# ./bin/hadoop-daemon.sh start datanode
[jayliu@Ubuntu4 hadoop]# ./bin/hadoop-daemon.sh start tasktracker
```

4）均衡 Block。执行命令如下。

```
[jayliu@Ubuntu4 hadoop]# ./bin/start-balancer.sh
```

为什么要执行 start-balancer.sh 操作呢？ 如果不执行 start-balancer.sh 操作，cluster 会把新的数据都存放在新的节点上，这样会降低 MapReduce 的工作效率 。

执行 start-balancer.sh 操作的优化如下。

1）设置平衡阈值，默认是 10%，值越低各节点越平衡，但消耗时间也更长。命令如下。

```
[jayliu@Ubuntu4 hadoop]# ./bin/start-balancer.sh -threshold 5
```

2）设置 balancer 的带宽，默认只有 1M/s。编辑 hdfs-site.xml 文件，添加如下配置信息。

```
<property>
    <name>dfs.balance.bandwidthPerSec</name>
    <value>10485760</value>
    <description>
       Specifies the maximum amount of bandwidth that each datanode
       can utilize for the balancing purpose in term of
       the number of bytes per second.
    </description>
 </property>
```

这里设置成 10M/S，在使用 start-balancer.sh 时，如果在 hdfs-site.xml 里面没有配置 dfs.balance.bandwidthPerSec，集群 HDFS 内部默认使用 1M/S 的速度移动数据，我们可以

通过在 hdfs-site.xml 里面配置 dfs.balance.bandwidthPerSec 来加快 balance 操作的速度。

dfs.balance.bandwidthPerSec 值的设置要根据现有 Hadoop 集群的配置不断进行调试，找到最佳的值。大了或小了都会影响 balance 操作的效率。

注意 必须确保 Slave 的 firewall 已关闭；确保新 Slave 的 IP 已经添加到 Master 及其他 Slave 的 /etc/hosts 中，反之也要将 Master 及其他 Slave 的 IP 添加到新 Slave 的 /etc/hosts 中。

4.8.5 删除节点

例如 Hadoop 集群里面有的设备坏了或者不想使用这台设备了，应该怎么做呢？此时需要删除 Hadoop 集群的节点，具体步骤如下。

1）集群配置。

修改 conf/hdfs-site.xml 文件。添加如下内容。

```
<property>
     <name>dfs.hosts.exclude</name>
     <value>/data/soft/hadoop/conf/excludes</value>
     <description>Names a file that contains a list of hosts that are
     not permitted to connect to the namenode.  The full pathname of the
     file must be specified.  If the value is empty, no hosts are
     excluded.</description>
</property>
```

2）确定要下架的机器。

dfs.hosts.exclude 定义的文件内容为每个需要下线的机器，一行一个。这将阻止它们连接 NameNode。例如 /data/soft/hadoop/conf/excludes 的内容为如下。

```
Ubuntu4
```

这样就把 Ubuntu4 这台设备从 Hadoop 集群中删除了。还有一种方法可以实现 Ubuntu4 设备从 Hadoop 集群中删除，配置如下。

```
<property>
     <name>dfs.hosts.exclude</name>
     <value>Ubuntu4</value>
     <description>Names a file that contains a list of hosts that are
     not permitted to connect to the namenode.  The full pathname of the
     file must be specified.  If the value is empty, no hosts are
     excluded.</description>
</property>
```

3）强制重新加载配置。

```
[root@master hadoop]# ./bin/hadoop dfsadmin  -refreshNodes
```

它会在后台进行 Block 块的移动。

4）关闭节点。

等待刚刚的操作结束后，需要下架的机器就可以安全关闭了。

```
[root@master hadoop]# ./bin/ hadoop dfsadmin -report
```

可以查看到现有集群上连接的节点信息。正在执行 Decommission 会显示以下信息。

```
Decommission Status : Decommission in progress
```

执行完毕后会显示以下信息。

```
Decommission Status : Decommissioned
```

5）再次编辑 excludes 文件。

一旦完成机器下架，下架的设备就可以从 excludes 文件移除了。登录要下架的机器，会发现 DataNode 进程没有了，但是 TaskTracker 依然存在，需要手动关闭。

4.9 HDFS 权限管理

Hadoop 分布式文件系统实现了一个和 POSIX 系统类似的文件和目录的权限模型。每个文件和目录有一个所有者（owner）和一个组（group）。

文件或目录对其所有者、同组的其他用户以及所有其他用户分别有着不同的权限。对文件而言，当读取这个文件时需要有 r 权限，当写入或者追加到文件时需要有 w 权限。对目录而言，当列出目录内容时需要具有 r 权限，当新建或删除子文件或子目录时需要有 w 权限，当访问目录的子节点时需要有 x 权限。不同于 POSIX 模型，HDFS 权限模型中的文件没有 sticky、setuid 或 setgid 位，因为这里没有可执行文件的概念。简单起见，这里也没有目录的 sticky、setuid 或 setgid 位。

总的来说，文件或目录的权限就是它的模式。HDFS 采用了 UNIX 表示和显示模式的习惯，包括使用八进制数来表示权限。当新建一个文件或目录，所有者即客户进程的用户，它的所属组是父目录的组（BSD 的规定）。

每个访问 HDFS 的用户进程的标识分为两部分，分别是用户名和组名列表。每次用户进程访问一个文件或目录 foo，HDFS 都要对其进行权限检查，检查的过程如下。

1）如果用户即 foo 的所有者，则检查所有者的访问权限；

2）如果 foo 关联的组在组名列表中出现，则检查组用户的访问权限；

3）否则检查 foo 其他用户的访问权限；

4）如果权限检查失败，则客户的操作会失败。

4.9.1 用户身份

Hadoop 1.0.3 版本中，客户端用户身份是通过宿主操作系统给出。在类 UNIX 系统中：

- 用户名等于 who am i；
- 组列表等于 bash -c groups。

将来会增加其他的方式来确定用户身份（比如 Kerberos、LDAP 等）。期待用上文中提到的第一种方式来防止一个用户假冒另一个用户是不现实的。这种用户身份识别机制结合

权限模型允许一个协作团体以一种有组织的形式共享文件系统中的资源。

不管怎么样，用户身份机制对 HDFS 本身来说只是外部特性。HDFS 并不提供创建用户身份、创建组或处理用户凭证等功能。

4.9.2 权限管理的原理

每次文件或目录操作都传递完整的路径名给 NameNode，每一个操作都会对此路径做权限检查。客户框架会隐式地将用户身份和与 NameNode 的连接关联起来，从而减少改变现有客户端 API 的需求。

经常会有这种情况，对一个文件的某一操作成功后，之后同样的操作却会失败，这是因为文件或路径上的某些目录已经不复存在了。比如，客户端首先开始读一个文件，它向 NameNode 发出一个请求以获取文件第一个数据块的位置。但接下去获取其他数据块的第二个请求可能会失败。另一方面，删除一个文件并不会撤销客户端已经获得的对文件数据块的访问权限。而权限管理能使得客户端对一个文件的访问许可在两次请求之间被收回。重复一下，权限的改变并不会撤销当前客户端对文件数据块的访问许可。

MapReduce 框架通过传递字符串来指派用户身份，没有做其他特别的安全方面的考虑。文件或目录的所有者和组属性是以字符串的形式保存，而不是像传统的 UNIX 方式转换为用户和组的数字 ID。

这个发行版本的权限管理特性并不需要改变 DataNode 的任何行为。DataNode 上的数据块上并没有任何 Hadoop 所有者或权限等关联属性。

4.9.3 设置权限的 Shell 命令

HDFS 系统提供了 Shell 命令来设置文件的权限，如改变文件的用户、所属组等操作。具体说明如下。

1）改变文件的执行权限，命令如下：

```
chmod [-R] mode file …
```

只有文件的所有者或者超级用户才有权限改变文件执行权限。

2）改变文件所属组，命令如下：

```
chgrp [-R] group file …
```

使用 chgrp 命令的用户必须属于特定的组并且是文件的所有者，或者用户是超级用户。

3）改变文件所属用户，命令如下：

```
chown [-R] [owner][:[group]] file …
```

文件的所有者只能被超级用户更改。

4.9.4 超级用户

超级用户即运行 NameNode 进程的用户。宽泛地讲，如果你启动了 NameNode，你就

是超级用户。超级用户干任何事情，因为超级用户能够通过所有的权限检查。没有永久记号保留谁过去是超级用户；当 NameNode 开始运行时，进程自动判断谁现在是超级用户。

HDFS 的超级用户不一定是 NameNode 主机上的超级用户，也不需要所有集群的超级用户都是一个。同样，在个人工作站上运行 HDFS 的实验者，不需任何配置就已成为他部署实例的超级用户。另外，管理员可以用配置参数指定一组特定的用户，如果做了设定，这个组的成员也会是超级用户。

4.9.5 HDFS 权限配置参数

HDFS 权限配置的参数在 hdfs-site.xml 下面，参数的解释如表 4-10 所示。

表 4-10 HDFS 权限配置参数

属 性	功能描述
dfs.permissions = true	如果是 true，打开前文所述的权限系统。如果是 false，权限检查就是关闭的，但是其他的行为没有改变。这个配置参数的改变并不改变文件或目录的模式、所有者和组等信息。不管权限模式是开还是关，chmod、chgrp 和 chown 总是会检查权限。这些命令只有在权限检查背景下才有用，所以不会有兼容性问题。这就能让管理员在打开常规的权限检查之前可以可靠地设置文件的所有者和权限
dfs.web.ugi = webuser,webgroup	Web 服务器使用的用户名。如果将这个参数设置为超级用户的名称，则所有 Web 客户就可以看到所有的信息。如果将这个参数设置为一个不使用的用户，则 Web 客户就只能访问到“other”权限可访问的资源了。额外的组可以加在后面，形成一个用逗号分隔的列表
dfs.permissions.supergroup = supergroup	超级用户的组名
dfs.upgrade.permission = 777	升级时的初始模式。文件永不会被设置 x 权限。在配置文件中，可以使用十进制数 51110
dfs.umask = 022	umask 参数在创建文件和目录时使用。在配置文件中，可以使用十进制数 1810

4.10 本章小结

本章主要是讲解了 Hadoop 分布式文件系统（HDFS），从 HDFS 的原理到 HDFS 的使用做了全面的介绍。HDFS 是 Hadoop 里一个核心组件，任何基于 Hadoop 应用的工具都要用到 HDFS，学习这章对 Hadoop 系统的维护有很大的帮助。

第5章　Hadoop 文件 I/O 详解

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

在介绍 Hadoop 的 MapReduce 编程之前，先介绍一下 Hadoop 的 I/O 知识，以免在后面的章节中看到 IntWritable、LongWritable、Text、NullWritable 等概念时会不知所措。其实，IntWritable 就是其他语言（如 Java、C++）里的 int 类型，LongWritable 就是其他语言里的 long，Text 类似 String，NullWritable 就是 Null。学习其他编程语言之前，必须先学习数据类型，因此，在学习 Hadoop 的 MapReduce 编程之前，先学习 Hadoop 的 I/O 知识。

5.1　Hadoop 文件的数据结构

Hadoop 的 HDFS 和 MapReduce 子框架主要是针对大数据文件来设计的，在小文件的处理上不但效率低，而且十分消耗磁盘空间（每一个小文件占用一个 Block，HDFS 默认 Block 大小为 64MB）。解决办法通常是选择一个容器，将这些小文件组织起来统一存储。

HDFS 提供了两种类型的容器，分别是 SequenceFile 和 MapFile。

5.1.1　SequenceFile 存储

SequenceFile 的存储类似于 LogFile，不同的是 LogFile 记录的是纯文本数据，而 SequenceFile 记录是可序列化的字符数组。

SequenceFile 完成新记录的添加操作 API 为 fileWriter.append(key,value)；可以看到，每条记录以键值对的方式进行组织，但前提是 Key 和 Value 需具备序列化和反序列化的功能。

SequenceFile 主要由一个 Header 后跟多条 Record 组成，存储结构如图 5-1 所示。

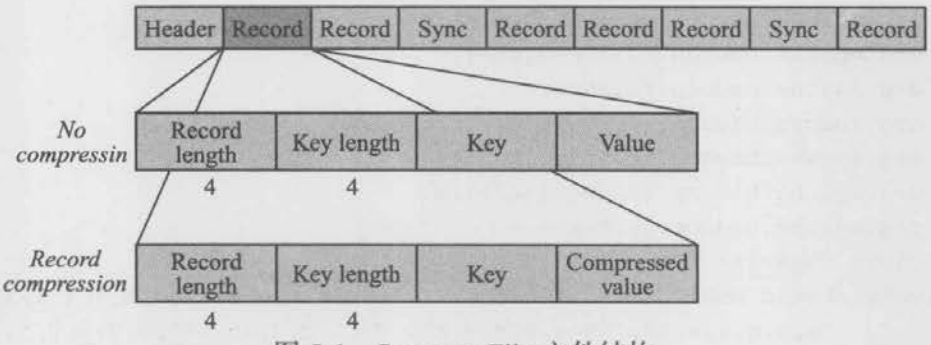


图 5-1　SequenceFile 文件结构

Header 主要包含了 Key classname、Value classname 存储压缩算法、用户自定义元数据等信息。此外，还包含了一些同步标识，用于快速定位到记录的边界。

每条 Record 以键值对的方式进行存储，用来表示它的字符数组可依次解析成：记录的长度、Key 的长度、Key 值和 Value 值，并且 Value 值的结构取决于该记录是否被压缩。

数据压缩有利于节省磁盘空间和加快网络传输，SequenceFile 支持两种格式的数据压缩，分别是 Record compression 和 Block compression。

- Record compression 是对每条记录的 Value 进行压缩。
- Block compression 是将一连串的 Record 组织到一起，统一压缩成一个 Block。

如图 5-2 所示，Block 信息主要存储了：块所包含的记录数、每条记录 Key 长度的集合、每条记录 Key 值的集合、每条记录 Value 长度的集合和每条记录 Value 值的集合。

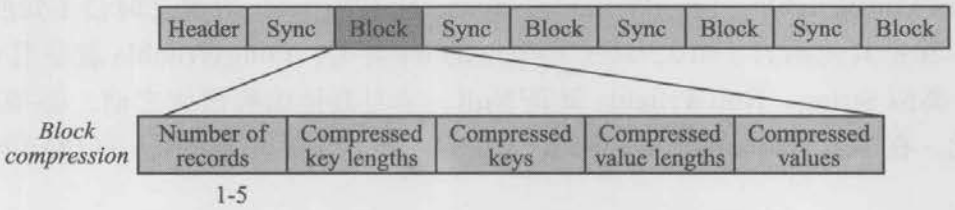


图 5-2 Block 结构模型

1. SequenceFile 写操作

通过 createWriter() 静态方法可以创建一个 SequenceFile 对象，并且返回一个 SequenceFile.Writer 实例。该静态方法有多个重载方法，但都需要指定写入的数据流（FSDataOutputStream、FileSystem 对象和 Path 对象）、Configuration 对象，以及键和值的类型。

存储在 SequenceFile 中的键和值不一定是 Writer 类型，任何一种通过 Serialization 类实现的序列化和反序列化类型均可被使用。

SequenceFile.Writer 实例可以通过 append() 方法在文件的末尾附加键值对。写完后，可以调用 close() 方法关闭写入流。

写入 SequenceFile 对象的操作如代码清单 5-1 所示。

代码清单 5-1 io\ SeqWrite.java

```
import java.net.URI;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.SequenceFile;
import org.apache.hadoop.io.Text;
public class SeqWrite {
    private static final String[] data = { "a,b,c,d,e,f,g", "h,i,j,k,l,m,n",
            "o,p,q,r,s,t", "u,v,w,x,y,z", "0,1,2,3,4", "5,6,7,8,9," };
```

```
    // Write an Sequrence File
    public static void main(String[] args) throws Exception {
        // 获取 Configuration 实例
        Configuration conf = new Configuration();
        // HDFS 文件系统
        FileSystem fs = FileSystem.get(URI.create("hdfs://localhost:9000"), conf);
        // 序列化文件路径
        Path path = new Path("test.seq");
        // 打开 SequenceFile.Writer 对象，写入键值
        SequenceFile.Writer writer = null;
        IntWritable key = new IntWritable();
        Text value = new Text();
        try {
            writer = SequenceFile.createWriter(fs, conf, path, key.getClass(),
                    value.getClass());
            for (int i = 0; i < 10000; i++) {
               key.set(i);
               value.set(SeqWrite.data[i % SeqWrite.data.length]);
               writer.append(key, value);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
           IOUtils.closeStream(writer);
        }
    }
}
```

顺序文件中存储的键是从 100 到 1 降序排列的整数，表示为 IntWritable 对象，值为 Text 对象。将每条记录追加到 SequenceFile.Writer 实例末尾之前，需要使用 getLength() 方法来获取文件访问的当前位置。

2. SequenceFile 读操作

从头到尾读取顺序文件的过程，是创建 SequenceFile.Reader 实例后反复调用 next() 方法迭代读取记录的过程。读取的是哪条记录与使用的序列化框架相关。如果使用的是 Writable 类型，通过键和值作为参数的 next() 方法可以将数据流中的下一条键值对读入变量中。

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

如果读取成功则返回 true，如果以读到文件尾则返回 false。如果读取非 Writable 类型的序列化框架，则需要使用以下方法。

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

这种情况下，请确保在 io.serializations 属性已经设置了你想使用的序列化框架。如果 next() 方法返回非空对象，可以从数据流中读取键值对，并且可以通过 getCurrentValue() 方

法读取该值；否则返回 null 表示到文件尾。

读取 SequenceFile 文件操作如代码清单 5-2 所示。

代码清单 5-2 io\ SeqRead.java

```
import java.io.IOException;
import java.net.URI;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.SequenceFile;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.util.ReflectionUtils;
public class SeqRead {
   public static void main(String[] args) throws Exception {
       // Configuration
       Configuration conf = new Configuration();
       // HDFS File Sytem
       FileSystem fs = FileSystem.get(URI.create("hdfs://localhost:9000"), conf);
       // Seq File Path
       Path path = new Path("test.seq");
       // Read SequenceFile
       SequenceFile.Reader reader = null;
       try {
            // Get Reader
            reader = new SequenceFile.Reader(fs, path, conf);
            // Get Key/Value Class
            Writable key = (Writable) ReflectionUtils.newInstance(
                     reader.getKeyClass(), conf);
            Writable value = (Writable) ReflectionUtils.newInstance(
                     reader.getValueClass(), conf);
            // Read each key/value
            while (reader.next(key, value)) {
                System.out.println(key + "\t" + value);
            }
       } catch (Exception e) {
            e.printStackTrace();
       } finally {
           IOUtils.closeStream(reader);
       }
    }
}
```

查看结果如下。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs -text test.seq |head
#结果（部分）
0       a,b,c,d,e,f,g
1       h,i,j,k,l,m,n
```

```
2       o,p,q,r,s,t
3       u,v,w,x,y,z
4       0,1,2,3,4
5       5,6,7,8,9,
6       a,b,c,d,e,f,g
7       h,i,j,k,l,m,n
8       o,p,q,r,s,t
9       u,v,w,x,y,z
```

SequenceFile 有三种压缩态。

- Uncompressed：未进行压缩的状态。
- Record Compressed：对每一条记录的 Value 值进行压缩（文件头中包含使用哪种压缩算法的信息）。
- Block Compressed：当数据量达到一定大小后，将停止写入进行整体压缩，其方法是把所有的 keylength、key、vlength、value 分别整合在一起进行压缩。

5.1.2 MapFile 存储

MapFile 的写入类似 SequenceFile 的写入，首先新建一个 MapFile.Writer 实例，然后调用 append() 方法将内容顺序写入。如果不按顺序写入内容，将抛出一个 IOException 异常，键必须是 WriableComparable 类型的实例，值必须是 Writable 类型的实例，这与 SequenceFile 中对应的正好相反。

新建一个 MapFile 对象，并写入一些内容，如代码清单 5-3 所示。

代码清单 5-3 io\ MapFileWriteDemo.java

```
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);
        }
      }
    }
```

执行以下程序。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$  hadoop MapFileWriteDemo  /user/hadoop/io/
test.map
```

观察 MapFile 会发现，实际上它是一个包含 data 和 index 两个文件的文件夹。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs -ls /user/hadoop/io/numbers.map
Found 2 items
-rw-r--r-3 jayliu supergroup 47898 2013-2-23 20:45 /user/hadoop/io/test.map/data
-rw-r--r-3 jayliu supergroup 47898 2013-2-23 20:45 /user/hadoop/io/test.map/index
```

data 和 index 两个文件都是 SequenceFile 格式，data 文件内容如下。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$  hadoop fs -text  /user/hadoop/io/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 包含一部分键和 data 文件中键到该键偏移量的映射，内容如下。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs -text /user/hadoop/io/numbers.map /index
1    128
129  6079
257  12054
385  18030
513  24002
641  29976
769  35947
897  41922
```

MapFile 是排序后的 SequenceFile，通过观察其目录结构可以看到 MapFile 由两部分组成，分别是 data 和 index。index 作为文件的数据索引，主要记录了每个 Record 的 Key 值，以及该 Record 在文件中的偏移位置。在 MapFile 被访问的时候，索引文件会被加载到内存，通过索引映射关系可迅速定位到指定 Record 所在文件位置。因此，相对 SequenceFile 而言，MapFile 的检索效率是高效的，缺点是会消耗一部分内存来存储 index 数据。

注意 MapFile 并不会把所有 Record 都记录到 index 中，默认情况下，每隔 128 条记录存储一个索引映射。当然，记录间隔可人为修改，通过 MapFIle.Writer 的 setIndexInterval() 方法，或修改 io.map.index.interval 属性。

与 SequenceFile 不同的是，MapFile 的 KeyClass 一定要实现 WritableComparable 接口，即 Key 值是可比较的。

注意，使用 MapFile 或 SequenceFile 虽然可以解决 HDFS 中小文件的存储问题，但也有一定局限性。例如：

- 文件不支持复写操作，不能向已存在的 SequenceFile（或 MapFile）追加存储记录。
- 当 write 流不关闭的时候，没有办法构造 read 流。也就是在执行文件写操作的时候，该文件是不可读取的。

5.1.3 SequenceFile 转换为 MapFile

在 MapFile 中搜索就相当于在索引和排序过的 SequenceFile 中搜索。所以自然就能联想到把 SequenceFile 转换为 MapFile。

SequenceFile 转化为 MapFile 如代码清单 5-4 示。

代码清单 5-4 io\SequenceFileConvertToMapFile.java

```
import java.net.URI;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.MapFile;
import org.apache.hadoop.io.SequenceFile;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.util.ReflectionUtils;
public class SequenceFileConvertToMapFile {
 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 class MapFileFixerTest{
  public void testMapFix(String sqFile) throws Exception{
   String uri=sqFile;
   Configuration conf=new Configuration();
   FileSystem fs=FileSystem.get(URI.create(uri),conf);
   Path path=new Path(uri);
   Path mapData=new Path(path,MapFile.DATA_FILE_NAME);
```

```
        SequenceFile.Reader reader=new SequenceFile.Reader(fs, mapData, conf);
        Class keyClass=reader.getKeyClass();
        Class valueClass=reader.getValueClass();
        reader.close();
        long entries=MapFile.fix(fs, path, keyClass, valueClass, false, conf);
        System.out.printf("create MapFile from sequenceFile about %d entries!", entries);
       }
      }
      public static void main(String[] args) throws Exception{
         SequenceFileConvertToMapFile fixer=new SequenceFileConvertToMapFile ();
         fixer.testMapFix(args[0]);
       }
    }
```

编译之后，把以下两个文件复制到 Hadoop 的 $HADOOP_HOME 下。

- SequenceFileConvertToMapFile $ MapFileFixerTest.class
- SequenceFile ConvertToMapFile.class

1）查看 test.map 目录，操作如下。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$  hadoop fs -ls /user/hadoop/test.map/
Found 2 items
-rw-r--r—3 jayliu supergroup 47898 2013-2-23 20:45  /user/hadoop/test.map/data
-rw-r--r—3 jayliu supergroup 47898 2013-2-23 20:45  /user/hadoop/test.map/index
```

2）删除 /user/hadoop/test.map/index 文件，操作如下。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$  hadoop fs -rm  /user/hadoop/test.map/index
```

3）运行代码清单 5-4，命令如下。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop SequenceFileConvertToMapFile /user/
hadoop/io/test.map
13/04/12 19:23:11 WARN util.NativeCodeLoader:Unable to load native-hadoop library
for your platform... using builtin-java classes where applicable
13/04/12 19:23:11 INFO compress.CodecPool: Got brand-new decompressor
13/04/12 19:23:11 INFO compress.CodecPool: Got brand-new compressor
create MapFile from sequenceFile about 1024 entries!
```

4）查看运行结果。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$  hadoop fs -ls /user/hadoop/test.map/
Found 2 items
-rw-r--r—3 jayliu supergroup 47898 2013-2-23 20:45  /user/hadoop/test.map/data
-rw-r--r—3 jayliu supergroup 47898 2013-2-23 20:45  /user/hadoop/test.map/index
```

index 文件在第二步被删除了，结果又生成了一个新的 index 文件，这样 SequenceFile 转化为 MapFile 就完成了。

5.2 HDFS 数据完整性

由于每一个磁盘或者网络上的 I/O 操作可能会对正在读写的数据处理不慎而出现错误，所以 HDFS 提供了两种数据检验方式：数据块检测程序以及校验和，以此来保证数据的完整性，而且这两种检验方式在 DataNode 节点上同时工作。

5.2.1 校验和

检测损坏数据的常用方法是在第一次启动 Hadoop 时，计算数据的校验和（checksum），在通道传输过程中，如果新生成的校验与原始的不完全匹配，那么数据就会被认为是损坏的。

从某个 DataNode 获取的数据块有可能是损坏的，损坏可能是由 DataNode 的存储设备错误、网络错误或者软件 bug 造成的。HDFS 客户端软件实现了对 HDFS 文件内容的校验和检查。当客户端创建一个新的 HDFS 文件，会计算这个文件每个数据块的校验和，并将校验和作为一个单独的隐藏文件保存在同一个 HDFS 名字空间下。当客户端获取文件内容后，它会检验从 DataNode 获取的数据跟相应的校验和文件中的校验和是否匹配，如果不匹配，客户端可以选择从其他 DataNode 获取该数据块的副本。

1. 客户端校验类 LocalFileSystem

LocalFileSystem 是客户端校验的类，使用 LocalFileSystem 写文件时，例如写入一个 filename 文件，文件系统客户端会自动创建一个 .filename.crc 文件。校验文件大小的字节数由 io.bytes.per.checksum 属性设置，默认是 512 bytes，即每 512 字节就生成一个 CRC-32 校验和。 .filename.crc 文件会保存 io.bytes.per.checksum 的信息，在读取的时候，会根据此文件进行校验。如果检测到错误，LocalFileSystem 将会抛出一个 ChecksumException 异常。

事实上 LocalFileSystem 是通过继承 ChecksumFileSystem 来实现校验的工作。

实现一个自定义文件上传校验的功能，如代码清单 5-5 所示。

代码清单 5-5　io\WriteToLocal.java

```
import java.io.IOException;
import java.io.OutputStream;
import java.net.URI;
import java.net.URISyntaxException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.LocalFileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.fs.RawLocalFileSystem;
public class WriteToLocal {
    public static void main(String[] args) throws IOException, URISyntaxException {
        Configuration conf = new Configuration();
        LocalFileSystem fs =  new LocalFileSystem(new RawLocalFileSystem());
```

```
        fs.initialize(new URI("file:///home/jayliu/file1.txt"), conf);
        OutputStream out = fs.create(new Path("file:/// home/jayliu/file1.txt "));
        for(int i = 0; i < 512*10;i++){
             out.write(97);
        }
        out.close();
        Path file = fs.getChecksumFile(new Path("file:/// home/jayliu/file1.txt"));
        System.out.println(file.getName());
        fs.close();
    }
}
```

2. ChecksumFileSystem 类

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

```
FileSystem rawFS = ...
FileSystem checksumFS = new ChecksumFileSystem(rawFS);
```

底层文件系统称为“源文件系统”，获取它可以使用 ChecksumFileSystem 实例的 getRaw FileSystem() 方法。ChecksumFileSystem 类还有其他一些与校验和有关的有用方法，比如 getChecksunFile() 方法用于获取任意一个文件的校验和文件路径。如果读取文件时，ChecksumFileSystem 类检测到错误，它会调用自己的 reportChecksumFailure() 方法，默认实现为空方法，但 LocalFileSystem 类会将这个出错的文件及其校验和移到用存储设备上一个名为 bad_files 的边际文件夹（side directory）中，管理员应定期检查这个坏文件并采取相应的处理。

5.2.2 数据块检测程序

数据块检测程序原理是，在 DataNode 节点上开启一个后台线程，用来定期验证存储在其之上的所有块，防止物理介质出现损减情况而造成的数据损坏。

关于校验和，HDFS 以透明的方式检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。验证数据的每一个校验块，都会创建一个单独的校验和，默认校验块大小是 512 字节，对应的校验和是 4 字节。DataNode 节点负载在存储数据（当然包括数据的校验和）之前验证它们收到的数据，如果此 DataNode 节点检测到错误，客户端会收到一个 CheckSumException。客户端读取 DataNode 节点上的数据时，会验证校验和，即将其与 DataNode 上存储的校验和进行比较。每一个 DataNode 节点都会维护着一个连续的校验和以及验证日志，里面有每一个 Block 的最后验证时间。客户端成功验证 Block 之后，便会告诉 DataNode 节点，DataNode 节点随之更新日志。这一点就涉及数据块检测程序 DataBlockScanner。

DataBlockScanner 相关联的类如图 5-3 所示。

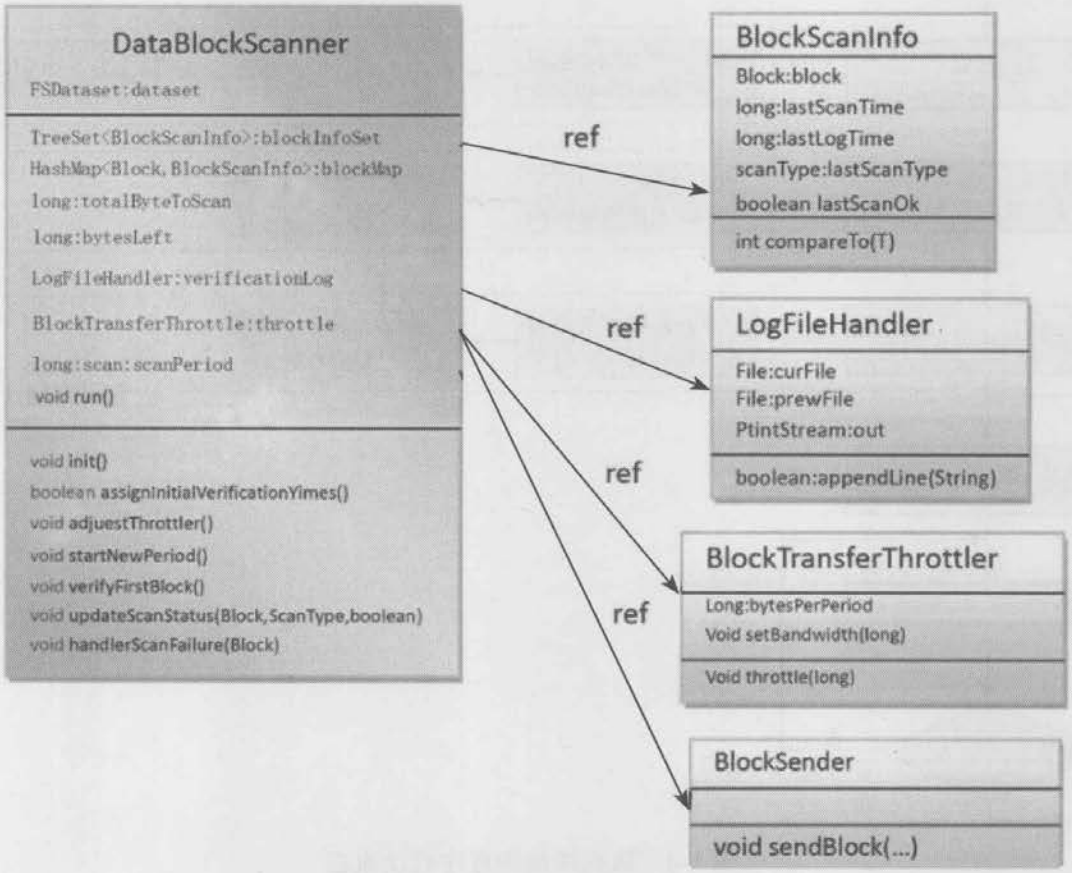


图 5-3 DataBlockScanner 类关系图

DataBlockScanner 类属性的含义解释如下。

- dataset：数据块管理器。
- blockInfoSet：数据块扫描信息集合，按照上一次扫描时间和数据块 ID 升序排序，以便快速获取验证到期的数据块。
- blockMap：数据块和数据块扫描信息的映射，以便能够根据数据块快速获取对应的扫描信息。
- totalBytesToScan：一个扫描周期中需要扫描的总数据量。
- bytesLeft：一个扫描周期中还剩下需要扫描的数据量。
- throttler：扫描时 I/O 速度控制器，需要根据 totalBytesToScan 和 bytesLeft 信息衡量。
- verificationLog：数据块的扫描验证日志记录器。
- scanPeriod：表示一个扫描周期，可以由 Datanode 的配置文件来设置，配置属性为 dfs.datanode.scan.period.hours，单位是小时。

数据块检测程序的流程如图 5-4 所示，DataBlockScanner 作为 DataNode 的一个后台工作线程，跟着 DataNode 一块启动。

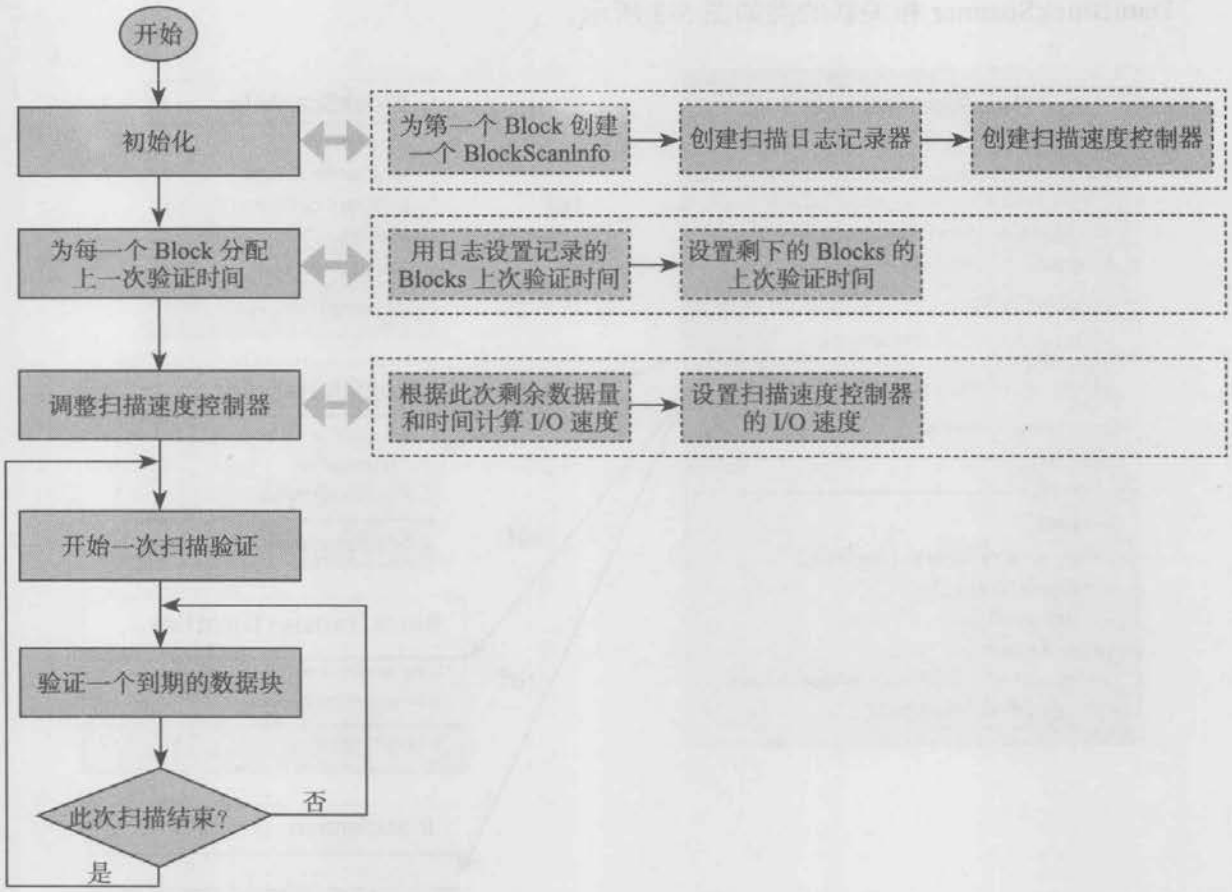


图 5-4 数据块检测程序启动流程

5.3 文件序列化

序列化和反序列化在分布式数据处理中，主要应用于进程间通行和永久存储两个领域。

序列化（Serialization）就是结构化的对象转化为字节流，这样可以方便在网络上传输和写入磁盘进行永久存储。

反序列化（Deserialization）就是指将字节流转回结构化对象的逆过程。

5.3.1 进程间通信对序列化的要求

在 Hadoop 系统中，系统中多个节点上的进程间通信是通过远程过程调用（Remote Procedure Call，RPC）实现的，RPC 协议将消息序列转化为二进制流后发送到远程节点，远程节点接着将二进制流反序列化为消息，所以 RPC 对于序列化有以下要求（也就是进程间通信对于序列化的要求）。

- **紧凑**。紧凑的格式可以提高传输效率，充分利用网络带宽，要知道网络带宽是数据中心的一种非常重要的资源。
- **快速**。进程间通信是分布式系统的重要内容，所以必须减少序列化和反序列化的开

销，这样可以提高整个分布式系统的性能。

- **可扩展**。通信协议为了满足一些新的需求，比如在方法调用的过程中增加新的参数，或者新的服务器系统要能够接受老客户端旧格式的消息，这样就需要直接引进新的协议，序列化必须满足可扩展的要求。
- **互操作**。可以支持不同语言的客户端（比如 C++、Java、Python 等）与服务器交互。

前面说了，序列化的目的是可以方便在网络上传输和写入磁盘进行永久存储，下面说一说数据永久存储对于序列化的要求。前面说的序列化的 4 个要求也是数据永久存储所要求的，存贮格式紧凑可以高效地利用存储空间，快速可以减少读写数据的额外开销，可扩展就可以方便地读取原始格式的数据，互操作即可满足不同的编程语言来读写永久存贮的数据。

5.3.2 Hadoop 文件的序列化

Hadoop 文件的序列化，没有采用 Java 的序列化机制，而是自己开发了一套文件的序列化框架。既然 Java 自带了一套序列化框架，为什么 Hadoop 不用，而是重新开发了一个序列化框架，原因有两个。

1）Java 的序列化机制在每个类的对象第一次出现的时候保存了每个类的信息，比如类名，第二次出现的类对象会有一个类的 reference，导致空间的浪费。

2）Java 序列化机制不能复用对象，Java 反序列化的时候，每次要构造出新的对象。在 Hadoop 的序列化机制中，反序列化的对象是可以复用的。

Hadoop 中定义了两个序列化相关的接口：Writable 和 Comparable，这两个接口可以合成一个 WritableComparable 接口，接下来分别介绍这两个接口。

5.3.3 Writable 接口

在 Hadoop 中，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 流读取其状态。Writable 接口如下。

```
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;
}
```

例如一个特别的 Writable，可以对它进行哪些操作。IntWritable 是对 Java 的 int 类型的封装，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，也可以使用构造函数来赋值：

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

写一个小的辅助方法用于检查 IntWritable 的序列化形式，把 java.io.ByteArrayOutputStream 封装到 java.io.DataOutputStream 中（java.io.DataOutput 的一个实现），以此来捕获序列数据流中的字节，实现代码如下。

```
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 4 字节写入，然后使用 JUnit 来断言是否正确，实现代码如下。

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

字节使用顺序写入，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，以 Hadoop 的 StringUtils 为例，来表示十六进制，实现代码如下。

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

再来试试反序列化。创建一个帮助方法从 1 字节数组读取一个 Writable 对象，实现代码如下。

```
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;
}
```

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，最后发现它的值（使用 get 方法检索得到）还是原来的值 163，实现代码如下。

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

deserialize() 方法对 163 进行反序列化，然后使用 get() 方法获取反序列化的值，结果是 163。

5.3.4 WritableComparable 接口

WritableComparable 接口是 Writable 和 java.lang.Comparable 接口的子接口，并且具有这两个接口的功能。WritableComparable 的接口声明如下所示。

```
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，键和键之间的比较在排序阶段完成。Hadoop 提供的一个优化方法是从 Java Comparator 的 RawComparator 扩展而来的。RawComparator 接口如下。

```
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);
}
```

RawComparator 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，IntWritable 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。

- 提供一个默认对原始 compare() 函数的调用，对从数据流比较的对象进行反序列化，然后调用对象的 compare() 方法。
- 充当 RawComparator 实例的一个工厂方法（Writable 方法已经注册）。

例如，获得 IntWritable 的 Comparator，代码如下。

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

Comparator 是对象比较器，方法 int compare(Object o1，Object o2) 返回一个基本类型的整型，返回负数表示 o1 小于 o2，返回 0 表示 o1 和 o2 相等，返回正数表示 o1 大于 o2。下面以两个 IntWritable 类型为例，来比较它们的大小，代码如下。

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

w1 和 w2 的序列化比较大小如下。

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

5.3.5 自定义 Writable 接口

Hadoop 自带一系列 Writable 实现，如 IntWritable、LongWritable 等，可以满足一些简单的数据类型。但有时，复杂的数据类型需要自己自定义实现。通过自定义 Writable，能够完全控制二进制表示和排序顺序。

Writable 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。

1. 自定义一个 Writable 类型 TextPair

为了演示如何创建一个自定义 Writable，编写一个表示一对字符串的实现，名为 TextPair，如代码清单 5-6 所示。

代码清单 5-6 io\ TextPair.java

```
package io;
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，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，会重写 java.lang.Object 的 hashCode()、equals() 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定 reduce 函数的分区在大小上是相当的。

TextPair 是 WritableComparable 的实现，所以它提供了 compareTo() 方法的实现，加入我们希望的排序：通过一个一个 String 逐个排序。

注意 TextPair 不同于前面的 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。

2. 实现一个快速的 RawComparator

上例中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 对象呢?

事实证明，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，利用第一或者第二个字符串的偏移量来调用它。下面例子给出了具体方法（注意，该代码嵌套在 TextPair 类中）。

```
public static class Comparator extends WritableComparator {
  public Comparator() {
    super(TextPair.class);
  }
  @Override
  public int compare(byte[] b1, int s1, int l1,
  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，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 first L1 和 first L2，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() 返问）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。

3. 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。WritableUtils 工具类提供的方法也非常方便。

如果可能，还应把自定义 Comparator 继承 RawComparator 类。RawComparator 类自带的 Comparator 实现排序顺序不同于默认 Comparator 定义的自然排序。下面的例子实现了

TextPair 类的比较器（comparator），名为 First Comparator。这里实现了对 TextPair 类里的 first 属性进行排序。实现的代码如代码清单 5-7 所示。

代码清单 5-7 io\ FirstComparator.java

```
public static class FirstComparator extends WritableComparator {
   private static final Text.Comparator TEXT_COMPARATOR = new Text.Comparator();
   public FirstComparator() {
     super(FirstComparator.class);
   }
   @Override
   public int compare(byte[]b1, ints1, intl1, byte[]b2, ints2, intl2) {
     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, first L1, b2, s2, first L2);
     } 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);
    }
}
```

我们重写了 public int compare(WritableComparable a, WritableComparable b) 方法，在该方法里对 first 属性进行排序，以下两个方法的语义是相同的。

```
public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2)
public int compare(WritableComparable a, WritableComparableb)
```

虽然两个 compare() 方法的语义是相同的，只是实现的方式不一样，第一个方法的实现就是使用了 WritableUtils 工具类。

5.3.6 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，并不是所有类型都能使用。为了提供相应的支持，Hadoop 有一个简便的序列化框架，在 org.apache.hadoop.io.serializer 包中。包中的 WritableSerialization 类就是 Writable 类型的序列化实现。除了 WritableSerialization，还有一个 JavaSerialization 类，此类让我们能方便地在 MapReduce 程序中使用标准的 Java 类型，不过不如 Writable 对象有效，不推荐使用。

5.3.7 数据序列化系统 Avro

Avro 是一个独立于编程语言的数据序列化系统，托管在 Apache 上，由 Doug Cutting 创建（Hadoop 也出自他手）的，用来解决 Hadoop 中 Writable 类型的不足：缺乏语言的可移植性，拥有一个可被多种语言（C、C++、Java、Python 等）处理的数据格式。Avro 通常是用 JSON 编写的。

Avro 依赖于模式（Schema），Avro 数据的读写操作是很频繁的，而这些操作都需要使用模式，以减少写入每个数据资料的开销，使得序列化快速而又轻巧。这种数据及其模式的自描述方便于动态脚本语言的使用。

当 Avro 数据存储到文件中时，它的模式也随之存储，这样任何程序都可以对文件进行处理。如果需要以不同的模式读取数据，这也很容易解决，因为两个模式都是已知的。

在 RPC 中使用 Avro 时，服务器和客户端可以在握手连接时交换模式。服务器和客户端有着彼此全部的模式，因此相同命名字段、缺失字段和多余字段等信息之间通信中需要解决的一致性问题就可以容易解决。

Avro 模式是用 JSON（一种轻量级的数据交换模式）定义的，这样对于已经拥有 JSON 库的语言可以容易实现。

Avro 提供着与诸如 Thrift 和 Protocol Buffers 等系统相似的功能，但是在一些基础方面还是有区别的，主要区别如下。

- ❑ 动态类型：Avro 并不需要生成代码，模式和数据存放在一起，而模式使得整个数据的处理过程并不生成代码、静态数据类型等。这方便了数据处理系统和语言的构造。
- ❑ 未标记的数据：由于读取数据的时候模式是已知的，需要和数据一起编码的类型信息就很少了，这样序列化的规模也就小了。
- ❑ 不需要用户指定字段号：即使模式改变，处理数据时新旧模式都是已知的，所以通过使用字段名称可以解决差异问题。

1. Avro 的基本类型

Avro 框架提供了如下类型。

- ❑ null：0 字节。
- ❑ boolean：1 字节，0（false）或 1（true）。
- ❑ int 和 long：使用变长的 zig-zag 编码。
- ❑ float：4 字节。
- ❑ double：8 字节。
- ❑ bytes：1 个 long，后边跟着字节序列。
- ❑ string：1 个 long，后边跟着 UTF-8 编码的字符。

2. Avro 的消息格式

Avro 的消息通信使用 JSON 格式，用一个案例来介绍 Avro 的通信格式，代码如下所示。

```
{
    "namespace": "avro",
    "protocol": "messageProtocol",
    "doc": "This is a message.",
    "name": "Message",
    "types": [
        {"name":"message", "type":"record",
            "fields":[
                {"name":"name", "type":"string"},
                {"name":"type", "type":"int"},
                {"name":"price", "type":"double"},
                {"name":"valid", "type":"boolean"},
                {"name":"content", "type":"bytes"}
        ]}
    ],
    "messages":    {
        "sendMessage":{
            "doc" : "test",
            "request" :[{"name":"message","type":"message" }],
            "response" :"message"
        }
    }
}
```

上面的 Avro 通信格式定义了 1 种类型叫做 message，有 5 个成员：name、type、price、valid、content；还定义了 1 个消息服务叫做 sendMessage，输入有 1 个参数，类型是 message，返回 message。

如果想全面学习 Avro 请访问地址 http://avro.apache.org/docs/1.7.4/#intro。

5.4 Hadoop 的 Writable 类型

Hadoop 中，并没有使用 Java 自带的基本类型类（Integer、Float 等），而是使用自己开发的类，包括 IntWritable、FloatWritable、BooleanWritable、LongWritable、ByteWritable、BytesWritable、DoubleWritable。

5.4.1 Writable 类的层次结构

Writable 接口是一个序列化对象的接口，能够将数据写入流或者从流中读出。实现之后，能够进行特定类型数据的异地传输。

Writable 类的层次结构如图 5-5 所示，它们都实现 Writable Comparable 接口。除了这些基本类型的定义，还添加了 VLongWritable 和 VIntWritable，V 指可变长度，例如，long 型的 1 实际只需要 1 字节的空间，由于是 long 型的，所以会占用 8 字节的空间，而 VLongWritable 中，会根据数值的大小分配适当的空间（仅分配 1 字节），以节省空间。在

基本数据类型的 Writable 类中，readFields(DataInput in) 方法是直接调用 in.readLong()（以 LongWritable 为例），而在 VLongWritable 与 VIntegerWritable 中，readFields (DataInputin) 方法使用了静态类 WritableUtils 中的 readVLong(DataInput in) 方法。WritableUtils 是一个工具类，用于提供 I/O 中的 Writable 类的一些静态方法。

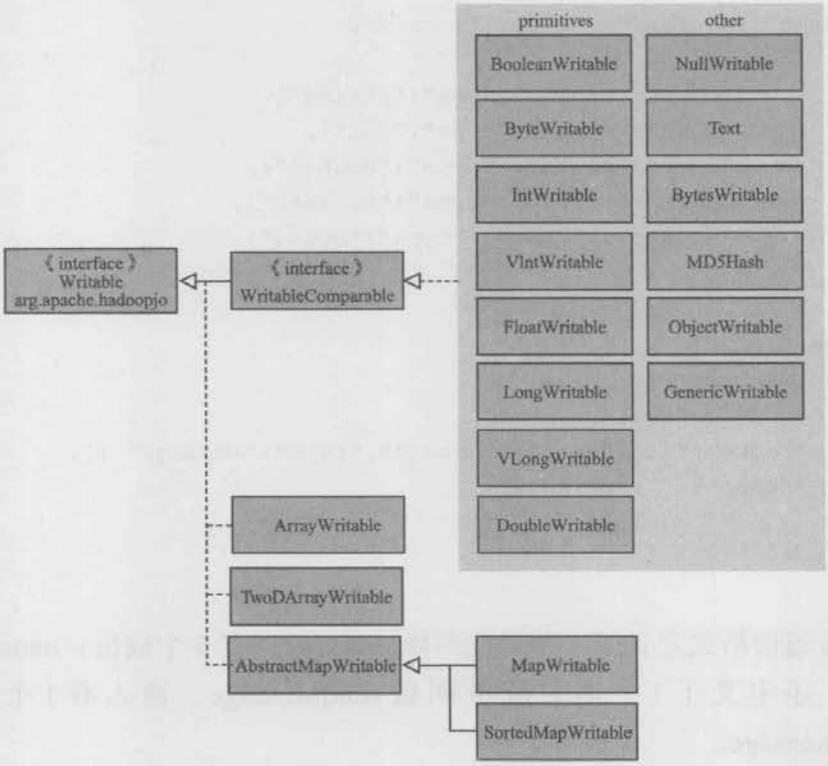


图 5-5 Hadoop 自带 Writable 类层次结构

表 5-1 展示 Java 基本类型和 Writable 的对应关系。

表 5-1 Java 基本类型和 Writable 的对应关系

Java 基本类型	Writable 实现	字节	Java 基本类型	Writable 实现	字节
boolean	BooleanWritable	1	float	FloatWritable	4
byte	ByteWritable	1	long	LongWritable VlongWritable	8 1~9
int	IntWritable VintWritable	4 1~5	double	DoubleWritable	8

5.4.2 Text 类型

Text 类是一种 UTF-8 格式的 Writable 类型。可以将它理解为一种与 java.lang.String 类似的 Writable 类型。

Text 使用 int 型在字符编码中存储字节数，最大值是 2GB。它有一个 find() 方法，相当于 Java 的 String 中的 indexOf() 方法。

Text 类常用方法如下。

```
void append(byte[] utf8,int start,int len)  //增加一段字节到末尾
int byteToCodePoint(ByteBuffer bytes)       //在当前 buffer 位置返回下一个 code 的位置
void clear()                                //清空字符串
String decode(byte[] utf8)                  //使用 UTF-8 格式把字节数组转为 string
ByteBuffer encode(String string)            //把 string 转为字节
void set(byte[] utf8)                       //设置内容
```

set(byte[] utf8) 方法可以用来重用 Text 实例，代码如下。

```
Text text = new Text("hadoop");
text.set("autoNavi");
```

5.4.3 NullWritable 类型

NullWritable 是一个特殊的 Writable 类型，它的序列化长度是零长度的。没有字节被写入流或从流中读出，一般当作占位符。

例如，在 MapReduce 中，如果不需要使用键或值，可以将键或值声明为 NullWritable 类型，结果是存储常量空值。NullWritable 是一个不可变的单实例类型，通过调用 NullWritable.get() 方法可以获得 NullWritable 的实例。

5.4.4 ObjectWritable 类型

针对一些常用 Java 类，ObjectWritable 是一种多用途的封装，它使用 Hadoop 的 RPC 来封送（marshal）和反封送（unmarshal）方法参数和返回类型。

当一个字段中包含多个类型时，ObjectWritable 是非常有用的，如果 SequenceFile 中的值包含多个类型，就可以将值类型声明为 ObjectWritable。

5.4.5 GenericWritable 类型

很多时候，特别是处理大数据的时候，我们希望一个 MapReduce 过程就可以解决几个问题。这样可以避免再次读取数据。比如，在做文本聚类和分类的时候，Mapper 读取语料进行分词后，要同时算出每个词条的频率以及它的文档的频率，前者对于每个词条来说其实是个向量，它代表此词条在 N 篇文档中的词频；而后者就是一个非负整数。这时候就可以借助一种特殊的 Writable 类 GenericWritable。

GenericWritable 用法是，继承这个类，然后把要输出 Value 的 Writable 类型加进它的 Class 静态变量里。

5.5 文件压缩

文件压缩有两大好处：1）减少文件存储所需要的空间；2）加快数据在网络和磁盘上

的传输。处理大量文件时，需要仔细考虑在 Hadoop 中如何使用压缩，以及使用哪种压缩。

5.5.1 Hadoop 支持的压缩格式

Hadoop 目前支持很多种压缩格式，Hadoop 下各种压缩算法的压缩比、压缩时间等比较如表 5-2 所示。

表 5-2 Hadoop 中压缩比较

压缩算法	原始文件大小（GB）	压缩后的文件大小（GB）	压缩速度（MB/s）	解压缩速度（MB/s）
gzip	8.3	1.8	17.5	58
bzip2	8.3	1.1	2.4	9.5
LZO-bset	8.3	2	4	60.6
LZO	8.3	2.9	49.3	74.6

在考虑那些将由 MapReduce 处理的数据压缩时，压缩格式是否支持分割是很重要的考虑因素。考虑存储在 HDFS 中的未压缩文件，其大小为 1GB，HDFS 的块大小为 64MB，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 1 个输入分片（split，也称为“分块”。对于 Block，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 1GB。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。

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。

在假设的例子中，如果是一个 LZO 格式的文件，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 PI 近似值），因此它支持分割机制。

对于文件的收集，这些问题会稍有不同。ZIP 是存档格式，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。

5.5.2 Hadoop 中的编码器和解码器

编码器和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，例如，GzipCodec 封装了 gzip 的压缩和解压算法。表 5-3 列出了 Hadoop 可用的编码器和解码器。

表 5-3 Hadoop 的编码器和解码器

压缩格式	Hadoop 压缩编码器和解码器
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，因此，它的 Hadoop 编码器和解码器必须单独下载，地址是 http://code.google.com/p/hadoop-gpl-compression/。

lzop 编码器和解码器兼容 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）。接下来介绍编码器和解码器两个重要的类 CompressionCodec 和 CompressionCodecFactory。

1. CompressionCodec 类

CompressionCodec 是对流进行压缩和解压缩，有两个方法用于压缩或解压缩数据。想对正在被写入一个输出流的数据进行压缩，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，想对从输入流读取的数据进行解压缩，则调用 createInputStream (InputStreamin) 函数获得一个 CompressionInputStream，进而从底层的流读取未压缩的数据。CompressionOutputStream 和 CompressionInputStream 类，类似 Java 中的 java.util.zip. DeflaterOutputStream 和 java.util.zip.DeflaterInputStream 类，前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 文件格式。

下例使用 API 压缩从标准输入读取的数据并将它写到标准输出，代码如下。

```
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 的合法全名作为命令行的第一个参数。使用

ReflectionUtils 来建立一个新的实例，然后获得一个压缩好的 System.out。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，向压缩程序表明结束向压缩流写入数据，但不关闭流。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，具体操作如下。

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

2. CompressionCodecFactory 类

用 CompressionCodecFactory 类可以推断 CompressionCodecs 的压缩格式，在阅读一个压缩文件时，通常可以从其扩展名来推断出它的编码 / 解码器。例如，以 gz 结尾的文件可以用 GzipCodec 阅读。每种压缩格式的比较如表 5-4 所示。

表 5-4 压缩格式的比较

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

CompressionCodecFactory 提供了 getCodec() 方法，用于将文件扩展名映射到相应的 CompressionCodec，此方法接受一个 Path 对象。下面的例子显示了一个应用程序，此程序使用这个功能来解压缩文件。

```
public class FileDecompressor {
    public static void main(String[] args) throws Exception {
        String uri = args[0];
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(URI.create(uri), conf);
        Path inputPath = new Path(uri);
        CompressionCodecFactory factory = new CompressionCodecFactory(conf);
        CompressionCodec codec = factory.getCodec(inputPath);
        if (codec == null) {
            System.err.println("No codec found for " + uri);
            System.exit(1);
        }
        String outputUri =
        CompressionCodecFactory.removeSuffix(uri, codec.getDefaultExtension());
        InputStream in = null;
        OutputStream out = null;
        try {
            in = codec.createInputStream(fs.open(inputPath));
            out = fs.create(new Path(outputUri));
```

```
            IOUtils.copyBytes(in, out, conf);
        } finally {
            IOUtils.closeStream(in);
            IOUtils.closeStream(out);
        }
    }
}
```

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，调用如下程序便把一个名为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。表5-5所示为默认情况下，Hadoop提供的所有编码/解码器。如果你有一个要注册的编码/解码器（如外部托管的LZO编码/解码器）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与已知扩展名相匹配的编码/解码器。

表5-5 Hadoop默认压缩/解压

属性名	类型	默认值	功能描述
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列表

5.5.3 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）。Java和本地提供的每个压缩格式的实现如表5-6所示。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5-6 本地库列表

压缩格式	Java实现	本地实现	压缩格式	Java实现	本地实现
DEFLATE	是	是	bzip2	是	否
gzip	是	是	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 库被使用。

5.5.4 可分割压缩 LZO

在 Hadoop 中使用 LZO 算法可以减小数据的大小，以及减少数据的磁盘读写时间，在 HDFS 中存储压缩数据，可以使集群能保存更多的数据，延长集群的使用寿命。不仅如此，由于通常 MapReduce 作业的瓶颈都在 I/O 上，存储压缩数据就意味着更少的 I/O 操作，Job 运行更加高效。

但是在 Hadoop 上使用压缩也有两个比较麻烦的地方。

- ❑ 有些压缩格式不能被分块，以及并行处理，比如 gzip。
- ❑ 另外一些压缩格式虽然支持分块处理，但是解压的过程非常缓慢，使 Job 的瓶颈转移到 CPU 上，例如 bzip2。

如果能够拥有一种压缩算法，即能够被分块、并行处理、速度也非常快，那就非常的理想。这种方式就是 LZO。

LZO 的压缩文件是由许多小的 Block 组成（约 256KB），使 Hadoop 的 Job 可以根据 Block 的划分来切割任务（split job）。不仅如此，LZO 在设计时就考虑到了效率问题，其解压速度是 gzip 的两倍，这就让它能够节省很多的磁盘读写，它的压缩比不如 gzip，压缩后的文件大约比 gzip 压缩的大一半，但是这样仍然比没有经过压缩的文件节省 20% ～ 50% 的存储空间，这样在效率上极大地提高 Job 执行速度。

5.5.5 压缩文件性能比较

对于 Hadoop 文件常用的几种压缩方法，写一个 Java 程序进行比较。代码清单 5-8 给出一个大文件（bigfile.txt），用各种方式压缩后最终复制到 HDFS 中。

代码清单 5-8 io\ HadoopCodec.java

```
package com.charles.Hadoop.fs;
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.URI;
import org.apache.Hadoop.conf.Configuration;
import org.apache.Hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.Hadoop.io.IOUtils;
import org.apache.hadoop.io.compress.CompressionCodec;
import org.apache.Hadoop.io.compress.CompressionCodecFactory;
```

```
import org.apache.hadoop.io.compress.GzipCodec;
import org.apache.Hadoop.util.ReflectionUtils;
public class HadoopCodec {
    public static void main(String[] args) throws Exception {
        // TODO Auto-generated method stub
        String inputFile = "bigfile.txt";
        String outputFolder = "hdfs://192.168.129.35:9000/user/Hadoop-user/codec/";
        // String outputFile="bigfile.gz";
        // 读取 Hadoop 文件系统的配置
        Configuration conf = new Configuration();
        conf.set("Hadoop.job.ugi", "hadoop-user,hadoop-user");
        //测试各种压缩格式的效率
        //gzip
        long gzipTime = copyAndZipFile(conf, inputFile, outputFolder,
            "org.apache.Hadoop.io.compress.GzipCodec", "gz");
        //bzip2
        long bzip2Time = copyAndZipFile(conf, inputFile, outputFolder,
            "org.apache.Hadoop.io.compress.BZip2Codec", "bz2");
        //deflate
        long deflateTime = copyAndZipFile(conf, inputFile, outputFolder,
            "org.apache.Hadoop.io.compress.DefaultCodec", "deflate");
        System.out.println("被压缩的文件名为: "+inputFile);
        System.out.println("使用 gzip 压缩, 时间为: "+gzipTime+" 毫秒!");
        System.out.println("使用 bzip2 压缩, 时间为: "+bzip2Time+" 毫秒!");
        System.out.println("使用 deflate 压缩, 时间为: "+deflateTime+" 毫秒!");
    }
    public static long copyAndZipFile(Configuration conf, String inputFile,
            String outputFolder, String codecClassName,
            String suffixName) throws Exception {
        long startTime = System.currentTimeMillis();
        // 因为本地文件系统是基于 java.io 包的, 所以创建一个本地文件输入流
        InputStream in = new BufferedInputStream(new FileInputStream(inputFile));
        //去掉扩展名提取 basename
        String baseName = inputFile.substring(0, inputFile.indexOf("."));
        //构造输出文件名, 它是路径名 + 基本名 + 扩展名
        String outputFile = outputFolder + baseName + "."+suffixName;

        FileSystem fs = FileSystem.get(URI.create(outputFile), conf);
        // 创建一个编码解码器, 通过反射机制根据传入的类名来动态生成其实例
        CompressionCodec codec = (CompressionCodec) ReflectionUtils.newInstance
            (Class.forName(codecClassName), conf);
        // 创建一个指向 HDFS 目标文件的压缩文件输出流
        OutputStream out = codec.createOutputStream(fs.create(new Path(outputFile)));
        // 用 IOUtils 工具将文件从本地文件系统复制到 HDFS 目标文件中
        try {
            IOUtils.copyBytes(in, out, conf);
        } finally {
            IOUtils.closeStream(in);
            IOUtils.closeStream(out);
        }
```

```
            long endTime = System.currentTimeMillis();
            return endTime - startTime;
        }
    }
```

案例运行结果如下。

```
被压缩的文件名为: bigfile.txt
使用 gzip 压缩，时间为: 11807 毫秒！
使用 bzip2 压缩，时间为: 44982 毫秒！
使用 deflate 压缩，时间为: 3696 毫秒！
```

5.5.6 Snappy 压缩

Snappy 是用来压缩和解压缩的 C++ 开发包，其目标不是最大限度压缩，而且不兼容其他压缩格式。Snappy 旨在提供高速压缩速度和合理的压缩率。Snappy 比 zlib 更快，但文件相对要大 20% ～ 100%。在 64 位模式的 Core i7 处理器上，可达每秒 250 ～ 500 兆的压缩速度。Snappy 是一个压缩库，其前身是 Zippy。虽然只是一个数据压缩库，它却被 Google 用于许多内部项目，其中包括 BigTable、MapReduce 和 RPC。Google 宣称它在这个库本身及其算法做了数据处理速度上的优化，并没有考虑输出大小以及和其他类似工具的兼容性问题。Snappy 特意为 64 位 x86 处理器做了优化，在单个 Intel Core i7 处理器内核上能够达到至少每秒 250MB 的压缩速率和每秒 500MB 的解压速率。

如果允许损失一些压缩率的话，那么可以达到更高的压缩速度，虽然生成的压缩文件可能会比其他库的要大上 20% ～ 100%，但是，相比其他的压缩库，Snappy 却能够在特定的压缩率下拥有惊人的压缩速度，“压缩普通文本文件的速度是其他库的 1.5 ～ 1.7 倍，HTML 能达到 2 ～ 4 倍，但是对于 JPEG、PNG 以及其他已压缩的数据，压缩速度不会有明显改善”。

5.5.7 gzip、LZO 和 Snappy 比较

目前，gzip、LZO 和 Snappy 是 Hadoop 的压缩中用的最多，可以说根据公司对业务和存储的需求，会选择这三种压缩中的一种，下面用一张表格来比较这三种格式，如表 5-7 所示。

表 5-7 gzip、LZO 和 Snappy 压缩性能比较

压缩算法	是否可分割	是否需要创建索引	压缩比 (%)	压缩速度 (%)	解压速度 (%)
gzip	否	否	13.4	21	118
LZO	是	是	20.5	135	410
Snappy	否	否	22.2	172	409

比较 gzip、LZO 和 Snappy 三种压缩的性能。

- gzip：压缩率最高，但是其实 CPU 密集型的，对 CPU 的消耗比其他算法要多，压缩和解压速度也慢。

- LZO：压缩率居中，比 gzip 要低一些，但是压缩和解压速度明显要比 gzip 快很多，其中解压速度快的更多。
- Zippy/Snappy：压缩率最低，而压缩和解压速度要比 LZO 稍快一些。

5.6 本章小结

本章主要讲一些 Hadoop I/O 底层的知识，让读者了解 Hadoop 底层的一些原理，如数据在 Hadoop 上是怎么传输的。学完这章对 Hadoop 的输入和输出、序列化和反序列化、Writable 类型等有所理解，学习下面的章节才能得心应手。

第6章　MapReduce 工作原理

MapReduce 是一种编程模型，用于大规模数据集（大于 1TB）的并行运算。“Map”（映射）和“ Reduce”（化简）的概念和它们的主要思想，都是从函数式编程语言借用来的，还有从矢量编程语言里借来的特性。极大地方便了编程人员在不会分布式并行编程的情况下，将自己的程序运行在分布式系统上。

当前的软件实现是指定一个 map 函数，用来把一组键值对映射成一组新的键值对，指定并发的 reduce 函数，用来保证所有映射的每一个键值对共享相同的键组。

6.1　MapReduce 的函数式编程概念

MapReduce 的创意和灵感来源于函数式编程。map 和 reduce 是函数式编程中两个常用函数。在函数式编程中，map 函数对列表的每个元素执行操作或函数。例如，在列表 [1, 2, 3, 4] 上执行 multiple-by-two 函数会产生另一个列表 [2, 4, 6, 8]。执行这些函数时，原有列表不会被修改。函数式编程认为，应当保持数据不可变，避免在多个进程或线程间共享数据。这意味着演示的 map 函数虽然很简单，却可以通过两个或更多线程在同一个列表上同时执行，线程之间互不影响，因为列表本身没有改变。

6.1.1　列表处理

MapReduce 程序是用来并行计算大规模海量数据的，这需要把工作流分划到大量的机器上去，如果组件（component）之间可以任意地共享数据，那这个模型就没法扩展到大规模集群上去（数百或数千个节点），用来保持节点间数据的同步而产生的通信开销，会使系统在大规模集群上变得不可靠和效率低下。 实际上，所有在 MapReduce 上的数据元素都是不可变的，这就意味着它们不能够被更新。如果在一个 Mapping 任务中改变一个输入键值对，它并不会反馈到输入文件；节点间的通信只在产生新的输出键值对时发生，Hadoop 系统会把这些输出传到下一个执行阶段。

从概念上讲，MapReduce 程序将输入数据元素列表转变成输出数据元素列表。一个 MapReduce 程序会重复这个步骤两次，并用两个不同的术语描述：map 和 reduce，这些术语来自于列表处理（List Processing）语言，如 LISP、Scheme 或 ML。

在 MapReduce 中，没有一个值是单独的，每一个值都会有一个键与其关联，键标识相关的值。举个例子，从多辆车中读取的时间编码车速表日志可以由车牌号码标识，如下所示。

```
AAA-123    65mph,12:00pm
ZZZ-789    50mph,12:02pm
AAA-123    40mph,12:05pm
CCC-456    25mph,12:15pm
```

6.1.2 Mapping 数据列表

MapReduce 程序的第一步叫做 Mapping，在这一步会有一些数据元素作为 map 函数的输入数据，每次一个，Mapper 会把每次 Map 得到的结果单独传到一个输出数据元素里。Mapper 模型如图 6-1 所示。

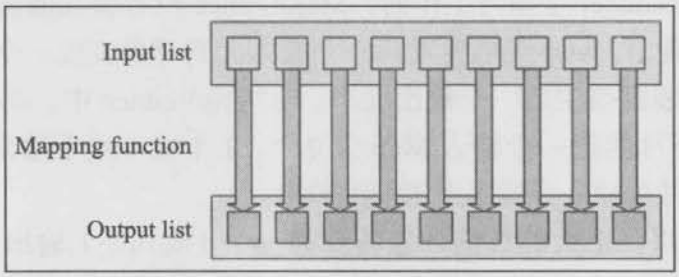


图 6-1 Mapper 模型

Mapping 通过对输入数据列表中的每一个元素使用一个函数创建了一个新的输出数据列表。这里举一个 Map 功能的例子：假设有一个函数 toUpper(str)，用来返回输入字符串的大写版本。可以在 Map 中使用这个函数把常规字符串列表转换成大写的字符串列表。注意，这里并没有改变输入字符串：返回一个新的字符串，它是新输出列表的组成部分之一。

6.1.3 Reducing 数据列表

Reducing 可以把数据聚集在一起。reduce 函数接收来自输入列表的迭代器，它会把这些数据聚合在一起，然后返回一个输出值。Reducer 模型如图 6-2 所示。

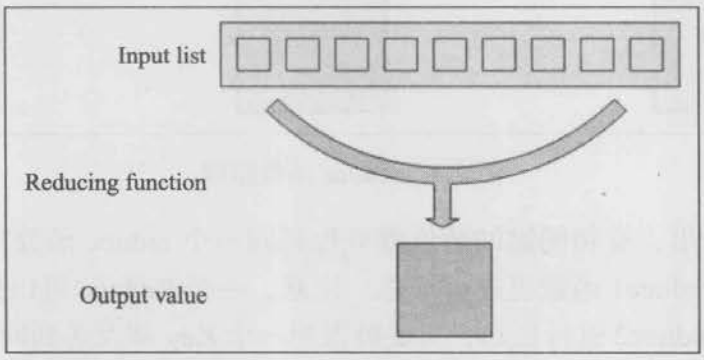


图 6-2 Reducer 模型

通过列表迭代器对输入数据进行 Reducing 操作输出聚合结果。Reducing 一般用来生成汇总数据，把大规模的数据转变成更小的总结数据。比如，“+”可以用作一个 reduce 函数，

返回输入数据列表的值的总和。

Hadoop 的 MapReduce 框架使用了上面的这些概念并用来处理大规模的数据信息。MapReduce 模型有两个组件：一个是抽象类 Mapper，另一个是抽象类 Reducer。上面描述的 Mapper 和 Reducer 术语在 Hadoop 中有了更细微的扩展，但基本的概念是相同的。

map 和 reduce 函数不接收数值（如 int、String 类型的值），而接收 Key-Value 对。这些函数的每一个输出都是一个键和一个值，它们将被送到数据流的下一个列表。

6.1.4 Mapper 和 Reducer 如何工作

对于 Mapper 和 Reducer 是如何工作的，MapReduce 没有像其他语言那样严格。在函数式 Map 和 Reduce 设置中，map 函数针对每一个输入元素都要生成一个输出元素，reduce 函数针对每一个输入列表都要生成一个输出元素。在 MapReduce 中，每一个阶段都可以生成任意的数值；Mapper 可能把一个输入 Map 为 0 个、1 个或 100 个输出。Reducer 可能计算超过一个的输入列表并生成一个或多个不同的输出。

reduce 函数的作用是把大的数值列表转变为一个（或几个）输出数值。在 MapReduce 中，所有的输出值一般不会被 Reduce 在一起，除非设置 Reduce 的个数为 1。但是有相同键的所有数值会被一起送到一个 reduce 函数里处理。作用在有着不同键关联的数值列表上的 Reduce 操作之间是独立执行的。reduce 函数原理如图 6-3 所示。

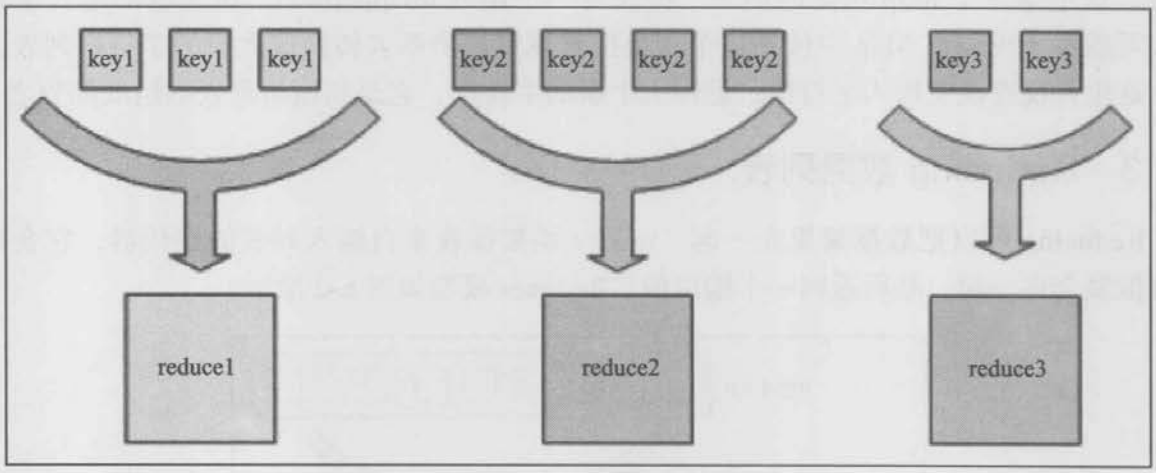


图 6-3 reduce 函数原理

从图 6-3 可以看出，有相同键的数值都被传到同一个 reduce 函数里进行汇总，如键为 key1 的值被发送到 reduce1 函数里进行汇总。注意，一个 Reduce 可以处理多个 Key，例如 key1 也可以发送到 reduce2 进行汇总，只要确保同一个 Key 被发送到同一个 Reduce。

为什么相同的 Key 要发送到同一个 reduce 呢？

假设统计“hello”出现的次数，那么 Key 的值就为“hello”，有两个 Reduce，分别为 reduce1 和 reduce2。如果 Key 为“hello”发送到 reduce1 和 reduce2 处理汇总，reduce1 汇总的结果为 <"hello"，3>，reduce2 汇总的结果为 <"hello",7>, 这样写入 HDFS 结果文件

Key 为“hello”的就会出现两次，分别为：

```
hello  3
hello  7
```

这不是我们想要的结果，我们想要的结果应该是“hello 10”，所以相同的 Key 必须发送同一个 reduce 函数进行处理。

6.1.5 应用实例：词频统计

写一个简单的 MapReduce 程序用来统计不同的词在一个文件集中出现的次数。比如，我们有以下两个文件。

- 文件 test1.txt；内容为 hadoop is easy for everybody 。
- 文件 test2.txt；内容为 easyhadoop is open source for everybody。

我们期望输出结果为：

```
hadoop      1
is          2
for         2
everybody   2
easy        1
easyhadoop  1
open        1
source      1
```

写一个 MapReduce 程序来计算得到这个输出。MapReduce 模型的伪代码实现如下所示。

```
mapper (filename, file-contents):
   for each word in file-contents:
     emit (word, 1)
reducer (word, values):
   sum = 0
   for each value in values:
     sum = sum + value
   emit (word, sum)
```

上面为 MapReduce 词频统计伪代码，若干个 Mapper 函数的实例会被创建在集群的不同机器上，每个实例接收一个不同的输入文件（这里假设有很多个文件）。 map 函数输出的（word，1）键值对会被转到 reduce 函数的输入。若干个 reduce 方法实例也会在不同机子上被实例化。每个 reduce 函数负责处理不同词的数值列表，数值列表中的值都是 1 ；reduce 函数把这些“1”值总和到一个关联某个词的最终计数里。然后 reduce 函数生成最终的（word，count）输出，并把它写到一个输出文件里。

6.2 MapReduce 框架结构

MapReduce 是一个用于大规模数据处理的分布式计算模型，最初是由 Google 工程师设计并实现的，Google 已经将完整的 MapReduce 论文公开发布了。其中的定义是，

MapReduce 是一个编程模型（Programming Model），是一个用于处理和生成大规模数据集（processing and generating large data sets）的相关的实现。用户定义一个 map 函数来处理一个 Key-Value 对以生成一批中间的 Key-Value 对，再定义一个 reduce 函数将所有这些中间的有相同 Key 的 Value 合并起来。很多现实世界中的任务都可用这个模型来表达。

6.2.1 MapReduce 模型

Hadoop MapReduce 模型主要有 Mapper 和 Reducer 两个抽象类。Mapper 端主要负责对数据的分析处理，最终转化为 Key-Value 的数据结构；Reducer 端主要是获取 Mapper 出来的结果，对结果进行统计。MapReduce 模型如图 6-4 所示。

Hadoop MapReduce 实现存储的均衡，但未实现计算的均衡，这是它天生的缺陷。目前，通常采用规避的办法来解决这个问题，由编写 MapReduce 代码的程序员来保证，这也使得 MapReduce 的调度器实现得不够灵活，多调度策略无法使用，而只能分集群多调度器并存。

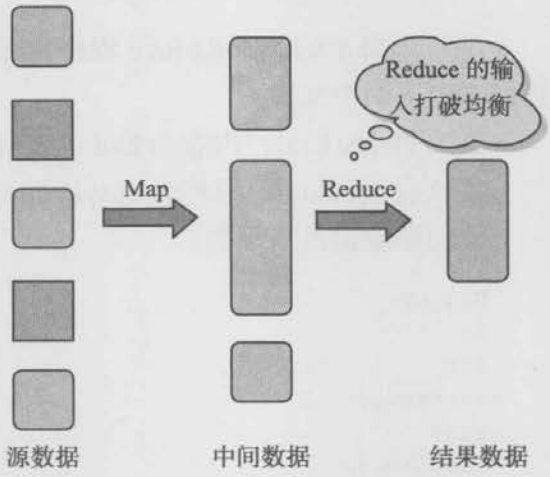


图 6-4 MapReduce 模型

6.2.2 MapReduce 框架组成

图 6-5 描述了 MapReduce 框架中主要组成和它们之间的关系。

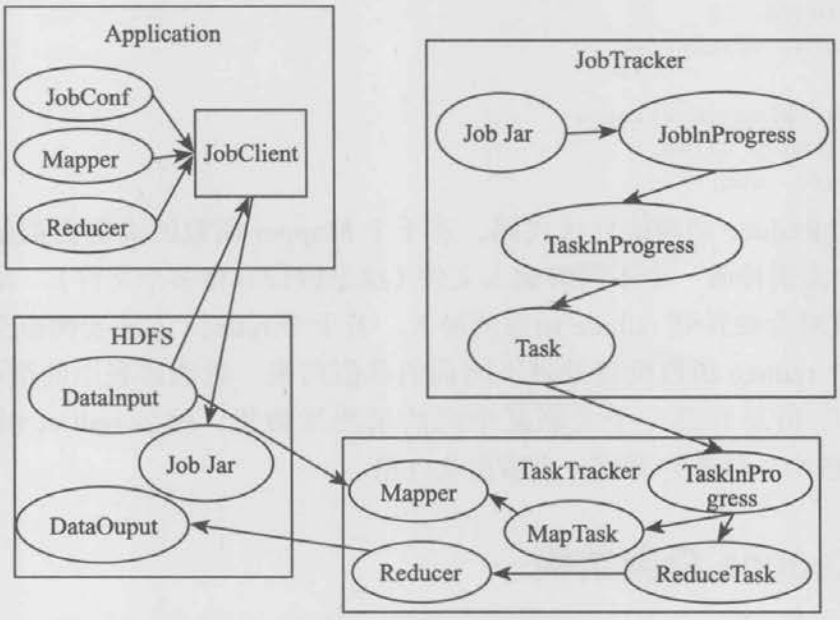


图 6-5 MapReduce 的架构结构

注意 TaskTracker 都需要运行在 HDFS 的 DataNode 上。

1. Mapper 和 Reducer

运行于 Hadoop 的 MapReduce 应用程序最基本的组成部分包括：一个 Mapper 抽象类和一个 Reducer 抽象类，以及创建 JobConf 的执行程序，在一些应用中还可以包括 Combiner 类，Combiner 实际也是 Reducer 的实现。

2. JobTracker

JobTracker 是一个 master 服务，软件启动之后 JobTracker 接收 Job，负责调度 Job 的每一个子任务 Task 运行于 TaskTracker 上，并监控它们，如果发现有失败的 Task 就重新运行它。一般情况应该把 JobTracker 部署在单独的机器上。

3. TaskTracker

TaskTracker 是运行在多个节点上的 slaver 服务。TaskTracker 主动与 JobTracker 通信（与 DataNode 和 NameNode 相似，通过心跳来实现）接收作业，并负责直接执行每一个任务。

4. JobClient

每一个 Job 都会在用户端通过 JobClient 类将应用程序以及配置参数 Configuration 打包成 JAR 文件存储在 HDFS，并把路径提交到 JobTracker 的 master 服务，然后由 master 创建每一个 Task（即 MapTask 和 ReduceTask）将它们分发到各个 TaskTracker 服务中去执行。

5. JobInProgress

JobClient 提交 Job 后，JobTracker 会创建一个 JobInProgress 来跟踪和调度这个 Job，并把它添加到 Job 队列里。JobInProgress 会根据提交的任务 JAR 中定义的输入数据集（已分解成 FileSplit）创建对应的一批 TaskInProgress 用于监控和调度 MapTask，同时创建指定数目的 TaskInProgress 用于监控和调度 ReduceTask，默认为 1 个 ReduceTask。

6. TaskInProgress

JobTracker 启动任务时通过每一个 TaskInProgress 来运行 Task，这时会把 Task 对象（即 MapTask 和 ReduceTask）序列化写入相应的 TaskTracker 服务中，TaskTracker 收到后会创建对应的 TaskInProgress（此 TaskInProgress 实现非 JobTracker 中使用的 TaskInProgress，作用类似）用于监控和调度该 Task。启动具体的 Task 进程通过 TaskInProgress 管理，通过 TaskRunner 对象来运行。TaskRunner 会自动装载任务 JAR 文件并设置好环境变量后，启动一个独立的 Java Child 进程来执行 Task，即 MapTask 或者 ReduceTask，但它们不一定运行在同一个 TaskTracker 中。

7. MapTask 和 ReduceTask

一个完整的 Job 会自动依次执行 Mapper、Combiner（在 JobConf 指定 Combiner 时执行）和 Reducer，其中 Mapper 和 Combiner 是由 MapTask 调用执行，Reducer 则由 ReduceTask

调用，Combiner 实际也是 Reducer 接口类的实现。Mapper 会根据 Job JAR 中定义的输入数据集按 <key1,value1> 对读入，处理完成生成临时的 <key2,value2> 对，如果定义了 Combiner，MapTask 会在 Mapper 完成调用该 Combiner 将相同 Key 的值做合并处理，以减少输出结果集。MapTask 的任务全完成即交给 ReduceTask 进程调用 Reducer 处理，生成最终结果 <key3,value3> 对。这个过程在下一部分详细介绍。

6.3 MapReduce 运行原理

Hadoop 中的 MapReduce 是一个使用简单的软件框架，基于它写出来的应用程序能够运行在由上千个商用机器组成的大型集群上，并以一种可靠容错式并行处理 TB 级别的数据集。

一个 MapReduce 作业（Job）通常会把输入的数据集切分为若干独立的数据块，由 Map 任务以完全并行的方式处理它们。框架会对 map 函数的输出先进行排序，然后把结果输入给 Reduce 任务。通常作业的输入和输出都会被存储在文件系统中。整个框架负责任务的调度和监控，以及重新执行已经失败的任务。

通常，MapReduce 框架和分布式文件系统是运行在一组相同的节点上的，也就是说，计算节点和存储节点通常在一起。这种配置允许框架在那些已经存好数据的节点上高效地调度任务，这可以使整个集群的网络带宽被非常高效地利用。

6.3.1 作业的提交

JobClient 的 runJob() 方法用于新建 JobClient 实例并调用其 submitJob() 方法，这种便捷方式提交作业后，runJob() 每秒轮询作业的进度，如果发现自上次上报后的信息有改动，便把进度报告输出到控制台。作业完成后，如果成功，就显示作业计数器，如果失败，导致作业失败的错误也会被输出到控制台。

Hadoop 运行 MapReduce 作业的工作原理如图 6-6 所示。

JobClient 的 submitJob() 方法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 JobTracker 的 getNewJobId() 获取）。

2）检查作业的输出说明。例如，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，并将错误返回给 MapReduce 程序。

3）计算作业的输出划分。如果划分无法计算，例如，因为输入路径不存在，作业就不会被提交，并将错误返回给 MapReduce 程序。

4）将运行作业所需要的资源（包括作业的 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 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（图 6-6 中步骤 3）。

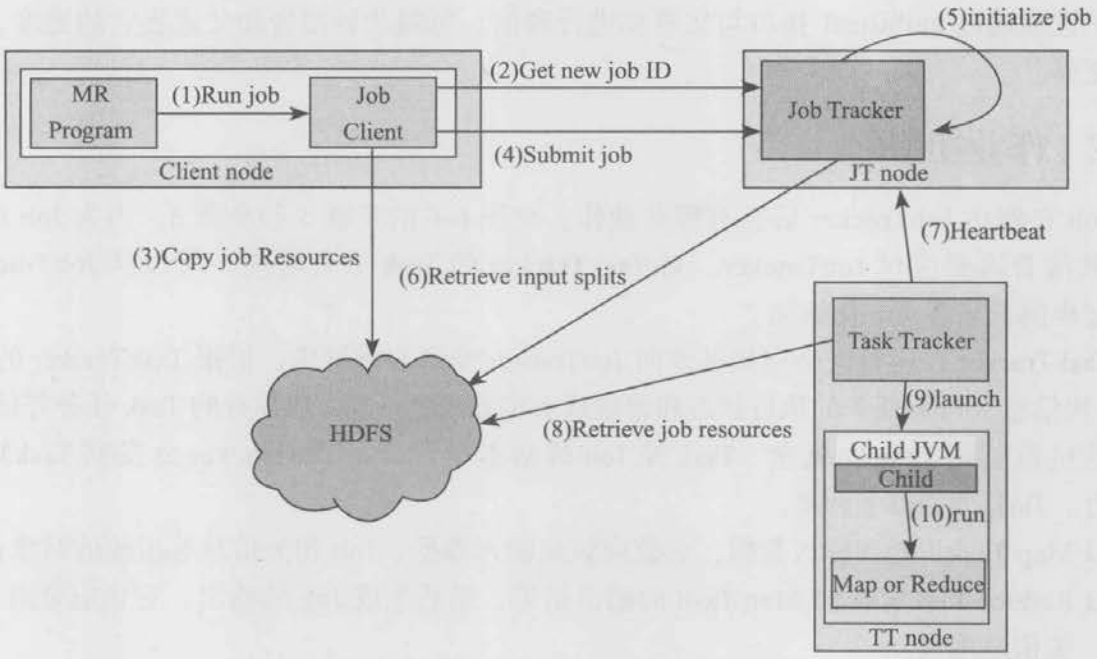


图 6-6 MapReduce 作业的工作原理

5）通过调用 JobTracker 的 submitJob() 方法，告诉 JobTracker 作业准备执行（图 6-6 中步骤 4）。

6）JobTracker 接收到对其 submitJob() 方法调用后，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对象，它封装任务和记录信息，以便跟踪任务的状态和进程（图 6-6 中步骤 5）。

7）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 已经计算好的输入划分信息（图 6-6 中步骤 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() 方法设定的，然后调度器便创建指定个数的 Reduce 来运行任务。任务在此时指定 ID。

8）TaskTraker 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，是否存活，同时充当两者之间的消息通道。作业心跳方法调用指 TaskTracker 会向 JobTracker 汇报当前的状态，如果正常，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（图 6-6 中步骤 7）。

现在，TaskTracker 已经被分配了任务，下面是运行任务步骤。

1）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本地磁盘。

2）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。

3）新建一个 TaskRunner 实例来运行任务。

TaskRunner 启动一个新的 Java 虚拟机（图 6-6 中步骤 9）来运行每个任务（图 6-6 中步骤 10），使得用户执行任务启动 map 和 reduce 函数的任何缺陷都不会影响 TaskTracker（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。

子进程通过 umbilical 接口与父进程进行通信。每隔几秒便告知父进程它的进度，直到任务完成。

6.3.2 作业初始化

Job 在到达 JobTracker 后会有哪些动作，见图 6-6 的步骤 5 和步骤 6。因为 Job 在初始化后紧接着需要应付 JobTracker，对 TaskTracker 的 Task 分发响应，所以从 JobTracker 的分发过程倒着来看 Job 初始化。

TaskTracker 在运行时会周期性地向 JobTracker 发送心跳请求，汇报 TaskTracker 的状态、Block 块信息、Task 任务的执行状态和希望从 JobTracker 得到可以执行的 Task 任务等信息。

这里需明确 Task 的概念。Task 是 Job 的基本单元，由 JobTracker 分发到 TaskTracker 来执行。Task 分为以下两类。

- Map Task：处理输入数据，它就应该是输入数据、Job 相关信息等组成的对象；
- Reduce Task：汇总 Map Task 的输出结果，最后生成 Job 的输出，它也应是由 Job 相关信息组成。

Task 在 TaskTracker 上执行，所以 Task 需要的信息必须与 JobTracker 的状态信息有关，以便 Task 执行时需要。

在前一节 Job 提交过程中说过，Job 将所有输入数据组装成了逻辑分片，这些逻辑分片只是 HDFS 上物理数据 Block 的索引及存储信息。Map Task 依赖于这些信息来决定将 Task 分发到哪些 TaskTracker 上。JobTracker 可以取到 Job 相关的 metadata 信息，然后由这些信息来决定如何分发 Task。因此，上一节的这些分片的相关信息就存放在特定的目录下，JobTracker 通过 JobId 可以访问到。

Reduce Task 不管在哪个 TaskTracker 上执行，都得从其他那些执行 Map Task 的 TaskTracker 上拉取数据，所以对它的分发 JobTracker 不需要准备什么，只要在合适的时候放到某台 TaskTracker 上执行即可。JobTracker 主要还是关注 Map Task 的准备工作（Reduce Task 并不是从所有 Map Task 拉取临时数据。如果有多个 Reduce Task，每个 Reduce Task 只拉取一部分 Map Task 的临时数据。关于 Partition 相关细节以后会介绍）。

这里介绍 JobTracker 分发 Task 过程中比较重要的一步：本地数据（data locality）级别。

Map Task 的执行效率依赖于读取输入数据的效率。输入数据越靠近执行 Task 的 TaskTracker，Map Task 执行得就越快。根据数据所处的位置与 TaskTracker 的距离，有三种本地数据级别，如表 6-1 所示。

表 6-1 本地数据级别

表 识	级 别	功能描述
0	node-local	输入分片就在 TaskTracker 本地
1	rack-local	输入分片在 TaskTracker 所在 Rack 的其他节点上
2	off-switch	输入分片在其他的 Rack 内

JobTracker 在 Task 分发时应充分考虑本地数据级别。分发策略对 Job 执行效率的影响很大程度是如何优化 Map Task 的本地数据。具体流程如图 6-7 所示。

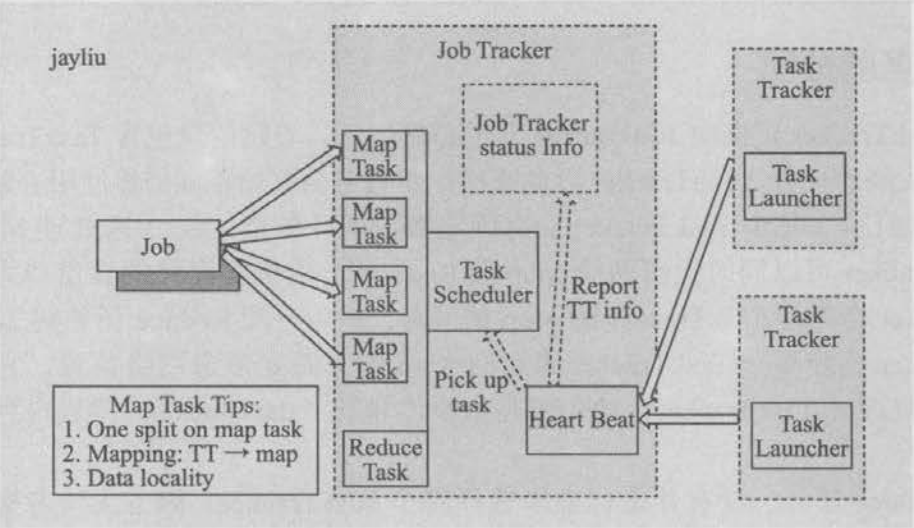


图 6-7 Map Task 的本地数据优化流程图

JobTracker 可以从 Job 的 metadata 中得到并维护这样一种映射关系：

job split → HDFS Block && slave node

这种映射关系就是生成 Map Task 的基础。有多少个 Split，就会有多少个 Map Task。如上所述，响应心跳而选择 Map Task 的处理步骤如下所示。

1）根据 TaskTracker 的机器，查看 JobTracker 中是否存在一个 Map Task，它关联的 Block（假设一个 Block 划分为一个 Split）存储在 TaskTracker 的本地磁盘上，那么就优先执行这个 Map Task。

2）如果没有 1 可选的 Map Task，那么查看是否有 Map 关联的 Block 在 TaskTracker 所在的 Rack 内。

3）如果上面两步都没有选到某个 Map Task，那么就根据情况看是否执行跨 Rack 的 Task 或其他推测式执行 Task。

初始化的基本工作就很少一些，主要的选择逻辑还是 Task Scheduler 应该做的事情。既然提到 Scheduler，还得说 MapReduce 默认的 JobQueue Scheduler 在理论上有个严重的问题。如果它没有从 1 或 2 中选取到 Map Task，它就会执行某个跨 Rack 的 Map Task。理想情况下应该尽量避免这样做，当前的 TaskTracker 本地或 Rack 内不存在 Task 所需的 Split，但其他 Rack 内的 TaskTracker 上肯定会存在那个 Split，那个有 Split 存在的 TaskTracker 之后也会从 JobTracker 取 Task，把这种 Map Task 放到有数据的 TaskTracker 上做肯定比跨 Rack 取数据快。所以如果可能的话，尽量按照实际情况选取合适的 Scheduler（以上是理论，也是因为大部分的执行环境都没有与 Rack 有关物理拓扑结构）。

Scheduler 还得注意的一点是，当用户开启 Task 推测式执行，推测式执行就会发生在 JobTracker 意识到某个 Task 执行效率低的时候，尽量要让推测式 Task 是 node local 级别的。

如果推测式 Task 要跨 Rack 取数据，比原来的 Task 执行还慢，就没有现实意义了。

Job 初始化过程主要是在 JobTracker 建立一个 slave node 对 Task 的映射模型，其他都是附属工作。我会把 Job 生命周期的重点放在 Task 执行的过程，也就是那个“奇迹发生的地方”。

6.3.3 任务的分配

每个 TaskTracker 定期向 JobTracker 发送心跳信息，心跳信息包含 TaskTracker 的状态，是否可以接收新的任务。JobTracker 以此来决定将任务分配给谁（仍然使用心跳的返回值与 TaskTracker 通信）。每个 TaskTracker 会有固定数量的任务槽（slot）来处理 Map 和 Reduce（表示 TaskTracker 可以同时运行两个 Map 和 Reduce），由机器内核的数量和内存大小来决定。JobTracker 会先将 TaskTracker 的 Map 槽填满，然后分配 Reduce 任务到 TaskTracker。

JobTracker 选择哪个 TaskTracker 来运行 Map 任务需要考虑网络位置，它会选择一个离输入分片较近的 TaskTracker，优先级是数据本地化（data-local），然后再到机架本地化（rack-local）。

对于 Reduce 任务，没有什么标准来选择哪个 TaskTracker，因为无法考虑数据的本地化。Map 的输出始终是需要经过整理（切分、排序和合并）后，Reduce 通过开启多线程去抓取 Map 的输出结果。可能多个 Map 的输出会切分出一部分送给一个 Reduce，所以 Reduce 任务没有必要选择和 Map 相同或最近的机器上。

6.3.4 任务的执行

TaskTracker 分配到一个任务后，首先从 HDFS 中把作业的 JAR 文件复制到 Task Tracker 所在的本地文件系统（JAR 本地化用来启动 JVM）。同时将应用程序所需要的全部文件从分布式缓存复制到本地磁盘。接下来 TaskTracker 为任务新建一个本地工作目录 work，并把 JAR 文件的内容解压到这个文件夹下。

TaskTracker 新建一个 TaskRunner 实例来运行该任务。TaskRunner 启动一个新的 JVM 来运行每个任务，以便客户的 MapReduce 不会影响 TaskTracker 守护进程。但在不同任务之间重用 JVM 还是可能的。子进程通过 umbilical 接口与父进程进行通信。任务的子进程每隔几秒便告知父进程的进度，直到任务完成。

6.3.5 进度和状态的更新

一个作业和每个任务都有一个状态信息，包括作业或任务的运行状态（running, successful, failed）、Map 和 Reduce 的进度、计数器值、状态消息和描述等。

这些信息通过一定的时间间隔由 Child JVM → TaskTracker → JobTracker 汇聚。JobTracker 将产生一个表明所有运行作业及其任务状态的全局视图。可以通过 Web UI 查看。同时 JobClient 通过每秒查询 JobTracker 来获得最新状态，输出到控制台上。

Streaming 和 Pipes 是用来运行其他语言编写的 Mapper 和 Reducer。Streaming 任务特指任务使用标准输入 / 输出 Steaming 与进程通信，可以是任何语言编写的，如 Python、Derby、Shell 等。Pipes 特指 C++ 语言编写的任务，其通过 Socket 来通信。

执行的 Streaming 和 Pipes 与 TaskTracker 及子进程的关系如图 6-8 所示。

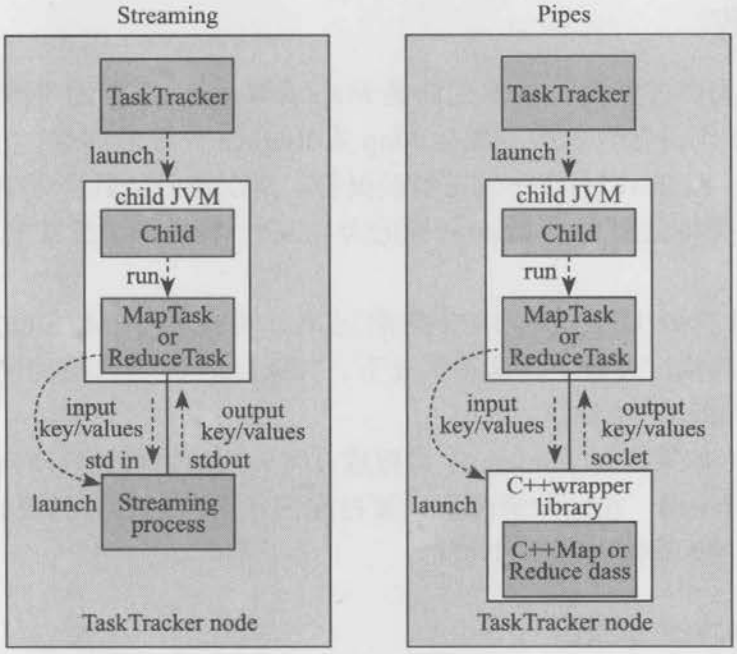


图 6-8 执行的 Streaming 和 Pipes 与 TaskTracker 及子进程的关系

6.3.6 MapReduce 的进度组成

进度并非总是可以预测的，但是无论如何它能告诉 Hadoop 有几个任务正在运行。例如，写输出记录的任务也可以表示进度，尽管它不能用总的需要写的百分比来表示，因为即使通过任务产生输出，也无法知道后面的情况。

任务执行的进度条对于 MapReduce 来说很重要，因为这意味着 Hadoop 不会让正在运行的任务失败。MapReduce 构成的所有操作如下。

- 读入一条输入记录（在 Mapper 或 Reducer 中）；
- 写入一条输入记录（在 Mapper 或 Reducer 中）；
- 在一个 Context 中设置状态描述；
- 增加计数器 Counter；
- 调用 progress() 方法。

6.3.7 任务完成

当 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成“成功”。然后在 JobClient 查询状态时，便知道任务已经完成。于是 JobTracker 打印一条消息到控制台上，最后调用 jobRun() 方法返回。

6.4 MapReduce 容错

实际情况下，用户代码存在软件错误，进程会崩溃，机器会产生故障。使用 Hadoop 最

大的好处之一就是能够处理此类故障并且成功运行完任务。

6.4.1 任务失败

首先考虑任务失败的情况，最常见的是 Map 或 Reduce 任务的失败，即由于用户写的 Map 代码和 Reduce 代码导致失败。发生 Map 或 Reduce 失败的时候，子任务 JVM 进程会在退出之前向其上一级 TaskTracker 发送错误报告。错误报告最后会记录在用户的错误日志里面，TaskTracker 会将此次 task attempt 标记为 failed，释放一个任务槽（slot）用来运行另外一个任务。

另一个错误情况就是子进程 JVM 突然退出，可能由 JVM 的 bug 导致，从而导致 MapReduce 用户代码执行失败。在这种情况下，TaskTracker 会监控到进程以便退出，并将此次尝试标记为“failed”失败。

还有一种情况，如果超时（timeout）设置成 0 将关闭超时检测，所以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所占用的任务槽，并随着时间的推移会降低整个集群的性能。

6.4.2 TaskTracker 失败

作业运行期间，TaskTracker 会通过心跳机制不断与系统 JobTracker 通信。如果某个 TaskTracker 运行缓慢或者失败，出现故障，TaskTracker 就会停止或者很少向 JobTracker 发送心跳，JobTracker 会注意到此 TaskTracker 发送心跳的情况，从而将此 TaskTracker 从等待任务调度的 TaskTracker 池中移除，JobTracker 会安排此 TaskTracker 上一个成功运行的 Map 任务返回。

下面介绍两种 TaskTracker 失败情况。

1）如果它们属于未完成的作业，Reduce 阶段无法获取本地 Map 输出的文件结果，任务都需要重新调度和执行，只要是 Map 阶段失败必然是重新执行这个任务。

2）如果是 Reduce 阶段，自然是执行未完成的 Reduce 任务。因为 Reduce 只要执行完了就会把输出写到 HDFS 上。

6.4.3 JobTracker 失败

JobTracker 失败应该说是最严重的失败方式，而且在 Hadoop 中存在单点故障的情况下是相当严重的，因为在这种情况下作业最终失败。尽管这种故障的概率极小，但还是应该避免，可以通过启动多个 JobTracker，在这种情况只运行一个主 JobTracker。通过一种机制来确定哪个是主 JobTracker。可以使用 ZooKeeper 作为 JobTracker 的协调机制来决定哪一个是主 JobTracker。

6.4.4 子任务失败

一个 MapReduce 任务里面包含成千上万的 Map 任务和 Reduce 任务，避免不了由于各

种原因而失败，下面总结 Map 任务和 Reduce 任务失败的三种情况。

1）当 Map 或者 Reduce 子任务中的代码抛出异常，JVM 进程会在退出之前向 TaskTracker 进程发送错误报告，TaskTracker 会将此（任务尝试）task attempt 标记为 failed 状态，释放一个槽以便运行另外一个任务。

2）对于流任务，如果流进程以非零退出代码退出运行，会标记为 failed。

3）子 JVM 突然退出，即 JVM 错误，这时 TaskTracker 会注意到进程已经退出，标记为 failed。

TaskTracker 将子任务标记为失败后会将自身计数器减 1，为了 JobTracker 申请新的任务，也是通过心跳告知 JobTracker 本地的一个任务尝试失败。JobTracker 接到任务失败的通知后，会将其重新加入到调度队列重新分配给其他的 TaskTracker 执行（避免将失败的任务分配给执行失败的 TaskTracker）。但是这个尝试也是有次数限制的，默认情况下，任务尝试 4 次后仍然没有完成，就不会再重试（JobTracker 会将其标记为 killed），此时整个作业就执行失败了。

6.4.5 任务失败反复次数的处理方法

当 Map Task 执行失败后会重试，超过重试次数（由 mapred.map.max.attempts 指定，默认为 4）整个 Job 会失败。Hadoop 提供配置参数 mapred.max.map.failures.percent 解决这个问题。

如果一个 Job 有 200 个 Map Task，该参数设置为 5，则单个 Job 最多允许 10 个 Map Task（200 ×5% =10）失败，这是我们想要的。

把下面配置放到 mapred-site.xml 文件中，重启 JobTracker（hadoop-daemon.sh stop/start jobtracker）。

```
<property>
  <name>mapred.max.map.failures.percent</name>
  <value>5</value>
</property>
```

Reduce Task 也有类似配置 mapred.max.reduce.failures.percent 属性，具体的含义和配置与 Map Task 一样。

6.5 Shuffle 阶段和 Sort 阶段

接下来，剖析 MapReduce 的核心过程——Shuffle 和 Sort。Shuffle 是指从 Map 的输出开始，包括系统执行排序以及传送 Map 输出到 Reduce 作为输入的过程。Sort 阶段就是对 Map 端输出的 Key 进行排序。在这里我们将探究 Shuffle 和 Sort 是如何工作的。

reduce 函数的输入是已分类的 map 函数的输出。在 Shuffle 阶段，框架通过 HTTP 取出所有 Map 输出的结果。由于不同的 Map 可能输出相同的 Key，在这个阶段，框架将 Reduce 的输入通过 Key 分成组。相同的 Key 发送到同一个 Reduce 端处理。Sort 阶段就是对 Map 端输出的 Key 进行排序。

Shuffle 阶段可以分为 Map 端 Shuffle 和 Reduce 端 Shuffle，接下来我们分别介绍这两个阶段 Shuffle 的功能和原理。

6.5.1 Map 端的 Shuffle

MapReduce 框架的核心步骤主要分两部分：Map 和 Reduce。当向 MapReduce 框架提交一个计算作业时，它会首先把计算作业拆分成若干个 Map 任务，然后分配到不同的节点上去执行，每一个 Map 任务处理输入数据中的一部分，当 Map 任务完成后，它会生成一些中间文件，这些中间文件将会作为 Reduce 任务的输入数据。Reduce 任务的主要目标就是把前面若干个 Map 的输出汇总到一起并输出。从高层次抽象来看，MapReduce 的数据流图如图 6-9 所示。

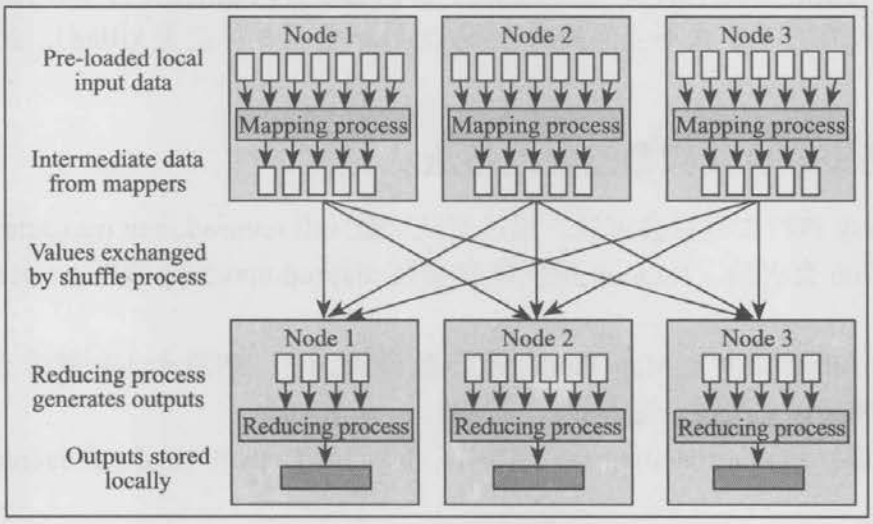


图 6-9 MapReduce 数据流图

首先从 Map 端开始分析。当 Map 开始产生输出时，它并不是简单地把数据写到磁盘，因为频繁的磁盘操作会导致性能严重下降。它的处理过程更复杂，数据首先写到内存中的一个缓冲区，并做一些预排序，以提升效率，如图 6-10 所示。

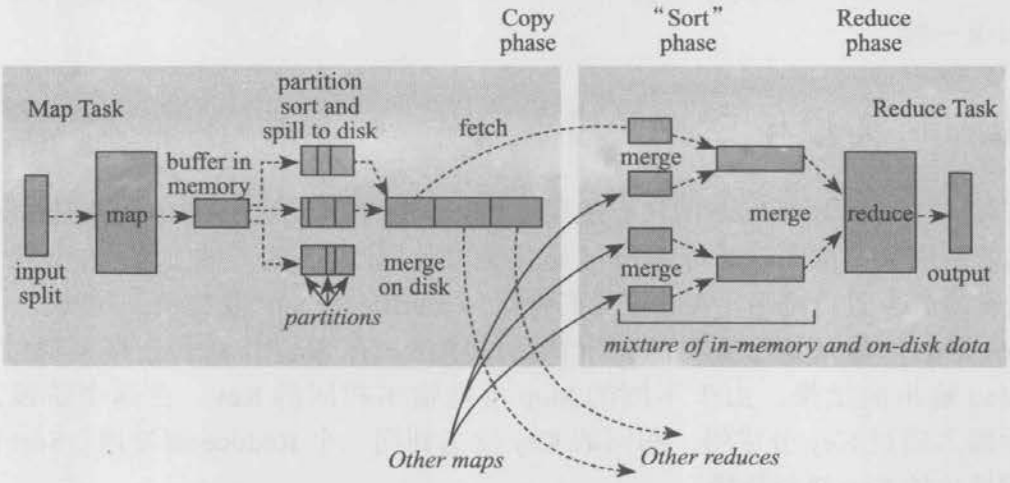


图 6-10 MapReduce 中的 Shuffle 和 Sort 过程

每个 Map 任务都有一个用来写入输出数据的循环内存缓冲区。这个缓冲区默认大小是 100MB，可以通过 io.sort.mb 属性来设置具体大小。当缓冲区中的数据量达到一个特定阈值时（io.sort.mb * io.sort.spill.percent，其中 io.sort.spill.percent 默认是 0.80），系统将会启动一个后台线程把缓冲区中的内容 spill 到磁盘。

在 spill 过程中，Map 的输出将会继续写入到缓冲区，但如果缓冲区已满，Map 就会被阻塞直到 spill 完成。spill 线程在把缓冲区的数据写到磁盘前，会对它进行一个二次快速排序，首先根据数据所属的 Partition 排序，然后每个 Partition 中再按 Key 排序。输出包括一个索引文件和数据文件。如果设定了 Combiner，将在排序输出的基础上运行。

Combiner 就是一个 Mini Reducer，它在执行 Map 任务的节点本身运行，先对 Map 的输出做一次简单 Reduce，使得 Map 的输出更紧凑，更少的数据会被写入磁盘和传送到 Reduce 端。spill 文件保存在由 mapred.local.dir 指定的目录中，Map 任务结束后删除。

每当内存中的数据达到 spill 阈值的时候，都会产生一个新的 spill 文件，所以在 Map 任务写完它的最后一个输出记录时，可能会有多个 spill 文件。在 Map 任务完成前，所有的 spill 文件将会被归并排序为一个索引文件和数据文件，如图 6-11 所示。这是一个多路归并过程，最大归并路数由 io.sort.factor 控制（默认是 10）。如果设定了 Combiner，并且 spill 文件的数量至少是 3（由 min.num.spills.for.combine 属性控制），Combiner 将在输出文件被写入磁盘前运行以压缩数据。

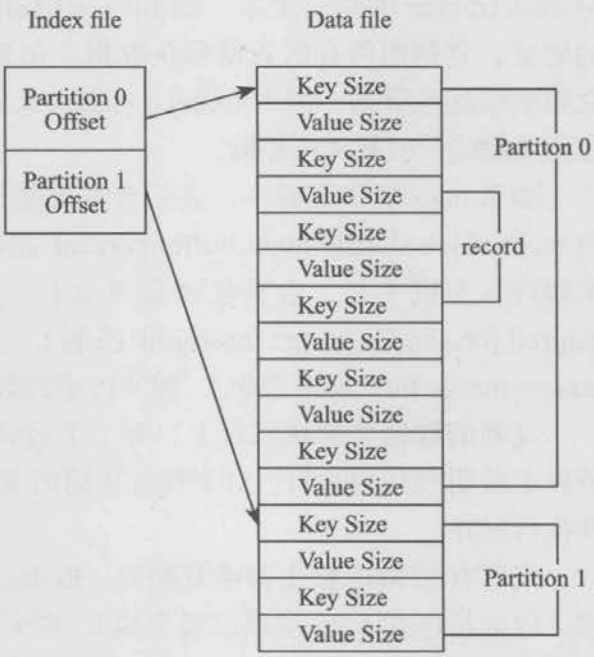


图 6-11 Map 任务的最终输出文件

对写入到磁盘的数据进行压缩（这种压缩同 Combiner 的压缩不一样）通常是一个很好的方法，因为这样做使得数据写入磁盘的速度更快，节省磁盘空间，并减少需要传送到 Reduce 的数据量。默认输出是不被压缩的，但可以很简单的设置 mapred.compress.map.output 为 true 启用该功能。压缩所使用的库由 mapred.map.output.compression.codec 来设定。

当 spill 文件归并完毕后，Map 将删除所有的临时 spill 文件，并告知 TaskTracker 任务已完成。Reduce 端通过 HTTP 获取对应的数据。用来传输 partitions 数据的工作线程个数由 tasktracker.http.threads 控制，这个设定是针对每一个 TaskTracker 的，并不是单个 Map，默认值为 40，在运行大作业的大集群上可以增加以提升数据传输速率。

现在转到 Shuffle 的 Reduce 部分。Map 的输出文件放置在运行 Map 任务的 TaskTracker 的本地磁盘上（注意，Map 输出总是写到本地磁盘，但 Reduce 输出不是，一般是写到 HDFS），它是运行 Reduce 任务的 TaskTracker 所需要的输入数据。Reduce 任务的输入数

据分布在集群内的多个 Map 任务的输出中，Map 任务可能会在不同的时间内完成，只要有其中的一个 Map 任务完成，Reduce 任务就开始复制它的输出。这个阶段称为 Cope 阶段。Reduce 任务拥有多个 cope 线程，可以并行的获取 Map 输出。可以通过设定 mapred.reduce.parallel.copies 来改变线程数，默认是 5。

6.5.2 Reduce 端的 Shuffle

Reduce 端怎么知道从哪些 TaskTracker 中获取 Map 端输出呢？当 Map 任务完成之后，会通知它们的父 TaskTracker，告知状态更新，然后 TaskTracker 再转告 JobTracker。这些通知信息是通过心跳通信机制传输的。因此针对一个特定的作业，JobTracker 知道 Map 输出与 TaskTracker 的映射关系。Reduce 端中有一个线程会间歇地向 JobTracker 询问 Map 输出的地址，直到把所有的数据都获取到。在 Reduce 取走了 Map 输出之后，TaskTracker 不会立即删除这些数据，因为 Reduce 可能会失败。它们会在整个作业完成后，JobTracker 告知它们要删除的时候才去删除。

如果 map 输出足够小，它们会被复制到 Reduce TaskTracker 的内存中（缓冲区的大小由 mapred.job.shuffle.input.buffer.percent 控制，制定了用于此目的的堆内存的百分比）；如果缓冲区空间不足，会被复制到磁盘上。当内存中的缓冲区用量达到一定比例阈值（由 mapred.job.shuffle.merge.threshold 控制），或者达到了 Map 输出的阈值大小（由 mapred.inmem.merge.threshold 控制），缓冲区中的数据将会被归并然后 spill 到磁盘。

复制的数据叠加在磁盘上，有一个后台线程会将它们归并为更大的排序文件，这样做节省了后期归并的时间。对于经过压缩的 Map 输出，系统会自动把它们解压到内存方便对其执行归并。

当所有的 Map 输出都被复制后，Reduce 任务进入排序阶段（更恰当地说应该是归并阶段，因为排序在 Map 端就已经完成），这个阶段会对所有的 Map 输出进行归并排序，这个工作会重复多次才能完成。

假设有 50 个 Map 输出（可能有保存在内存中的），并且归并因子是 10（由 io.sort.factor 控制，就像 Map 端的 merge 一样），最终需要 5 次归并。每次归并会把 10 个文件归并为一个，最终生成 5 个中间文件。在这一步之后，系统不再把 5 个中间文件归并成一个，而是排序后直接“喂”给 reduce 函数，省去向磁盘写数据这一步。最终归并的数据可以是混合数据，既有内存上的也有磁盘上的。由于归并的目的是归并最少的文件数目，使得在最后一次归并时总文件个数达到归并因子的数目，所以每次操作所涉及的文件个数在实际中会更微妙些。如果有 40 个文件，并不是每次都归并 10 个最终得到 4 个文件，相反第一次只归并 4 个文件，然后再实现 3 次归并，每次 10 个，最终得到 4 个归并好的文件和 6 个未归并的文件。要注意，这种做法并没有改变归并的次数，只是最小化写入磁盘的数据优化措施，因为最后一次归并的数据总是直接送到 reduce 函数那里。

在 Reduce 阶段，reduce 函数会作用在排序输出的每一个 Key 上。这个阶段的输出被直接写到输出文件系统，一般是 HDFS。在 HDFS 中，因为 TaskTracker 节点也运行着一个

DataNode 进程，所以第一个块备份会直接写到本地磁盘。

Shuffle 在 Reduce 端的过程用三点来概括。当前 Reduce Copy 数据的前提是它要从 JobTracker 获得有哪些 Map Task 已执行结束。Reducer 真正运行之前，所有的时间都是在拉取数据，进行 merge 操作，且不断重复地在做。如前面的方式一样，下面分段描述 Reduce 端的 Shuffle 细节。

1）**Copy 过程**：简单地拉取数据。Reduce 进程启动一些数据 copy 线程（Fetcher），通过 HTTP 方式请求 Map Task 所在的 TaskTracker 获取 Map Task 的输出文件。因为 Map Task 早已结束，这些文件就归 TaskTracker 管理在本地磁盘中。

2）**Merge 阶段**：这里的 merge 如 Map 端的 merge 动作，只是数组中存放的是不同 Map 端复制来的数值。复制过来的数据会先放入内存缓冲区中，这里的缓冲区大小要比 Map 端的更灵活，它基于 JVM 的 heap size 设置，因为 Shuffle 阶段 Reduce 不运行，所以应该把绝大部分的内存都给 Shuffle 用。这里需要强调的是，Merge 有三种形式：内存到内存；内存到磁盘；磁盘到磁盘。

默认情况下第一种形式不启用。当内存中的数据量到达一定阈值，就启动内存到磁盘的 merge 操作。与 Map 端类似，这也是溢写的过程，这个过程中如果设置 Combiner，也是会启用的，然后在磁盘中生成了众多的溢写文件。第二种 Merge 方式一直在运行，直到没有 Map 端的数据时才结束。然后启动第三种磁盘到磁盘的 Merge 方式生成最终的文件。

3）**Reducer 的输入文件**：不断地 merge 操作后，最后会生成一个“最终文件”。为什么加引号？因为这个文件可能存在于磁盘上，也可能存在于内存中。对于我们来说，当然希望它存放于内存中，直接作为 Reduce 的输入，但默认情况下，这个文件是存放于磁盘中的。至于怎样才能让这个文件出现在内存中，之后的性能优化篇再详细说明。当 Reduce 的输入文件已定，整个 Shuffle 才最终结束。然后就是 Reduce 执行，把结果放到 HDFS 上。

6.5.3 Shuffle 过程参数调优

现在我们有较好的基础对 MapReduce 中的 Shuffle 过程进行调优，从而提高 MapReduce 性能，表 6-2 和表 6-3 总结了 Map 端和 Reduce 端的相关参数设置和默认值。这些设置以作业（Job）为单位，默认值适用于常规的作业。

表 6-2 Map 端的调优属性

参数名称	类 型	默 认 值	功能描述
io.sort.mb	int	100	排序 Map 输出是所使用内存缓冲区的大小，以 MB 为单位
io.sort.record.percent	float	0.05	用作存储 Map 输出记录边界的 io.sort.mb 的比例，剩余的空间用来存储 Map 输出记录本身
io.sort.spill.percent	Float	0.8	Map 输出内存缓冲和用来开始磁盘溢出写过程的记录边界索引，是两者使用比例的阈值

（续）

参数名称	类 型	默认值	功能描述
Io.sort.factor	boolean	false	排序文件时，一次最后合并的流数，这个属性也在 Reduce 端使用，将阈值增加到 100 是很常用的
mapred.compress.map.output			压缩 Map 的输出
mapred.map.out.compression.codec	class	org.apache.hadoop.io.compress.DefaultCodec	用于 Map 输出的压码编码解码
min.num.spills.for.combinar	int	3	运行 Combinar 所需要的最少溢出写文件数
tasktracker.http.threads	int	40	每个 TaskTracker 工作的线程数，用于将 Map 输出到 Reduce 这是集群范围的设置，不能有单个作业来设置

表 6-3 Reduce 端的调优属性

选项名称	类 型	默认值	功能描述
mapred.reduce.parallel.copies	int	5	每个 Reduce 并行下载 Map 结果的最大线程数
mapred.reduce.copy.backoff	int	300	Reduce 下载线程最大等待时间（in sec）
io.sort.factor	int	10	同上
mapred.job.shuffle.input.buffer.percent	float	0.7	用来缓存 Shuffle 数据的 reduce task heap 百分比
mapred.job.shuffle.merge.percent	float	0.66	缓存的内存中多少百分比后开始做 merge 操作
mapred.job.reduce.input.buffer.percent	float	0.0	sort 完成后 Reduce 计算阶段用来缓存数据的百分比

Shuffle 调优的总原则就是给 Shuffle 过程尽量多地提供内存空间，然后有一个平衡的问题，要确保 map 函数和 reduce 函数能够得到足够的内存来运行，这就是编写 map 函数和 reduce 函数时要尽量少用内存的原因。

运行 Map 任务和 Reduce 任务的 JVM，其内存大小由参数 mapred.child.java.opts 来设置，任务节点上面的内存大小应尽量大。

6.6 任务的执行

前面讲解了 MapReduce 作业的运行机制，结合整个作业的运行背景知道了 MapReduce 是如何执行作业的，本节介绍 MapReduce 用户对任务执行的更多控制。

6.6.1 推测执行

Hadoop 系统有一个问题，它把任务分派到很多个节点，其中很可能有一些慢的节点会限制剩下程序的执行速度。举个例子，如果有个节点内有一个比较慢的磁盘控制器，它读取输入数据的速度可能只有其他节点速度的 10%。所以当 99 个 Map 任务都已经完成了，系统仍在等待最后那个比较耗时的 Map 任务完成。

通过强迫任务独立运行于其他的任务，使得单个任务之间不会知道它们的输入数据来自哪里。任务相信 Hadoop 平台会派送合适的输入到它们那里。因此，对于相同的输入数据，我们可以并行多次处理以利用不同机器的负载能力。因为作业中大多数的任务都已经完成了，Hadoop 平台会在几个空闲的节点上调度执行剩余任务的复制，这个过程叫做推测性执行。当任务完成时，它会向 JobTracker 通告。任何一个首先完成的复制任务将成为权威复制，如果其他复制任务还在推测性执行中，Hadoop 会告诉 TaskTracker 去终止这些任务并丢弃它们的输出，接着 Reducer 会从首先完成的 Mapper 那里获取输入数据。

推测性执行默认是启用的，设置 JobConf 中的 mapred.map.tasks.speculative.execution 和 mapred.reduce.tasks.speculative.execution 为 false，用来禁止 Map 和 Reduce 的推测性执行。

6.6.2 任务 JVM 重用

mapred.job.reuse.jvm.num.tasks 的默认值为 1，即每一个 Task 都新启动一个 JVM 来运行任务，而当值为 -1 时，表示 JVM 可以无限制重用。

结合上面的分析，当 mapred.job.reuse.jvm.num.task 设置为 1 时，每执行一个 Map 或 Reduce 任务都会杀死 JvmRunner 和 spawn New Jvm 的进程，造成了系统资源的浪费，同时影响了 MapReduce 的性能。而且在值为 -1 时，TaskTracker 也是先判断当前节点是否有空闲的 slot 剩余，如果没有空闲的 slot 槽位，才会判断当前分配的 slot 槽位中的 JVM 是否已经将当前的 Task 任务运行完，如果 Task 已经运行完，才会复用当前 JVM（当前只针对同一个 Job 的 Task 才会进行 JVM 的复用）。

因此，当一个 Job 的 Task（尤其是 Task 的耗时很小）数目很大，由于频繁的 JVM 停启会造成很大的开销，进行 JVM 的复用也可以使同一个 Job 的一些静态数据得到共享，从而使集群的性能得到极大的提升。但是 JVM 的重用也会造成在同一个 JVM 中的碎片增加，导致 JVM 的性能变差。但是这一负面影响不是很大，总的来说，JVM 重用还是值得使用的，尽管相对于那些长时间且 Task 数少的 Job 来说，JVM 重用几乎没有什么性能提升。

但是有一点值得注意，由于 mapred.job.reuse.jvm.num.task 是客户端参数（也可以在服务端 TaskTracker 节点上将其声明为 final 使之生效），在 JVM 重用时，会导致 map（reduce）函数中 static 类型的变量在使用时可能没有达到预期目的，因为再次使用该变量时，静态变量的值仍为上次 Task 运行结束时的值。因此在使用该参数时，对于在 map（reduce）函数中静态变量的使用一定要小心，应该考虑是否需要对其进行初始化或者仍然应用上次使用的值，以达到数据共享目的。

6.6.3 跳过坏的记录

数据的一些记录不符合规范，处理时抛出异常，MapReduce 可以将这次记录标为坏记录，重启任务时会跳过该记录。

默认情况下该属性是关闭的，通过 mapred.map.max.attemps 和 mapred.reduce.max.attemps 两个参数进行设置。

6.6.4 任务执行的环境

在 MapReduce 程序中，可以通过某些环境属性（Configuration）得知作业和任务的信息。属性介绍如表 6-4 所示。

表 6-4 任务属性解释

属性名称	功能描述
mapred.job.id	作业 ID，如 job_201104121233_0001
mapred.tip.id	任务 ID，如 task_201104121233_0001_m_000003
mapred.task.id	任务尝试 ID，如 attempt_201104121233_0001_m_000003_0
mapred.task.partition	作业中任务 ID，如 3
mapred.task.is.map	此任务是否为 Map 任务，如 true

6.7 作业调度器

Hadoop 作业调度器采用的是插件机制，即作业调度器是动态加载的、可插拔的，同时第三方可以开发自己的作业调度器替代 Hadoop 默认的调度器。目前，Hadoop 作业调度器主要有以下三个。

- 先进先出调度器（FIFO Scheduler）
- 容量调度器（Capacity Scheduler）
- 公平调度器（Fair Scheduler）

6.7.1 先进先出调度器

最早的 Hadoop MapReduce 计算架构中，JobTracker 在进行作业调度时使用的是 FIFO（First In First Out）算法。所有用户的作业都被提交到一个队列中，然后由 JobTracker 先按照作业的优先级高低，再按照作业提交时间的先后顺序选择将被执行的作业。FIFO 是 Hadoop 的默认调度器。

FIFO 优点是调度算法简单明了，JobTracker 工作负担轻。FIFO 缺点是忽略了不同作业的需求差异。例如，类似对海量数据进行统计分析的作业长期占据计算资源，在其后提交的交互型作业有可能迟迟得不到处理，从而影响用户的体验。

6.7.2 容量调度器

容量调度器（Capacity Scheduler）的每个队列中采用的调度策略是 FIFO 算法。

Capacity Scheduler 默认情况下不支持优先级，但是可以在配置文件中开启此选项，如果支持优先级，调度算法就是带有优先级的 FIFO。

Capacity Scheduler 不支持优先级抢占，一旦一个作业开始执行，在执行完成之前它的资源不会被高优先级作业所抢占。

Capacity Scheduler 对队列中同一用户提交的作业能够获得的资源百分比进行限制，以避免同属于一个用户的作业独占资源的情况。

1. 容量调度器的配置

容量调度器的配置步骤如下所示。

步骤 1 将 Capacity Scheduler 的 JAR 包复制到 $HADOOP_HOME/lib 目录下。

```
cp $HADOOP_HOME/contrib/capacity-scheduler/hadoop-*-capacity-scheduler.jar
$HADOOP_HOME/lib/
```

步骤 2 配置 mapred-site.xml 文件。

添加一个参数 mapred.jobtracker.taskScheduler，设置值为 org.apache.hadoop.mapred.CapacityTaskScheduler。这里分两个组，在 mapred.queue.names 里定义，同时组的具体配额在 capacity-scheduler.xml 里定义。具体的配置操作如下所示。

1）打开 mapred-site.xml 文件，添加如下内容。

```
<property>
   <name>mapred.jobtracker.taskScheduler</name>
   <value>org.apache.hadoop.mapred.CapacityTaskScheduler</value>
</property>
<property>
   <name>mapred.queue.names</name>
   <value>default,queue1,queue2</value>
</property>
```

mapred.queue.names 默认值为 default，queue1 和 queue2 是新设置的两个队列。

2）设置具体的 queue 配额信息在 capacity-scheduler.xml 里进行，前面两组分别配置 default 和 quenen1 组，完整的配置如下所示。

```
<?xml version="1.0"?>
<configuration>
   <property>
      <name>mapred.capacity-scheduler.queue.default.capacity</name>
      <value>5</value>
   </property>
   <property>
      <name>mapred.capacity-scheduler.queue.default.supports-priority</name>
      <value>true</value>
   </property>
   <property>
      <name>mapred.capacity-scheduler.queue.default.minimum-user-limit-percent</name>
      <value>100</value>
   </property>
   <property>
      <name>mapred.capacity-scheduler.queue.default.maximum-initialized-jobs-
      per-user</name>
```

```
        <value>2</value>
    </property>
    <property>
        <name>mapred.capacity-scheduler.queue. quenen1.capacity</name>
        <value>95</value>
    </property>
    <property>
        <name>mapred.capacity-scheduler.queue. quenen1.supports-priority</name>
        <value>true</value>
    </property>
    <property>
        <name>mapred.capacity-scheduler.queue. quenen1.minimum-user-limit-
        percent</name>
        <value>100</value>
    </property>
    <property>
        <name>mapred.capacity-scheduler.queue. quenen1.maximum-initialized-jobs-
        per-user</name>
        <value>2</value>
    </property>
    <property>
        <name>mapred.capacity-scheduler.default-supports-priority</name>
        <value>true</value>
    </property>
    <property>
        <name>mapred.capacity-scheduler.default-minimum-user-limit-percent</name>
        <value>100</value>
    </property>
    <property>
        <name>mapred.capacity-scheduler.default-maximum-initialized-jobs-per-
        user</name>
        <value>2</value>
    </property>
    <property>
        <name>mapred.capacity-scheduler.init-poll-interval</name>
        <value>5000</value>
    </property>
    <property>
        <name>mapred.capacity-scheduler.init-worker-threads</name>
        <value>5</value>
    </property>
</configuration>
```

步骤 3 配置完成后，重启 JobTracker，命令如下。

```
stop-mapred.sh
start-mapred.sh
```

注意 在提交 Job 时要设置 job.set()，指定组或者 pool。如 job.set("queue1")，表示把任务提交到 queue1 队列里执行，默认是在 default 里执行。

2. 如何选择合适的作业执行

为队列定义一个指标，即队列中正在运行的任务数与其应该分得的计算资源（配置文件中为此队列分配了相应数量的资源，而实际中该队列可能没有分配到）之间的比值。当系统中出现空闲的 TaskTracker，算法会首先选择一个比值最低的队列。 队列被选中后，将按照作业优先级（如果支持）和提交时间顺序选择执行的作业。

在选择作业的时候，还需要考虑作业所属的用户是否已经超出了所能使用的资源限制。此外，还会考虑 TaskTracker 内存资源是否满足作业的要求。

3. 内存资源的有效管理

Capacity Scheduler 能有效地对 Hadoop 集群的内存资源进行管理，以支持内存密集型应用。 作业对内存资源需求高时，调度算法将把该作业的相关任务分配到内存资源充足的 Task Tracker 上。

在作业选择过程中，Capacity Scheduler 会检查空闲 TaskTracker 上的内存资源是否满足作业要求。TaskTracker 上的空闲资源（内存）数量值可以通过 TaskTracker 的内存资源总量减去当前已经使用的内存数量得到，而后者包含在 TaskTracker 向 JobTracker 发送的周期性心跳信息中。

目前，基于内存的调度只能在 Linux 平台起作用，关于内存调度的相关参数可以通过配置文件来设置。

6.7.3 公平调度器

公平调度器（Fair Scheduler）的设计思想是，尽可能保证所有的作业都能够获得等量的资源份额。系统中只有一个作业执行时，它将独占集群所有资源。有其他作业被提交时就会有 TaskTracker 被释放并分配给新提交的作业，以保证所有的作业都能够获得大体相同的计算资源。

1. 作业池

用户提交的作业将会放进一个能够公平共享资源的池（pool）中。 每个作业池设定了一个最低资源保障（guaranteed minimum share），当一个池中包含 Job 时，它至少可以获得 minimum share 的资源，这就是最低保障资源份额机制。

池中的作业获得一定份额的资源，可以通过配置文件限制每个池中的作业数量。 默认情况下，每个作业池中选择将要执行作业的策略是 FIFO 策略，先按照优先级高低排序，然后按照提交时间排序。

2. 作业和作业池的权值 weight

默认情况下，Fair Scheduler 会为每一个用户建立一个单独的 pool。所有用户能够获得等量的资源份额而无论提交了多少作业，而每个 pool 中，各个作业将平分分配所在池的资源。实际应用中，无论是作业池还是作业，都被赋予一定的权值，并以此为依据获得相应

比例的资源。这种情况下，作业池和作业在资源分配时不是严格的平均分配，但这有利于根据作业的重要程度及实际需求合理分配资源。

3. 如何选择合适的作业执行

默认是 FIFO 策略，此外还有一种基于赤字的策略。 Fair Scheduler 为每个作业定义了一个赤字（deficit）指标。deficit 是一个作业在理想情况下获得的计算资源和实际中获得的计算资源之间的差距。

Fair Scheduler 会每隔几百毫秒观察每个作业中有多少任务已经在这个时间间隔内执行，并将结果与它应得的资源份额比较，以更新该作业的 deficit 值。一旦有空闲的 Task Tracker 出现，首先分配给当前具有最高 deficit 值的作业。

例外情况是，如果系统中存在着尚未获得最低资源保障的作业池，该池中的作业将会优先调度，而选择池中的作业需要根据它们的 deficit 来决定。这样做是为了尽可能满足作业池最低保障资源份额的机制。

4. 如何确定每个作业的资源份额

默认情况是平分资源，此外提供一种基于权值的资源分配方法。作业资源份额的计算是根据作业的权值将集群的资源总量划分给各个可以运行的作业。默认情况下，权值基于作业优先级，每个优先级对应的权值是低一个优先级的 2 倍（优先级有 VERY_HIGH、HIGH、NORMAL、LOW、VERY_LOW 五个等级，则 VERY_HIGH 具有 4 倍 NORMAL 的权值）。 作业和作业池的权值可以在池配置文件中进行设定，例如，可以基于作业的大小和提交时间来设定。 作业池的最低资源保障也是按照权值比例分配给其中的作业。

5. 公平调度器的配置

将 Fair Scheduler 的 JAR 包复制到 $HADOOP_HOME /lib 目录下，命令如下。

```
cp $HADOOP_HOME/contrib/fairscheduler/hadoop-*-fairscheduler.jar $HADOOP_HOME/lib/
```

配置 mapred-site.xml 文件，内容如下所示。

```
<property>
   <name>mapred.jobtracker.taskScheduler</name>
   <value>org.apache.hadoop.mapred.FairScheduler</value>
</property>
<property>
   <name>mapred.fairscheduler.allocation.file</name>
   <value>$HADOOP_CONF/pools.xml</value>
</property>
<property>
   <name>mapred.fairscheduler.preemption</name>
   <value>true</value>
</property>
<property>
   <name>mapred.fairscheduler.assignmultiple</name>
```

```
    <value>true</value>
</property>
<property>
    <name>mapred.fairscheduler.poolnameproperty</name>
    <value>mapred.queue.name</value>
    <description>job.set("mapred.queue.name",pool);
    // pool is set to either 'high' or 'low' </description>
</property>
<property>
    <name>mapred.fairscheduler.preemption.only.log</name>
    <value>true</value>
</property>
<property>
    <name>mapred.fairscheduler.preemption.interval</name>
    <value>15000</value>
</property>
<property>
    <name>mapred.queue.names</name>
    <value>default,quenen1</value>
</property>
```

配置 pools.xml 文件，内容如下所示。

```
<?xml version="1.0"?>
<allocations>
    <pool name=" quenen1">
      <minMaps>90</minMaps>
      <minReduces>20</minReduces>
      <maxRunningJobs>20</maxRunningJobs>
      <weight>2.0</weight>
      <minSharePreemptionTimeout>30</minSharePreemptionTimeout>
    </pool>
    <pool name="default">
      <minMaps>9</minMaps>
      <minReduces>2</minReduces>
      <maxRunningJobs>20</maxRunningJobs>
      <weight>1.0</weight>
      <minSharePreemptionTimeout>30</minSharePreemptionTimeout>
    </pool>
</allocations>
```

6. 配额文件格式

配额文件为每一个资源池配置最小共享资源、运行作业限制、权重和抢占超时时间。与 Linux 系统下面的文件配额类似。

HADOOP_HOME/conf/fairscheduler.xml.template 提供了一个示例。配额文件可以包含元素的类型如表 6-5 所示。

表 6-5 配额文件包含的元素

元 素	功能描述
pool	配置各个资源池。可能包含下列子元素： • minMaps 和 minReduces，设置资源池最小共享的任务时间片 • maxRunningJobs，限制从资源池同时运行的作业数量（默认是无限） • weight，以非比例的方式与其他资源池共享集群（默认是 1.0） • minSharePreemptionTimeout，如果资源池的资源低于它的最小共享资源，在结束其他资源池的任务之前等待的秒数（默认是无限）
user	可能包含一个 maxRunningJobs 属性用来限制作业。默认情况下要注意，每一个用户有一个资源池，所以每用户限制不是必需的
poolMaxJobsDefault	为没有指定运行作业限制的资源池设定默认值
userMaxJobsDefault	为没有指定运行作业限制的用户设定默认值
defaultMinSharePreemptionTimeout	为没有指定最小共享资源抢占超时时间的资源池设定默认值
fairSharePreemptionTimeout	设置作业拥有低于其公平共享资源的一半的抢占超时时间

pool 和 user 元素仅在对资源池 / 用户设定了非默认值时才是必需的。也就是说，在运行公平调度器前，不必在配置文件中声明所有的用户及资源池。如果一个用户或资源池没有列在配置文件中，将会使用运行作业限制、抢占超时时间等默认值。

下面给出一个配额文件示例。

```
<?xml version="1.0"?>
<allocations>
<pool name="sample_pool">
  <minMaps>5</minMaps>
  <minReduces>5</minReduces>
  <weight>2.0</weight>
</pool>
  <user name="sample_user">
   <maxRunningJobs>6</maxRunningJobs>
  </user>
<userMaxJobsDefault>3</userMaxJobsDefault>
</allocations>
```

以上示例创建一个资源池 sample_pool，保证最小有 5 个 Map 时间片和 5 个 Reduce 时间片。资源池有 2.0 的权重，意味着它与其他资源池相比，拥有相对 2 倍的集群共享资源（默认权重是 1）。最后，除了 sample_user 可并行运行 6 个作业外，示例限制了其他用户的运行作业数为 3。任何未在配额文件中定义的资源池将获得无保证的容量及 1.0 的权重。与此同时，任何没有在配额文件中设置最大运行作业数的资源池或用户将被允许运行无限个作业。

想了解更多公平调度器的属性配置，可以参考 Hadoop 自带的 fairscheduler.xml.template 文件，位置在 HADOOP_HOME/conf/ 下，这个示例更详细，并且设定了抢占超时时间。

7. 公平调度器管理

公平调度器通过两种途径提供运行时的管理支持。

1）在运行时修改配额文件中的最小共享资源、运行作业限制、权重和抢占超时时间。调度器在检测到配额文件被修改过后 20 分钟内会重新加载它。

2）在 JobTracker 的 Web 接口 http://<JobTracker URL>/scheduler 中查看当前的作业、资源池和公平共享资源。利用这个接口，可以修改作业的优先权或把作业从一个资源池转移到另外一个，并可以看到在公平共享资源上的效果（这个需要 JavaScript）。

在 Web 接口上可以看到以下作业的字段。

- Submitted：作业提交的日期和时间。
- JobID、User 和 Name：作业标识，与标准 Web 用户接口上的一样。
- Pool：作业当前所属的资源池。选择另外一个值把作业移动到另外一个资源池。
- Priority：当前优先权。选择另外一个值来改变作业的优先权。
- Maps/Reduces Finished：已完成的任务。
- Maps/Reduces Running：当前正在运行的任务。
- Map/Reduce Fair Share：根据公平共享的方法，这个作业在任意时刻应该得到的平均任务时间片。实际的任务时间片会根据作业已有的计算时间上下波动，但一般它会获取到它公平共享资源的量。
- MapReduce Weight：作业在公平共享计算中的权重。这个依赖于优先权，如果启用了 sizebasedweight 和 NewJobWeightBooster，还跟作业的大小及年龄有关。
- MapReduce Deficit：作业在机器上的调度缺额。根据作业的公平共享资源所应得到资源量，减去它的实际所得。正的缺额意味着作业不久将会被调度，因为需要达到它的公平共享资源。调度器会首先调度拥有较高缺额的作业。

6.8 自定义 Hadoop 调度器

在 Hadoop 中，调度器是一个可插拔的模块，用户可以根据自己的实际应用要求设计调度器，然后在配置文件中指定相应的调度器，当 Hadoop 集群启动时，便会加载该调度器。

当前 Hadoop 自带了三种调度器，分别是 FIFO（默认调度器）、Capacity Scheduler 和 Fair Scheduler，通常境况下，这些调度器很难满足公司复杂的应用需求，因而往往需要开发自己的调度器。本文介绍 Hadoop 调度器的基本编写方法。

6.8.1 Hadoop 调度器框架

Hadoop 的调度器是在 JobTracker 中加载和调用的，用户可以在配置文件 mapred-site.xml 中的 mapred.jobtracker.taskScheduler 属性中指定调度器。本节分析 Hadoop 调度器的框架，实际上分析两个重要类 TaskScheduler 和 JobTracker 的关系。

1. TaskScheduler 类

如果用户要编写自己的调度器，需要继承抽象类 TaskScheduler，该类的接口如下。

```
abstract class TaskScheduler implements Configurable {
    protected Configuration conf; // 配置文件
    protected TaskTrackerManager taskTrackerManager; // 一般会设为 JobTracker
    public Configuration getConf() {
      return conf;
    }
    public void setConf(Configuration conf) {
      this.conf = conf;
    }
    public synchronized void setTaskTrackerManager(
     TaskTrackerManager taskTrackerManager) {
     this.taskTrackerManager = taskTrackerManager;
  }
  public void start() throws IOException { //初始化函数，如加载配置文件等
  // do nothing
  }
  public void terminate() throws IOException { //结束函数
  // do nothing
  }
  //最重要的函数，为该 taskTracker 分配合适的 task
  public abstract List<Task> assignTasks(TaskTrackerStatus taskTracker)
  throws IOException;
  //根据队列名字获 job 列表
  public abstract Collection<TaskInProgress> getJobs(String queueName);
}
```

2. JobTracker 类

JobTracker 是 Hadoop 的核心组件，它监控整个集群中的作业运行情况并对资源进行管理和调度。

每个 TaskTracker 每隔 3 秒（默认值，可配置）通过 heartbeat 向 JobTracker 汇报自己管理机器的一些基本信息，包括内存使用量、内存剩余量、正在运行的 Task、空闲的 slot 数目等，一旦 JobTracker 发现该 TaskTracker 出现了空闲的 slot，便会调用调度器中的 AssignTasks 方法为该 TaskTracker 分配 Task。

下面分析 JobTracker 调用 TaskScheduler 的具体流程。

```
    ...
    private final TaskScheduler taskScheduler; // 声明调度器对象
    public static JobTracker startTracker(JobConf conf, String identifier) {
    ...
    result = new JobTracker(conf, identifier);
    result.taskScheduler.setTaskTrackerManager(result); // 设置调度器的 manager
    ...
}
    // 创建调度器
```

```
    JobTracker(JobConf conf, String identifier) {
    ...
    // Create the scheduler
    Class<? extends TaskScheduler> schedulerClass = conf.getClass("mapred.
    jobtracker.taskScheduler",
    JobQueueTaskScheduler.class, TaskScheduler.class);
    taskScheduler = (TaskScheduler) ReflectionUtils.newInstance(schedulerClass,
    conf);
    ...
  }
//run forever
public void offerService() {
    ...
    taskScheduler.start(); //启动调度器
    ...
}
HeartbeatResponse heartbeat(TaskTrackerStatus status,
    boolean restarted,
    boolean initialContact,
    boolean acceptNewTasks,
    short responseId) {
    ...
    // Check for new tasks to be executed on the tasktracker
    if (recoveryManager.shouldSchedule() && acceptNewTasks && !isBlacklisted) {
    ...
    //使用调度器，为该 taskTracker 分配作业
    tasks = taskScheduler.assignTasks(taskTrackerStatus);
    ...
    }
}
```

从上面的分析可以知道，Scheduler 和 JobTracker 之间会相互包含（实际上是组合模式），Scheduler 中要包含 JobTracker（实际上就是 TaskTrackerManager）对象，以便获取整个 Hadoop 集群的一些信息，如 slot 总数、QueueManager 对象和添加 JobInProgressListener（以便增加或删除 Job 时通知 Scheduler）。JobTracker 中要包含 Scheduler 对象，以便可以对每个 TaskTracker 分配 Task 任务。

6.8.2 编写 Hadoop 调度器

假设要编写一个新的调度器 MyScheduler，需要进行以下工作。

1. 用户需要自己实现的类

- MyHadoopSchedulerConf：配置文件管理类，读取自己的配置文件，并保存到合适的数据结构中。一般而言，这个类应该支持动态加载配置文件。
- MyHadoopSchedulerListener：编写自己的 JobInProgressListener，并调用 JobTracker 的 addJobInProgressListener() 将之加到系统的 Listener 队列中，以便系统中添加或

删除 Job 后，JobTracker 可立刻告诉调度器。

- MyHadoopScheduler：调度器的核心实现算法。

2. 用户要用到的系统类

实现自定义任务调度器时，JobTracker、TaskInProgress 和 JobConf 类是我们必需掌握的，接下来介绍这三个类。

（1）JobTracker 类

JobTracker 在 startTracker 函数中，会将 MyHadoopScheduler 的 taskTrackerManager 赋值为 JobTracker 对象，这样，在 MyHadoopScheduler 中，可调用 JobTracker 中的所有 public 方法和成员变量，常用的有以下几个。

- getClusterStatus()：获取集群的状态，如 TaskTracker 列表、map slot 总数、reduce slot 总数、当前正在运行的 map/reduce task 总数等。
- getQueueManager()：如果 MyHadoopScheduler 支持多队列，需要使用该方法获取 QueueManager 对象，通过该对象，会用可以获取系统的所有队列名称和每个队列的 ACL（Access Control List）⊖。
- kill job：可以调用该函数杀死某个 Job。
- kill task：如果调度器支持资源抢占，可调用该函数杀死某个 Task 以便进行资源抢占。

（2）JobInProgress 类

用户向 Hadoop 中提交一个 Job 后，Hadoop 会为该 Job 创建一个 JobInProgress 对象，该对象中包含了 Job 相关的基本信息，且它会伴随某个 Job 的一生（与 Job 共存亡）。该对象中包含的 Job 信息有：该 Job 包含的所有 Task 的信息（如正在运行的 Task 列表、已经完成的 Task 列表、尚未运行的 Task 列表等）、作业的优先级、作业的提交时间、开始运行时间，以及运行结束时间等信息。

在 JobInProgress 的 Task 列表中，每个 Task 以对象 TaskInProgress 的形式保存，该对象中包含了每个 Task 的基本信息，包括：Task 要处理的数据 split、Task 创建时间、Task 开始执行时间、Task 结束时间等。这些信息肯定会在调度器中使用。

（3）JobConf 类

每个作业的运行参数和配置选项被保存到一个 JobConf 对象中，该对象包含了配置文件 mapred-site.xml、core-site.xml 和 hdfs-site.xml 设置的选项和该作业的特有属性（用户名、InputFormat、Mapper 等），一般是以 Key-Value 形式保存。比如想获取当前用户名，可以这样：

```
JobConf  conf;
...
String username = conf.get("user.name");
```

⊖ 具体参考 http://hadoop.apache.org/common/docs/current/service_level_auth.html。

用户也可以通过该对象传递一些自己定义的全局属性，如用户自己定义一个属性mapred.job.deadline（作业的 deadline 时间），可以在提交作业时设定该值。

```
hadoop jar hadoop-examples.jar wordcount -files cachefile.txt \
-D mapred.job.deadline=100000 \
input  output
```

然后在调度器中获取该属性的值。

```
JobConf  conf;
…
int deadline=conf.getInt("mapred.job.deadline", -1);
```

获取 mapred.job.deadline 属性，如果没有设置，默认返回负数 1。

提示 调度器是 Hadoop 的中枢，其重要性可想而知。用户如果要设计 Hadoop 调度器，需要对 Hadoop 的整个框架有比较深入的理解，同时阅读一些很重要的类（如 JobTracker 和 TaskInProgress 等）的源码，以便利用这些类完成你的调度算法。

Hadoop 目前自带了三个比较常用的调度器，分别为 FIFO Scheduler、Capacity Scheduler 和 Fair Scheduler，它们是学习 Hadoop 调度器的最好资料，代码都是开源的。

6.9 YARN 介绍

YARN 是在 Hadoop 1.0 基础上衍化而来的，它具有比 Hadoop 1.0 更为先进的理念和思想，充分吸取了 Hadoop 1.0 的优势，同时增加了很多新的特性和改进。即使不使用 YARN，研究 YARN 对于改进当前 Hadoop 版本仍有巨大的帮助。

6.9.1 异步编程模型

Hadoop 0.23.0 是一个通用的资源分配框架，不仅支持 MapReduce 计算框架，也支持流式计算框架、迭代计算框架、MPI 并行计算等，实现时采用了基于事件驱动机制，异步编程模型如图 6-12 所示。

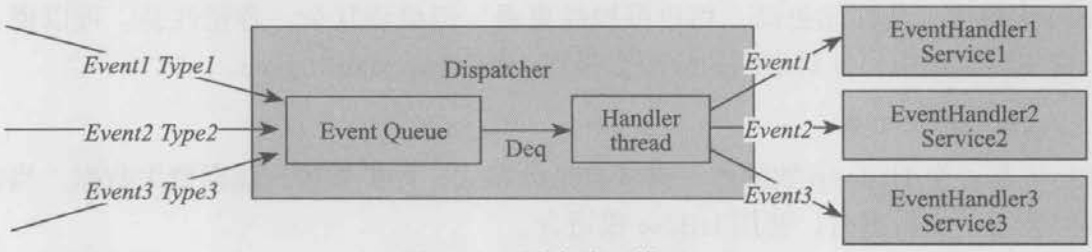


图 6-12 异步编程模型

EventHandler 称为事件处理器，每种事件类型对应一种 EventHandler，其对事件的处理过程通过状态机来描述，Handler 接收到一个事件后，使处理对象单签的状态跳转到另一个状态，同时触发一个动作。

6.9.2 YARN 支持的计算框架

YARN 是 Hadoop 2.0 及以上版本的下一代集群资源管理和调度平台，能支持多种计算框架，具体包括以下几种。

1. MapReduce

首先是大家熟悉的 MapReduce，在第二代 MapReduce（MR2）之前，Hadoop 包括 HDFS 和 MapReduce，作为 Hadoop 上唯一的分布式计算框架，其优点是用户可以很方便地编写分布式计算程序，并支持许多应用，如 Hive、Mahout 和 Pig 等；其缺点是无法充分利用集群资源，不支持 DAG、迭代式计算等。为了解决这些问题，Yahoo 提出了 YARN（next generation mapreduce），一个分布式集群资源管理和调度平台。这样除了 MapReduce 外，还可以支持各种计算框架。

2. Spark

Spark 是一种与 MapReduce 相似的开源计算框架，不同之处在于 Spark 在某些工作负载方面表现更优，因为它使用了内存分布式数据集。另外，除了提供交互式查询外，它还可以优化迭代工作负载。

3. Apache Hama

Apache Hama 是一个运行在 HDFS 上的 BSP（Bulk Synchronous Parallel，大容量同步并行）计算框架，主要针对大规模科学计算，如矩阵、图像和网络算法等。当前它的功能包括：作业提交和管理接口；单节点上运行多个任务；输入 / 输出格式化；备份恢复；支持通过 Apache Whirr 运行在云端；支持与 YARN 一起运行。

4. Apache Giraph

图像处理平台上运行这大型算法（如 page rank、shared connections 和 personalization-based popularity 等）已经很流行，Giraph 采用 BSP 模型，可用于迭代类算法。

5. Open MPI

这是一个高性能计算函数库，通常在 HPC（High Performance Computing）中采用。与 MapReduce 相比，其性能更高，用户可控性更强，但编程复杂，容错性差。可以说，各有所长，在实际应用中，针对不同的应用会采用 MPI 或者 MapReduce。

6. Apache HBase

HBase 是一个 Hadoop 数据库，其特点是分布式，可扩展的，能存储大数据。当有需要随机实时读写的大数据时，使用 HBase 很适合。

6.9.3 YARN 架构

YARN 最基本的设计思想是将 JobTracker 的两个主要功能，即资源管理和作业调度、监控分成两个独立的进程，如图 6-13 所示。在该解决方案中包含两个组件：全局的

ResourceManager（RM）、与每一个应用相关的 ApplicationManager（AM）。

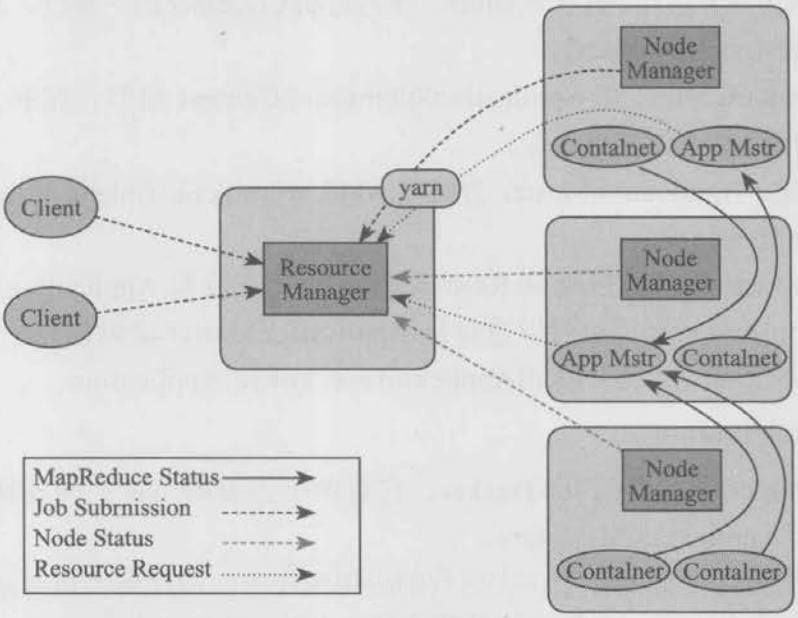


图 6-13 YARN 架构图

这里，“应用”指的是一个单独的 MapReduce 作业或者 DAG 作业。RM 与 NM（NodeManager）共同组成整个计算框架。RM 是系统中将资源分配给各个应用的最终决策者。AM 实际上是一个具体的框架库，它的任务是与 RM 协商获取应用所需资源和与 NM 合作，以完成执行和监控 Task 的任务。

6.9.4 YARN 工作流程

首先说的概念是“Application Submission Client”，它负责将“Application”提交到 YARN 的 ResourceManager。客户端通过 ClientRMProtocol 协议与 ResourceManager 联系，如果需要 Client，会通过 ClientRPProtocol::getNewApplication 来获取新的 ApplicationId，然后通过 ClientRMProtocol::submitApplication 将应用提交运行。

作为 ClientRMProtocol::submitApplication 调用的一部分，客户端需要足够的信息给 ResourceManager 来运行应用的第一个 Container，也就是 ApplicationMaster。需要提供以下一些信息。

- 你的应用程序运行时所需要的本地文件和本地 JAR 文件；
- 执行时所运行的命令（包括必要的命令参数）；
- UNIX 环境变量设置（可选的）。

实际上需要为 ApplicationMaster 提供 UNIX 进程的描述信息。YARN 的 ResourceManager 会在一个获得的 Container 上启动 ApplicationMaster。ApplicationMaster 通过 AMRMProtocol 协议与 ResourceManager 通信，首先 ApplicationMaster 需要将自身注册到 ResouceManager。ApplicationMaster 为了完成交给它的任务，会通过 AMRMProtocol::allocate 申请 Container。

如果获得了 Container，ApplicationMaster 会通过 ContainerManager::startContainer 和 NodeManager 联系为任务启动一个 Container。作为启动 Container 的一部分，ApplicationMaster 需要指定 ContainerLaunchContext。

ContainerLaunchContext 和 ApplicationSubmissionContext 相似，包括一些启动时需要的信息，如命令行命令、环境变量等。

一旦任务完成，ApplicationMaster 会通过 AMRMProtocol::finishApplicationMaster 通知 ResourceManager 任务完成。

与此同时，Client 可以通过查询 ResourceManager 来获取 Application 的状态信息，或者如果 ApplicationMaster 支持也可以直接从 ApplicationMaster 查询信息。如果需要，Client 可以通过 ClientRMProtocol::forceKillApplication 来 kill 掉 Application。

1. ResourceManager

ResourceManager 有点类似 JobTracker，它有两个主要的组件：调度器（Scheduler）和应用程序管理器（ApplicationManager）。

Scheduler 负责分配资源到各个正在运行的应用程序中。调度器不执行监控和跟踪应用程序的状态，从这个意义上说，它是纯粹的调度器。此外，它也不保证重启失败的任务。调度器是基于资源的请求来执行它的调度功能，它是基于资源容器的抽象概念，这种资源容器包括内存、CPU、磁盘、网络等。在第一个版本中只支持内存。调度器支持可插入的策略。

ApplicationManager 负责接送提交的作业，协商第一个执行该任务的容器，并提供失败作业的重启。NodeManager 是每个节点的框架代理，它负责监控资源的使用情况，并报告给 ResourceManager。每个应用的 ApplicationMaster 负责与调度器谈判资源占用的 Container 数量，追踪状态和监控进程。

2. NodeManager

NodeManager 类似 TaskTracker，它负责启动应用程序 Container（类似 JVM），监控 Container 的资源（CPU、内存、磁盘、网络等），并将信息上报给 ResouceManager。需要注意的是，调度器就是根据应用程序的 Container 进行调度的。

6.10 本章小结

本章主要介绍了 MapReduce 的工作原理，主要讲解了 JobTracker、TaskTracker、任务的调度、任务工作的原理以及 MapReduce 的 Shuffle 和 Sort 机制等，此外还介绍第二代 MapReduce（YARN），读者通过这章内容可以深刻理解 MapReduce 的原理，对日后 MapReduce 的编程开发奠定了基础。

第7章　Eclipse 插件的应用

Hadoop 是一个强大的并行框架，它允许任务在其分布式集群上并行处理。但是编写、调试 Hadoop 程序都有很大难度。正因为如此，在 Hadoop 的开发环境中嵌入了 Eclipse，从而实现了开发环境的图形化，降低了编程难度。

在安装插件配置 Hadoop 的相关信息之后，如果用户创建 Hadoop 程序，插件会自动导入 Hadoop 编程接口的 JAR 文件，这样用户就可以在 Eclipse 的图形化界面中编写、调试、运行 Hadoop 程序（包括单机程序和分布式程序），也可以在其中查看自己程序的实时状态、错误信息和运行结果，还可以查看、管理 HDFS 以及文件。总的来说，Hadoop Eclipse 插件安装简单，使用方便，功能强大，尤其是在 Hadoop 编程方面，是入门必不可少的工具。

Eclipse 是目前最主流的 IDE 工具，一个好的开发环境和工具对于开发人员来说非常的重要。用 Eclipse 插件可以调试 Hadoop 源码和 MapReduce 程序，还可用来编译 Hadoop 源码和对 Hadoop 进行二次开发，本章主要介绍利用 Eclipse 编译 Hadoop 源码和调试 MapReduce 程序。

7.1　编译 Hadoop 源码

Hadoop 是一个开源的框架，既然是开源的，我们就可以修改 bug、commit 代码以及在 Hadoop 上进行二次开发，但最终我们需要了解 Hadoop 是怎么编译的。

Hadoop 源码包含 HDFS、MapReduce、Hadoop 的 Eclipse 插件、common 等组件。下面以编译 common 组件为例来介绍 Hadoop 源码的编译过程。

7.1.1　下载 Hadoop 源码

Hadoop 各成员源代码下载地址是 http://svn.apache.org/repos/asf/hadoop。请使用 SVN 下载，在 SVN 浏览器中将 trunk 目录下的源代码检索（check-out）到本地即可。

注意，只检索 SVN 上 trunk 目录下的内容，如 http://svn.apache.org/repos/ asf/ hadoop/ common/trunk，而不是 http://svn.apache.org/repos/asf/hadoop/common。原因是 http:// svn.apache.org/ repos/asf/hadoop/ common 目录包括很多非源代码文件，很庞大，check-out 需要很长的时间。

7.1.2　准备编译环境

Hadoop 的版本差异比较大，这里以 hadoop-1.0.3 为例，因为这个版本是在企业里用得最多的。

编译 Hadoop 会依赖很多第三方库，但编译工具 Ant 会自动从网上下载缺少的库，所以必须保证机器能够访问 Internet。

编译 Hadoop 要用 JDK1.6 以上，网址为 http://java.sun.com/javase/ downloads/index.jsp。

安装好之后，请设置 JAVA_HOME 环境变量，如表 7-1 所示。

表 7-1 JDK 环境变量配置

HOME	C:\Program Files\Java\jdk1.6.0_02
Path	%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin
CLASSPATH	.;%JAVA_HOME%lib;%JAVA_HOME%lib\tools.jar

使用 Ant 工具编译 Hadoop，可以从 http://ant.apache.org/ivy/download.cgi 下载，下载以后解压 Ant 文件，最后把 Ant 加入环境变量 Path 中，如图 7-1 所示。

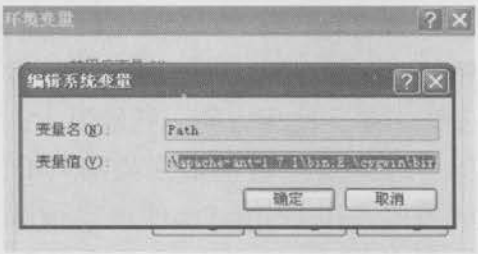


图 7-1 添加 Ant 环境变量

Cygwin 是一个在 Windows 平台上运行的 UNIX 模拟环境，Hadoop 是运行在 Linux 环境下，如果想用 Eclipse 实现 MapReduce 的运行，需要在 Windows 系统上安装 Cygwin，如果在 Linux 下面使用 Eclipse 开发 MapReduce 程序，就不需要安装 Cygwin 程序了。

安装 Cygwin 很简单，在 http://www.cygwin.com/setup.exe 下载 setup.exe 执行文件，然后单击安装。安装过程会很慢，因为是在线安装的。

Eclipse 可以从 http://www.eclipse.org/downloads/ 下载。64 位操作系统一定要下载 x64 版本，否则启动会报错，找不到 dll 库。这里下载 Eclipse 3.6 版本（Helios）。

7.1.3 编译 common 组件

本节介绍如何编译 Hadoop 家庭成员 common，其他成员的编译方法类似。

步骤 1 创建一个 Java 工程。

在 Eclipse 的 Package 视图中单击右键，选择 New → Java Project，如图 7-2 所示。

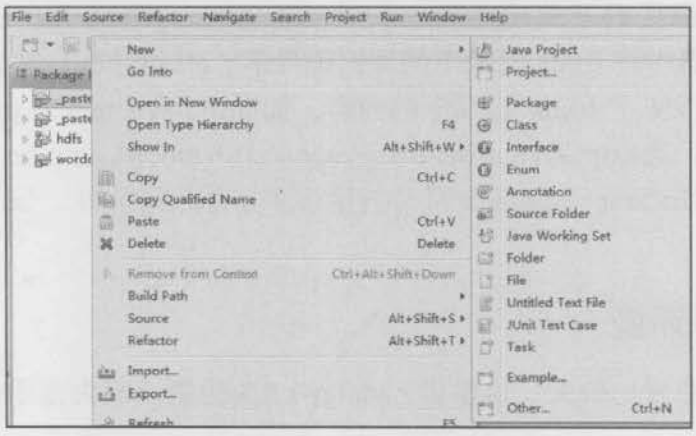


图 7-2 创建 Java 工程

步骤 2 选择源代码目录，设置 Project 名。

在图 7-3 所示的对话框中，单击 Browse 按钮，选择 common 源代码目录，并设置 Project name 为 common。

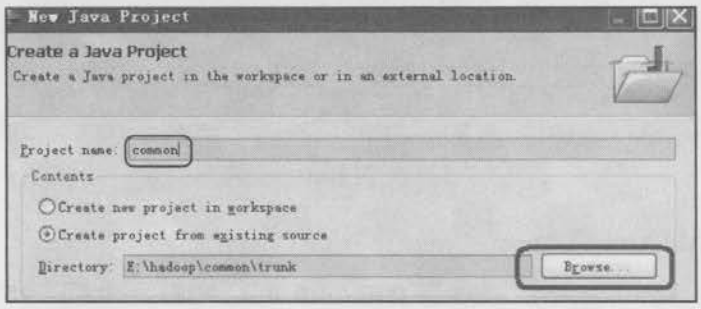


图 7-3 设置工程名称

工程导入完成后，进入 Eclipse 主界面，可以看到 common 已经导入进来，但 common 上有红叉，因为 Eclipse 默认使用了 Java Builder，而不是 Ant Builder，所以下一步就是设置使用 Ant Builder。

步骤 3 设置 Builder 名称。

右键 common → Properties → Builders，如图 7-4 所示。

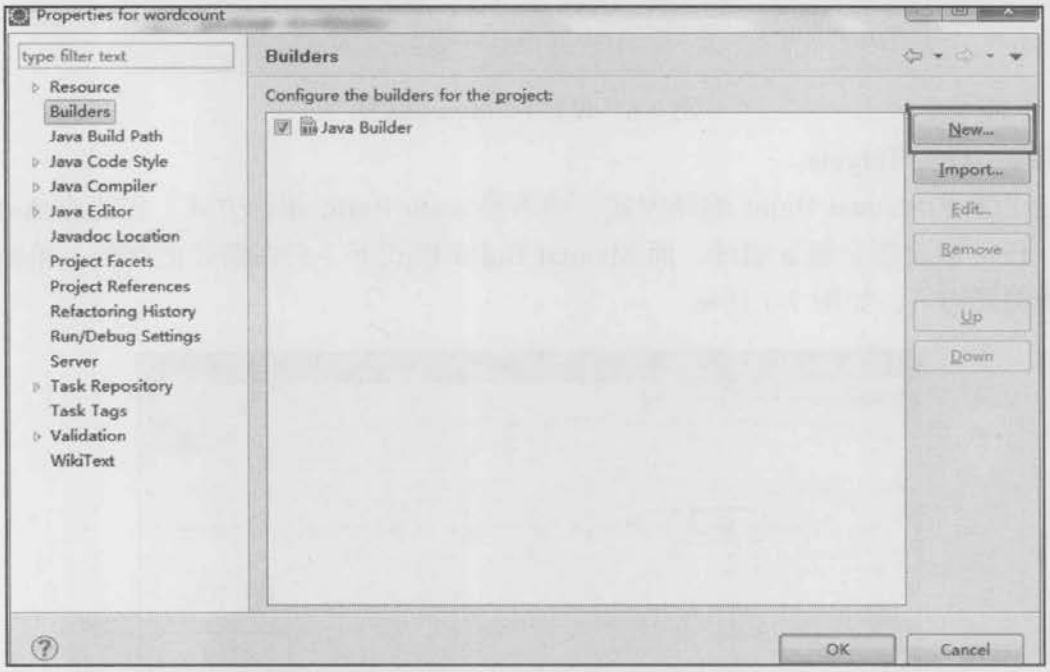


图 7-4 设置 Builders 名称

在图 7-4 所示的界面中，单击 New 按钮，在弹出的对话框中选 Ant Builder，确定之后会弹出如下对话框，如图 7-5 所示。

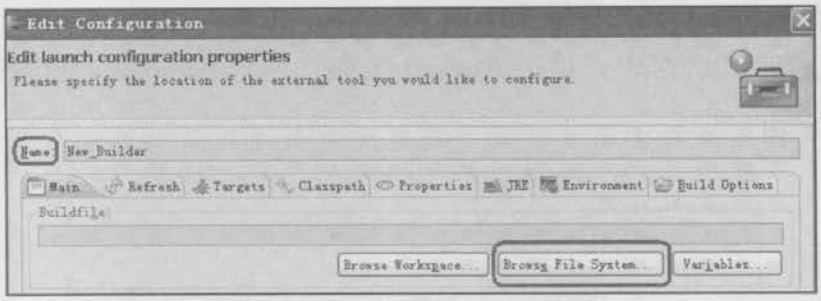


图 7-5 设置 Builder 名称

步骤 4 设置 build.xml 路径。

单击 Browse File System 按钮，选择 common 源代码目录下的 build.xml 文件，并设置 Name 为 common_Builder（可以使用其他的名称，但建议使用 common_Builder，因为这样名副其实，比较规范），操作结果如图 7-6 所示。

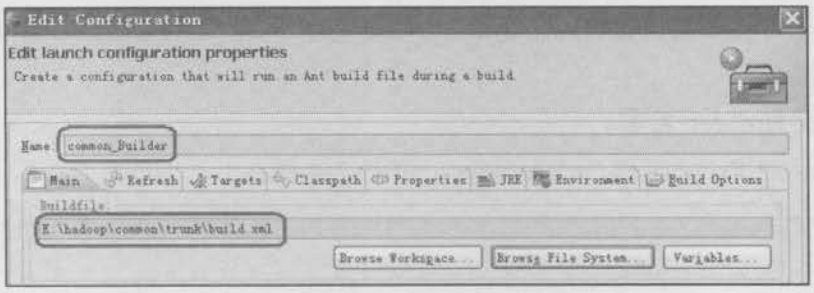


图 7-6 设置 build.xml 路径

步骤 5 设置 Targets。

建议设置成 Manual Build 编译方式，而不是 Auto Build 编译方式。因为在 Auto Build 模式下，任何修改都会触发编译，而 Manual Build 模式下，只在需要的时候，单击编译按钮或菜单编译即可，如图 7-7 所示。

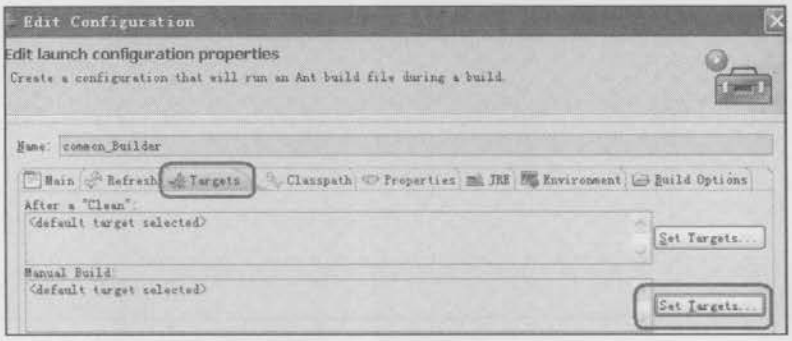


图 7-7 设置 Targets

步骤 6 设置编译的 JAR 文件。

Hadoop 各成员都需要编译成 JAR 文件，如图 7-8 所示修改。

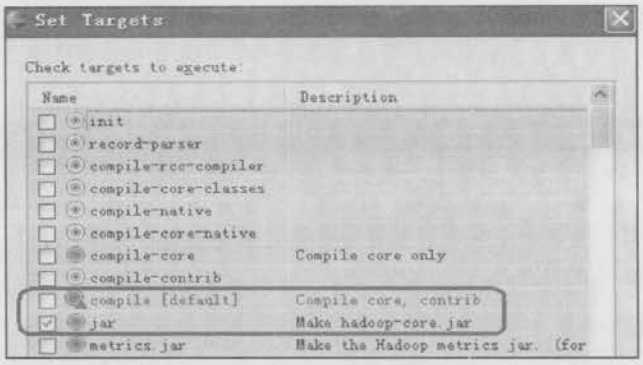


图 7-8 设置编译的 JAR 文件

确定之后，返回图 7-9 所示的 Edit Configuration 对话框。

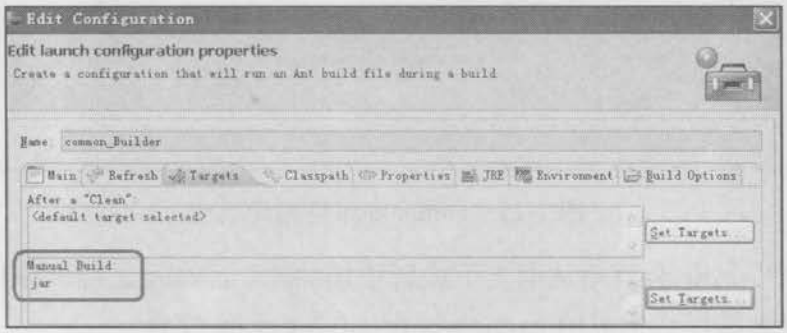


图 7-9 设置编译目标

步骤 7 上面步骤完成后，回到 Builder 主对话框，将对话框中的 Java Builder 下移，并取消它前面的勾选。

步骤 8 进入 Eclipse 主界面，之前选择了 Manual Build，所以需要人工方式驱动编译。

右键 common → Build Project 编译 common 组件，如图 7-10 所示。如果菜单中的 Build Automatically 被勾选，在 common 的右键菜单中可能不会出现 Build 子菜单。

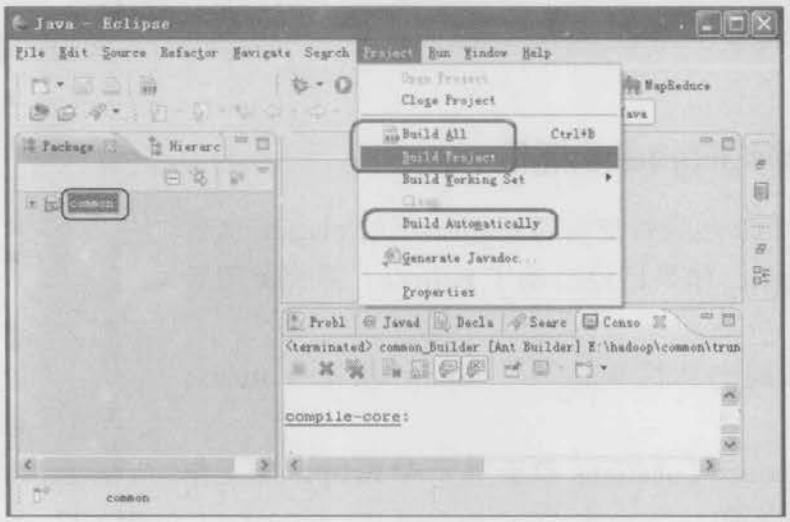


图 7-10 编译 Hadoop 的 common 模块

编译需要 5 分钟左右，如果出现 BUILD SUCCESSFUL，说明 common 组件编译成功，如图 7-11 所示。

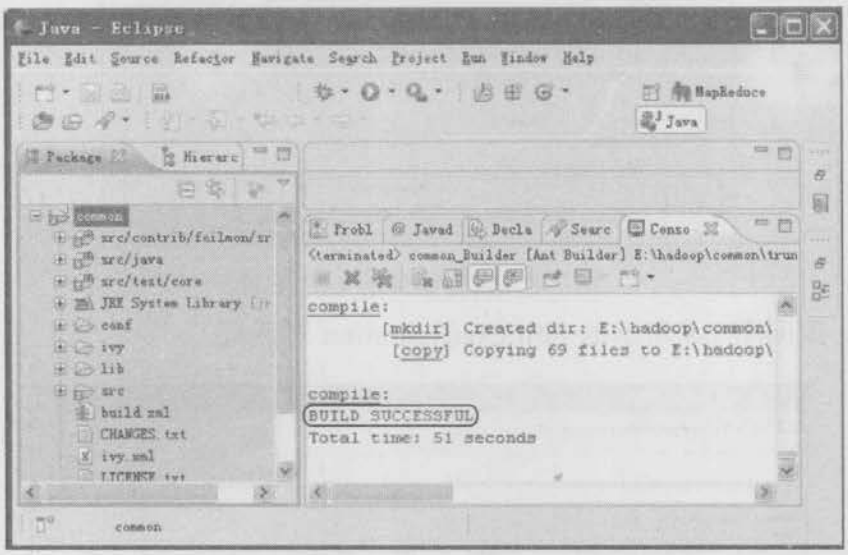


图 7-11 common 组件编译成功

在编译过程中，Ant 会自动从网上下载所依赖的库。common 的编译成功结束后，可以在 build 目录下找到编译后生成的 hadoop-core-1.0.3-dev.jar 文件。

提示 HDFS、MapReduce 和 Hive 等模块的编译方式与 common 相同。

7.2 Eclipse 安装 MapReduce 插件

Hadoop 自带了 Eclipse 插件，笔者用自带的插件没有配置成功，重新编译一下，名为 hadoop-eclipse-plugin-dev-1.0.3.jar。具体编译的过程可以参照 7.1 节的操作。这个插件对 Eclipse 的版本也是有要求的，笔者使用 Eclipse 3.6（Helios）版本。

7.2.1 查找 MapReduce 插件

把编译好的 JAR 包直接复制到 Eclipse 下的 plugins 文件夹里。启动 Eclipse，如果已经启动了 Eclipse，需要关闭后重新启动。

启动以后找到工具栏 Window → Show View → others，如图 7-12 所示。

双击 Map/Reduce Locations 就会显示在 Eclipse 的工作区间，如图 7-13 所示。

图 7-12 查找 MapReduce 插件

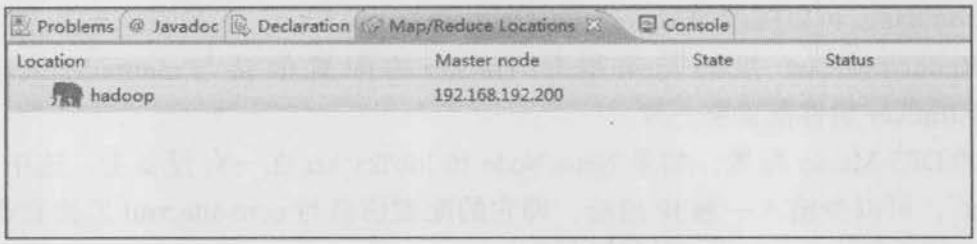


图 7-13 Hadoop 插件的工作台

7.2.2 新建一个 Hadoop location

第一次打开 Map/Reduce Locations 应该是空白的，没有“hadoop”和“192.148.192.200”配置信息，因为这是笔者刚刚配好的。鼠标放到空白处右击，出现如图 7-14 的对话框。

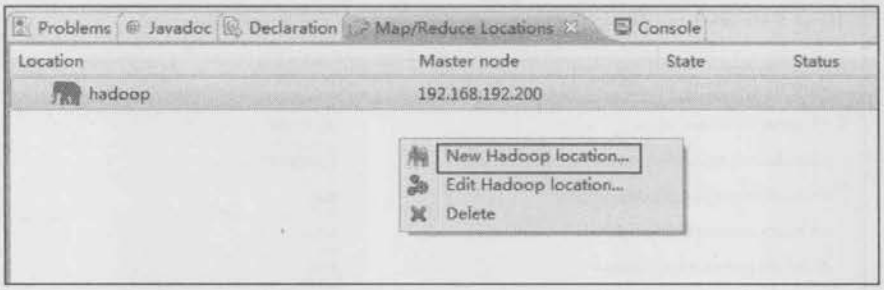


图 7-14 新建一个 Hadoop location

1. 配置 General

选择 New Hadoop location 出现如面 7-15 的对话框。

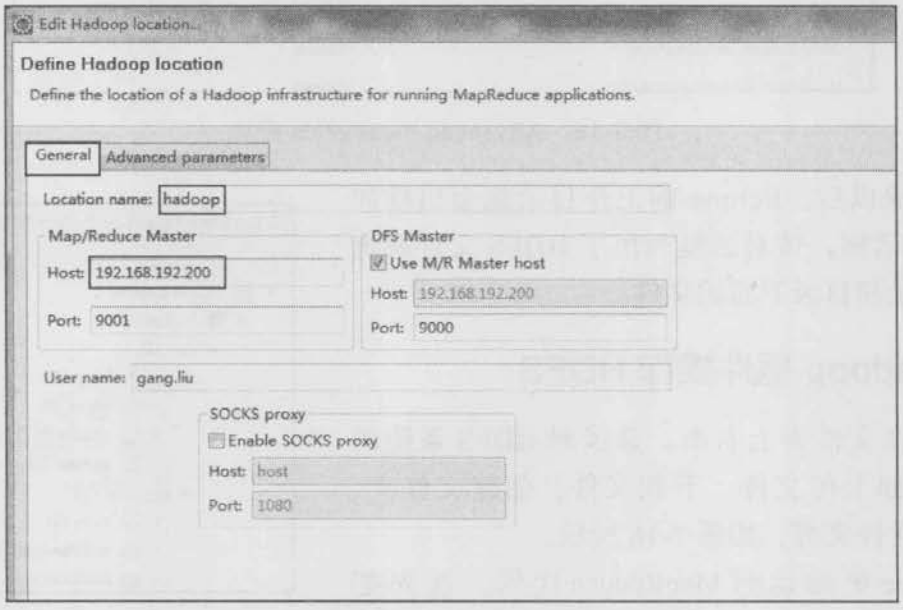


图 7-15 Hadoop location 配置对话框

Location name 可以随便填写，这里为 hadoop。

Map/Reduce Master 指的是集群 JobTracker 的配置信息与 mapred-site.xml 里面的 mapred.job.tracker 属性配置要一致。

右边的 DFS Master 配置，如果 NameNode 和 JobTracker 在一台设备上，选中“Use M/R Master host”，可以少输入一遍 IP 地址，即它的配置信息与 core-site.xml 里的 fs.default.name 属性一致。到这里 General 这项就配置好了。

2. 配置 Advanced parameters 选项

如图 7-16 所示。这里的参数比较多，但是要修改的地方很少。找到 hadoop.job.ugi 参数，把它改成 Hadoop 用户和用户组就可以了。最后单击完成。

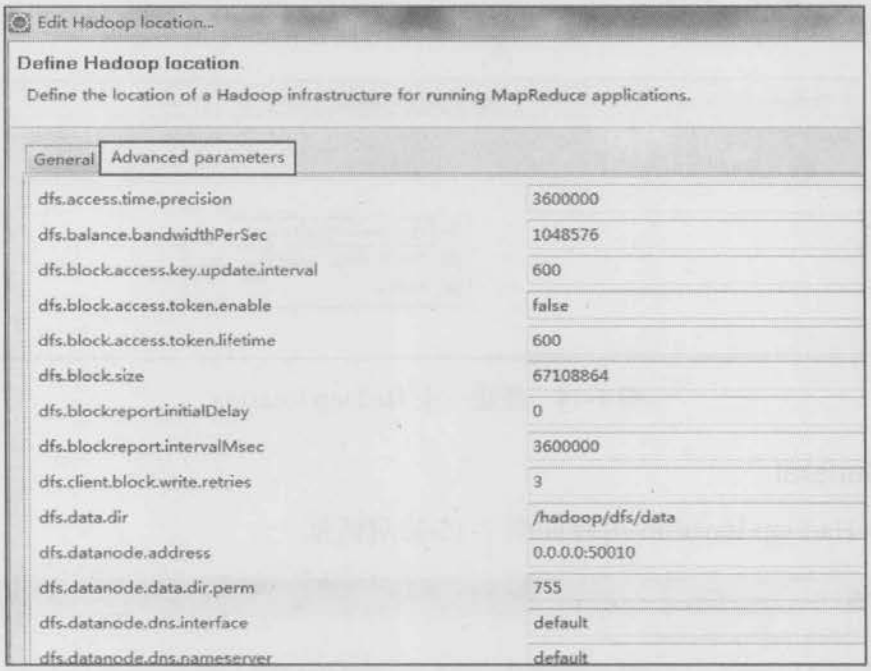


图 7-16 Advanced parameters 配置

配置完成以后，Eclipse 的工作目录就会出现如图 7-17 的对话框，该对话框列出了 HDFS 文件系统里的所有目录和目录下面的文件。

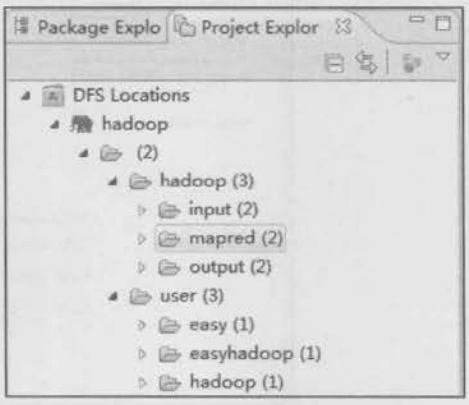


图 7-17 显示 HDFS 文件系统下的文件

7.2.3 Hadoop 插件操作 HDFS

鼠标放在文件夹上右击，显示对 HDFS 系统的一些操作，如上传文件、下载文件、创建文件夹、上传文件或文件夹等，如图 7-18 所示。

在 Eclipse 里面运行 MapReduce 代码，首先把 Hadoop 的 examples 下自带的 WordCount 案例复制到

工作目录下，接下来设置输入和输出的参数，如图 7-19 所示。

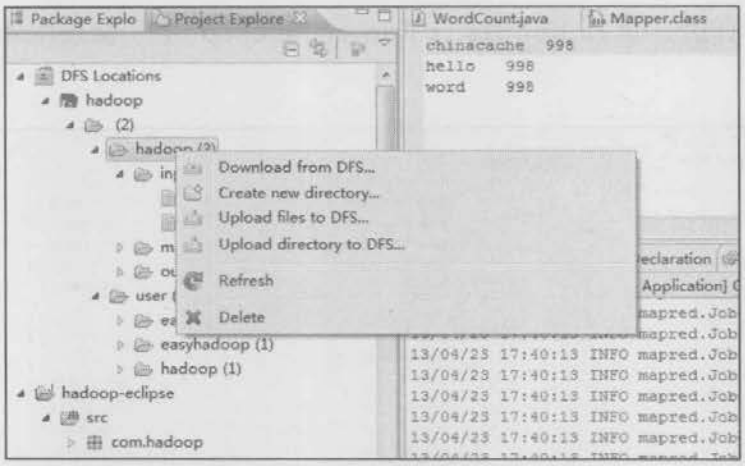


图 7-18 Hadoop 插件操作 HDFS

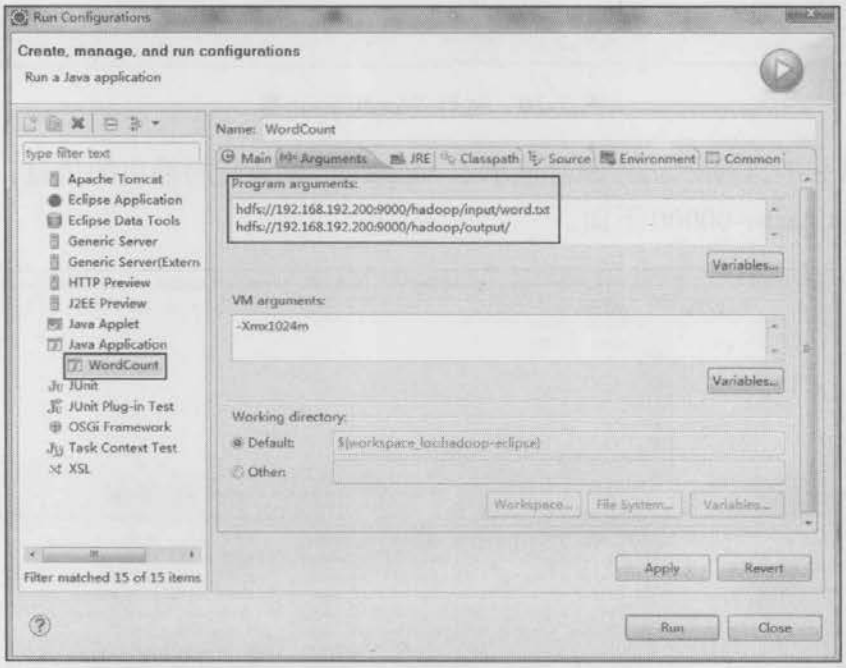


图 7-19 设置 WordCount 的输入和输出

Program arguments 里设置了输入文件和输出文件，一行一个。图中输入的文件是 /hadoop/input/word.txt，把结果文件存放到 /hadoop/output 目录下。

VM arguments 参数设置了 JDK 的堆内存大小，默认为内存的 1/6。例如机器的内存为 2GB，默认的 JDK 堆内存为 341MB。

注意 JDK 默认的内存比较小，在 MapReduce 分析大文件的时候会报内存溢出（Out Of Memory，OOM），这里可以在 VM arguments 里设置 JDK 的内存，以免产生 OOM。

7.2.4 运行 MapReduce 的驱动类

鼠标在类名上右击，选择 Run As → Run on Hadoop 运行 MapReduce 的驱动类，如图 7-20 所示。

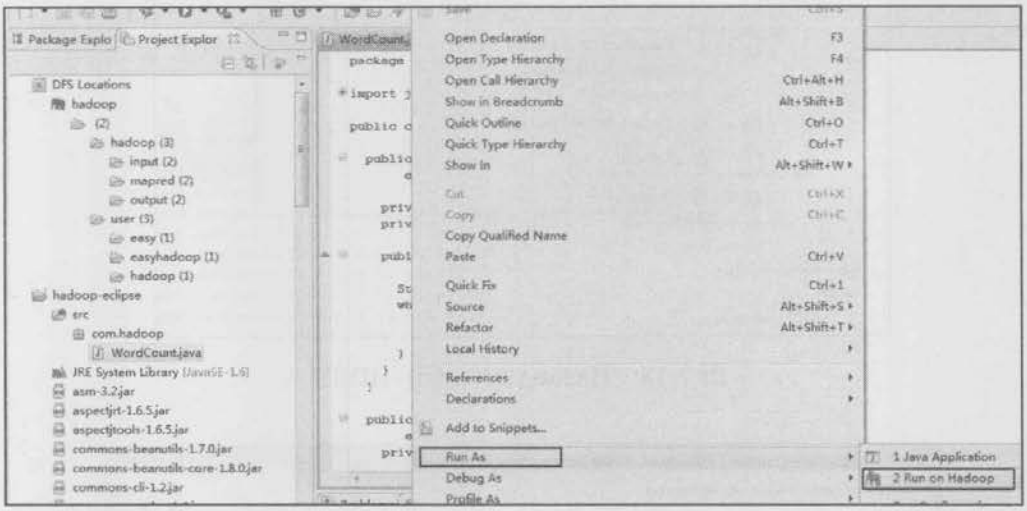


图 7-20 运行 WordCount 类

运行结果如图 7-21 所示。控制台显示了 MapReduce 执行时输出的信息，结果文件存放在 /hadoop/output/part-r-00000 下面。

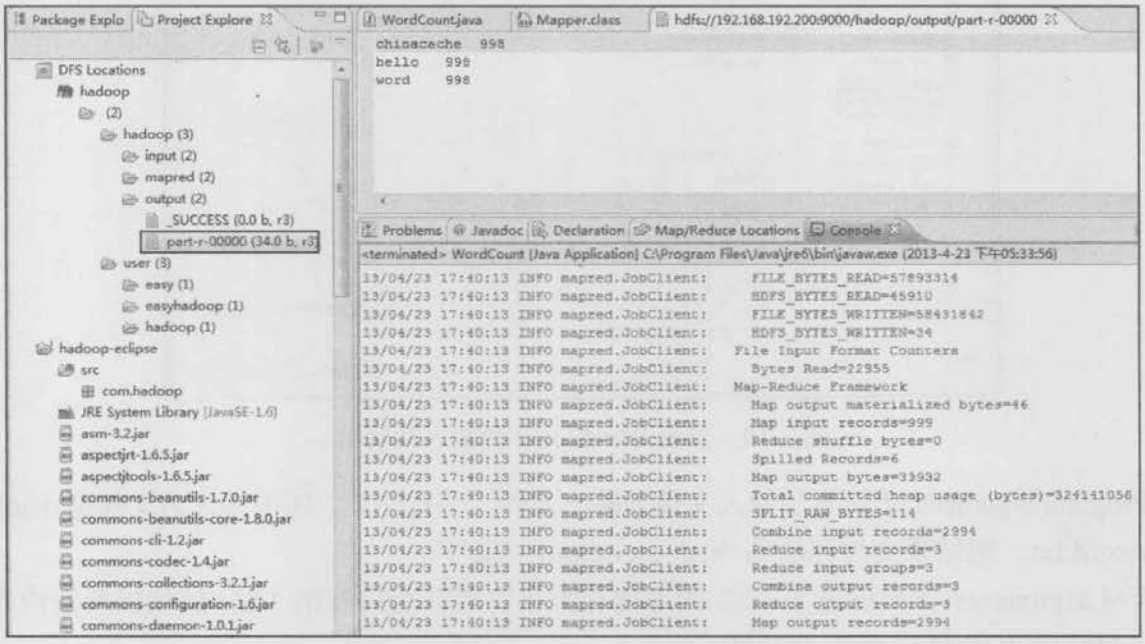


图 7-21 WordCount 运行效果

到这里 Eclipse 的 MapReduce 插件就配置好了，这个插件在实际工作中非常实用，减少了 MapReduce 程序的测试时间，减少了不必要的打包、上传等一系列重复的操作。

7.3 MapReduce 的 Debug 调试

对于程序员看来说 40% ～ 50% 的时间用来编写代码，剩下的时间就是测试了。分布式的代码是最难调试的，而 MapReduce 就是分布式运行的，下面来看 MapReduce 代码怎么用 Debug 来调试。Debug 调试步骤如下所示。

7.3.1 进入 Debug 运行模式

步骤 1 选定一台调试机器，修改 mapred-site.xml 文件，添加如下配置。

```
1. <property>
2.     <name>mapred.child.java.opts</name>
3.     <value>-agentlib:jdwp=transport=dt_socket,address=8883,server=y,
           suspend=y</value>
4. </property>
```

步骤 2 关闭所有的 TaskTracker，只保留上面配置的一台需要调试的 TaskTracker。

步骤 3 启动 MapReduce Job。

步骤 4 添加一个新的测试。

右键 Hadoop 项目，选择“Debug As”→“Debug Configurations”→“Remote Java Application”，添加一个新的测试，输入远程 Host IP 和监听端口，端口为 8883，然后单击 Apply，如图 7-22 所示。

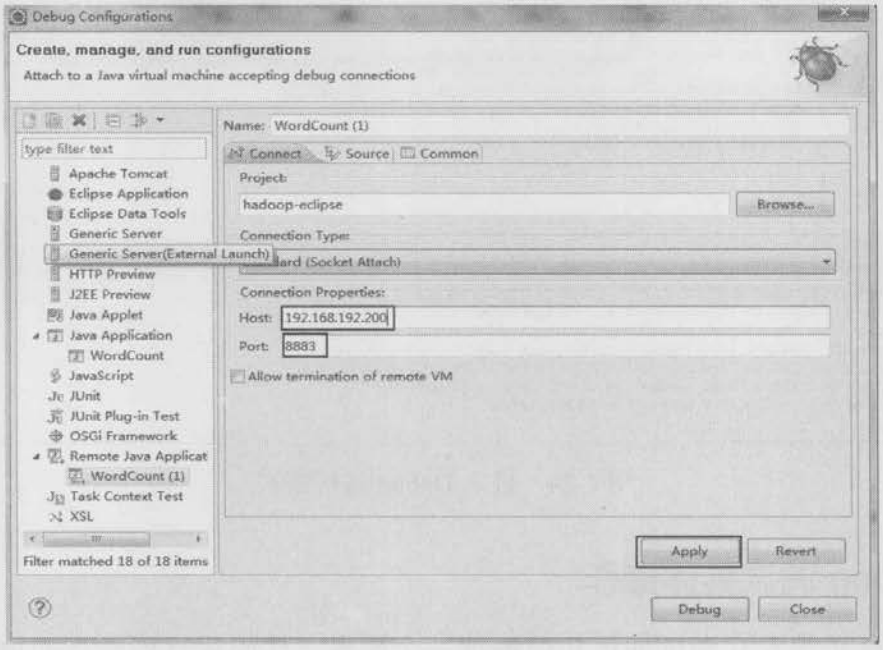


图 7-22 配置 Remote Java Application

步骤 5 运行 WordCount 类。

在 WordCount 类上右击，选择 Debug As → Java Application，如图 7-23 所示。

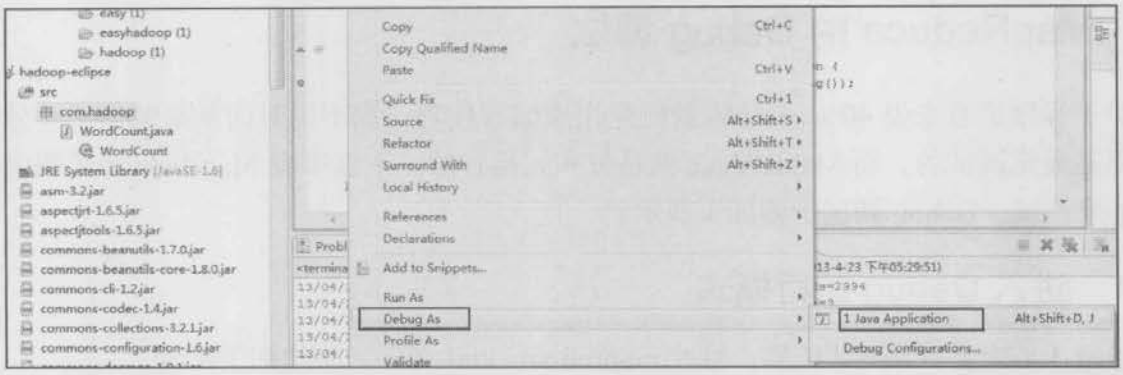


图 7-23 Debug 运行

步骤 6 进入 Debug 运行模式。

单击 Java Application，会弹出一个对话框，如图 7-24 所示，确认是否进入 Debug 模式，选择 Yes，进入 Debug 模式。

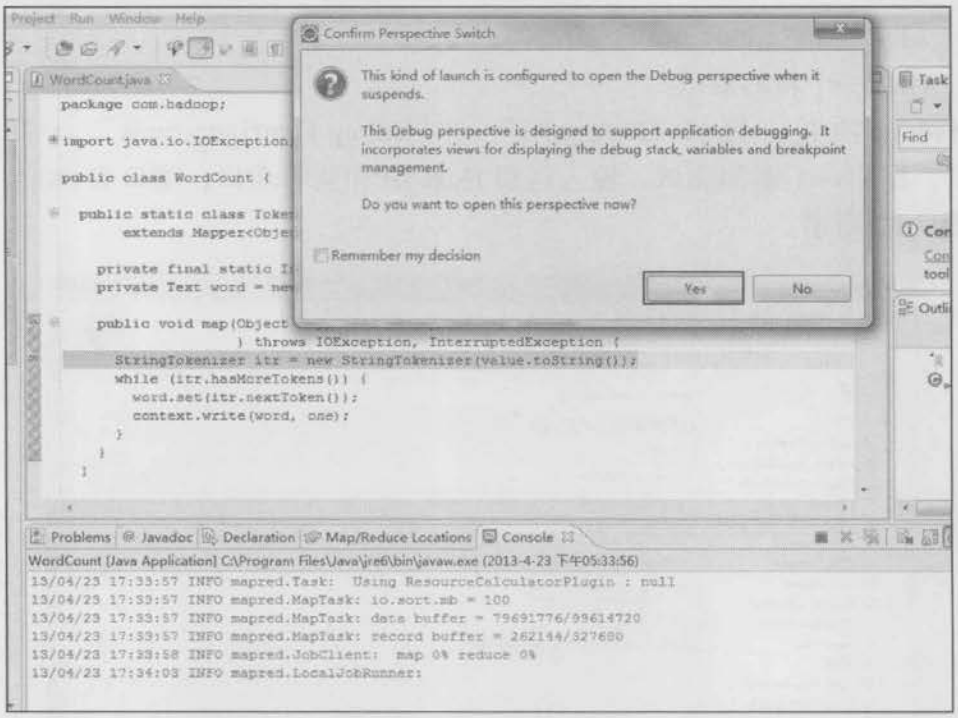


图 7-24 进入 Debug 运行模式

7.3.2 Debug 调试具体操作

Debug 调试必须要在调试的语句前面加一个断点，代码运行到断点的地方就会停止，然后我们就可以一条一条往下运行代码。

图 7-25 显示 map 函数的调试。图的右上角有个变量（Variables）对话框，显示所有变量和值，每执行一步可以查看变量的值和我们期望的是否一样，一样就继续往下执行，不

一样就说明在这行代码附近有问题。例如，图中 key 的值为“chinacache hello world”是期望的值，因为分析的文件第一行内容就是“chinacache hello world”。

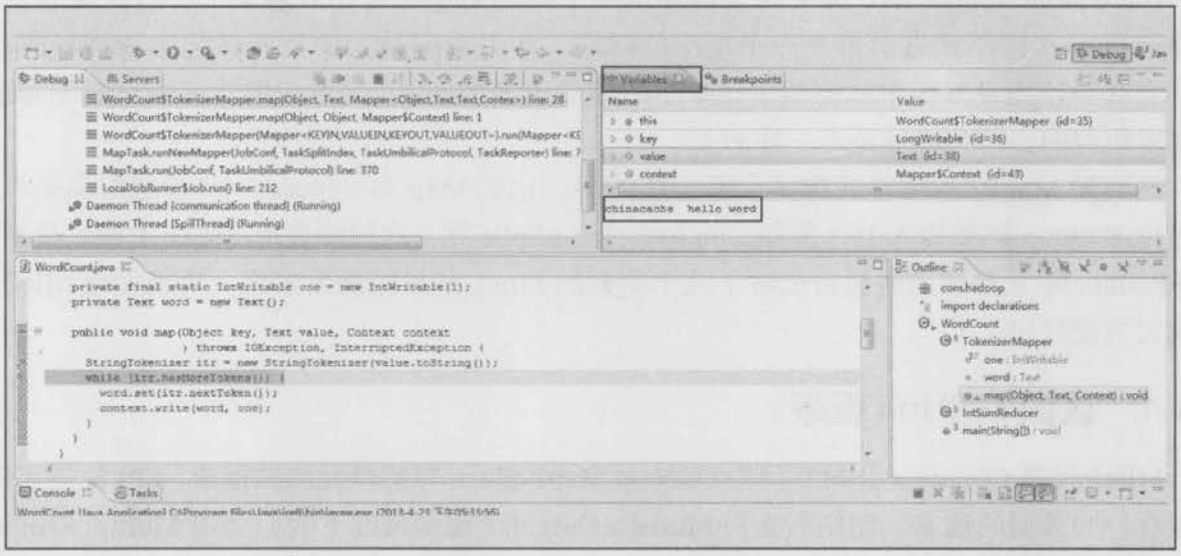


图 7-25 Debug 调试

Reduce 函数的调试如图 7-26 所示。

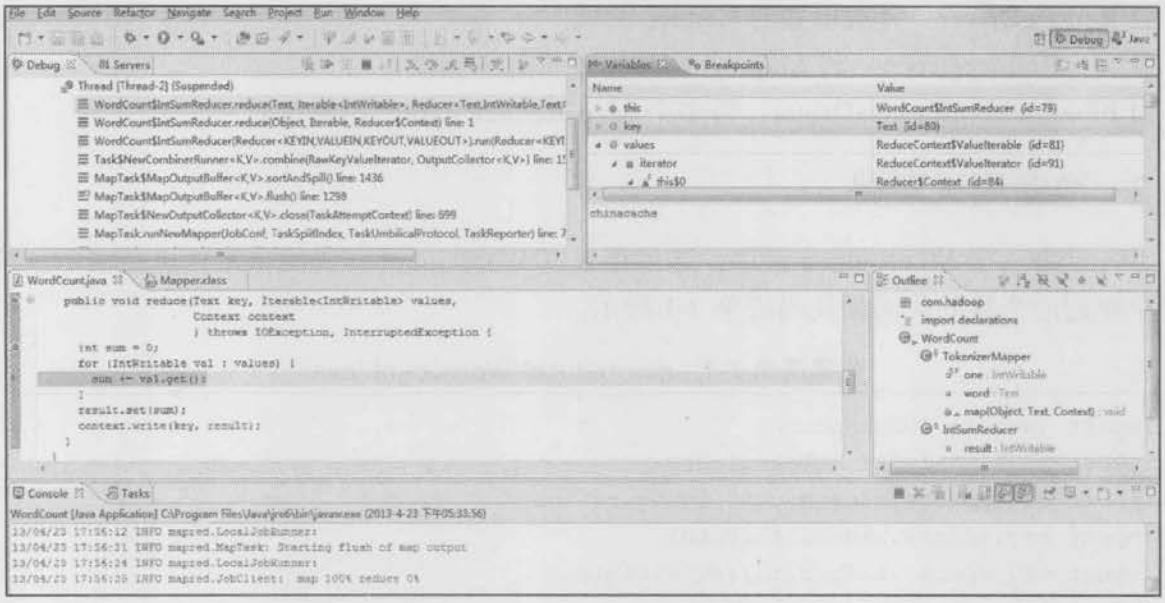


图 7-26 Reduce 代码调试

Reducer 调试和 Mapper 调试一样，从图 7-26 可以看出 key 的值为 chinacache。

MapReduce 代码在 Eclipse 里面的调试就讲完了，读者记得一定要照着步骤做下来。

提示 这里介绍 Eclipse 中的 Debug 操作快捷键。F5 进入方法；F6 下一步；F7 从方法里跳出；F8 调到下个断点，如果没有断点就结束程序。

7.4 单元测试框架 MRUnit

Hadoop 的 MapReduce 提交到集群环境中出问题的定位是比较麻烦的，有时需要一遍遍修改代码和打印日志来排查一个很小的问题，如果数据量大的话调试起来相当耗时间。因此有必要使用良好的单元测试手段来尽早消除明显的 bug（当然仅有单元测试是不够的，毕竟跟集群的运行环境还是不一样的）。

然而做 MapReduce 单元测试会有一些障碍，比如 Map 和 Reduce 一些参数对象是在运行时由 Hadoop 框架传入的，例如 Context、InputSplit 等，这就需要有 Mock 手段。最初写 MapReduce 单元测试的时候自己写了几个简单的 Mock 基本能满足需要，后来发现 MRUnit 比我写的要好用。

7.4.1 认识 MRUnit 框架

MRUnit 是 Cloudera 公司专门为 Hadoop MapReduce 写的单元测试框架，API 简洁实用。但也有一些薄弱的地方，比如不支持 MultipleOutputs（很多情况下我们会用 MultipleOutputs 作为多文件输出，后面将介绍如何增强 MRUnit 使之支持 MultipleOutputs）。

MRUnit 针对不同测试对象分别使用以下几种 Driver。

- MapDriver：针对单独的 Map 测试。
- ReduceDriver：针对单独的 Reduce 测试。
- MapReduceDriver：将 Map 和 Reduce 连贯起来测试。
- PipelineMapReduceDriver：将多个 Map-Reduce pair 贯串测试。

7.4.2 准备测试案例

首先准备一个 MapReduce 程序，这里还是以 WordCount 作为测试案例进行演示。统计文件中单词的个数的代码如代码清单 7-1 所示。

代码清单 7-1 mapreduce\ Wordcount.java

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
```

```
public class Wordcount {
    public static class TokenizerMapper
                extends Mapper<Object, Text, Text, IntWritable>{
      private final static IntWritable one = new IntWritable(1);
      private Text word = new Text();
      public void map(Object key, Text value, Context context )
                    throws IOException, InterruptedException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
          word.set(itr.nextToken());
          context.write(word, one);
        }
      }
    }
    public static class IntSumReducer
         extends Reducer<Text,IntWritable,Text,IntWritable> {
      private IntWritable result = new IntWritable();
      public void reduce(Text key, Iterable<IntWritable> values,
                         Context context
                         ) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
          sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
      }
    }
    public static void main(String[] args) throws Exception {
      Configuration conf = new Configuration();
      String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
      if (otherArgs.length != 2) {
        System.err.println("Usage: wordcount <in> <out>");
        System.exit(2);
      }
      Job job = new Job(conf, "word count");
      job.setJarByClass(Wordcount.class);
      job.setMapperClass(TokenizerMapper.class);
      job.setCombinerClass(IntSumReducer.class);
      job.setReducerClass(IntSumReducer.class);
      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(IntWritable.class);
      FileInputFormat.addInputPath(job, new Path("/user/hadoop/input/word/"));
      FileOutputFormat.setOutputPath(job, new Path("/user/hadoop/output"));
      System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
  }
```

准备好这个类以后，我们开始对这个类进行单元测试。

7.4.3 Mapper 单元测试

Mapper 的逻辑就是把一个字符串转化成 Key-Value 的形式，如字符串“hello hello hadoop”，map 的结果应该为 <"hello", 1>、<"hello", 1> 和 <"hadoop", 1>。

Mapper 的简单例子如代码清单 7-2 所示。

代码清单 7-2 test\ WordCountMapperTest.java

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mrunit.mapreduce.MapDriver;
import org.junit.Before;
import org.junit.Test;
@SuppressWarnings("all")
public class WordCountMapperTest {
  private Mapper mapper;
  private MapDriver driver;
  @Before
  public void init() {
     mapper = new Wordcount.TokenizerMapper();
     driver = new MapDriver(mapper);
  }
  @Test
  public void test() throws IOException {
     String line = "easyhadoop is a great open source of hadoop";
     driver.withInput(null, new Text(line))
           .withOutput(new Text("easyhadoop"), new IntWritable(1))
           .withOutput(new Text("is"), new IntWritable(1))
           .withOutput(new Text("a"), new IntWritable(1))
           .withOutput(new Text("great"), new IntWritable(1))
           .withOutput(new Text("open"), new IntWritable(1))
           .withOutput(new Text("source"), new IntWritable(1))
           .withOutput(new Text("of"), new IntWritable(1))
           .withOutput(new Text("hadoop"), new IntWritable(1))
           .runTest();
  }
}
```

鼠标放在 WordCountMapperTest 类上右击，选择 Run As → JUnit test，运行结果如图 7-27 所示。

测试方法为 test() 方法，左边的对话框里显示“Runs：1/1，Errors：0，Failures：0”说明 Mapper 测试成功了。

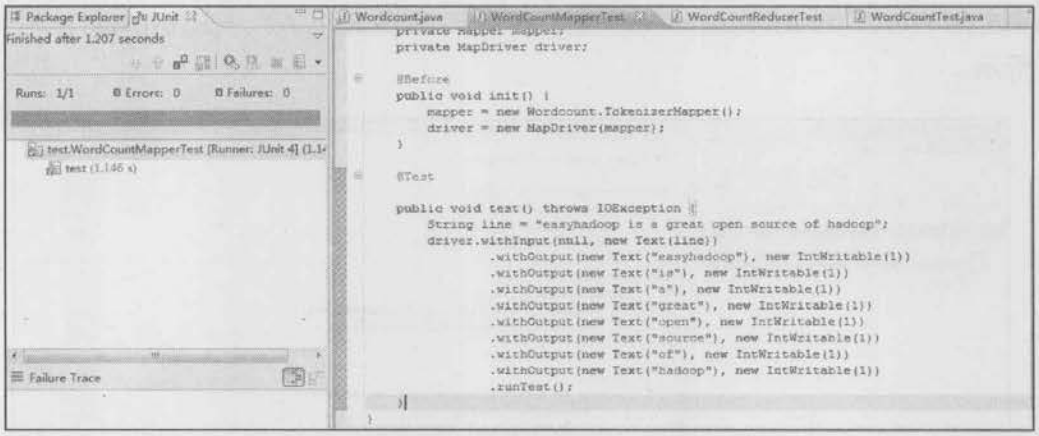


图 7-27 Mapper 函数单元测试

7.4.4 Reducer 单元测试

Reduce 函数的逻辑就是把 Key 相同的 Value 值相加，Reducer 单元测试如代码清单 7-3 所示。

代码清单 7-3 test\ WordCountReducerTest.java

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mrunit.mapreduce.ReduceDriver;
import org.junit.Before;
import org.junit.Test;

@SuppressWarnings("all")
public class WordCountReducerTest {
  private Reducer reducer;
  private ReduceDriver driver;
  @Before
  public void init() {
     reducer = new Wordcount.IntSumReducer();
     driver = new ReduceDriver(reducer);
  }
  @Test
  public void test() throws IOException {
     String key = "easyHadoop";
     List values = new ArrayList();
     values.add(new IntWritable(2));
     values.add(new IntWritable(5));
     driver.withInput(new Text("easyHadoop"), values)
         .withOutput(new Text("easyHadoop"), new IntWritable(7)).runTest();
  }
}
```

鼠标放在 WordCountReducerTest 类上右击，选择 Run As → JUnit test，运行结果如图 7-28 所示。

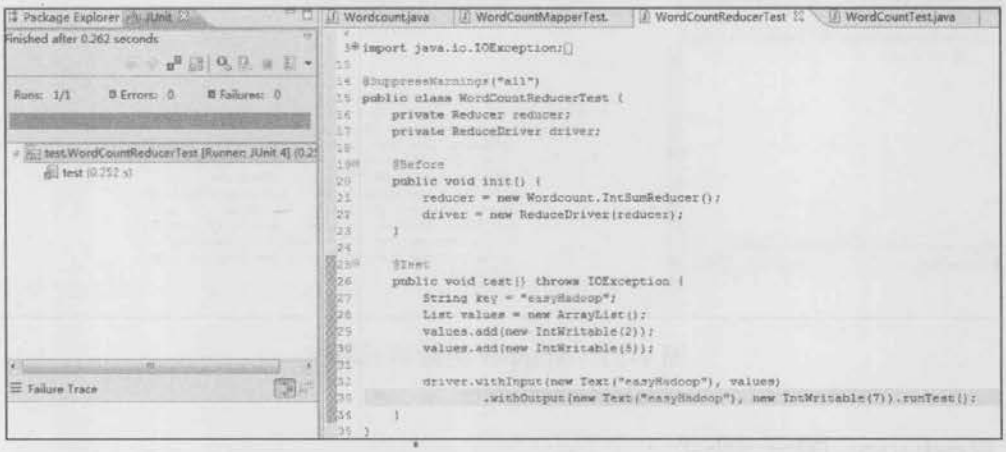


图 7-28 Reducer 单元测试

7.4.5 MapReduce 单元测试

把 Mapper 和 Reducer 的测试集成起来测试案例如代码清单 7-4 所示。

代码清单 7-4 test\ WordCountTest.java

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mrunit.mapreduce.MapReduceDriver;
import org.junit.Before;
import org.junit.Test;

@SuppressWarnings("all")
public class WordCountTest {
  private Mapper mapper;
  private Reducer reducer;
  private MapReduceDriver driver;
  @Before
  public void init() {
     mapper = new Wordcount.TokenizerMapper();
     reducer = new Wordcount.IntSumReducer();
     driver = new MapReduceDriver(mapper, reducer);
  }
  @Test
  public void test() throws RuntimeException, IOException {
     String line = "easyhadoop is a great open source of hadoop is it";
     driver.withInput("", new Text(line))
         .withOutput(new Text("a"), new IntWritable(1))
```

```
            .withOutput(new Text("easyhadoop"), new IntWritable(1))
            .withOutput(new Text("great"), new IntWritable(1))
            .withOutput(new Text("hadoop"), new IntWritable(1))
            .withOutput(new Text("is"), new IntWritable(2))
            .withOutput(new Text("it"), new IntWritable(1))
            .withOutput(new Text("of"), new IntWritable(1))
            .withOutput(new Text("open"), new IntWritable(1))
            .withOutput(new Text("source"), new IntWritable(1))
            .runTest();
    }
}
```

鼠标放在 WordCountTest 类上右击，选择 Run As → JUnit test，运行结果如图 7-29 所示。

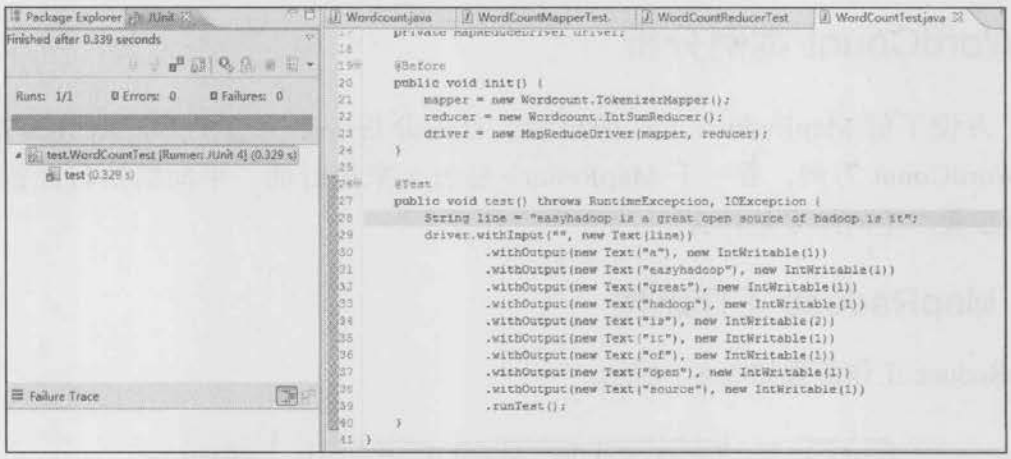


图 7-29 MapReduce 单元测试

注意 MapReduce 输出的 Key 是排序的，所以输出结果要按照 Key 排序，否则测试不通过。

7.5 本章小结

本章主要介绍了 Eclipse 的 MapReduce 插件安装、利用插件调试 MapReduce 代码，以及 MapReduce 单元测试三部分内容。主要目的是提高读者编写 MapReduce 调试、测试的效率。选择和使用一个好的工具是项目成功的一半。希望读者自己动手操作一下。

第8章　MapReduce 编程开发

第 3 章介绍了 MapReduce 的快速入门，大家大概了解开发一个 MapReduce 程序、运行等所需要的步骤，如打包、部署、提交 Job 等操作。第 7 章介绍了 Hadoop 的 Eclipse 插件，主要理解 MapReduce 的打包、部署等一些过程。这个插件为 MapReduce 程序的开发、测试提供了方便，提高工作效率，本章介绍 MapReduce 编程。

8.1　WordCount 案例分析

为了方便了解 MapReduce 工作的机制，首先分析一个完整的 MapReduce 执行流程，还是以 WordCount 为例，看一下 MapReduce 是怎么来执行的，中间的执行经过了那些步骤，每个步骤产生的结果是什么。

8.1.1　MapReduce 工作流程

MapReduce 工作流程如图 8-1 所示。

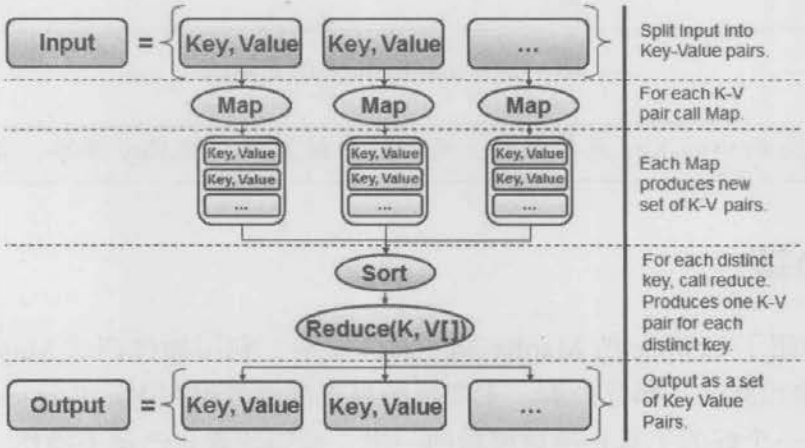


图 8-1　MapReduce 流程图

Input 从 HDFS 里面并行读取文本中的内容，经过 MapReduce 模型，最终把分析出来的结果用 Output 封装，持久化到 HDFS 里面。

WordCount 案例运行的输入和输出如图 8-2 所示。

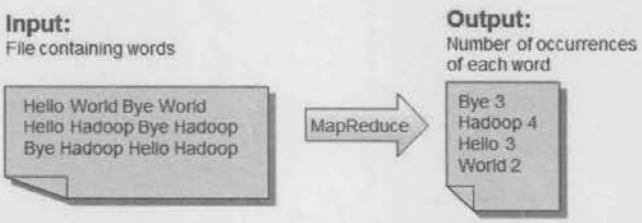


图 8-2　WordCount 案例的输入和输出

8.1.2 WordCount 的 Map 过程

使用三个 Map 任务并行读取三行文件的内容，对读取的单词进行 map 操作，每个词都以 <key,value> 形式生成，如图 8-3 所示。

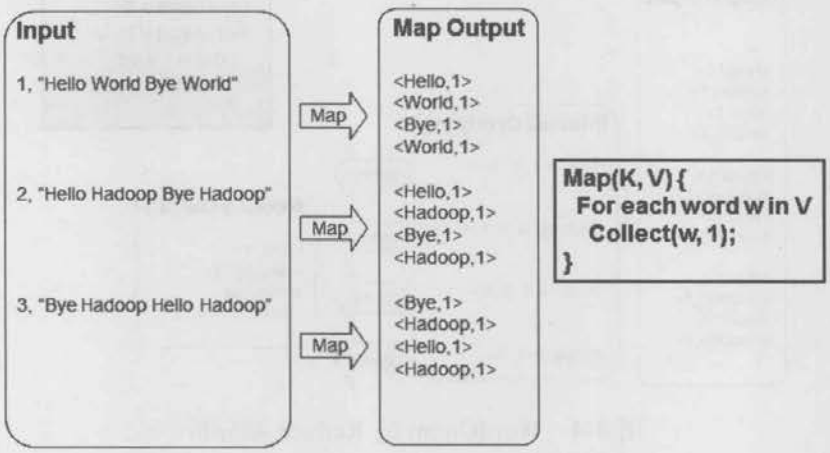


图 8-3 WordCount 的 Map 端分析

Map 端代码如代码清单 8-1 所示。

代码清单 8-1 mapred\ WordMap.java

```
/**
 * WordMap 继续自 Mapper<Object, Text, Text, IntWritable>
 * [ 一个文件就一个 Map，两个文件就会有两个 Map]
 * Map[ 这里读入输入文件内容以 " \t\n\r\f" 进行分割，然后设置 word ==> one 的 Key-Value 对 ]
 * @param Object  Input key Type:
 * @param Text    Input value Type:
 * @param Text    Output key Type:
 * @param IntWritable Output value Type:
 *
 * Writable 的主要特点是使 Hadoop 框架知道对一个 Writable 类型的对象怎样进行 Serialize 以
 * 及 Deserialize
 * WritableComparable 在 Writable 的基础上增加了 compareTo 接口
 * 使得 Hadoop 框架知道怎样对 WritableComparable 类型的对象进行排序
 * @author jayliu
 */
 public  class WordMap
     extends Mapper<Object, Text, Text, IntWritable>{
   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();
   public void map(Object key, Text value, Context context
                   ) throws IOException, InterruptedException {
     StringTokenizer itr = new StringTokenizer(value.toString());
     while (itr.hasMoreTokens()) {
       word.set(itr.nextToken());
       context.write(word, one);
     }
}
```

8.1.3 WordCount 的 Reduce 过程

Reduce 操作是对 Map 的结果进行排序、合并等操作最后得出词频，如图 8-4 所示。

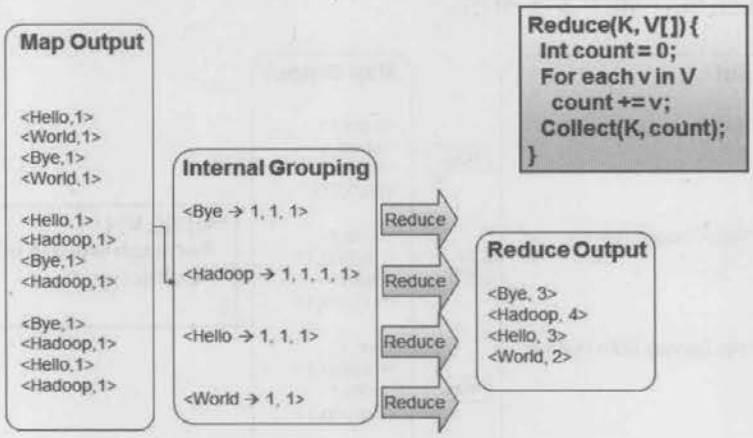


图 8-4 WordCount 的 Reduce 端分析

Reduce 端代码如代码清单 8-2 所示。

代码清单 8-2 mapred\ WordReduce.java

```
/**
  * WordReduce 继承自 Reducer<Text,IntWritable,Text,IntWritable>
  * [ 不管几个 map，都只有一个 reduce，这是一个汇总 ]
  * reduce[ 循环所有的 map 值，把 word ==> one 的 Key-Value 对进行汇总 ]
  * 这里的 Key 为 Mapper 设置的 word[ 每一个 Key-Value 都会有一次 reduce]
  * 循环结束后，context 就是最后的结果
  * @author jayliu
  */
  public   class WordReduce
        extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();
    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val:values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key,result);
  }
}
```

8.1.4 每个过程产生的结果

1）Input 输入。

```
Hello World Bye World
Hello Hadoop Bye Hadoop
Bye Hadoop Hello Hadoop
```

2）Map 结果。

```
<Hello,1>
<World,1>
<Bye,1>
<World,1>
<Hello,1>
<Hadoop,1>
<Bye,1>
<Hadoop,1>
<Bye,1>
<Hadoop,1>
<Hello,1>
<Hadoop,1>
```

3）Sort 结果。

```
<Bye,1>
<Bye,1>
<Bye,1>
<Hadoop,1>
<Hadoop,1>
<Hadoop,1>
<Hadoop,1>
<Hello,1>
<Hello,1>
<Hello,1>
<World,1>
<World,1>
```

4）Combine 结果。

```
<Bye,1,1,1>
<Hadoop,1,1,1,1>
<Hello,1,1,1>
<World,1,1>
```

5）Reduce 结果。

```
<Bye,3>
<Hadoop,4>
<Hello,3>
<World,2>
```

6）MergeSort 的过程。

MergSort 采用了二分法排序的原理。就是 Sort 产生有序的结果，具体分析过程如下。

第一步：用“|”分割符对 Map 的结果进行第一次二分法。

```
<Hello,1><World,1><Bye,1><World,1><Hello,1><Hadoop,1>|
<Bye,1><Hadoop,1><Bye,1><Hadoop,1><Hello,1><Hadoop,1>
```

第二步：用“||”分割符对 Map 的结果进行第二次二分法（在第一次的基础上）。

```
<Hello,1><World,1><Bye,1> || <World,1><Hello,1><Hadoop,1> | <Bye,1><Hadoop,1><Bye,1> ||
<Hadoop,1><Hello,1><Hadoop,1>
```

第三步：用“|||”分割符对 Map 的结果进行第三次二分法（在第二次的基础上）。

```
<Hello,1><World,1> ||| <Bye,1> || <World,1><Hello,1> ||| <Hadoop,1> | <Bye,1><Hadoop,1> |||
<Bye,1> || <Hadoop,1><Hello,1> ||| <Hadoop,1>
```

第四步：MergeArray 第一次排序结果。

```
<Hello,1><World,1> ||| <Bye,1> || <Hello,1><World,1> ||| <Hadoop,1> |
<Bye,1><Hadoop,1> ||| <Bye,1> || <Hadoop,1><Hello,1> ||| <Hadoop,1>
```

第五步：MergeArray 第二次排序结果。

```
<Bye,1><Hello,1><World,1> || <Hadoop,1><Hello,1><World,1> | <Bye,1><Bye,1><Hadoop,1> ||
<Hadoop,1><Hadoop,1><Hello,1>
```

第六步：MergeArray 第三次排序结果。

```
<Bye,1><Hadoop,1><Hello,1><World,1><Hello,1><World,1> |
<Bye,1><Bye,1><Hadoop,1> <Hadoop,1><Hello,1><Hadoop,1>
```

第七步：MergeArray 最终排序结果。

```
<Bye,1><Bye,1><Bye,1><Hadoop,1><Hadoop,1><Hadoop,1><Hadoop,1><Hello,1><Hello,1>
<Hello,1><World,1><World,1>
```

排序完成。

8.1.5 Mapper 抽象类

我们写的 Map 类是继承 Mapper<Object, Text, Text, IntWritable> 抽象类，方法如代码清单 8-3 所示。

代码清单 8-3 mapred\ Mapper.java

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
    public class Context  extends MapContext<KEYIN,VALUEIN,KEYOUT,VALUEOUT> {
      public Context(Configuration conf, TaskAttemptID taskid,
             RecordReader<KEYIN,VALUEIN> reader,
             RecordWriter<KEYOUT,VALUEOUT> writer,
             OutputCommitter committer,
             StatusReporter reporter,
             InputSplit split) throws IOException, InterruptedException {
        super(conf, taskid, reader, writer, committer, reporter, split);
      }
    }
    /**
    * Called once at the beginning of the task.
    */
```

```
    protected void setup(Context context)
                        throws IOException, InterruptedException {
      // NOTHING
    }
    /**
     * Called once for each key/value pair in the input split. Most applications
     * should override this, but the default is the identity function.
     */
    @SuppressWarnings("unchecked")
    protected void map(KEYIN key, VALUEIN value, Context context)
                        throws IOException, InterruptedException {
      context.write((KEYOUT) key, (VALUEOUT) value);
    }
    /**
     * Called once at the end of the task.
     */
    protected void cleanup(Context context)
                            throws IOException, InterruptedException {
    // NOTHING
    }
    /**
     * Expert users can override this method for more complete control over the
     * execution of the Mapper.
     * @param context
     * @throws IOException
     */
    public void run(Context context) throws IOException, InterruptedException {
      setup(context);
      while (context.nextKeyValue()) {
        map(context.getCurrentKey(), context.getCurrentValue(), context);
      }
      cleanup(context);
    }
}
```

我们写 MapReduce 程序的时候，Mapper 都要继承这个 Mapper.class，通常会重写 map 方法，map() 每次接受一个 Key-Value 对，然后对这个 Key-Value 对进行处理，再分发出处理后的数据。可能重写 setup() 以对这个 Map Task 进行一些预处理，比如创建一个 List 等；也可能重写 cleanup() 方法做一些处理后的工作，当然也可能在 cleanup() 中写出 Key-Value 对。举个例子，InputSplit 的数据是一些整数，然后要在 map 中算出它们的和。可以先设置 sum 属性，然后 map 函数处理一个 Key-Value，就是将其加到 sum 上，最后在 cleanup() 函数中调用 context.write(key,value)。

Mapper.class 中的 run() 方法相当于 Map Task 的驱动，可以看到 run() 方法首先调用 setup() 进行初始操作，然后对每个 context.nextKeyValue() 获取的 Key-Value 对，就调用 map() 函数进行处理，最后调用 cleanup() 做最后的处理。事实上，text.nextKeyValue() 就是使用了相应的 RecordReader 来获取 Key-Value 对的。

run() 方法提供了 setup()->map()->cleanup() 的执行模板。

Mapper<Object, Text, Text, IntWritable> 抽象类的四个参数中，前两个为 map() 函数的输入，后两个为 map() 函数的输出。下面看 map() 方法。

```
protected void map(Object key, Value value, Context context)
        throws IOException, InterruptedException {
    context.write((KEYOUT) key, (VALUEOUT) value);
}
```

Object key 参数表示偏移量，Value value 表示一行文本的值，Context context 参数为 MapReduce 模型 Map 端的上下文对象。例如，输入文件 test.txt 的内容如下。

```
hello world
easy hadoop
```

map() 方法被执行两次。

- 第一次：Object key 的值为 0，Value value 值为 hello world。
- 第二次：Object key 的值为 11，Value value 值为 easy hadoop。

8.1.6 Reducer 抽象类

我们写的 Reduce 类是继承 Reducer<Text, IntWritable, Text, IntWritable > 抽象类，方法如代码清单 8-4 所示。

代码清单 8-4 mapred\Reduce.java

```
public class Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT> {
  public class Context extends ReduceContext<KEYIN,VALUEIN,KEYOUT,VALUEOUT> {
    public Context(Configuration conf, TaskAttemptID taskid,
           RawKeyValueIterator input,
           Counter inputKeyCounter,
           Counter inputValueCounter,
           RecordWriter<KEYOUT,VALUEOUT> output,
           OutputCommitter committer,
           StatusReporter reporter,
           RawComparator<KEYIN> comparator,
           Class<KEYIN> keyClass,
           Class<VALUEIN> valueClass
           ) throws IOException, InterruptedException {
      super(conf, taskid, input, inputKeyCounter, inputValueCounter,
           output, committer, reporter,
           comparator, keyClass, valueClass);
    }
  }
  /**
   * Called once at the start of the task.
   */
```

```
    protected void setup(Context context)
                        throws IOException, InterruptedException {
      // NOTHING
    }
    /**
      * This method is called once for each key. Most applications will define
      * their reduce class by overriding this method. The default implementation
      * is an identity function.
      */
      @SuppressWarnings("unchecked")
      protected void reduce(KEYIN key, Iterable<VALUEIN> values, Context context)
                            throws IOException, InterruptedException {
        for(VALUEIN value: values) {
          context.write((KEYOUT) key, (VALUEOUT) value);
        }
    }
    /**
      * Called once at the end of the task.
      */
    protected void cleanup(Context context)
                           throws IOException, InterruptedException {
    // NOTHING
  }
  /**
    * Advanced application writers can use the
    * {@link #run(org.apache.hadoop.mapreduce.Reducer.Context)} method to
    * control how the reduce task works.
    */
  public void run(Context context) throws IOException, InterruptedException {
    setup(context);
    while (context.nextKey()) {
      reduce(context.getCurrentKey(), context.getValues(), context);
    }
    cleanup(context);
  }
}
```

Reducer<Text, IntWritable, Text, IntWritable> 抽象类里面的方法和 Mapper<Object, Text, Text, IntWritable> 有点相似，除了 reduce() 方法不一样外，其他的都一样，功能也一样。

Reducer<Text, IntWritable, Text, IntWritable> 抽象类解释如下。前面个参数为 Reduce 的输入（即 Map 的输出），后两个参数为 Reduce 的输出。

功能解释如下。

1）获取 map 函数的中间结果。

2）将中间结果中的 Value 按 Key 划分组（group），而 group 按照 Key 排序。形成了 <key, (collection of values)> 的结构，此时 Key 是唯一的。

3）处理 group 中的所有 Value，相同 Key 的 Value 相加。最终 Key 对应的 Value 唯一，<key, value> 序对形成。

reduce() 方法如下。

```
protected void reduce(Text key, Iterable<IntWritable> values, Context context)
                throws IOException, InterruptedException {
for(Iterable<IntWritable> value: values) {
  context.write(Text key, IntWritable value);
}
```

Text key 参数表示 Map 端输出 Key 的值。Iterable<IntWritable> values 参数表示 Key 相同的值，Value 的集合。Context context 参数表示 MapReduce 的 Meduce 端上下文。

例如 Map 端输出的结果如下。

```
<hello,1>
<hello,1>
<hello,1>
<hello,1>
```

reduce() 方法中 Text key 的值为 hello，Iterable<IntWritable> values 的值为 [1,1,1,1]。

8.1.7 MapReduce 驱动

Hadoop MapReduce 程序的最后一个组件叫做 Driver，简单地说就是 Java 的 Main 函数。它会初始化 Job 和指示 Hadoop 平台在输入文件集合上执行你的代码，并控制输出文件的放置地址。如代码清单 8-5 所示。

代码清单 8-5 mapred\WordMain.java

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
public class WordMain {
  public static void main(String[] args) throws Exception {
      Configuration conf = new Configuration();
      String[] otherArgs = new GenericOptionsParser(conf, args).
      getRemainingArgs();
      /**
       * 这里必须有输入 / 输出
       */
      if (otherArgs.length != 2) {
        System.err.println("Usage: wordcount <in> <out>");
        System.exit(2);
      }
      Job job = new Job(conf, "word count");
      job.setJarByClass(WordMain.class);                    // 主类
      job.setMapperClass(WordMapper.class);                 // mapper
```

```
        job.setCombinerClass(WordReducer.class);              //作业合成类
        job.setReducerClass(WordReducer.class);               //reducer
        job.setOutputKeyClass(Text.class);                    //设置作业输出数据的关键类
        job.setOutputValueClass(IntWritable.class);           //设置作业输出值类
        FileInputFormat.addInputPath(job, new Path(otherArgs[0])); //文件输入
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1])); //文件输出
        System.exit(job.waitForCompletion(true) ? 0 : 1);     //等待完成退出
        }
}
```

main() 方法建立了一个在给定输入文件夹（InputPath 参数）里的文件上执行词频统计程序的作业（Job）。Reduce 的输出被写到 OutputPath 指定的文件夹内，用于运行 Job 的配置信息保存在 Job 对象里。通过 setMapperClass() 和 setReducerClass() 方法可以设定 map 和 reduce 函数。

Reduce 生成的数据类型由 setOutputKeyClass() 和 setOutputValueClass() 方法设定。默认情况下，假定这些也是 Map 的输出数据类型。如果想设定不同的数据格式的话，可以通过 Job 的 setMapOutputKeyClass() 和 setMapOutputValueClass() 方法设定。

Map 的输入数据类型由 InputFormat 控制。默认的输入格式是“TextInputFormat”，它会以（LongWritable，Text）键值对的方式加载数据。LongWritable 值表示某一行在文件中的偏移量，Text 对象则保存某一行的字符串内容。

通过调用 Job.runJob(conf) 即可向 MapReduce 提交 Job，这个调用会阻塞直到 Job 完成。如果 Job 失败了，它会抛出一个 IOException。

8.1.8 MapReduce 最小驱动

如果没有指定 Mapper 类和 Reducer 类就运行 MapReduce，会发生什么情况呢？下面运行一个没有 Mapper 和 Reducer 的最小 MapReduce 程序，看看结果会是什么？假如输入一个文件 file1.txt，文件的内容如下。

```
Hello world
Hello hadoop
```

MapReduce 最小驱动实现如代码清单 8-6 所示。

代码清单 8-6　mapred\MiniMapReduceDriver.java

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
public class MiniMapReduceDriver {
  public static void main(String[] args) throws Exception {
     Configuration conf = new Configuration();
     String[] otherArgs = new GenericOptionsParser(conf, args).
```

```
        getRemainingArgs();
        /**
         * 这里必须有输入/输出
         */
        if (otherArgs.length != 2) {
          System.err.println("Usage: wordcount <in> <out>");
          System.exit(2);
        }
        Job job = new Job(conf, "MiniDriver");              //设置Job的名称
        job.setJarByClass(MiniMapReduceDriver.class);       //主类
        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));//文件输入
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));//文件输出
        System.exit(job.waitForCompletion(true) ? 0 : 1);  //等待完成退出
      }
    }
```

运行 MiniMapReduceDriver 命令如下：

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop MiniMapReduceDriver  /user/hadoop/
input/file1.txt  /user/hadoop/output
```

查看结果：

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs -text /user/hadoop/output/
part-r-00000
0       Hello world
11      Hello hadoop
```

从输出的结果可以看出，第一行输出的是整数，其次是 Tab 键，然后是原来数据的记录。这个案例虽然没有实际的使用价值，但是对我们理解 Hadoop 的默认 MapReduce 作业是很有用的。代码清单 8-7 与代码清单 8-6 完成的功能一模一样，但是它显式地把作业环境设置成默认值。

最小的 MapReduce 驱动程序默认值显示设置如代码清单 8-7 所示。

代码清单 8-7 mapred\MiniMapReduceDriverDefault.java

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class MiniMapReduceDriver {
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    /**
     * 这里必须有输入/输出
     */
    if (otherArgs.length != 2) {
```

```
            System.err.println("Usage: wordcount <in> <out>");
            System.exit(2);
        }
        Job job = new Job(conf, "MiniDriver");          //设置 Job 的名称
        job.setJarByClass(MiniMapReduceDriver.class); //主类
            conf.setInputFormat(TextInputFormat.class);
            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);
        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));//文件输入
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));//文件输出
        System.exit(job.waitForCompletion(true) ? 0 : 1);//等待完成退出
    }
}
```

运行结果和代码清单 8-6 运行结果是一样的。加粗部分代码是执行 MapReduce 作业最关键的，接下来我们逐一介绍每个默认类的含义。

默认的输入格式是 TextInputFormat。它产生的键类型是 LongWritable（文件中每行文件的偏移量值），值类型是 Text（文本行），这也解释了最后输出的整数的含义：行的偏移量。

默认的 Mapper 类型是 IdentityMapper，它将输入的键和值原封不动地写到输出中：

```
public class IdentityMapper<K, V>
    extends MapReduceBase implements Mapper<K, V, K, V> {
    /* The identify function. Input key/value pair is written directly tooutput.*/
  public void map(K key, V val,
                  OutputCollector<K, V> output, Reporter reporter)
                  throws IOException {
    output.collect(key, val);
  }
}
```

IdentityMapper 是一个泛型类型（generic type），可以接受任意键和值的类型，只需要 Map 输入和输出的键类型相同，输入和输出类型的值相同就 ok 了。在上面这个例子中，Map 的输出键是 LongWritable 类型，Map 的输出值是 Text 类型。

提示 读者看到 public void map(K key, V val, OutputCollector<K, V> output, Reporter reporter) 方法的时候，会想与 public void map(K key, V val,Context context) 有什么不一样。其实没有什么不一样，OutputCollector<K, V> output 和 Reporter reporter 两个参数是 Hadoop-0.20 之前版本用的 API，到 Hadoop-0.20 版本以后就把这两个参数合并成一个参数，即 Context context。本书全部使用新的 API 来讲解。

Map 任务是由 MapRunner 负责运行的，MapReunner 是 MapRunnable 的默认实现，它顺序地为每一条记录调用一次 Mapper 的 map() 方法。

默认的 partitioner 是 HashPartitioner，它对每一条记录的键进行哈希操作已决定该记录应该属于哪个分区。每个分区对应一个 Reduce 任务，所以分区的个数等于 Reduce 的个数，默认 HashPartitioner 的代码如下。

```
public class HashPartitioner<K2, V2> implements Partitioner<K2, V2> {
  public void configure(JobConf job) {}
  /** Use {@link Object#hashCode()} to partition. */
  public int getPartition(K2 key, V2 value,int numReduceTasks) {
    return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
  }
}
```

键的哈希码被转换为一个非负整数，它由哈希值与最大的整型值做一次按位与操作而获得，然后用分区数进行取模操作，来决定该记录属于哪个分区的索引。

默认情况下，只有一个 Reduce，因此只有一个分区，在这种情况下，partitioner 操作将由于所有数据都已放入同一个分区而无关紧要了。然而，如果有很多 Reduce，了解 HashPartitioner 的作用是非常重要。

默认的 Reduce 类是 IdentityReducer，它也是一个泛型，简单地将所有的输入写到输出中，默认的 IdentityReducer 代码如下。

```
public class IdentityReducer<K, V>
    extends MapReduceBase implements Reducer<K, V, K, V> {
  /** Writes all keys and values directly to output. */
  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，输出的键是 LongWritable 类型，而值是 Text 类型。事实上，对于 MapReduce 程序来说，所有键都是 LongWritable 类型，所有值都是 Text 类型。

下面用表格来描述 MapReduce 最小驱动的默认设置，如表 8-1 所示。

表 8-1 MapReduce 最小驱动的默认设置

类 型	默 认 值	类 型	默 认 值
InputFormat（输入）	TextInputFOrmat	PartitionerClass	HashPartitioner
MapperClass（Map 类）	IdentityMapper	ReduceClass	IdentityReduce
MapRunnerClass（Map 启动类）	MapRunner	OutputKeyClass	LongWritable
MapOutputKeyClass	LongWritable	OutputValueClass	Text
MapOutputValueClass	Text	OutputFormatClass	TextOutputFormat

8.2 输入格式

文件是 MapReduce 任务数据的初始存储地。正常情况下，输入文件一般是存在于 HDFS 里。这些文件的格式可以是任意的；我们可以使用基于行的日志文件，也可以使用二进制格式，多行输入记录或其他一些格式。这些文件会很大，有的可能数十 GB 或更大。

8.2.1 InputFormat 接口

InputFormat 接口决定了输入文件如何被 Hadoop 分块（split up）与接受。

1. InputFormat 类的层次结构

InputFormat 类的层次结构如图 8-5 所示。

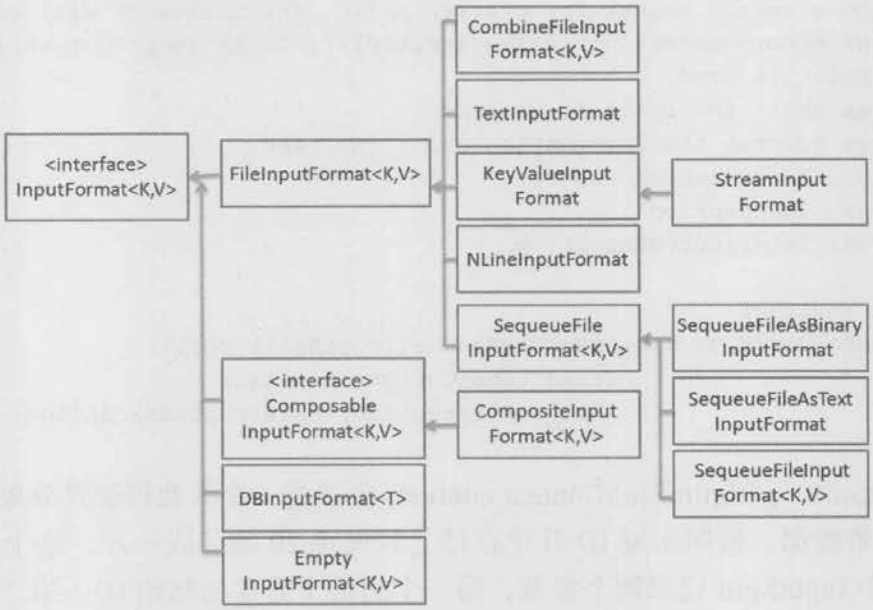


图 8-5 InputFormat 类的层次结构

TextInputFormat 是 InputFormat 的默认实现，对于输入数据中没有明确的 Key-Value 时非常有效，TextInputFormat 返回的 Key 表示这行数据的偏移量，Value 为这行的内容。

MapReduce 程序如果用 TextInputFormat 作为输入格式，可以这样做：

```
conf.setInputFormat(TextInputFormat.class);
```

2. InputFormat 的抽象类方法

InputFormat 的抽象类方法如下。

```
public abstract class InputFormat<K, V> {
  /**
   * Logically split the set of input files for the job.
   *
   * <p>Each {@link InputSplit} is then assigned to an individual {@link Mapper}
```

```
     * for processing.</p>
     *
     * <p><i>Note</i>: The split is a <i>logical</i> split of the inputs and the
     * input files are not physically split into chunks. For e.g. a split could
     * be <i>&lt;input-file-path, start, offset&gt;</i> tuple. The InputFormat
     * also creates the {@link RecordReader} to read the {@link InputSplit}.
     *
     * @param context job configuration.
     * @return an array of {@link InputSplit}s for the job.
     */
     public abstract
      List<InputSplit> getSplits(JobContext context)
                               throws IOException, InterruptedException;

    /**
     * Create a record reader for a given split. The framework will call
     * {@link RecordReader#initialize(InputSplit, TaskAttemptContext)} before
     * the split is used.
     * @param split the split to be read
     * @param context the information about the task
     * @return a new record reader
     * @throws IOException
     * @throws InterruptedException
     */
     public abstract
      RecordReader<K,V> createRecordReader(InputSplit split,
                              TaskAttemptContext context)
                              throws IOException, InterruptedException;
}
```

List<InputSplit> getSplits(JobContext context) 负责将一个大数据逻辑分成多片。比如数据库表有 100 条数据，按照主键 ID 升序存储。假设每 20 条分成一片，这个 List 的大小就是 5，然后每个 InputSplit 记录两个参数，第一个为这个分片的起始 ID，第二个为这个分片数据的大小，这里是 20。很明显 InputSplit 并没有真正存储数据。只是提供了一个如何将数据分片的方法

RecordReader<K,V> createRecordReader(InputSplit split, TaskAttemptContext context) 根据 InputSplit 定义的分片方法，返回一个能够读取分片记录的 RecordReader。getSplits 用来获取由输入文件计算出来的 InputSplit，后面会看到计算 InputSplit 时，会考虑输入文件是否可分割、文件存储时分块的大小和文件大小等因素；而 createRecordReader() 提供了前面所说的 RecordReader 的实现，将 Key-Value 对从 InputSplit 中正确读出来，比如 LineRecordReader 以偏移值为 Key，每行的数据为 Value，这使所有 createRecordReader() 返回 LineRecordReader 的 InputFormat 都是以偏移值为 Key，一行数据为 Value 的形式读取输入分片的。

3. InputFormat 输入格式

Hadoop 自带了许多输入格式。其中有一个抽象类 FileInputFormat，所有操作文件的 InputFormat 类都是从这里继承功能和属性。当开启 Hadoop 作业时，FileInputFormat 会得

到一个路径参数，这个路径内包含了所需要处理的文件，FileInputFormat 会读取这个文件夹内的所有文件（默认不包括子文件夹内的），然后它会把这些文件拆分成一个或多个的 InputSplit。可以通过 JobConf 对象的 setInputFormat() 方法来设定应用到你的作业输入文件上的输入格式。表 8-2 给出了一些标准的输入格式。

表 8-2 InputFormat 的输出格式

输入格式	功能描述	键类型	值类型
TextInputFormat	默认格式，读取文件的行	行的字节偏移量（LongWriable）	行的内容（Text）
KeyValueInputFormat	把行解析为键值对	第一个 Tab 字符前的所有字符（Text）	行剩下的内容（Text）
SequenceFileInputFormat	Hadoop 定义的高性能二进制格式	用户自定义	用户自定义

默认的输入格式是 TextInputFormat，它把输入文件每一行作为单独的记录，但不做解析处理。这对那些没有被格式化的数据或是基于行的记录来说是很有用的，比如日志文件。

更有趣的一个输入格式是 KeyValueInputFormat，这个格式也是把输入文件每一行作为单独的记录。不同的是 TextInputFormat 把整个文件行当做值数据，KeyValueInputFormat 通过搜寻 Tab 字符把行拆分为键值对，把一个 MapReduce 的作业输出作为下一个作业的输入时特别有用，因为默认输出格式（下面有更详细的描述）正是按 KeyValueInputFormat 格式输出数据。

最后讲讲 SequenceFileInputFormat，它会读取特殊的特定于 Hadoop 的二进制文件，这些文件包含了很多能让 Hadoop 的 Map 快速读取数据的特性。Sequence 文件是块压缩的，提供了对几种数据类型（不仅仅是文本类型）直接的序列化与反序列化操作。Squence 文件可以作为 MapReduce 任务的输出数据，用它做一个 MapReduce 作业到另一个作业的中间数据是很高效的。

8.2.2 InputSplit 类

一个输入块描述了构成 MapReduce 程序中单个 Map 任务的一个单元。把一个 MapReduce 程序应用到一个数据集上，也就是说一个作业会由几个（也可能几百个）任务组成。Map 任务可能读取整个文件，但一般是读取文件的一部分。

默认情况下，FileInputFormat 及其子类会以 64MB（与 HDFS 的 Block 默认大小相同，Hadoop 建议 Split 大小与此相同）为基数来拆分文件。可以在 hadoop-site.xml（0.20.* 版本以后是在 mapred-default.xml 里）文件内设定 mapred.min.split.size 参数来控制具体划分大小，或者在具体 MapReduce 作业的 JobConf 对象中重写这个参数。

通过以块形式处理文件，可以让多个 Map 任务并行地操作一个文件。如果文件非常大，这个特性可以通过并行处理大幅提升性能。更重要的是，因为多个块（Block）组成的文件可能会分散在集群内的好几个节点上（事实上就是这样），这样就可以把任务调度在不同的节点上；因此所有的单个块都是本地处理的，而不是把数据从一个节点传输到另外

一个节点。当然，日志文件能够以块处理方式进行处理，但是有些文件格式不支持块处理方式。针对这种情况，可以写一个自定义的 InputFormat，这样就可以控制文件如何被拆分（或不拆分）成文件块。

输入格式定义了组成 Mapping 阶段的 Map 任务列表，每一个任务对应一个输入块。接着根据输入文件块所在的物理地址，这些任务会被分派到对应的系统节点上，可能会有多个 Map 任务被分派到同一个节点上。任务分派好后，节点开始运行任务，尝试去最大并行化执行。节点上的最大任务并行数由 mapred.tasktracker.map.tasks.maximum 参数控制。

我们知道 Map 的输入是一个一个的输入分片，称为 InputSplit。InputSplit 是一个抽象类，它在逻辑上包含了提供给处理这个 InputSplit 的 Mapper 的所有 Key-Value 对。InputSplit 的抽象类里面的抽象方法如下。

```
public abstract class InputSplit {
  public abstract long getLength() throws IOException, InterruptedException;

  public abstract
    String[] getLocations() throws IOException, InterruptedException;
}
```

- getLength() 用来获取 InputSplit 的大小，以支持对 InputSplits 进行排序。
- getLocations() 用来获取存储分片的位置列表。

1. InputSplit 的子类

InputSplit 的子类有 FileSplit、CombineFileSplit 等，下面分别介绍这两个子类。

FileSplit 是由文件路径、分片开始位置、分片大小和存储分片数据的 host 列表组成，根据这些信息我们就可以从输入文件中切分出提供给单个 Mapper 的输入数据。这些属性由 Constructor 设置，我们在后面介绍在 InputFormat 的 getSplits() 中构造这些分片。

CombineFileSplit 同样包含文件路径、分片起始位置、分片大小和存储分片数据的 host 列表，由于 CombineFileSplit 是针对小文件的，它把很多小文件包在一个 InputSplit 内，这样一个 Mapper 就可以处理很多小文件。

要知道我们上面介绍的 FileSplit 是对应一个输入文件的，也就是说，如果用 FileSplit 对应的 FileInputFormat 作为输入格式，即使文件特别小，也是单独计算成一个输入分片来处理的。当输入是由大量小文件组成时，就会导致有同样大量的 InputSplit，从而需要同样大量的 Mapper 来处理，每个 Mapper 的启动需要消耗一定的系统资源，大量的 Mapper 启动会导致 MapReduce 的性能很慢，想想有一堆 Map Task 要运行！这是不符合 Hadoop 的设计理念的，Hadoop 是为处理大文件优化的。

2. 文件避免切分

有些应用程序不希望文件被切分，而是用一个 Mapper 完整地处理每一个输入文件，例如检查一个文件中所有记录是否有序，一个简单的方法就是顺序扫描这个文件每一行记录，并且比较后一条是否比前一条小，如果将它实现为一个 Map 任务，那么只有一个 Map 操作

整个文件时，这个算法才可行。有两种方法可以实现。

第一种方法是设置最小分片的大小，将它设置成大于要处理的最大文件大小，把它设置成最大值 Long.MAX_VALUE 即可。

第二种方法是使用 FileInputFormat 的子类，并且重载 isSplitable() 方法，把返回值设成 false，例如以下就是一个不可分割的 TextInputFormat。

```
import org.apache.hadoop.fs.*
import org.apache.hadoop.mapred.TextInputFormat;
public class NonSplittableTextInputFormat extends TextInputFormat{
  @override
  protected Boolean isSplittable(FileSystem fs,Path path){
   Return false;
   }
}
```

文件输入分片属性如表 8-3 所示。

表 8-3 文件输入分片属性

属性名称	类 型	功能描述
map.input.file	String	输入文件的路径
map.input.start	Long	分片开始的字节偏移量
map.input.length	Long	分片的长度

8.2.3 RecordReader 类

InputSplit 定义了如何切分工作，但是没有描述如何去访问它。RecordReader 类则是实际用来加载数据并把数据转换为适合 Mapper 类里面 map() 方法读取的键值对。

RecordReader 实例是由输入格式定义的，默认的输入格式为 TextInputFormat，提供了一个 LineRecordReader，这个类会把输入文件的每一行作为一个新值，关联到每一行的键则是该行在文件中的字节偏移量。

RecordReader 会在输入块上被重复调用，直到整个输入块被处理完毕，每一次调用 RecordReader 都会调用 Mapper 的 map() 方法。RecordReader 的接口如下。

```
public interface RecordReader<K, V> {
  boolean next(K key, V value) throws IOException;
  K createKey();
  V createValue();
  long getPos() throws IOException;
  public void close() throws IOException;
  float getProgress() throws IOException;
}
```

自定义一个 RecordReader 类需要实现 RecordReader 接口，主要实现接口的 next() 方法。

比如 LineRecordReader 继承了 RecordReader<LongWritable, Text>,TextInputFormat 使用了这个 Reader。KeyValueLineRecordReader 用在了 KeyValueTextInputFormat 类中。

8.2.4 应用实例：随机生成 100 个小数并求最大值

在编写 MapReduce 的时候，自带的输入格式有时候满足不了我们的需求，这个时候就要自己来定制 InputFormat、InputSplit 和 RecordReader。

MapReduce 自带的输入类型都是基于 HDFS 的，本实例的功能是，不用从 HDFS 上面读取输入的内容，在内存里面随机生成 100 个 (0-1)float 型的小数，然后求这 100 个小数的最大值，这个案例很具有代表性，下面我们看看代码的实现。

1. 自定义 InputFormat 类

定义一个 FindMaxValueInputFormat 类继承 InputFormat 抽象类，如代码清单 8-8 所示。实现里面的以下两个方法。

- List<InputSplit> getSplits(JobContext context)
- RecordReader<K,V> createRecordReader(InputSplit split, TaskAttemptContext context)

代码清单 8-8 mapred\output\ FindMaxValueInputFormat.java

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import org.apache.hadoop.io.ArrayWritable;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.InputFormat;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.JobContext;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
public class FindMaxValueInputFormat extends
        InputFormat<IntWritable, ArrayWritable> {
  public static float[] floatvalues;
  @Override
  public RecordReader<IntWritable, ArrayWritable> createRecordReader(
              InputSplit split, TaskAttemptContext context) throws IOException,
              InterruptedException {
        // TODO Auto-generated method stub
        return new FindMaxValueRecordReader();
  }
  @Override
  public List<InputSplit> getSplits(JobContext context) throws IOException,
  InterruptedException {
     // TODO Auto-generated method stub
     int NumOfValues = context.getConfiguration().getInt("NumOfValues", 100);
     floatvalues = new float[NumOfValues];
     Random rand = new Random();
     for(int i = 0; i < NumOfValues; i++) {
         floatvalues[i] = rand.nextFloat();
         //System.out.print(floatvalues[i]+ " ");
```

```
    }
    int NumSplits = context.getConfiguration().getInt("mapred.map.tasks", 2);
    int beg = 0;
    int length = (int)Math.floor(NumOfValues / NumSplits);
    //System.out.println("\n splits : " + NumSplits);
    ArrayList<InputSplit> splits = new ArrayList<InputSplit>();
    int end= length-1;
    for(int i = 0; i < NumSplits - 1; i++)
    {
        FindMaxValueInputSplit split = new FindMaxValueInputSplit(beg, end);
        splits.add(split);
        //System.out.println("FromTo: " + split.getM_StartIndex() + "--" +
        split.getM_EndIndex());
        beg = end + 1;
        end = beg + length -1;
    }
    FindMaxValueInputSplit split = new FindMaxValueInputSplit(beg, NumOfValues-1);
    splits.add(split);
    //System.out.println("FromTo: " + split.getM_StartIndex() + "--" + split.
    getM_EndIndex());
    return splits;
   }
}
```

List<InputSplit> getSplits(JobContext context) 方法主要实现了随机生产 100 个 0 ～ 1 的小数，放到一个 float 数组里面。

以下代码表示指定两个 map 函数来处理这 100 个小数，这里取了平均值，每个 map 函数处理 50 个小数。

```
int NumSplits = context. getConfiguration(). getInt("mapred. map. tasks", 2);
```

2. 自定义 InputSplit 类

这个自定义分片类主要是记录两个 map 函数的开始索引和结束索引，第一个 map 函数处理 1 ～ 50 个小数，第二个 map 函数处理 50 ～ 100 的数据，实现如代码清单 8-9 所示。

代码清单 8-9　mapred\output\FindMaxValueInputSplit.java

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

import org.apache.hadoop.io.ArrayWritable;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.mapreduce.InputSplit;
public class FindMaxValueInputSplit extends InputSplit implements Writable{
    private int m_StartIndex;
    private int m_EndIndex;
    private ArrayWritable m_FloatArray=new ArrayWritable(FloatWritable.class);
```

```
    public  FindMaxValue InputSplit(){}
    public FindMaxValue InputSplit(int start, int end){
      m_StartIndex = start;
      m_EndIndex = end;
      int len = m_EndIndex - m_StartIndex + 1;
      int Index = m_StartIndex;
      FloatWritable[] result = new FloatWritable[len];
      for (int i = 0; i < len; i++) {
         //System.out.println(m_StartIndex + "--" + m_EndIndex);
         float f=FindMaxValueInputFormat.floatvalues[Index];
         //System.out.println("floatvalues[" + Index + "]= " + FindMaxValueInputFormat.
         //floatvalues[Index]);
         FloatWritable fw=new FloatWritable();
         fw.set(f);
         //System.out.println(fw.get());
         result[i]=fw;
         Index ++;
      }
      m_FloatArray.set(result);
    }
  @Override
  public long getLength() throws IOException, InterruptedException {
         // TODO Auto-generated method stub
         return (this.m_EndIndex-this.m_StartIndex+1) ;
  }
  @Override
  public String[] getLocations() throws IOException, InterruptedException {
         return new String[]{ "hadoop-2", "hadoop-1" };
  }
  public int getM_StartIndex() {
         return m_StartIndex;
  }
  public int getM_EndIndex() {
         return m_EndIndex;
  }
   public void readFields(DataInput in) throws IOException {
         this.m_StartIndex = in.readInt();
         this.m_EndIndex = in.readInt();
         this.m_FloatArray.readFields(in);
  }
  public void write(DataOutput out) throws IOException {
         out.writeInt(this.m_StartIndex);
         out.writeInt(this.m_EndIndex);
         this.m_FloatArray.write(out);
  }
  public ArrayWritable getM_FloatArray() {
         return m_FloatArray;
  }
}
```

3. 自定义 RecordReader 类

这个自定义 RecordReader 类的主要功能是：定义输入到 map 函数的数据格式，Key 为偏移量，Value 为一个 float 的数组，长度为 50。如代码清单 8-10 所示。

代码清单 8-10 mapred\output\ FindMaxValueRecordReader.java

```
import java.io.IOException;

import org.apache.hadoop.io.ArrayWritable;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
public class FindMaxValueRecordReader extends RecordReader<IntWritable,
       ArrayWritable>{
  private int m_End;
  private int m_Index;
  private int m_Start;
  private IntWritable key = null;
  private ArrayWritable value = null;
  private FindMaxValueInputSplit fmvsplit=null;
  @Override
  public void close() throws IOException {
     // TODO Auto-generated method stub
  }
  @Override
  public IntWritable getCurrentKey() throws IOException, InterruptedException {
     // TODO Auto-generated method stub
     return key;
  }
  @Override
  public ArrayWritable getCurrentValue() throws IOException, InterruptedException {
     // TODO Auto-generated method stub
     return value;
  }
  @Override
  public float getProgress() throws IOException, InterruptedException {
     // TODO Auto-generated method stub
     if(this.m_Index  == this.m_End) {
            return 0.0f;
     }
     else{
         return Math.min(1.0f, (this.m_Index - this.m_Start) / (float)(this.m_
         End - this.m_Start));
     }
  }
  @Override
  public void initialize(InputSplit split, TaskAttemptContext context)
            throws IOException, InterruptedException {
```

```
        // TODO Auto-generated method stub
        fmvsplit = (FindMaxValueInputSplit)split;
        this.m_Start = fmvsplit.getM_StartIndex();
        this.m_End = fmvsplit.getM_EndIndex();
        this.m_Index = this.m_Start;
    }
    @Override
    public boolean nextKeyValue() throws IOException, InterruptedException {
        // TODO Auto-generated method stub
        if (key == null) {
            key = new IntWritable();
        }
        if (value == null) {
            value = new ArrayWritable(FloatWritable.class);
        }
        //System.out.println(m_Start + "--" + m_End + " " + m_Index);
        if( m_Index <= m_End ) {
            key.set(m_Index);
            value=fmvsplit.getM_FloatArray();
            m_Index = m_End +1;
            return true;
       }
       else
            return false;
    }
}
```

4. 编写 Mapper 类

这里的 Mapper 类实现和一般的 Mapper 有所区别，一般 Mapper 类中的 map 方法如下。

```
public void map(IntWritable key, Text  value, Context context)
```

而下面我们要实现的 Mapper 类中的 map 方法如下。

```
public void map(IntWritable key, ArrayWritable value, Context context)
```

ArrayWritable 表示输入 map 函数的 value 为一个数组，而不是一个字符串。代码实现如代码清单 8-11 所示。

代码清单 8-11 mapred\output\FindMaxValueMapper.java

```
public class FindMaxValueMapper extends
            Mapper<IntWritable, ArrayWritable, IntWritable, FloatWritable> {
    private final static IntWritable one = new IntWritable(1);
    // private PrintWriter file=null;
    public void map(IntWritable key, ArrayWritable value, Context context)
                    throws IOException, InterruptedException {
          FloatWritable[] floatArray = (FloatWritable[]) value.toArray();
        float maxfloat = floatArray[0].get();
        float tmp;
        for (int i = 1; i < floatArray.length; i++) {
```

```
                tmp = floatArray[i].get();
                if (tmp > maxfloat) {
                   maxfloat = tmp;
                }
            }
            context.write(one, new FloatWritable(maxfloat));
        }
    }
```

看以下方法的内容：

```
public void  map(IntWritable key, ArrayWritable value, Context context)
throws IOException, InterruptedException
```

变量 key 为偏移量，变量 value 是一个大小为 50 的数组。

这个 MapReduce 程序启动了两个 map，第一个 map 对前 50 个小数求最大值，第二个 map 对后 50 个小数求最大值，最后把两个 map 求出来的最大值发到 Reduce 端，再求出最大值。

5. 编写 Reducer 类

Reducer 类主要是合并两个 map 出来的最大值。如第一个 map 输出结果为 <0,0.9912>，第二个 map 输出的结果为 <0,0.9967>，那么 Reduce 端比较这两个数，求出最大值 <0.0.9967> 写到 HDFS 指定的目录下面。代码实现如代码清单 8-12 所示。

代码清单 8-12　mapred\output\ FindMaxValueReducer.java

```
public static class FindMaxValueReducer extends Reducer<IntWritable,
             FloatWritable, Text, FloatWritable> {
   public void reduce(IntWritable key, Iterable<FloatWritable> values, Context context)
                throws IOException, InterruptedException {
      Iterator it = values.iterator();
      float maxfloat=0, tmp;
      if(it.hasNext())
         maxfloat = ((FloatWritable)(it.next())).get();
      else {
        context.write(new Text("Max float value: "), null);
        return;
      }
      //System.out.print(maxfloat + " ");
      while(it.hasNext()){
        tmp = ((FloatWritable)(it.next())).get();
        //System.out.print(tmp + " ");
        if (tmp > maxfloat) {
            maxfloat = tmp;
        }
      }
      context.write(new Text("Max float value:"), new FloatWritable(maxfloat));
    }
}
```

6. 驱动类

驱动类编写没有什么特别的地方，实现如代码清单 8-13 所示。

代码清单 8-13 mapred\output\ MaxValueDriver.java

```
public class MaxValueDriver {
  public static void main(String[] args) throws Exception  {
     // TODO Auto-generated method stub
     Configuration conf = new Configuration();
     String[] paths = { "hdfs://192.168.153.100:9000/user/hadoop/output3/" };
     String[] otherArgs = new GenericOptionsParser(conf, paths)
                 .getRemainingArgs();
     // String[] otherArgs = new GenericOptionsParser(conf,
     // args).getRemainingArgs();
     if (otherArgs.length != 1) {
          System.err.println("Usage: FindMaxValue <OutputDir>");
          //System.exit(2);
     }
     conf.set("NumOfValues", "111");
     conf.set("hadoop.job.ugi", "jayliu,jayliu");
     conf.set("mapred.system.dir",
          "/home/hadoop/HadoopHome/tmp/hadoop-hadoop/mapred/system/");
     Job job = new Job(conf, "FindMaxValue");
     // job.setNumReduceTasks(2);
     job.setJarByClass(FindMaxValue.class);
     job.setMapperClass(FindMaxValueMapper.class);
     job.setReducerClass(FindMaxValueReducer.class);
     job.setMapOutputKeyClass(IntWritable.class);
     job.setMapOutputValueClass(FloatWritable.class);
     job.setOutputKeyClass(Text.class);
     job.setOutputValueClass(FloatWritable.class);
     job.setInputFormatClass(FindMaxValueInputFormat.class);
     FileOutputFormat.setOutputPath(job, new Path(otherArgs[0]));
     System.out.println(conf.get("mapred.job.tracker"));
     System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

这里只指定了输出的路径，没有输入的路径，因为输入的数据是在内存里面生成的。

7. 运行

按照第 3 章的步骤运行即可，这里的 JAR 包名为 maxValue.jar，我们看看结果吧。

执行如下命令运行 maxValue.jar 程序。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop jar  maxValue.jar mapred.output.
MaxValueDriver
```

查看结果。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs  -text
              /user/hadoop/output3/part-r-00000
Max float value:   0.9967
```

到这里，自定义 InputFormat、InputSplit、RecordReader 全部实现，理解这三个类或接口的含义，并且能够根据实际需求自己实现 MapReduce 的输入是非常重要的。

8.3 输出格式

MapReduce 提供了许多默认输出格式，如 TextOutputFormat、KeyValueOutputFormat 等。默认情况下只有一个 Reduce，输出只有一个文件，默认文件名为 part-r-00000，输出文件的个数与 Reduce 的个数一致。如果有两个 Reduce，输出的结果就有两个文件，第一个为 part-r-00000，第二个为 part-r-00001，依次类推。

8.3.1 OutputFormat 接口

OutputFormat 和 InputFormat 类似，但是有些地方不一样。所有的 OutputFormat 大多数继承自 FileOutputFormat，但是 NullOutputFormat 和 DBOutputFormat 是为专门领域的应用程序预留的。

TextOutputFormat 是默认的输入格式，key 和 value 用一个 Tab 分开，分割符也可以通过 mapred.textoutputformat.separator 属性设置。TextOutputFormat 的输出格式可以被 KeyValue-TextInputFormat 接受。

OutputFormat 提供了对 RecordWriter 的实现，从而指定如何序列化数据。 RecordWriter 类可以处理包含单个键值对的作业，并将结果写入 OutputFormat 中准备好的位置。RecordWriter 的实现主要包括两个函数：write 和 close。write 函数从 MapReduce 作业中取出键值对，并将其字节写入磁盘。close 函数会关闭 Hadoop 到输出文件的数据流。

OutputFormat 的层次结构如图 8-6 所示。

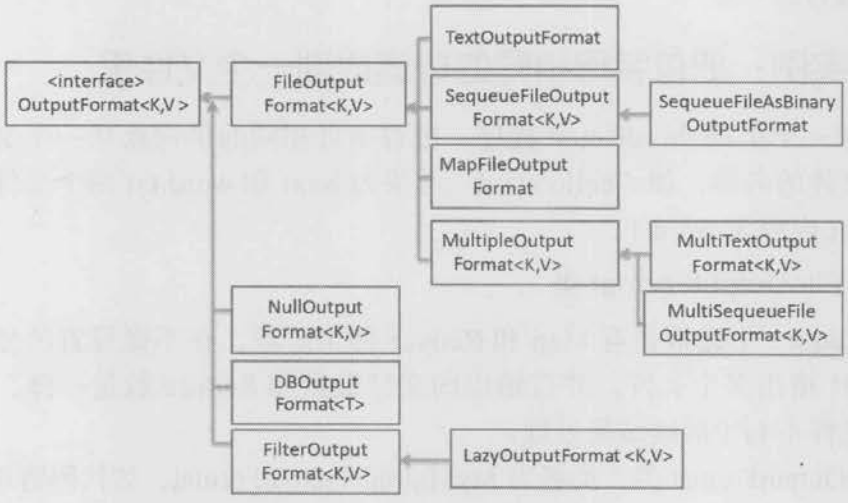


图 8-6 OutputFormat 类的层次结构

OutputFormat 是 MapReduce 输出的基类，所有实现 MapReduce 输出都实现了 OutputFormat

接口。实现接口类解释如下。

- FileOutputFormat：默认的输出格式。
- NullOutputFormat：任何结果都不输出。
- DBOutputFormat：输出到数据库如 MySQL、Oracle 等。
- FilterOutputFormat：对输出结果文件进行过滤。

FileOutputFormat（实现 OutputFormat 接口）是所有 OutputFormat 的基类。继承 FileOutputFormat 类如下所示。

- MapFileOutputFormat：一种使用部分索引键的格式。
- SequenceFileOutputFormat：二进制键值数据的压缩格式。
- MultipleOutputFormat：输出多个压缩格式的文件。

FileOutputFormat 的子类会产生文件（part-r-0000），即使文件是空的，有些应用倾向于不创建空文件（如 Hive 应用），此时 LazyOutputFormat 就有用了，它是一个封装输出格式，可以保证指定分区第一条记录输出时才真正创建文件，用 job.setOutputFormatClass() 方法即可。例如：

```
job.setOutputFormatClass(LazyOutputFormat.class)
```

8.3.2 RecordWriter 类

LineRecordWriter 是默认使用的 RecordWriter，它是前面提到的 TextOutputFormat 的一部分。写入的内容包括：

- 键（Key）的字节（由 getBytes() 函数返回）。
- 一个用以定界的制表符。
- 值（Value）的字节（同样由 getBytes() 函数返回）。
- 一个换行符。

8.3.3 应用实例：把首字母相同的单词放到一个文件里

这里我们举一个扩展 WordCount 程序，把首字母相同的单词放到一个文件里，并且以首字母来命名文件的名称，如“hello word”结果为 h.txt 和 word.txt 两个文件，h.txt 内容为 hello 1，word.txt 内容为 word 1。

1. 自定义 FileOutputFormat 类

普通 MapReduce 中通常是有 Map 和 Reduce 两个阶段，在不做设置的情况下，计算结果会以 part-000* 输出多个文件，并且输出的文件数量和 Reduce 数量一样，文件内容格式也不能改变。这样不利于后续结果处理。

自定义 FileOutputFormat 类，类名为 MyMultipleOutputFormat，如代码清单 8-14 所示。

代码清单 8-14 mapred\output\ MyMultipleOutputFormat.java

```
package com.easyhadoop.multi;
import java.io.DataOutputStream;
```

```
import java.io.IOException;
import java.util.HashMap;
import java.util.Iterator;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.compress.CompressionCodec;
import org.apache.hadoop.io.compress.GzipCodec;
import org.apache.hadoop.mapreduce.OutputCommitter;
import org.apache.hadoop.mapreduce.RecordWriter;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.output.FileOutputCommitter;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.ReflectionUtils;
public abstract class MyMultipleOutputFormat<K extends WritableComparable<?>,
        V extends Writable> extends FileOutputFormat<K, V> {
    private MultiRecordWriter writer = null;
    public RecordWriter<K, V> getRecordWriter(TaskAttemptContext job)
          throws IOException, InterruptedException {
        if (writer == null) {
            writer = new MultiRecordWriter(job, getTaskOutputPath(job));
        }
        return writer;
    }
    private Path getTaskOutputPath(TaskAttemptContext conf) throws IOException {
        Path workPath = null;
        OutputCommitter committer = super.getOutputCommitter(conf);
        if (committer instanceof FileOutputCommitter) {
            workPath = ((FileOutputCommitter) committer).getWorkPath();
        } else {
            Path outputPath = super.getOutputPath(conf);
            if (outputPath == null) {
                throw new IOException("Undefined job output-path");
            }
            workPath = outputPath;
        }
        return workPath;
    }
    /** 通过 key、value、conf 来确定输出文件名（含扩展名）*/
    protected abstract String generateFileNameForKeyValue(K key, V value,
            Configuration conf);
    public class MultiRecordWriter extends RecordWriter<K, V> {
        /**RecordWriter 的缓存 */
        private HashMap<String, RecordWriter<K, V>> recordWriters = null;
        private TaskAttemptContext job = null;
        /** 输出目录 */
        private Path workPath = null;
```

```
public MultiRecordWriter(TaskAttemptContext job, Path workPath) {
    super();
    this.job = job;
    this.workPath = workPath;
    recordWriters = new HashMap<String, RecordWriter<K, V>>();
}
@Override
public void close(TaskAttemptContext context)
                throws IOException, InterruptedException {
    Iterator<RecordWriter<K, V>> values = this.recordWriters.values().iterator();
    while (values.hasNext()) {
         values.next().close(context);
    }
    this.recordWriters.clear();
}
@Override
public void write(K key, V value) throws IOException, InterruptedException {
    //得到输出文件名
    String baseName = generateFileNameForKeyValue(key, value,
                    job.getConfiguration());
    RecordWriter<K, V> rw = this.recordWriters.get(baseName);
    if (rw == null) {
          rw = getBaseRecordWriter(job, baseName);
          this.recordWriters.put(baseName, rw);
    }
    rw.write(key, value);
}
// ${mapred.out.dir}/_temporary/_${taskid}/${nameWithExtension}
private RecordWriter<K, V> getBaseRecordWriter(TaskAttemptContext job,
          String baseName) throws IOException, InterruptedException {
    Configuration conf = job.getConfiguration();
    boolean isCompressed = getCompressOutput(job);
    String keyValueSeparator = ",";
    RecordWriter<K, V> recordWriter = null;
    if (isCompressed) {
          Class<? extends CompressionCodec> codecClass =
                  getOutputCompressorClass(job, GzipCodec.class);
          CompressionCodec codec = ReflectionUtils.
          newInstance(codecClass, conf);
          Path file = new Path(workPath, baseName +
          codec.getDefaultExtension());
          FSDataOutputStream fileOut =
          file.getFileSystem(conf).create(file, false);
          recordWriter = new LineRecordWriter<K, V>(new DataOutputStream
          (codec. createOutputStream(fileOut)), keyValueSeparator);
    } else {
          Path file = new Path(workPath, baseName);
          FSDataOutputStream fileOut =
                  file.getFileSystem(conf).create(file, false);
          recordWriter = new LineRecordWriter<K, V>
                  (fileOut, keyValueSeparator);
    }
```

```
            return recordWriter;
        }
    }
}
```

主要的方法解释如下。

- generateFileNameForKeyValue(K key, V value, Configuration conf)：最重要的实现方法，通过 key、value 和 conf 三个变量来确定输出文件名（含扩展名），如 key 为 hello，扩展名为 h.txt。
- public void write(K key, V value)：得到输出的文件名，如把 h.txt 文件写到 HDFS 上。

2. 添加 AlphabetOutputFormat 类

在 WordCount 类里添加静态类 AlphabetOutputFormat，该类继承了 MyMultipleOutputFormat 类，实现代码如代码清单 8-15 所示。

代码清单 8-15 mapred\output\ MultiWordCount.java

```
package com.easyhadoop.multi;
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
public class MultiWordCount {
    public static class TokenizerMapper extends Mapper<Object, Text, Text,
    IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
    }
    public static class IntSumReducer extends Reducer<Text, IntWritable, Text,
    IntWritable> {
        private IntWritable result = new IntWritable();
        public void reduce(Text key, Iterable<IntWritable> values, Context
        context) throws IOException, InterruptedException {
```

```
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }
    public static class AlphabetOutputFormat extends MyMultipleOutputFormat<Text,
    IntWritable> {
        @Override
        protected String generateFileNameForKeyValue(Text key, IntWritable value,
        Configuration conf) {
            char c = key.toString().toLowerCase().charAt(0);
            if (c >= 'a' && c <= 'z') {
                return c + ".txt";
            }
            return "other.txt";
        }
    }
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = new Job(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setOutputFormatClass(AlphabetOutputFormat.class);// 设置输出格式
        FileInputFormat.addInputPath(job, new Path("/user/hadoop/input/word/"));
        FileOutputFormat.setOutputPath(job, new Path("/user/hadoop/output"));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

以上代码基本维持Hadoop示例中的WordCount原样，只增加一个静态内部类AlphabetOutputFormat，这个类实现了MyMultipleOutputFormat，文件命名规则是：以英文字母开头的单词以“首字母.txt”为文件名保存，其他以“other.txt”保存。

3. 运行

JAR包名为multiInput.jar。测试数据为test.txt，内容为“hello word 123”。执行multiInput.jar命令如下所示：

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop jar
                multiInput.jar  mapred.output. WordCount
```

查看结果：

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs  -ls /user/hadoop/output
h. txt
w. txt
other.Txt
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs  -text  /user/hadoop/output/h.txt
hello 1
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs  -text  /user/hadoop/output/other.txt
123 1
```

多文件输出在 MapReduce 开发中很常用，例如把相同的客服信息存放到同一个文件下面，每个客户有一个唯一标示 ID，可以把客户的 ID 当做 Key，使用多文件输出，把结果文件存放到 Key.txt 中。

8.4 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。使用速度最快压缩（如 LZO 和 Snappy），还是使用压缩比最大（如 gzip），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。

8.4.1 如何在 MapReduce 中使用压缩

对于大型的、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序列文件（Sequence File）。

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。

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，如下例所示。

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

8.4.2 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 Reduce 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 8-4 显示了启用 Map 输出压缩和设置压缩格式的配置属性。

表 8-4 Map 端压缩属性列表

属性名称	类 型	默 认 值	功能描述
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);
```

注意 这里要查看压缩文件内容要用 -text 参数，-cat 和 -tail 查看为乱码。

8.5 MapReduce 优化

Combiner 和 Partitioner 是用来优化 MapReduce 的，可以提高 MapReduce 的运行效率。下面我们来看它们的作用和使用方法。

8.5.1 Combiner 类

通常，每一个 Map 可能会产生大量的输出，Combiner 的作用就是在 Map 端对输出先做一次合并，以减少传输到 Reduce 端的数据量。很多 MapReduce 程序受限于集群上可用的带宽，所以它会尽力最小化需要在 Map 和 Reduce 任务之间传输的中间数据。

因为 Combiner 本身只是一种优化，Hadoop 并不保证对于某个 Map 输出，这个方法会被调用多少次。换句话说，不管 Combiner 被调用多少次，对应的 Reduce 输出结果都应该是一样的。

Combiner 在系统中并没有自己的基类，而是用 Reducer 作为 Combiner 的基类，它们对外的功能是一样的，只是使用的位置和使用时的上下文不太一样而已。

下面我们通过一个例子来解释 Combiner 的原理，假设 1950 年的天气数据读取是由两个 Map 完成的，其中第一个 Map 的输出如下：

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

第二个 Map 的输出为：

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

而 Reduce 得到的输入为：

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

输出为：

```
(1950, 25)
```

由于 25 是集合中的最大值，可以使用一个类似于 Reduce 的 Combiner 来找出每个 Map 输出中的最大值，这样 Reduce 的输入就变成以下内容。

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

各个 function 对温度值的处理过程可以表示如下。

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

注意 不是所有函数都拥有这个属性，有这个属性的函数称为 commutative 和 associative，例如，要计算平均温度就不能这样使用 Combiner，因为 mean(0, 20, 10, 25, 15) = 14，而 mean(mean(0, 20, 10), mean(25, 15)) = mean(10, 20) = 15。

Combiner 并不能取代 Reduce（因为仍然需要 Reduce 处理来自不同 Map 的带有相同 Key 的记录）。但是它可以帮助减少需要在 Map 和 Reduce 之间传输的数据，因为这一点 Combiner 就值得考虑使用。

Combiner 的实现与 Reducer 类的实现一样，唯一的改变就是在 Job 中设置。Combiner 的具体使用如下。

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

8.5.2 Partitioner 类

Map 最终处理的键值对 <key, value> 需要送到 Reduce 去合并。合并的时候，有相同 Key 的 Key-Value 对会送到同一个 Reduce 上。Key 到 Reduce 的分配过程是由分区（Partitioner）规定的。

Partitioner 只提供了一个方法：

```
getPartition(Text key, Text value, int numPartitions)
```

< Text key, Text value > 两个参数为 Map 的 Key 和 Value，numPartitions 为 Reduce 的个数，默认为 1，Partitioner 的工作原理如图 8-7 所示。

Partitioner 有两个功能：

- 均衡负载。尽量将工作均匀地分配给不同的 Reduce。
- 效率。分配速度一定要快。

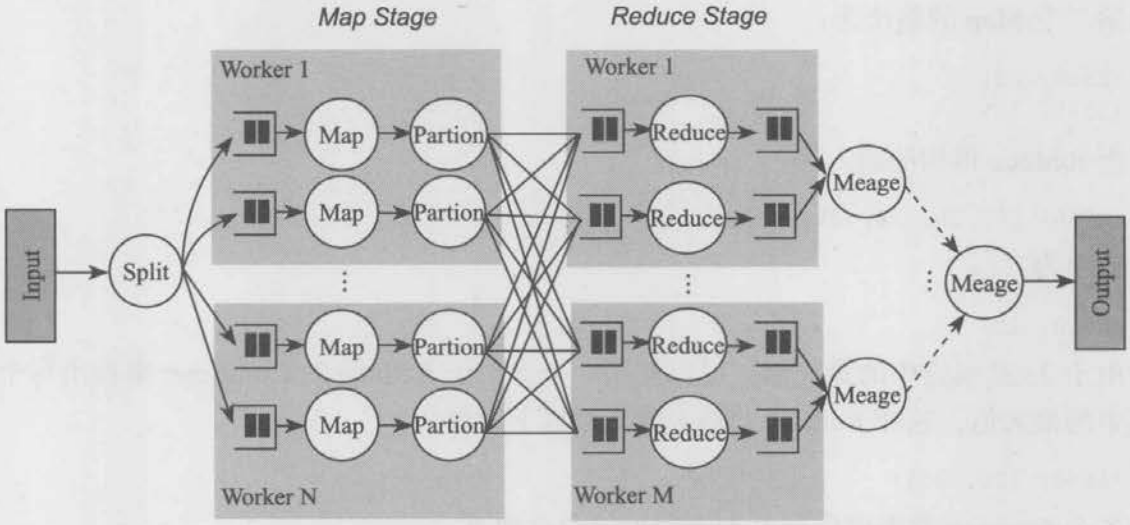


图 8-7 Partitioner 的工作原理

1. MapReduce 提供的 Partitioner

MapReduce 提供了 4 种 Partitioner，默认的是 HashPartitioner，如图 8-8 所示。

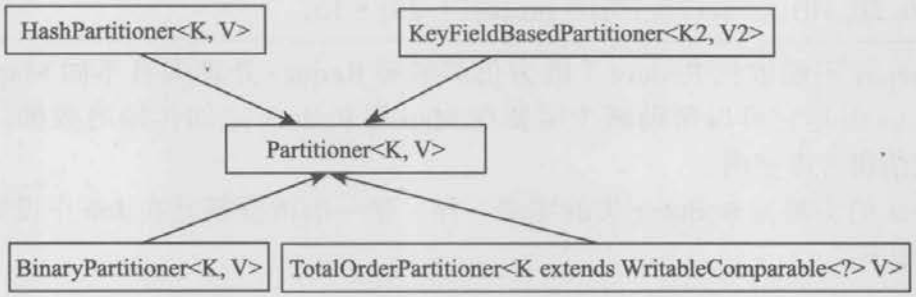


图 8-8 MapReduce 自带的 Partitioner

Partitioner 是一个基类，如果定制 Partitioner 需要继承该类。

HashPartitioner 是 MapReduce 的默认 Partitioner。计算方法如下。

```
which reducer=(key.hashCode() & Integer.MAX_VALUE) % numReduceTasks
```

取 Key 的 hashCode 码与 Integer 的最大值做或运算，得出的结果和 Reducer 的个数取模，最终得到这个 Key 发送到哪个 Reducer 上面。

BinaryPartitioner 继承于 Partitioner< BinaryComparable, V>，是 Partitioner 的字节码子类。该类提供 leftOffset 和 rightOffset，在计算 which reducer 时，仅对 Key-Value 中 Key 的 [rightOffset，leftOffset] 区间取 hash。

```
which reducer=(hash & Integer.MAX_VALUE) % numReduceTasks
```

KeyFieldBasedPartitioner 是基于 hash 的 Partitioner。与 BinaryPartitioner 不同，它提供了多个区间用于计算 hash。当区间数为 0 时，KeyFieldBasedPartitioner 就变成 HashPartitioner。

TotalOrderPartitioner 类可以实现输出的全排序。不同于以上 3 个 Partitioner，这个类并

不是基于 hash 的（后面会详细介绍 TotalOrderPartitioner）。

2. 自定义 Partitioner

下面自己实现一个 Partitioner。自定义 MyPartitionerPar .java 类，如代码清单 8-16 所示。

代码清单 8-16 mapred\partition\MyPartitionerPar.java

```
/**
 * partitioner 的输入就是 Map 的输出
 * @author Administrator
 */
public class MyPartitionerPar  extends Partitioner<Text, Text> {
      @Override
      public int getPartition(Text key, Text value, int numPartitions) {
            int result = 0;
            if (key.equals("three")) {
                  result = 0 % numPartitions;//0%3=0
            } else if (key.equals("four")) {
                  result = 1 % numPartitions;  //1%3=1
            } else if (key.equals("right")) {
                  result = 2 % numPartitions;  //2%3 =2
            }
        return result;
    }
}
```

MyPartitionerPar 的功能是把 Map 端输出的 Key 为“three”发送给 reduce-0 处理，结果存到 part-r-00000 里面；Key 为“four”发送给 reduce-1 处理，结果存到 part-r-00001 里面；Key 为“right”发送给 reduce-2 处理，结果存到 part-r-00002 里面。

3. TotalOrderPartitioner 类

每一个 Reduce 的输出在默认的情况下都是有顺序的，但是 Reduce 之间在输入是无序的情况下也是无序的。如果要实现输出是全排序的，就会用到 TotalOrderPartitioner。

要使用 TotalOrderPartitioner，需要给 TotalOrderPartitioner 提供一个 partition file。这个文件要求 Key（这些 Key 就是所谓的划分）的数量和当前 Reducer 的数量 −1 相同，并且是从小到大排列。对于为什么要用到这样一个文件，以及这个文件的具体细节下面还会提到。TotalOrderPartitioner 对不同 Key 的数据类型提供了两种方案。

1）对于非 BinaryComparable 类型的 Key，TotalOrderPartitioner 采用二分法查找当前的 Key 所在的 index。

例如 Reducer 的数量为 5，partition file 提供的 4 个划分为 [2，4，6，8]。如果当前的一个 Key-Value 值是 <4, “good” >，利用二分法查找到 index=1，index+1=2，那么这个 Key-Value 值将会发送到第二个 Reduce。如果一个 Key-Value 值为 <4.5, “good” >，那么二分法查找将返回 −3，同样对 −3 加 1 然后取反就是这个 Key-Value 值将要去的 Reducer。对于一些数值型的数据来说，利用二分法查找复杂度是 o（log (reducer count)）速度比较快。

2）对于 BinaryComparable 类型的 Key（也可以直接理解为字符串）。字符串按照字典顺序也是可以进行排序的。

这样的话也可以给定一些划分，让不同的字符串 Key 分配到不同的 Reduce 里。这里的处理和数值类型比较相近。例如 Reducer 的数量为 5，partition file 提供了 4 个划分为 [“abc”，“bce”，“eaa”，“fhc”]，那么“ab”这个字符串将会被分配到第一个 Reducer 里，因为它小于第一个划分“abc”。但是不同于数值型的数据，字符串的查找和比较不能按照数值型数据的比较方法。MapReduce 采用 Tire Tree 的字符串查找方法。查找的时间复杂度 o(m)，m 为树的深度，空间复杂度 o(255^m-1)。Tire Tree 的字符串查找是一个典型的空间换时间的案例。

Hadoop 自带的采样器（Sampler）也是基于 Partitioner 实现的，是一种海量数据排序的算法，下面介绍 Hadoop 自带的采样器。

4. Sampler 类

Hadoop 自带的 Sampler 可以对输入目录下的数据进行采样。提供了 3 种采样方法，如图 8-9 所示。

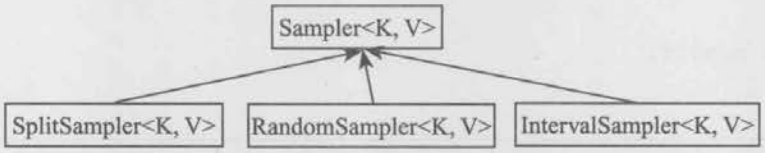


图 8-9 采样类结构图

输入采样一般在对海量数据进行排序的时候用得比较多，主要是用来缓解 Reduce 端的负载均衡，Hadoop 自带的 1TB 数据排序算法就用到了输入采样 Sampler。Hadoop 自带的采样方法如表 8-5 所示。

表 8-5 自带的输入采样

类名称	采样方式	构造方法	效率	特 点
SplitSampler<K,V>	对前 n 个记录进行采样	采样总数，划分数	最高	
RandomSampler<K,V>	遍历所有数据，随机采样	采样频率，采样总数，划分数	最低	
IntervalSampler<K,V>	固定间隔采样	采样频率，划分数	中	对有序的数据十分适用

InputSampler 类实现了三种采样方法：RandomSampler、SplitSampler 和 IntervalSampler。它们都是 InputSampler 的静态内部类，都实现了 InputSampler 的内部接口 Sampler，接口的方法如下。

```
public interface Sampler<K,V>{
    K[] getSample(InputFormat<K,V> inf,JobConf job) throws IOException;
}
```

getSample 方法根据 Job 的配置信息以及输入格式获得抽样结果，三个采样类各自有不同的实现。

InputSampler 里面的 writePartitionFile 方法定义如下。

```
public static <K,V> void writePartitionFile(JobConf job,
      org.apache.hadoop.mapreduce.lib.partition.InputSampler.Sampler<K,V> sampler)
      throws IOException, ClassNotFoundException, InterruptedException
```

writePartitionFile 方法是将采样的结果排序，然后按照分区的个数 n，将排序后的结果平均分为 n 份，取 n-1 个分割点，这个分割点具体取的时候，运用了一些 4 舍 5 入的方法，可以理解是取后 n-1 个组中每组的第一个采样值。

8.5.3 分布式缓存

分布式缓存在 MapReduce 中称为 DistributedCache，它可以方便 Map Task 之间或者 Reduce Task 之间共享一些信息，同时也可以将第三方包添加到其 CLASSPATH 路径中。Hadoop 会将缓存数据分发到集群中所有准备启动的节点上，复制到 mapred.temp.dir 中配置的目录。

1. DistributedCache 的使用方法

使用 DistributedCache 的本质其实是添加 Configuration 中的 mapred.cache.{files|archives} 属性。有两种方法可以将文件添加 DistributedCache。

第一种方法：使用 Configuration 对象的 set 方法，代码如下。

```
conf.set("mapred.cache.files", "/data/data");
conf.set("mapred.cache. archives", "/data/data.zip");
```

第二种方法：比较方便，直接使用 DistributedCache 的静态方法，代码如下所示。

```
DistributedCache. addCacheFile(URI, Configuration);
DistributedCache.addArchiveToClassPath(Path, Configuration, FileSystem);
```

需要注意的是，上面几行代码需要写在 Job 类初始化之前，否则在运行中会找不到文件，因为 Job 初始化时将传入 Configuration 对象给 JobContext 复制一份。

在 MapReduce 0.21 版本以后，org.apache.hadoop.mapreduce 均移到 org.apache.hadoop.mapred 包下。但文档中提供的 configure 方法是重写的 MapReduceBase 中的，而新版本中 Map 继承于 Mapper，Reduce 继承于 Reducer，所以 configure 方法一律改成 setup。要获得 cache 数据，就得在 Map/Reduce Task 中的 setup 方法中取得 cache 数据，再进行相应操作。

```
@Override
protected void setup(Context context) throws IOException, InterruptedException {
    super.setup(context);
    URI[] uris = DistributedCache.getCacheFiles(context .getConfiguration());
    Path[] paths = DistributedCache.getLocalCacheFiles(context .getConfiguration());
}
```

三方库的使用稍微简单，只需要将库上传至 HDFS，再用代码添加至 CLASSPATH。

```
DistributedCache.addArchiveToClassPath(new Path("/data/test.jar"), conf);
```

2. 使用 Symlink 缓存文件

Symlink 其实就是 HDFS 文件的一个快捷方式，只需要在路径名后加入“#linkname”，之后在 Task 中使用 linkname 使用相应文件，代码如下。

```
conf.set("mapred.cache.files", "/data/data#mData");
conf.set("mapred.cache. archives", "/data/data.zip#mDataZip");
@Override
protected void setup(Context context) throws IOException, InterruptedException {
    super.setup(context);
    FileReader reader = new FileReader(new File("mData"));
    BufferedReader bReader = new BufferedReader(reader);
    // TODO
}
```

在使用 Symlink 之前，需要告知 Hadoop 系统，默认是不支持 Symlink 操作的，要在 Configuration 里面设置属性 mapred.create.symlink 的值为 yes。具体代码如下所示。

```
conf.set("mapred.create.symlink", "yes"); // 是 yes，不是 true
DistributedCache.createSymlink(conf);
```

使用 createSymlink(Configuration conf) 方法创建一个 Symlink，在实际开发中用得非常多，主要用来管理第三方库、配置文件等。

注意事项：

- 缓存文件（数据、三方库）需上传至 HDFS，方能使用。
- 缓存较小的情况下，建议将数据全部读入相应节点内存，提高访问速度。
- 缓存文件是 read-only 的不能修改。若要修改得重新输出，将新输出文件作为新缓存进入下一次迭代。

8.6 辅助类

为了简化命令行方式运行作业，Hadoop 自带了一些辅助类，例如 GenericOptionsParser、Tool 和 ToolRunner。

GenericOptionsParser 用来解释常用的 Hadoop 命令行选项，并根据需要为 Configuration 对象设置相应的取值。GenericOptionsParser 接口如下。

```
public interface Tool extends Configurable {
    int run(String [] args) throws Exception;
}
```

通常不直接使用 GenericOptionsParser，更方便的方式是：实现 Tool 接口，通过 ToolRunner 来运行应用程序，ToolRunner 内部调用 GenericOptionsParser。

8.6.1 读取 Hadoop 配置文件

本节介绍一个非常简单的 Tool 实现，用来打印 Tool 的 Configuration 对象中所有属性

的键值对，实现代码如代码清单 8-17 所示。

代码清单 8-17 mapred\output\ ConfigurationPrinter.java

```
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，Configured 是 Configurable 接口的一个实现。Tool 的所有实现都需要实现 Configurable（因为 Tool 继承于 Configurable），Configured 子类通常是一种最简单的实现方式。run() 方法通过 Configurable 的 getConf() 方法获取 Configuration，然后重复执行，将每个属性打印到标准输出。

静态代码部分用来获取 HDFS 和 MapReduce 配置，以及核心配置（Configuration 已经取得核心配置）。ConfigurationPrinter 的 main() 方法没有直接调用自身的 run() 方法，而是调用 ToolRunner 的静态 run() 方法，该方法负责在调用 run() 方法之前，为 Tool 建立一个 Configuration 对象。ToolRunner 还使用了 GenericOptionsParser 来获取在命令行方式中指定的任何标准选项，然后在 Configuration 实例上进行设置。

运行下列代码，可以看到在 conf/mapred-site.xml 中设置的属性。

```
$ hadoop ConfigurationPrinter -conf conf/mapred-site.xml \
  | grep mapred.job.tracker=  mapred.job.tracker=192.168.192.100:8021
```

8.6.2 设置 Hadoop 的配置文件属性

Hadoop 的配置文件在 $HADOOP_HOME/conf 目录下面，也可以在 Hadoop 安装路径的 docs 目录中，查看所有公共属性的默认设置，相关文件包括 coredefault.html、hdfs-default.html 和 mapred-default.html 等 HTML 文件。每个属性都有解释属性作用和取值范围的描述。

使用我们编写的 ConfigurationPrinter 类来动态设置 Hadoop 属性，GenericOptionsParser 允许设置个别属性。例如：

```
$ hadoop ConfigurationPrinter -D color=yellow | grep color
  color=yellow
```

-D 选项用于将键 color 的配置属性值设置为 yellow。设置为 -D 的选项优先级要高于配置文件里的其他属性。这一点很有用：可以把默认属性放入配置文件中，然后在需要时，用 -D 选项来覆盖它们。一个常见的例子是：通过 -D mapred.reduce.tasks=n 设置 MapReduce 作业中 reduce 的数量。这样会覆盖集群上或客户端配置属性文件中设置的 Reduce 数量。

使用 -D property=value 选项，将 Hadoop 属性设置为 GenericOptionsParser（和 ToolRunner），不同于用 -Dproperty=value 选项将 JVM 系统属性设置为 Java 命令。JVM 系统属性的语法不允许 D 和属性名之间有任何空格，而 GenericOptionsParser 要求用空格分隔 D 和属性名。JVM 系统属性来自 java.lang.System 类，而 Hadoop 属性只能从 Configuration 对象中获取。所以，下面的命令行将没有任何输出，因为 ConfigurationPrinter 没有使用 System 类。

```
$  hadoop -Dcolor=yellow ConfigurationPrinter | grep color
```

注意 在客户端配置中设置某些属性，将不会产生影响。例如，如果在作业提交时想通过设置 mapred.tasktracker.map.tasks.maximum 来改变运行作业的 TaskTracker 的任务槽（task slot）数，结果会令你失望，因为这个属性只能在 TaskTracker 的 mapred-site.xml 文件中进行设置。一般情况下，可以通过属性名来告诉组件该属性应该在哪里进行设置，由于 mapred.tasktracker.map.tasks.maximum 以 mapred.tasktracker 开头，因此，我们知道它只能为 TaskTracker 守护进程设置。但是，这不是硬性的，在有些情况下，我们需要进行尝试，甚至去阅读源码。

8.6.3 GenericOptionsParser 选项

GenericOptionsParser 选项和 ToolRunner 选项参数解释如表 8-6 所示。

表 8-6 GenericOptionsParser 选项和 ToolRunner 选项

选项名称	功能描述
-D *property=value*	将指定值赋值给确定的 Hadoop 配置属性。覆盖配置文件里的默认属性或站点属性，或通过 -conf 选项设置的任何属性
-conf *filename* ...	将指定文件添加到配置的资源列表中。这是设置站点属性或同时设置一组属性的简便方法
-fs uri	用指定的 URI 设置默认文件系统。这是 -D fs.default.name=uri 的快捷方式
-jt host:port	用指定主机和端口设置 JobTracker。这是 -D mapred.job.tracker= host:port 的快捷方式
-files*file1,file2,...*	从本地文件系统（或任何指定模式的文件系统）中复制指定文件到 JobTracker 所用的共享文件系统（通常是 HDFS），确保在任务工作目录的 MapReduce 程序可以访问这些文件
-archives *archive1,archive2,...*	从本地文件系统（或任何指定模式的文件系统）复制指定存档到 JobTracker 所用的共享文件系统（通常是 HDFS），打开存档文件，确保任务工作目录的 MapReduce 程序可以访问这些存档
-libjars*jar1,jar2,...*	从本地文件系统（或任何指定模式的文件系统）复制指定 JAR 文件到被 JobTracker 使用的共享文件系统（通常是 HDFS），把它们加入 MapReduce 任务的类路径中。这个选项适用于传输作业需要的 JAR 文件

8.7 Streaming 接口

Streaming 框架允许任何程序语言实现的程序在 Hadoop MapReduce 中使用，方便已有程序向 Hadoop 平台移植。因此可以说对于 Hadoop 的扩展性意义重大。

8.7.1 Streaming 工作原理

Streaming 的原理是用 Java 实现一个包装用户程序的 MapReduce 程序，该程序负责调用 MapReduce Java 接口获取 Key-Value 对输入，创建一个新的进程启动包装的用户程序，将数据通过管道传递给包装的用户程序处理，然后调用 MapReduce 的 Java 接口将用户程序的输出切分成 Key-Value 对输出。

1）Streaming 优点包括：

- 开发效率高。便于移植只要按照标准输入输出格式进行编程，就可以满足 Hadoop 要求。因此单机程序稍加改动就可以在集群上进行使用。同样便于测试只要按照 cat input | mapper | sort | reducer > output 进行单机测试即可。如果单机测试通过，大多数情况是可以在集群上成功运行的，只要控制好内存就可以了。
- 提高程序效率。有些程序对内存要求较高，如果用 java 控制内存毕竟不如 C/C++。

2）Streaming 不足包括：

- Hadoop Streaming 默认只能处理文本数据，无法直接对二进制数据进行处理。
- Streaming 中的 Mapper 和 Reducer 默认只能向标准输出写数据，不能方便地处理多路输出。

8.7.2 Streaming 编程接口参数

Streaming 编程接口默认有很多参数，这些参数的作用其实和我们用 Java 开发 MapReduce 里面的驱动类有点相似。参数介绍如表 8-7 所示。

表 8-7 Streaming 参数列表

参数名称	功能描述
-input <*path*>	输入数据路径
-output <*path*>	输出数据路径
-mapper <cmd|JavaClassName>	Mapper 可执行程序或 Java 类
-reducer <cmd|JavaClassName>	Reducer 可执行程序或 Java 类
-file <*file*> Optional	分发本地文件
-cacheFile <*file*> Optional	分发 HDFS 文件
-cacheArchive <*file*> Optional	分发 HDFS 压缩文件
-numReduceTasks <*num*> Optional	Reduce 任务个数
-jobconf | -D NAME=VALUE Optional	作业配置参数
-combiner <*JavaClassName*> Optional	Combiner Java 类

（续）

参数名称	功能描述
-partitioner <*JavaClassName*> Optional	Partitioner Java 类
-inputformat <*JavaClassName*> Optional	InputFormat Java 类
-outputformat <*JavaClassName*> Optional	OutputFormat Java 类
-inputreader <*spec*> Optional	InputReader 配置
-cmdenv <*n*>=<*v*> Optional	传给 Mapper 和 Reducer 的环境变量
-mapdebug <*path*> Optional	Mapper 失败时运行的 debug 程序
-reducedebug <*path*> Optional	Reducer 失败时运行的 debug 程序
-verbose Optional	详细输出模式

下面是对各个参数的详细说明。

- -input <path>：指定作业输入，path 可以是文件或者目录，可以使用“*”通配符，-input 选项可以使用多次指定多个文件或目录作为输入。
- -output <path>：指定作业输出目录，path 必须不存在，而且执行作业的用户必须有创建该目录的权限，-output 只能使用一次。
- -mapper：指定 Mapper 可执行程序或 Java 类，必须指定且唯一。
- -reducer：指定 Reducer 可执行程序或 Java 类，必须指定且唯一。
- -file, -cacheFile, -cacheArchive：分别用于向计算节点分发本地文件、HDFS 文件和 HDFS 压缩文件。
- -numReduceTasks：指定 reduce 的个数，如果设置 0 或者 NONE 则没有 Reduce 程序，Mapper 的输出直接作为整个作业的输出。
- -jobconf | -D NAME=VALUE：指定作业参数，NAME 是参数名，VALUE 是参数值，可以指定的参数参考 hadoop-default.xml。

特别建议用以下代码设置作业名。

```
-jobconf mapred.job.name='My Job Name'
```

使用以下代码设置作业优先级。

```
-jobconf mapred.job.priority=VERY_HIGH | HIGH | NORMAL | LOW | VERY_LOW
```

使用以下代码设置同时最多运行 M 个 Map 任务。

```
-jobconf mapred.job.map.capacity=M
```

使用以下代码设置同时最多运行 N 个 Reduce 任务。

```
-jobconf mapred.job.reduce.capacity=N
```

8.7.3 作业配置属性

常见的作业配置属性如表 8-8 所示。

表 8-8 常见的作业配置参数

参数名称	功能描述
mapred.job.name	作业名
mapred.job.priority	作业优先级
mapred.job.map.capacity	最多同时运行 Map 任务数
mapred.job.reduce.capacity	最多同时运行 Reduce 任务数
hadoop.job.ugi	作业执行权限
mapred.map.tasks	Map 任务个数
mapred.reduce.tasks	Reduce 任务个数
mapred.job.groups	作业可运行的计算节点分组
mapred.task.timeout	任务没有响应（输入 / 输出）的最大时间
mapred.compress.map.output	Map 的输出是否压缩
mapred.map.output.compression.codec	Map 的输出压缩方式
mapred.output.compress	Reduce 的输出是否压缩
mapred.output.compression.codec	Reduce 的输出压缩方式
stream.map.output.field.separator	Map 输出分隔符

下面是对各个参数的详细说明。

- -combiner：指定 Combiner 的 Java 类，对应的 Java 类文件打包成 JAR 文件后用 -file 分发。
- -partitioner：指定 Partitioner 的 Java 类，Streaming 提供了一些实用的 Partitioner 实现，参考 KeyBasedFiledPartitoner 和 IntHashPartitioner。
- -inputformat 和 -outputformat：指定 InputFormat 和 OutputFormat 的 Java 类，用于读取输入数据和写入输出数据，分别要实现 InputFormat 和 OutputFormat 接口。如果不指定，默认使用 TextInputFormat 和 TextOutputFormat。
- -cmdenv NAME=VALUE：给 Mapper 和 Reducer 程序传递额外的环境变量，NAME 是变量名，VALUE 是变量值。
- -mapdebug 和 -reducedebug：分别指定 Mapper 和 Reducer 程序失败时运行的 Debug 程序。
- -verbose：指定输出详细信息，例如分发哪些文件，实际作业配置参数值等，可以用于调试。

8.7.4 应用实例：抓取网页的标题

Hadoop Streaming 帮助用户创建和运行一类特殊的 MapReduce 作业，这些特殊的 MapReduce 作业是由一些可执行文件或脚本文件充当 Mapper 或者 Reducer。通过 Streaming 工具，程序员可以使用 C++、Python、Perl、Ruby 等语言来开发 MapReduce 任务。

如何用 Python 开发 Hadoop 程序，Hadoop 源代码中提供了一个统计词频的例子 WordCount.py，不过这个例子程序是将 Python 代码通过 Jython 运行时转换为 JAR 包。然

后通过 Streaming 的方式运行这个 JAR 包，稍显麻烦。

德国的开发者 Michael G. Noll 提供了一种 Python 的实现方法：通过 STDIN 标准输入和 STDOUT 标准输出在 Map 和 Reduce 代码之间传输数据。Python 的 sys.stdin 方法读取输入数据，并通过 sys.stdout 打印输出数据，而数据的交互处理过程交由 Hadoop Streaming 来处理。

用 Python 写的典型示例程序 Hadoop Streaming，功能就是抓取网页的标题。

现在有一个输入文件 test.txt 内容为：

```
http://www.souhu.com
http://www.sina.com
http://www.baidu.com
http://www.autonavi.com
```

输出结果为：

```
http://www.souhu.com  搜狐
http://www.sina.com  新浪首页
http://www.baidu.com  百度一下，你就知道
http://www.autonavi.com  高德导航
```

下面看怎么用 Streaming 来实现这个功能。mapper.py 类编写如代码清单 8-18 所示。

代码清单 8-18 mapred\streaming\ mapper.py

```
#!/usr/bin/env python
import sys, urllib, re
title_re = re.compile("<title>(.*?)</title>",
re.MULTILINE | re.DOTALL | re.IGNORECASE)
for line in sys.stdin:
    url = line.strip()
    match =
title_re.search(urllib.urlopen(url).read())
    if match:
        print "mapper", url, "\t",
match.group(1).strip()
```

reducer.py 类编写如代码清单 8-19 所示。

代码清单 8-19 mapred\streaming\ reducer.py

```
#!/usr/bin/env python
from operator import itemgetter
import sys
for line in sys.stdin:
line = line.strip()
print "reducer" + line
```

运行上面 Streaming 的 MapReduce 程序，代码如下：

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop jar contrib/streaming/hadoop-1.0.3-
streaming.jar \
-mapper mapper.py \
```

```
-file mapper.py \
-reducer reducer.py \
-file reducer.py \
-input /test/input \test.txt
-output /user/output \
-jobconf mapred.reduce.tasks=3 \
-jobconf mapred.job.name= "my crawl job"
```

查看结果如下。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$  hadoop fs -text  /user/output/
http://www.souhu.com  搜狐
http://www.sina.com  新浪首页
http://www.baidu.com  百度一下，你就知道
http://www.autonavi.com  高德导航
```

直接用 Python 语言开发 Hadoop 程序有时显得不够灵活，比如 mapper.py 和 reducer.py 程序必须放在两个文件中。于是国外有人用 Python 对 Hadoop API 进行了封转。著名的音乐站点 Last.fm 发布了基于 Python 的 Dumbo（小飞象）项目，Dumbo 能够帮助 Python 开发者更方便地编写 Hadoop 应用，并且 Dumbo 为 MapReduce 应用提供了灵活易用的 Python API。Last.fm 的开发者，同时也是 Dumbo 项目发起人 Klaas Bosteels 认为，对于定制 Hadoop 应用，使用 Python 语言代替 Java 会让工作变得更有效率。

8.8 本章小结

本章主要讲解 MapReduce 编程开发用到的一些知识点，如 Combiner、Partitioner、DistributedCache 等，全面介绍了 Mapper 和 Reducer 两个抽象类，最后介绍了如何使用 Hadoop Streaming 接口开发 MapReduce 程序。希望读者仔细认真阅读本章内容，为下一章节 MapReduce 的高级应用做好铺垫。

第9章　MapReduce 高级应用

MapReduce 组件中包含许多高级的应用，如内置计数器（Counter）、ChainMapper、ChainReducer、二次排序等，这些高级特性在 MapReduce 性能调优和复杂业务的情况下是非常有用的，本章我们介绍一下 MapReduce 开发常用的一些高级应用。

9.1　计数器

MapReduce 计数器（Counter）为提供我们一个窗口，用于观察 MapReduce Job 运行期的各种细节数据。对 MapReduce 性能调优很有帮助，MapReduce 性能优化的评估大部分都是基于这些 Counter 的数值表现出来的。MapReduce 自带了许多默认 Counter，现在我分析这些默认 Counter 的含义，方便大家观察 Job 结果，如输入的字节数、输出的字节数、Map 端输入 / 输出的字节数和条数、Reduce 端输入 / 输出的字节数和条数等。

9.1.1　默认计数器

Counter 有组（group）的概念，用于表示逻辑上相同范围的所有数值。MapReduce Job 提供的默认 Counter 分为五个组，如表 9-1 所示一个 MapReduce 任务运行完成以后产生的计数器信息。

表 9-1　MapReduce 默认的 Counter 组

Group	Counter	Map	Reduce	Total
Job Counters	SLOTS_MILLIS_MAPS	0	0	82 984
	Launched reduce tasks	0	0	1
	Total time spent by all reduces waiting after reserving slots (ms)	0	0	0
	Total time spent by all maps waiting after reserving slots (ms)	0	0	0
	Launched map tasks	0	0	1
	Data-local map tasks	0	0	1
	SLOTS_MILLIS_REDUCES	0	0	18 104
File Output Format Counters	Bytes Written	0	77	77
File Input Format Counters	Bytes Read	130	0	130
File System Counters	FILE_BYTES_READ	0	136	136
	HDFS_BYTES_READ	239	0	239
	FILE_BYTES_WRITTEN	21 633	21 602	43 235
	HDFS_BYTES_WRITTEN	0	77	77

（续）

Group	Counter	Map	Reduce	Total
MapReduce Framework	Map output materialized bytes	136	0	136
	Map input records	6	0	6
	Reduce shuffle bytes	0	0	0
	Spilled Records	6	6	12
	Map output bytes	118	0	118
	CPU time spent (ms)	1 370	1 670	3 040
	Total committed heap usage (bytes)	202 768 384	7 929 856	210 698 240
	Combine input records	0	0	0
	SPLIT_RAW_BYTES	109	0	109
	Reduce input records	0	6	6
	Reduce input groups	0	4	4
	Combine output records	0	0	0
	Physical memory (bytes) snapshot	139 141 120	37 294 080	176 435 200
	Reduce output records	0	4	4
	Virtual memory (bytes) snapshot	372 322 304	375 513 088	747 835 392
	Map output records	6	0	6

接下来分别介绍一下每个属性的含义。

1. Job Counters 计数器组

Job Counters 描述与 Job 调度相关的统计，属性解释如表 9-2 所示。

表 9-2 Job Counters 计数器组属性

Counter	Map	Reduce	Total	功能描述
SLOTS_MILLIS_MAPS	0	0	82 984	所有 Map Task 占用 slot 的总时间，包含执行时间和创建 / 销毁子 JVM 的时间
Launched reduce tasks	0	0	1	此 Job 启动了多少个 Map Task 任务
Total time spent by all reduces waiting after reserving slots (ms)	0	0	0	当前 Job 为某些 Reduce Task 的执行保留了 slot，总共保留的时间是多少
Total time spent by all maps waiting after reserving slots (ms)	0	0	0	当前 Job 为某些 Map Task 的执行保留了 slot，总共保留的时间是多少
Launched map tasks	0	0	1	此 Job 启动了多少个 Reduce Task 任务
Data-local map tasks	0	0	1	Job 在被调度时，如果启动了一个 data-local（源文件的副本在执行 Map Task 的 TaskTracker 本地）
SLOTS_MILLIS_REDUCES	0	0	18 104	所有 Reduce Task 占用 slot 的总时间，包含执行时间和创建 / 销毁子 JVM 的时间

2. File Output Format Counters 计算器组

File Output Format Counters 表示 Reduce Task 写入文件内容的总大小，属性如表 9-3 所示。

表 9-3 File Output Format Counters 计数器组属性

Counter	Map	Reduce	Total	功能描述
Bytes Written	0	77	77	Reduce Task 写入文件的总大小

3. File Input Format Counters 计算器组

File Input Format Counters 表示 Map Task 读取文件内容（总输入数据），属性如表 9-4 所示。

表 9-4 File Input Format Counters 计数器组属性

Counter	Map	Reduce	Total	功能描述
Bytes Read	130	0	130	Map Task 的所有输入数据（字节），等于各个 Map Task 的 map 方法传入的所有 value 值字节之和

4. File System Counters 计算器组

File System Counters 表示 Job 与文件系统交互的读写统计，属性如表 9-5 所示。

表 9-5 File System Counters 计数器属性

Counter	Map	Reduce	Total	功能描述
FILE_BYTES_READ	0	136	136	Reduce 读取 Map 结果的数据大小
HDFS_BYTES_READ	239	0	239	Map 读取 HDFS 的数据大小
FILE_BYTES_WRITTEN	21 633	21 602	43 235	整个 Job 过程中写入文件的数据大小
HDFS_BYTES_WRITTEN	0	77	77	Reduce 写入 HDFS 数据大小

5. MapReduce Framework 计算器组

MapReduce Framework 包含了相当多的 Job 执行细节数据。这里需要有个概念认识，一般情况下，record 表示一行数据，byte 表示这行数据所占的字节数，group 表示经过 reduce merge 操作后的记录条数，如 { “bbb”, [3, 4, 7, 9, …]}。属性如表 9-6 所示。

表 9-6 MapReduce Framework 计数器组属性

Counter	Map	Reduce	Total	功能描述
Map output materialized bytes	136	0	136	Map 输出数据总大小
Map input records	6	0	6	Map 读记录的条数
Reduce shuffle bytes	0	0	0	Reduce 洗牌的数据大小
Spilled Records	6	6	12	Map 和 Reduce 读入总条数
Map output bytes	118	0	118	Map 输出数据总和
CPU time spent (ms)	1 370	1 670	3 040	CPU 执行的时间总和
Total committed heap usage (bytes)	202 768 384	7 929 856	210 698 240	JVM 堆使用的总大小
Combine input records	0	0	0	使用 Combine 输入的总条数
SPLIT_RAW_BYTES	109	0	109	切割行的总字节数
Reduce input records	0	6	6	Reduce 输入的总字节数

（续）

Counter	Map	Reduce	Total	功能描述
Reduce input groups	0	4	4	Reduce 处理的 Key 的个数
Combine output records	0	0	0	使用 Combine 输出的总记录数
Physical memory (bytes) snapshot	139 141 120	37 294 080	176 435 200	机器物理内存使用的大小
Reduce output records	0	4	4	Reduce 输出的总条数
Virtual memory (bytes) snapshot	372 322 304	375 513 088	747 835 392	机器虚拟内存使用的总大小

了解这些 Hadoop 自带计数器是非常重要的，这些计数器记录了 MapReduce 程序执行的所有信息，如输入文件的大小、条数，Map 中间结果的大小、条数，输入文件的大小、条数等好多有用的信息，对于 MapReduce 程序的调优是必不可少的。

9.1.2 自定义计数器

自定义计数器用的比较广泛，特别是统计日志错误条数的时候，我们就会用到计数器来记录错误日志的条数。下面我们实现一个自定义的计数器，功能为非法日志的行数。

假如有一个 log.txt 文件，内容如图 9-1 所示。

响应时间	IP	请求类型	响应状态
23	192.168.10.123	GET	200
24	192.168.10.124		404（不合法）
25	192.168.10.125	GET	505
23	192.168.10.123	GET	200
24	192.168.10.124		404（不合法）
25	192.168.10.125	GET	505

图 9-1 log.txt 文件

这里把长度为 4 认为是标准日志，即合法日志；长度为 3 认为不合法日志，即日志有丢失。统计不合法日志代码的实现如代码清单 9-1 所示。

代码清单 9-1 mapred\counter\ MyCounter.java

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Counter;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

```
import org.apache.hadoop.util.GenericOptionsParser;

public class MyCounter {
    public static class MyCounterMap extends Mapper<LongWritable, Text, Text, Text> {
        public static Counter ct = null;
        protected void map(LongWritable key, Text value,
                org.apache.hadoop.mapreduce.Mapper<LongWritable,
                Text, Text, Text>.Context context)
                throws java.io.IOException, InterruptedException {
            String arr_value[] = value.toString().split( "/t" );
            if (arr_value.length < 4) {
                ct = context.getCounter("ErrorCounter","ERROR_LOG_TIME");
                ct.increment(1);
            }
        }
    }

    public static void main(String[] args) throws IOException,
           InterruptedException, ClassNotFoundException {
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
        if (otherArgs.length != 2) {
            System.err.println("Usage: MyCounter <in> <out>");
            System.exit(2);
        }

        Job job = new Job(conf, "MyCounter");
        job.setJarByClass(MyCounter.class);
        job.setMapperClass(MyCounterMap.class);

        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

对于小于 4 个字段的日志自定义一个 Counter 为 ERROR_LOG_TIME 来统计，increment(1)方法用来统计个数，出现一次，ERROR_LOG_TIME 计数器就加 1。运行代码命令如下。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$  hadoop jar MyCounter.jar mapred.counter.MyCounter
/user/hadoop/input/log.txt    /user/hadoop/output/
```

控制台输出结果如图 9-2 所示。可以看到 ERROR_LOG_TIME=2，而 log.txt 文件中长度小于 4 的记录有 2 条，与我们期待的结果一样。

```
13/05/04 17:29:15 INFO mapred.JobClient: Job complete: job_201305042120_0019
13/05/04 17:29:15 INFO mapred.JobClient: Counters: 18
13/05/04 17:29:15 INFO mapred.JobClient:   Job Counters
13/05/04 17:29:15 INFO mapred.JobClient:     Launched reduce tasks=1
13/05/04 17:29:15 INFO mapred.JobClient:     Rack-local map tasks=1
13/05/04 17:29:15 INFO mapred.JobClient:     Launched map tasks=6
13/05/04 17:29:15 INFO mapred.JobClient:   ErrorCounter
13/05/04 17:29:15 INFO mapred.JobClient:     ERROR_LOG_TIME=2
13/05/04 17:29:15 INFO mapred.JobClient:   FileSystemCounters
13/05/04 17:29:15 INFO mapred.JobClient:     FILE_BYTES_READ=6
13/05/04 17:29:15 INFO mapred.JobClient:     HDFS_BYTES_READ=47
13/05/04 17:29:15 INFO mapred.JobClient:     FILE_BYTES_WRITTEN=234
13/05/04 17:29:15 INFO mapred.JobClient:   Map-Reduce Framework
13/05/04 17:29:15 INFO mapred.JobClient:     Reduce input groups=0
13/05/04 17:29:15 INFO mapred.JobClient:     Combine output records=0
13/05/04 17:29:15 INFO mapred.JobClient:     Map input records=5
13/05/04 17:29:15 INFO mapred.JobClient:     Reduce shuffle bytes=36
13/05/04 17:29:15 INFO mapred.JobClient:     Reduce output records=0
13/05/04 17:29:15 INFO mapred.JobClient:     Spilled Records=0
13/05/04 17:29:15 INFO mapred.JobClient:     Map output bytes=0
13/05/04 17:29:15 INFO mapred.JobClient:     Combine input records=0
13/05/04 17:29:15 INFO mapred.JobClient:     Map output records=0
13/05/04 17:29:15 INFO mapred.JobClient:     Reduce input records=0
```

图 9-2 自定义计数器结果图

9.1.3 获取计数器

在 Hadoop 作业执行完毕后，终端中会返回程序各个计数器的值。此外，还有两种方法获取计数器的值。

- 在 JobTracker 的 Web 页面中查看每个 Job 和 Job 中的每个 Task 的计数器值。
- 使用命令“hadoop job -counter [-counter]”获取计数器的值。

Hadoop 中，用户也可以使用 Java API 获取计数器的值，Java API 支持在 Job 运行期间就能获取计数器的值如下例，统计缺损日志记录所占的比例：

```
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class  ErrorLogPercent  extends Configured implements Tool {
    @Override
    public int run(String[] args) throws Exception {
        if (args.length != 1) {
            JobBuilder.printUsage(this, "<job ID>");
            return -1;
        }
        String jobID = args[0];
        JobClient jobClient = new JobClient(new JobConf(getConf()));
        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()) { // 判断 Job 是否运行完成
            System.err.printf("Job %s is not complete.\n", jobID);
            return -1;
        }
```

```
        Counters counters = job.getCounters();
        long missing = counters.getCounter
                    (new MyCounter.MyCounterMap().getCounter("ERROR_LOG_TIME"));
        long total = counters.getCounter(Task.Counter.MAP_INPUT_RECORDS);
        System.out.printf("Records with error log percent: %.2f%%\n",
                100.0 * missing / total);
        return 0;
    }

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new MyCounter (), args);
        System.exit(exitCode);
    }
}
```

在这个程序中使用了计数器 Java API 函数，Counters 类的 getCounter() 方法会返回一个 Counter 对象，这个对象包含了这个作业的所有用户自定义计数器和自带的计数器。

9.2 MapReduce 二次排序

默认情况下，Map 输出的结果会对 Key 进行默认排序，但是有时候需要对 Key 排序的同时再对 Value 进行排序，这时候就要用到二次排序。

9.2.1 二次排序原理

在 Map 阶段，使用 job.setInputFormatClass 定义的 InputFormat 将输入的数据集分割成小数据块 splite，同时 InputFormat 提供一个 RecordReader 的实现。本例子中使用的是 TextInputFormat，它提供的 RecordReader 会将文本的行号作为 Key，这一行的文本作为 Value。这就是自定义 Mapper 的输入是 <LongWritable, Text> 的原因。然后调用自定义 Mapper 的 map 方法，将一个个 <LongWritable, Text> 对输入给 Mapper 的 map 方法。

注意 输出应该符合自定义 Mapper 中定义的输出 <IntPair, IntWritable>。最终是生成一个 List<IntPair, IntWritable>。

在 Map 阶段的最后，会先调用 job.setPartitionerClass 对这个 List 进行分区，每个分区映射到一个 Reducer。每个分区内又调用 job.setSortComparatorClass 设置的 Key 比较函数类排序。可以看到，这本身就是一个二次排序。如果没有通过 job.setSortComparatorClass 设置 Key 比较函数类，则使用 Key 实现的 compareTo 方法。在第一个例子中，使用了 IntPair 实现的 compareTo 方法，而在下一个例子中，专门定义了 Key 比较函数类。

在 Reduce 阶段，reduce 接收到所有映射到这个 Reduce 的 map 输出后，也是会调用 job.setSortComparatorClass 设置的 Key 比较函数类对所有数据对排序。然后开始构造一个 Key 对应的 Value 迭代器。这时就要用到分组，使用 job.setGroupingComparatorClass 设置的分组函数类。只要这个比较器比较的两个 Key 相同，它们就属于同一个组，它们的 Value

放在一个 Value 迭代器，而这个迭代器的 Key 使用属于同一个组的所有 Key 的第一个 Key。最后就是进入 Reducer 的 reduce 方法，reduce 方法的输入是所有的 Key 和它的 Value 迭代器。同样注意输入与输出的类型必须与自定义的 Reducer 中声明的一致。

看下面的示例。输入文件 sort.txt 内容为：

```
40  20
40  10
40  30
40  5
30  30
30  20
30  10
30  40
50  20
50  50
50  10
50  60
```

输出文件内容（从小到大排序）如下：

```
  30  10
  30  20
  30  30
  30  40
================================================
  40  5
  40  10
  40  20
  40  30
================================================
  50  10
  50  20
  50  50
  50  60
```

从输出的结果可以看出 Key 实现了从小到大的排序，同时相同 Key 的 Value 也实现了从小到大的排序，这就是二次排序的结果，接下来我们用代码实现上面这个示例。

9.2.2 二次排序的算法流程

在 MapReduce 中，所有的 Key 是需要被比较和排序的，并且是二次，先根据 Partitioner，再根据大小。而本例中也是要比较两次。先按照第一字段排序，然后再对第一字段相同的按照第二字段排序。根据这一点，我们可以构造一个复合类 IntPair，它有两个字段，先利用分区对第一字段排序，再利用分区内的比较对第二字段排序。

二次排序的算法流程图如图 9-3 所示。

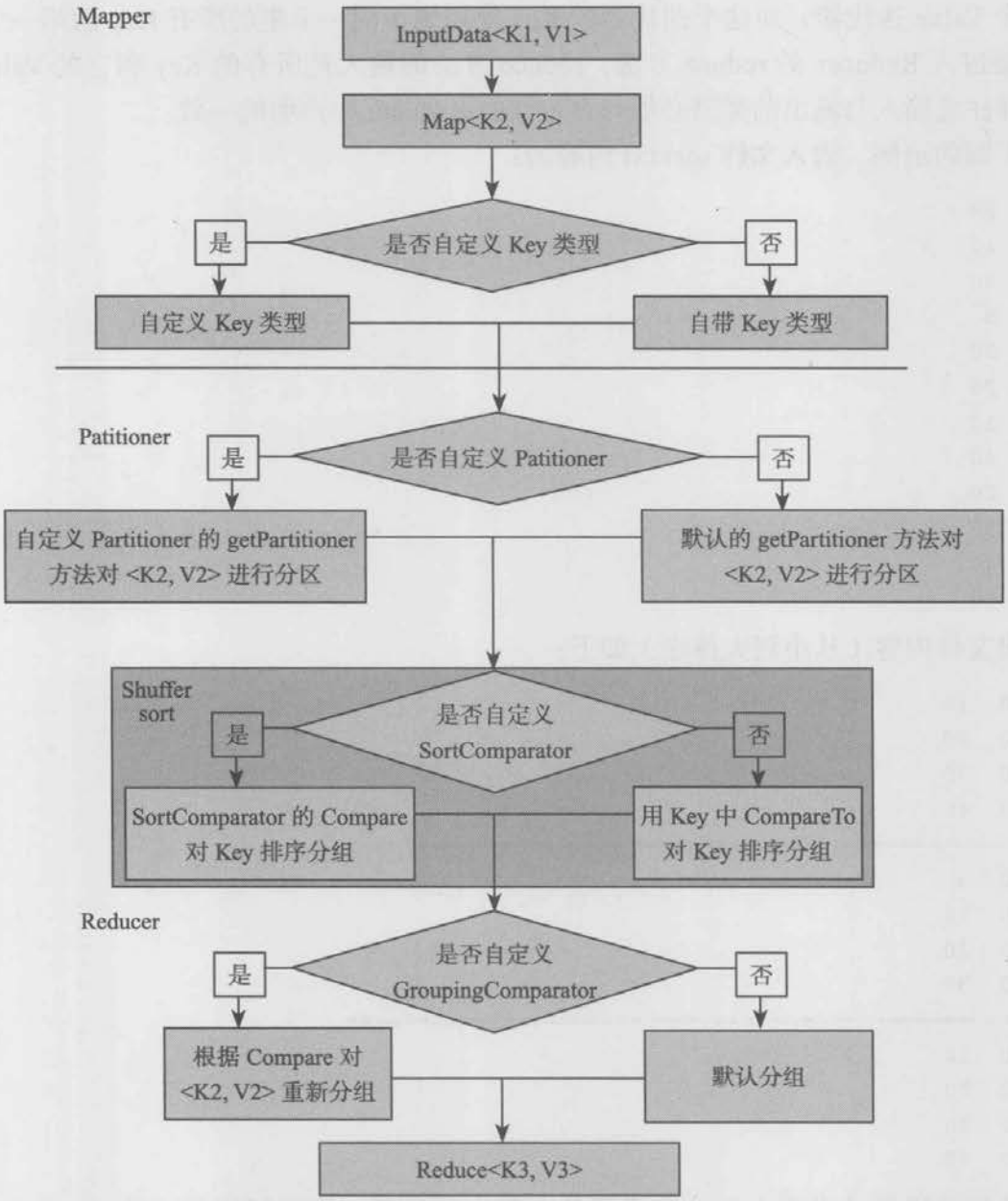


图 9-3 二次排序算法

1. 自定义 Key

所有自定义的 Key 应该实现接口 WritableComparable，因为是可序列的并且可比较的。WritableComparable 的方法如下。

```
// 反序列化，从流中的二进制转换成 IntPair
public void readFields(DataInput in) throws IOException
// 序列化，将 IntPair 转化成使用流传送的二进制
public void write(DataOutput out)
//key 的比较
```

```
public int compareTo(IntPair o)
// 另外新定义的类应该重写的两个方法
//The hashCode() method is used by the HashPartitioner (the default partitioner
//in MapReduce)
public int hashCode()
public boolean equals(Object right)
```

2. 自定义分区

自定义分区函数类。这是 Key 的第一次比较。

```
public static class FirstPartitioner extends Partitioner<IntPair,IntWritable>
```

提示 在 Job 中使用 setPartitionerClasss 设置 Partitioner。

3. Key 的比较类

这是 Key 的第二次比较。这是一个比较器，需要继承 WritableComparator。

```
public static class KeyComparator extends WritableComparator
```

必须有一个构造函数，并且重载以下方法。

```
public int compare(WritableComparable w1, WritableComparable w2)。
```

另一种方法是实现接口 RawComparator。

提示 在 Job 中设置使用 setSortComparatorClass。

4. 定义分组类函数

在 Reduce 阶段，构造一个与 Key 相对应的 Value 迭代器的时候，只要 first 相同就属于同一个组，放在一个 Value 迭代器。这是一个比较器，需要继承 WritableComparator。

```
public static class GroupingComparator extends WritableComparator
```

同 Key 比较函数类，必须有一个构造函数，并且重载如下方法。

```
public int compare(WritableComparable object1, WritableComparable object2)
```

同 Key 比较函数类，分组函数类另一种方法是实现接口 RawComparator。

提示 在 Job 中设置使用 setGroupingComparatorClass。另外注意的是，如果 Reduce 的输入与输出不是同一种类型，不要定义 Combiner 也使用 Reduce，因为 Combiner 的输出是 Reduce 的输入，除非重新定义一个 Combiner。

9.2.3 代码实现

Hadoop 的 example 包中自带了一个 MapReduce 的二次排序算法，下面这个示例是对 example 包中二次排序源码的改进。实现如代码清单 9-2 所示。

代码清单 9-2 mapred\sort\ SecondarySort.java

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import java.util.StringTokenizer;
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.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Partitioner;
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 SecondarySort {
  public static class IntPair implements WritableComparable<IntPair> {
    int first;
    int second;
    public void set(int left, int right) {
      first = left;
      second = right;
    }
    public int getFirst() {
      return first;
    }
    public int getSecond() {
      return second;
    }
    @Override
    public void readFields(DataInput in) throws IOException {
      // TODO Auto-generated method stub
      first = in.readInt();
      second = in.readInt();
    }
    @Override
    public void write(DataOutput out) throws IOException {
      // TODO Auto-generated method stub
      out.writeInt(first);
      out.writeInt(second);
    }
    @Override
```

```
    public int compareTo(IntPair o) {
      // TODO Auto-generated method stub
      if (first != o.first) {
        return first <o.first ? -1 : 1;
      } else if (second != o.second) {
        return second <o.second ? -1 : 1;
      } else {
        return 0;
      }
    }
    @Override
    //The hashCode() method is used by the HashPartitioner
    //(the default partitioner in MapReduce)
    public int hashCode() {
      return first * 157 + second;
    }
    @Override
    public boolean equals(Object right) {
      if (right == null)
        return false;
      if (this == right)
        return true;
      if (right instanceof IntPair) {
        IntPair r = (IntPair) right;
        return r.first == first &&r.second == second;
      } else {
        return false;
      }
    }
  }
  public static class FirstPartitioner extends Partitioner<IntPair,IntWritable>{
    @Override
    public int getPartition(IntPair key, IntWritable value,int numPartitions) {
      return Math.abs(key.getFirst() * 127) % numPartitions;
    }
  }
  public static class GroupingComparator implements RawComparator<IntPair> {
    @Override
    public int compare(IntPair o1, IntPair o2) {
      int l = o1.getFirst();
      int r = o2.getFirst();
      return l == r ? 0 : (l < r ? -1 : 1);
  }
  @Override
  public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2){
    // TODO Auto-generated method stub
    return WritableComparator.compareBytes(b1, s1, Integer.SIZE/8,
                b2, s2, Integer.SIZE/8);
  }
}*/
```

```
    public static class GroupingComparator extends WritableComparator {
      protected GroupingComparator() {
        super(IntPair.class, true);
      }
      @Override
      //Compare two WritableComparables.
      public int compare(WritableComparable w1, WritableComparable w2) {
        IntPair ip1 = (IntPair) w1;
        IntPair ip2 = (IntPair) w2;
        int l = ip1.getFirst();
      int r = ip2.getFirst();
      return l == r ? 0 : (l < r ? -1 : 1);
      }
    }
    public static class Map extends Mapper<LongWritable,
         Text, IntPair, IntWritable> {
      private final IntPair intkey = new IntPair();
      private final IntWritable intvalue = new IntWritable();
      public void map(LongWritable key, Text value, Context context)
           throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizertokenizer = new StringTokenizer(line);
        int left = 0;
        int right = 0;
        if (tokenizer.hasMoreTokens()) {
          left = Integer.parseInt(tokenizer.nextToken());
          if (tokenizer.hasMoreTokens())
              right = Integer.parseInt(tokenizer.nextToken());
          intkey.set(left, right);
          intvalue.set(right);
          context.write(intkey, intvalue);
        }
      }
    }
    public static class Reduce extends Reducer<IntPair, IntWritable, Text,
IntWritable> {
      private final Text left = new Text();
      private static final Text SEPARATOR =
           new Text("======================================");
    public void reduce(IntPair key, Iterable<IntWritable>values,
       Context context) throws IOException, InterruptedException {
      context.write(SEPARATOR, null);
      left.set(Integer.toString(key.getFirst()));
      for (IntWritable val : values) {
        context.write(left, val);
      }
     }
    }
```

```
    public static void main(String[] args) throws IOException, InterruptedException,
                ClassNotFoundException {
      Configuration conf = new Configuration();
      Job job = new Job(conf, "mysecondarysort");
      job.setJarByClass(SecondarySort.class);
      job.setMapperClass(Map.class);
      job.setReducerClass(Reduce.class);
      job.setPartitionerClass(FirstPartitioner.class);
      job.setGroupingComparatorClass(GroupingComparator.class);
      job.setMapOutputKeyClass(IntPair.class);
      job.setMapOutputValueClass(IntWritable.class);
      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(IntWritable.class);
      job.setInputFormatClass(TextInputFormat.class);
      job.setOutputFormatClass(TextOutputFormat.class);
      FileInputFormat.setInputPaths(job, new Path(args[0]));
      FileOutputFormat.setOutputPath(job, new Path(args[1]));
      System.exit(job.waitForCompletion(true) ? 0 : 1);
     }
    }
```

运行步骤可以参照第 3 章 MapReduce 快速入门里面运行部署 MapReduce 程序的方法。运行代码如下。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop jar myscondarysort.jar
              mapred.sort. SecondarySort
/user/hadoop/input/sort.txt   /user/hadoop/output/
```

运行结果应该和 9.2.1 节中介绍的示例结果是一样的。查看结果的命令如下。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$  hadoop fs -text  /user/hadoop/output/
part-r-0000
    30  10
    30  20
    30  30
    30  40
=====================================
    40  5
    40  10
    40  20
    40  30
=====================================
    50  10
    50  20
    50  50
    50  60
```

至此，MapReduce 的二次排序的原理和实现就介绍完了，理解起来有点难度，希望读者先看代码，把代码跑起来以后再去看前面的原理，这样理解起来就非常容易。

9.3 MapReduce 中的 Join 算法

在关系型数据库中 Join 是非常常见的操作，各种优化手段已经到了极致。在海量数据的环境下，不可避免的也会碰到这种类型的需求，例如在数据分析时需要连接从不同的数据源中获取到的数据。不同于传统的单机模式，在分布式存储的下采用 MapReduce 编程模型，也有相应的处理措施和优化方法。

9.3.1 Reduce 端 Join

假设两个表所在的文件分别为 Customers 和 Orders，以 CSV 格式存储在 HDFS 中[⊖]。

Customers.csv 文件的内容：

```
1,Stephanie Leung,555-555-5555
2,Edward Kim,123-456-7890
3,Jose Madriz,281-330-8004
4,David Stork,408-555-0000
```

Orders.csv 文件的内容：

```
3,A,12.95,02-Jun-2008
1,B,88.25,20-May-2008
2,C,32.00,30-Nov-2007
3,D,25.02,22-Jan-2009
```

这里 Customer ID 是连接的键，连接的结果如下。

```
1,Stephanie Leung,555-555-5555,B,88.25,20-May-2008
2,Edward Kim,123-456-7890,C,32.00,30-Nov-2007
3,Jose Madriz,281-330-8004,A,12.95,02-Jun-2008
3,Jose Madriz,281-330-8004,D,25.02,22-Jan-2009
```

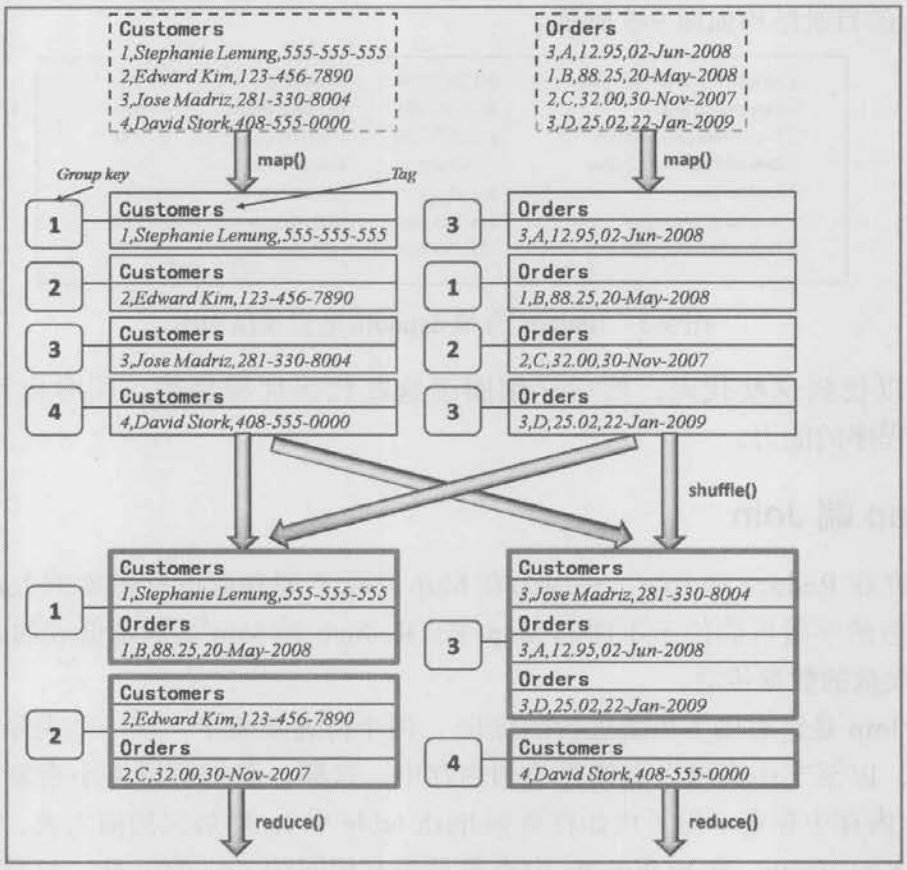
提示 回忆 Hadoop 中 MapReduce 的主要几个过程：依次是读取数据分块、Map 操作、Shuffle 操作、Reduce 操作，然后输出结果。简单来说，其本质在于大而化小，分拆处理。

显然我们想到的是将两个数据表中键值相同的元组放到同一个 Reduce 结点进行，关键问题在于如何做到？具体处理方法是将 Map 操作输出的 Key 值设为两表的连接键（如例子中的 Customer ID），在 Shuffle 阶段，Hadoop 中默认的 Partitioner 会将相同 Key 值的 Map 输出发送到同一个 Reduce 结点。整个过程如图 9-4 所示。

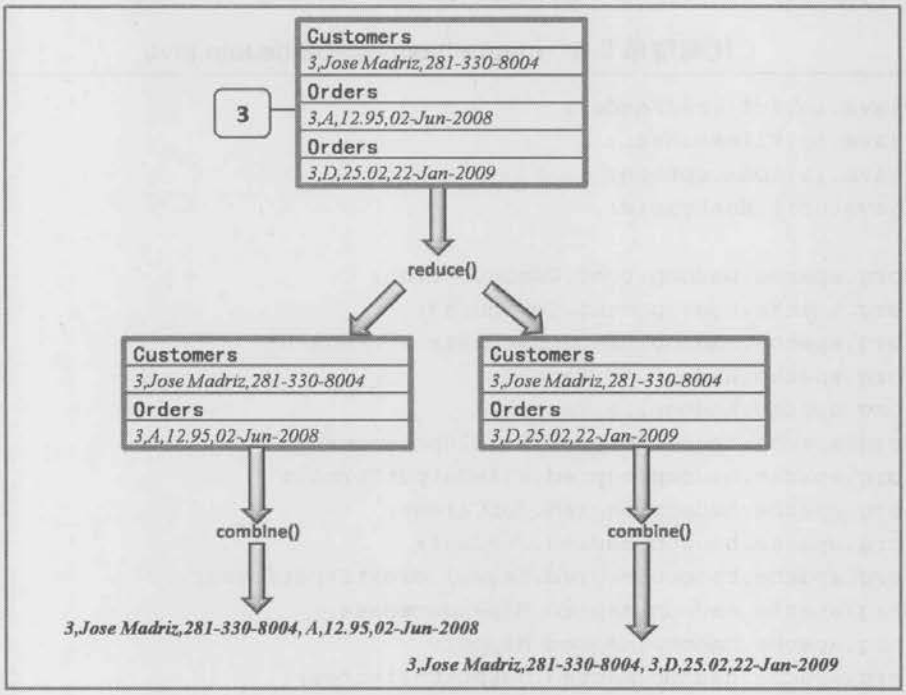
这种方法称为 Repartition Join，同时它进行 Join 操作是在 Reduce 阶段进行，也属于 Reduce 端 Join；在 Hadoop 中 contrib 目录下的 datajoin 案例就是采用的这种方法。实现代码可以参照 Hadoop 自带的 datajoin 案例[⊖]。

⊖ 文中所采用的例子选自《Hadoop in Action》一书中的 5.2 节。作者：Chuck Lam，出版社：Manning Publications，ISBN: 9781935182191。

⊖ 地址为 hadoop-1.0.3\src\contrib\ data_join\ src\ java\org\apache\hadoop\contrib。



a）Reduce Join 的 Map 流程



b）Reduce Join 的 Reduce 流程

图 9-4　Reduce Join 流程

datajoin 的目录结构如图 9-5 所示。

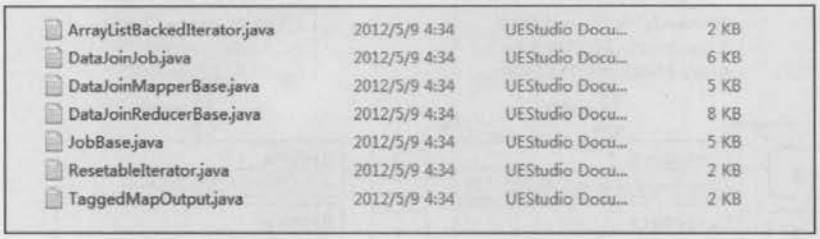


图 9-5 Hadoop 自带 datajoin 的目录结构图

读者可以找到这些代码，然后仔细阅读这些代码加深影响，同时也可以提高编写 MapReduce 程序的能力。

9.3.2 Map 端 Join

之所以存在 Reduce 端 Join，是因为在 Map 阶段不能获取所有需要的 Join 字段，即同一个 Key 对应的字段可能位于不同的 Map 中。Reduce 端 Join 是非常低效的，因为 Shuffle 阶段要进行大量的数据传输。

Map 端 Join 是针对以下场景进行的优化：两个待连接表中，有一个表非常大，而另一个表非常小，以至于小表可以直接存放到内存中。这样，我们可以将小表复制多份，让每个 Map Task 内存中存在一份（比如存放到 hash table 中），然后只扫描大表，对于大表中的每一条记录 Key-Value，在 hash table 中查找是否有相同的 Key 的记录，如果有，则连接后输出即可。MapReduce 的 Map 端 Join 算法的实现如代码清单 9-3 所示。

代码清单 9-3 mapred\join\ MapSideJoin.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Hashtable;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.filecache.DistributedCache;
import org.apache.hadoop.fs.Path;
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.KeyValueTextInputFormat;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.TextOutputFormat;
```

```
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

@SuppressWarnings("deprecation")
public class MapSideJoin extends Configured implements Tool{
  private final static String inputa = "hdfs://Ubuntu1:9000/user/hadoop/join/Customers.csv";
  private final static String inputb = "hdfs:// Ubuntu1:9000/user/hadoop/join/Orders.csv";
  private final static String output = "hdfs:// Ubuntu1:9000/user/hadoop/join/output";
  public static class MapClass extends MapReduceBase
       implements Mapper<Text, Text, Text, Text> {
    private Hashtable<String, String>joinData = new Hashtable<String, String>();
    @Override
    public void configure(JobConf conf) {
      try {
        Path [] cacheFiles = DistributedCache.getLocalCacheFiles(conf);
        if (cacheFiles != null &&cacheFiles.length> 0) {
          String line;
          String[] tokens;
          BufferedReader joinReader = new BufferedReader(
                          new FileReader(cacheFiles[0].toString()));
          try {
            while ((line = joinReader.readLine()) != null) {
              tokens = line.split(",", 2);
              joinData.put(tokens[0], tokens[1]);
            }
          }finally {
            joinReader.close();
          }}} catch (IOException e) {
            System.err.println("Exception reading DistributedCache: " + e);
          }
       }
    public void map(Text key, Text value,OutputCollector<Text, Text> output,
           Reporter reporter) throws IOException {
      String joinValue = joinData.get(key.toString());
      if (joinValue != null) {
        output.collect(key,new Text(value.toString() + "," + joinValue));
      }
    }
  }

  @Override
  public int run(String[] args) throws Exception {
    Configuration conf = getConf();
    DistributedCache.addCacheFile(new Path(inputa).toUri(), conf);
    JobConf job = new JobConf(conf, DataJoinDC.class);
    Path in = new Path(inputb);
    Path out = new Path(output);
    FileInputFormat.setInputPaths(job, in);
    FileOutputFormat.setOutputPath(job, out);
    job.setJobName("DataJoin with DistributedCache");
```

```
        job.setMapperClass(MapClass.class);
        job.setNumReduceTasks(0);
        job.setInputFormat(KeyValueTextInputFormat.class);
        job.setOutputFormat(TextOutputFormat.class);
        job.set("key.value.separator.in.input.line", ",");
        JobClient.runJob(job);
        return 0;
      }

      public static void main(String[] args) throws Exception{
        int res = ToolRunner.run(new Configuration(), new DataJoinDC(), args);
        System.exit(res);
      }
    }
```

运行命令如下所示。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop jar mapsidejoin.jar mapred.join. MapSideJoin
```

查看结果如下。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$  hadoop fs -text  /user/hadoop/join/output/
part-r-0000
1,Stephanie Leung,555-555-5555,B,88.25,20-May-2008
2,Edward Kim,123-456-7890,C,32.00,30-Nov-2007
3,Jose Madriz,281-330-8004,A,12.95,02-Jun-2008
3,Jose Madriz,281-330-8004,D,25.02,22-Jan-2009
```

这里是将 Customers 表作为较小的表传入 DistributedCache，代码如下。

```
DistributedCache.addCacheFile(new Path(inputa).toUri(), conf);
```

这句一定要放在 Job 初始化之前，否则在 Map 中读取不到文件。因为 Job 初始化时将传入 Configuration 对象复制一份给了 JobContext。

9.3.3 半连接 Semi Join

Semi Join（半连接）是从分布式数据库中借鉴来的方法。产生动机是：对于 Reduce 端 Join，跨机器的数据传输量非常大，这成了 Join 操作的一个瓶颈，如果能够在 Map 端过滤掉不会参加 Join 操作的数据，则可以大大节省网络 I/O。

实现方法很简单，选取一个小文件，假设是 File1，将其参与 Join 的 Key 抽取出来，保存到文件 File3 中，File3 文件一般很小，可以放到内存中。在 Map 阶段，使用 DistributedCache 将 File3 复制到各个 TaskTracker 上，然后将 File2 中不在 File3 中的 Key 对应的记录过滤掉，剩下的 Reduce 阶段的工作与 Reduce 端 Join 相同。

9.4 MapReduce 从 MySQL 读写数据

Hadoop 访问关系数据库主要通过 DBInputFormat 类实现，包的位置在 org.apache.hadoop.mapred.lib.db。

9.4.1 读数据

DBInputFormat 在 Hadoop 应用程序中通过数据库供应商提供的 JDBC 接口来与数据库进行交互，并且可以使用标准的 SQL 来读取数据库中的记录。学习 DBInputFormat 首先必须知道如下两个条件。

- 在使用 DBInputFormat 之前，必须将要使用的 JDBC 驱动复制到分布式系统各个节点的 $HADOOP_HOME/lib/ 目录下。
- MapReduce 访问关系数据库时，大量频繁地从 MapReduce 程序中查询和读取数据，这大大增加了数据库的访问负载，因此，DBInputFormat 接口仅适合读取小数据量的数据，而不适合处理数据仓库。

提示 处理数据仓库的方法有：利用数据库的 Dump 工具将大量待分析的数据输出为文本，并上传到 HDFS 中进行处理，处理的方法可参考 9.3 节 MapReduce 的 Join 算法。

1. 内置类

DBInputFormat 类中包含以下三个内置类。

1）protected class DBRecordReader implementsRecordReader<LongWritable, T>

该类用来从一张数据库表中读取一条元组记录。

2）public static class NullDBWritable implements DBWritable, Writable

该类主要用来实现 DBWritable 接口。DBWritable 接口要实现两个函数，write 和 readFileds。这两个函数都不难理解，一个是写，一个是读出所有字段。原型如下。

```
public void write(PreparedStatement statement) throws SQLException;
public void readFields(ResultSet resultSet) throws SQLException;
```

3）protected static class DBInputSplit implements InputSplit

该类主要用来描述输入元组集合的范围，包括 start 和 end 两个属性，start 表示第一条记录的索引号，end 表示最后一条记录的索引号。

2. 读取数据库记录的步骤

下面对使用 DBInputFormat 读取数据库记录进行详细介绍，具体步骤如下。

步骤 1 配置 JDBC 驱动、数据源和数据库访问的用户名和密码。代码如下。

```
DBConfiguration.configureDB (JobConf job, String driverClass,
        String dbUrl, String userName, String passwd)
```

MySQL 数据库的 JDBC 的驱动为“com.mysql.jdbc.Driver”，数据源为“jdbc:mysql://localhost/testDB”。其中 testDB 为访问的数据库，useName 一般为“root”，passwd 是数据库的密码。

步骤 2 使用 setInput 方法操作 MySQL 中的表，setInput 方法的参数如下。

```
DBInputFormat.setInput(JobConf job, Class<?extends DBWritable> inputClass,
        String tableName, String conditions,String orderBy, String... fieldNames)
```

这个方法的参数很容易看懂，inputClass 实现 DBWritable 接口。tableName 为表名，conditions 表示查询的条件，order By 表示排序的条件，fieldNames 是字段，这相当于把 SQL 语句拆分的结果。当然也可以用 SQL 语句进行重载，代码如下。

```
setInput(JobConf job, Class<?extends DBWritable> inputClass,
        String inputQuery, String inputCountQuery)
```

步骤 3 编写 MapReduce 函数，包括 Mapper 类、Reducer 类、输入 / 输出文件格式等，然后调用 JobClient.runJob(conf)。

3. 示例

上面讲了理论，下面举个例子。假设 MySQL 数据库中有数据库 student，假设数据库中的字段有“id”“name”“gender”“number”。

1）实现 DBwrite 和 write 数据接口。如代码清单 9-4 所示。

代码清单 9-4 mapred\mysql\ StudentRecord.java

```
public class StudentRecord implements Writable, DBWritable{
            int id;
            String name;
            String gender;
            String number;
            @Override
        public void readFields(DataInput in) throws IOException {
            // TODO Auto-generated method stub
            this.id = in.readInt();
            this.gender = Text.readString(in);
            this.name = in.readString();
            this.number = in.readString();
        }
            @Override
        public void write(DataOutput out) throws IOException {
            // TODO Auto-generated method stub
            out.writeInt(this.id);
            Text.writeString(out,this.name);
            out.writeInt(this.gender);
            out.writeInt(this.number);
        }

            @Override
        public void readFields(ResultSet result) throws SQLException {
            // TODO Auto-generated method stub
            this.id = result.getInt(1);
            this.name = result.getString(2);
            this.gender = result.getString(3);
            this.number = result.getString(4);
        }
```

```
        @Override
    public void write(PreparedStatement stmt) throws SQLException{
        // TODO Auto-generated method stub
        stmt.setInt(1, this.id);
        stmt.setString(2, this.name);
        stmt.setString(3, this.gender);
        stmt.setString(4, this.number);
    }
        @Override
    public String toString() {
        // TODO Auto-generated method stub
        return new String(this.name + " " + this.gender + " " +this.number);
     }
}
```

2）实现 Map 和 Reduce 类，如代码清单 9-5 所示。

代码清单 9-5　mapred\mysql\ DBAccessMapper.java

```
public class DBAccessMapper extends MapReduceBase implements
        Mapper<LongWritable, TeacherRecord, LongWritable, Text> {
    @Override
    public void map(LongWritable key, TeacherRecord value,
        OutputCollector<LongWritable, Text> collector, Reporter reporter)
        throws IOException {
      // TODO Auto-generated method stub
      new collector.collect(new LongWritable(value.id), new Text(value.toString()));
    }
  }
```

Reducer 类采用自带的 IdentityReuducer 类，在 src\mapred\org\ apache\ hadoop\ mapred\lib 下。

3）主函数的实现，如代码清单 9-6 所示。

代码清单 9-6　mapred\mysql\ DBAccessMapper.java

```
public class DBAccessReader {
  public static void main(String[] args) throws IOException {
     JobConf conf = new JobConf(DBAccessReader.class);
     conf.setOutputKeyClass(LongWritable.class);
     conf.setOutputValueClass(Text.class);
     conf.setInputFormat(DBInputFormat.class);
     FileOutputFormat.setOutputPath(conf, new Path("/user/hadoop/mysql/output/"));
     DBConfiguration.configureDB(conf, "com.mysql.jdbc.Driver",
     "jdbc:mysql://localhost/school","root","root");
     String [] fields = {"id", "name", "gender", "number"};
     DBInputFormat.setInput(conf, StudentRecord.class,"Student","id", fields)
     conf.setMapperClass(DBAccessMapper.class);
     conf.setReducerClass(IdentityReducer.class);
     JobClient.runJob(conf);
     }
}
```

4）运行命令如下。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop  jar
            mysql-mapreduce.jar  mapred.mysql. DBAccessReader
```

5）查看结果如下所示。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$  hadoop fs -text  /user/hadoop/mysql/output/
    1000    jayliu man 200612169
```

提示 MapReduce 操作 MySQL 数据库在实际工作中比较常用，例如把 MySQL 中的数据迁移到 HDFS 中，当然还有个很好的方法把 MySQL 或 Oracle 中的数据迁移到 DHFS 中，这个工具是 Pig，如果有这方面的需求建议使用 Pig。

9.4.2 写数据

数据处理结果的数据量一般不会太大，可能适合 Hadoop 直接写入数据库中。Hadoop 提供了数据库接口，把 MapReduce 的结果直接输出到 MySQL、Oracle 等数据库。主要的类如下所示。

- DBOutFormat：提供数据库写入接口。
- DBRecordWriter：提供向数据库中写入的数据记录的接口。
- DBConfiguration：提供数据库配置和创建链接的接口。

DBOutFormat 提供一个静态方法 setOutput(job, String table, String ...filedNames); 该方法的参数有表名和表里面的字段。示例如下。

假设往 student 表里插入一条数据，代码如下。

```
public static void main(String[] args) throws IOException {
    Configuration conf = new Configuration();
    JobConf conf = new JobConf();
    conf.setOutputFormat(DBOutputFormat.class);
    DBConfiguration.configureDB(conf,"com.mysql.jdbc.Driver",
          "jdbc:mysql://localhost/school","root","root");
    DBOutputFormat.setOutput(conf,"Student", 1000, "jayliu", "man", "200612169");
    JobClient.runJob(conf);
}
```

9.5 Hadoop 系统调优

Hadoop 系统的调优对于 Hadoop 的集群来讲是非常重要的，Hadoop 系统的调优也是非常困难的，要从很多角度考虑 Hadoop 系统的性能，如设备的配置、Hadoop 的配置文件、MapReduce 代码的优化、小文件优化等，接下来介绍两种比较常用的方法，小文件优化和 Map、Reduce 个数优化。

9.5.1 小文件优化

对于小文件问题，Hadoop 本身也提供了几个解决方案，分别是 Hadoop Archive、Sequence file 和 CombineFileInputFormat。

1. Hadoop Archive

Hadoop Archive（HAR）是一个文件存档工具，能高效地将小文件放入 HDFS 块中，它能够将多个小文件打包成一个 HAR 文件，这样在减少 NameNode 内存使用的同时，仍然允许对文件进行透明访问。

使用 HAR 时需要注意两点：

- 对小文件进行存档后，原文件并不会自动被删除，需要用户自己删除；
- 创建 HAR 文件的过程实际上是在运行一个 MapReduce 作业，因而需要有一个 Hadoop 集群运行此命令。

该方案需人工进行维护，适用管理人员的操作，而且 HAR 文件一旦创建，Archives 便不可改变，不能应用于多用户的互联网操作。

2. SequenceFile

SequenceFile 由一系列的二进制 Key-Value 组成，如果为 Key 小文件名，Value 为文件内容，则可以将大批小文件合并成一个大文件。

Hadoop-1.0.3 中提供了 SequenceFile，包括 Writer、Reader 和 SequenceFileSorter 类进行写、读和排序操作。

3. CombineFileInputFormat

CombineFileInputFormat 是一种新的 InputFormat，用于将多个文件合并成一个单独的 split，另外，它会考虑数据的存储位置。

提示 *还可以采用中间件的方法，目前也是很多公司采用的方法，就是在数据进入 Hadoop 的 HDFS 系统之前，对它进行手动合并，一般合并的大小与 Block 一致。这种方案效果比上面我们讲的三种方案要好很多，而且效果特别明显。*

9.5.2 Map 和 Reduce 个数设置

影响 MapReduce 性能主要因素就是 Map 和 Reduce 的个数，个数少了系统的资源没有得到充分的利用，多了系统的开销也要消耗资源，从而使 MapReduce 的性能下降。本节介绍一下 Map 和 Reduce 个数的设置。

1. 全局 Map 和 Reduce 个数设置

对于整个 Hadoop 集群来讲，Map 的个数不可以手动设置，根据 mapred.min.split.size 参数计算得到，默认为 1 的情况下，针对每个文件会按照以下大小拆分。

```
min (totalsize[所有文件总大小]/mapnum[jobconf设置的mapnum], blocksize)
```

并不是说文件小于 Block 就不去拆分。用户不能直接设置这个参数。

InputSplit 的大小决定了一个 Job 拥有多少个 Map。默认 InputSplit 的大小是 64MB（与 dfs.block.size 的默认值相同）。然而，如果输入的数据量巨大，那么默认的 64MB 的 Block 会有几万甚至几十万的 Map Task，集群的网络传输会很大，最严重的是会给 JobTracker 的调度、队列、内存带来很大压力。mapred.min.split.size 这个配置项决定了每个 InputSplit 的最小值，用户可以修改这个参数，从而改变 Map Task 的数量。

Reduce 的个数设置非常简单，用 Job.setNumMapTasks() 设置，默认为 1。

2. 单个节点 Map 和 Reduce 设置

Hadoop 集群有很多台设备组成，正常情况下每台设备的配置是一致的，但是如果个别设备资源没有的到充分的利用时，我们可以根据每台设备的性能来设置每台设备处理多少 Map 和 Reduce 的个数，设置参数在 mapred-site.xml 配置文件里面，两个参数如下。

```
mapred.tasktracker.map.tasks.maximum      // 设置Map的个数，默认为2
mapred.tasktracker.reduce.tasks.maximum   // 设置Reduce的个数，默认为2
```

增加 Task 的数量，一方面增加了系统的开销，另一方面增加了负载平衡和减小了任务失败的代价。下面给出设置最优的 Map 和 Reduce 个数。

- 每个节点的 Map 并行度是大约每个节点 10 ～ 100 个 Map，最好每个 Map 的执行时间在 1 ～ 2 分钟。
- 合适的 Reduce Task 数量是 0.95 或者

  0.75*(nodes * mapred.tasktracker.reduce.tasks.maximum)

  其中，mapred.tasktracker.tasks.reduce.maximum 的数量一般设置为各节点 CPU 核的数量，即能同时计算的 slot 数量。对于 0.95，当 Map 结束时，所有的 Reduce 能够立即启动；对于 1.75，较快的节点结束第一轮 Reduce 后，可以开始第二轮的 Reduce 任务，从而提高负载均衡。

9.6 本章小结

在本章中，我们介绍了 MapReduce 的一些高级应用、算法等知识点，算是对第 8 章学的知识进行应用。本章内容包含 MapReduce 开发和 Hadoop 系统调优常用的知识点，希望读者能够掌握这章的内容，对 MapReduce 的算法开发和调优打下基础。

第10章　数据仓库工具 Hive

Hive 是基于 Hadoop 的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供完整的 SQL 查询功能，可以将 SQL 语句转换为 MapReduce 任务进行运行。其优点是学习成本低，可以通过类 SQL 语句快速实现简单的 MapReduce 统计，不必开发专门的 MapReduce 应用，十分适合数据仓库的统计分析。

10.1　认识 Hive

Hive 是基于 Hadoop 构建的一套数据仓库分析系统，它提供了丰富的 SQL 查询方式来分析存储在 Hadoop 分布式文件系统中的数据，可以将结构化的数据文件映射为一张数据库表，并提供完整的 SQL 查询功能，可以将 SQL 语句转换为 MapReduce 任务进行运行，通过自己的 SQL 去查询分析需要的内容，这套 SQL 简称 Hive SQL，使不熟悉 MapReduce 的用户很方便地利用 SQL 语言查询、汇总、分析数据。

10.1.1　Hive 工作原理

Hive 工作原理如图 10-1 所示。

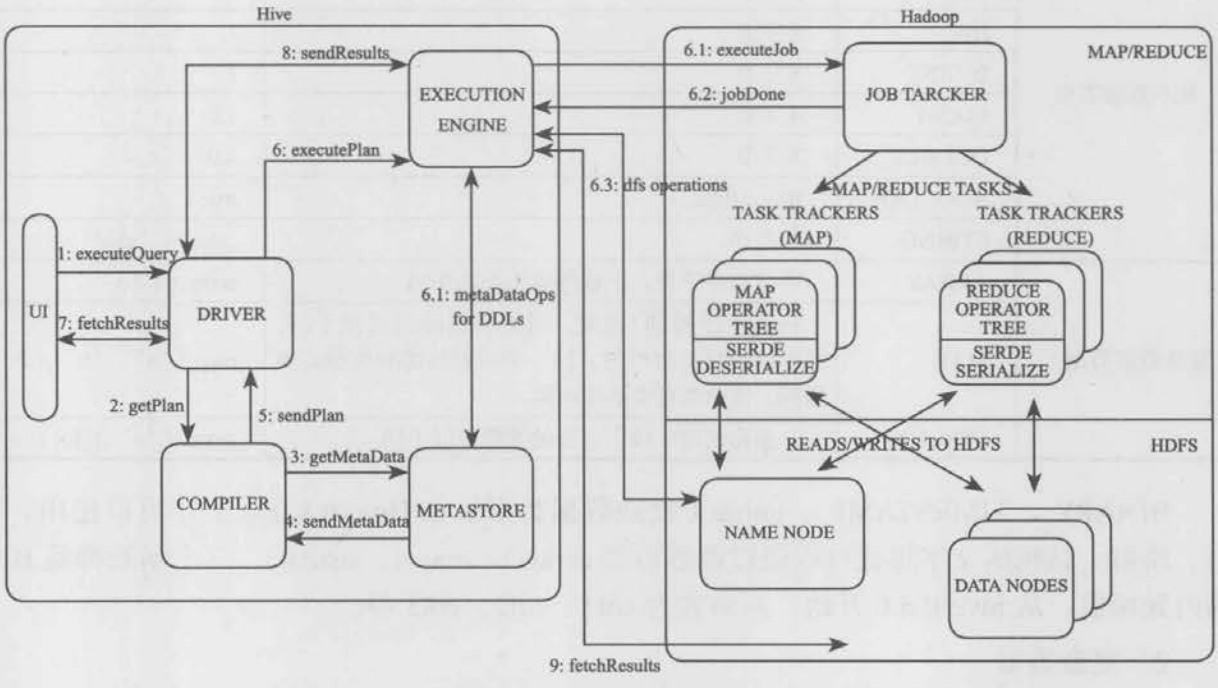


图 10-1　Hive 工作流程

MapReduce 开发人员可以把自己写的 Mapper 和 Reducer 作为插件支持 Hive 做更复杂的数据分析。它与关系型数据库的 SQL 略有不同，但支持了绝大多数的语句（如 DDL、DML ）以及常见的聚合函数、连接查询、条件查询等操作。

Hive 不适合用于联机（online）事务处理，也不提供实时查询功能。它最适合应用在基于大量不可变数据的批处理作业。Hive 的特点是可伸缩（在 Hadoop 的集群上动态的添加设备），可扩展、容错、输入格式的松散耦合。Hive 的入口是 DRIVER，执行的 SQL 语句首先提交到 DRIVER 驱动，然后调用 COMPILER 解释驱动，最终解释成 MapReduce 任务执行，最后将结果返回，接下来详细介绍 Hive 的解释器 COMPILER。

10.1.2 Hive 数据类型

Hive 提供了基本数据类型和复杂数据类型，复杂数据类型是 Java 语言所不具有的。本章介绍 Hive 的两种数据类型以及数据类型之间的转换。

1. 基本数据类型

Hive 支持原子和复杂数据类型，原子数据类型包括数据值、布尔类型和字符串类型，复杂的类型包括数组、映射和结构。Hive 的数据类型在表 10-1 中列出，注意，列出的是在 HiveSQL 中使用的形式而不是在表中序列化存储的格式。

表 10-1 Hive 的数据类型

类别	类型	描述	示例
基本数据类型	TINYINT	1 字节	1
	SMALLINT	2 字节	1
	INT	4 字节	1
	BIGINT	8 字节	1
	FLOAT	4 字节	1.0
	DOUBLE	8 字节	1.0
	BOOLEAN	true、flase	true
	STRING	字符串	‘abc’，“abc”
复杂数据类型	ARRAY	一组有序字段，字段的类型必须相同	array（1,2）
	MAP	一组无序的键 / 值对。键的类型必须是原子的，值可以是任何类型的。同一个映射的键的类型必须相同，值的类型也必须相同	map（‘a’，‘b’,1,2）
	STRUCT	一组命名的字段，字段的类型可以不同	struct（‘a’,1,1.0）

BINARY 、TIMESTAMP 、union（复杂数据类型）在 Hive 0.8.0 以上才可以使用。数组、映射、结构的文字形式可以通过函数得到 array()、map()、struct()，三个函数都是 Hive 的内置函数。从 hive-0.6.0 开始，列命名为 col1、col2、col3 等。

2. 复杂类型

Hive 有三种复杂数据类型 ARRAY、MAP 和 STRUCT。ARRAY 和 MAP 与 Java 中的

Array 和 Map 类似，而 STRUCT 与 C 语言中的 Struct 类似，它封装了一个命名的字段集合，复杂数据类型允许任意层次的嵌套。

复杂数据类型的声明必须使用尖括号指明其中数据字段的类型。定义三列，每列对应一种复杂的数据类型，如下所示。

```
CREATE TABLE complex(
  col1 ARRAY<INT>,
  col2 MAP<STRING,INT>,
  col3 STRUCT<a:STRING,b:INT,c:DOUBLE>
);
```

3. 类型转化

Hive 的原子数据类型是可以进行隐式类型转换的，类似于 Java 的类型转换，例如某表达式使用 INT 类型，TINYINT 会自动转换为 INT 类型，但是 Hive 不会进行反向转化，例如，某表达式使用 TINYINT 类型，TIN 不会自动转换为 TINYINT 类型，它会返回错误，除非使用 CAST 操作。

隐式类型转换规则如下。

- 任何整数类型都可以隐式地转换为一个范围更广的类型，如 TINGINT 可以转换成 INT，INT 可以转换成 BIGINT。
- 所有整数类型、FLOAT 和 String 类型都可以隐式地转换成 DOUBLE。
- TINYINT、SMALLINT、INT 都可以转换为 FLOAT。
- BOOLEAN 类型不可以转换为任何其他的类型。

可以使用 CAST 操作显式进行数据类型转换，例如 CAST('1' AS INT) 将把字符串 '1' 转换成整数 1；如果强制类型转换失败，如执行 CAST('X' AS INT)，表达式返回空值 NULL。

10.1.3 Hive 的特点

Hive 与关系数据库的比较如表 10-2 所示。

表 10-2 Hive 与关系数据库的比较

	Hive	RDBMS
查询语言	HQL	SQL
数据存储	HDFS	Raw Device or Local FS
索引	无	有
执行	MapReduce	Excutor
执行延迟	高	低
处理数据规模	大	小

1. 查询语言

由于 SQL 被广泛应用在数据仓库中，专门针对 Hive 的特性设计了类 SQL 的查询语言 HQL。熟悉 SQL 的开发者可以很方便地使用 Hive 进行开发。

2. 数据存储位置

Hive 建立在 Hadoop 之上，所有 Hive 的数据都存储在 HDFS 中。而数据库则可以将数据保存在块设备或者本地文件系统中。

3. 数据格式

Hive 中没有定义专门的数据格式，数据格式可以由用户指定，用户定义数据格式需要指定三个属性：列分隔符（通常为空格、“\t”、“\x001”）、行分隔符（“\n”）以及读取文件数据的方法（Hive 中默认有三个文件格式 TextFile、SequenceFile 以及 RCFile）。由于在加载数据的过程中，不需要从用户数据格式到 Hive 定义的数据格式的转换，Hive 在加载的过程中不会对数据本身进行任何修改，只是将数据内容复制或者移动到相应的 HDFS 目录中。

而在数据库中，不同数据库有不同的存储引擎，定义自己的数据格式。所有数据都会按照一定的组织存储，因此，数据库加载数据的过程会比较耗时。

4. 数据更新

Hive 是针对数据仓库应用设计的，而数据仓库的内容读多写少。因此，Hive 中不支持对数据的改写和添加，所有数据都是在加载时确定的。而数据库中的数据通常是需要经常修改的，因此可以使用“INSERTINTO ... VALUES”添加数据，使用“UPDATE ... SET ”修改数据。

5. 索引

之前说过，Hive 在加载数据的过程中不会对数据进行任何处理，甚至不会对数据进行扫描，因此也没有对数据中的某些 Key 建立索引。Hive 要访问数据中满足条件的特定值时，需要暴力扫描整个数据，因此访问延迟较高。由于 MapReduce 的引入，Hive 可以并行访问数据，因此即使没有索引，对于大数据量的访问，Hive 仍然可以体现出优势。

数据库中，通常会针对一个或者几个列建立索引，因此对于少量特定条件数据的访问，数据库可以有很高的效率，较低的延迟。由于数据的访问延迟较高，决定了 Hive 不适合在线数据查询。

6. 执行

Hive 中大多数查询的执行是通过 Hadoop 提供的 MapReduce 实现的（类似 select * from tablel 的查询不需要 MapReduce）。而数据库通常有自己的执行引擎。

7. 执行延迟

之前提到，Hive 在查询数据时没有索引，需要扫描整个表，因此延迟较高。另外一个导致 Hive 执行延迟高的因素是 MapReduce 框架。MapReduce 本身具有较高的延迟，在利用 MapReduce 执行 Hive 查询时，也会有较高的延迟。相对而言，数据库的执行延迟较低。当然，这个低是有条件的，即数据规模较小，当数据规模大到超过数据库的处理能力的时候，Hive 的并行计算显然能体现出优势。

8. 可扩展性

Hive 建立在 Hadoop 之上，因此 Hive 的可扩展性和 Hadoop 的可扩展性是一致的。而数据库由于 ACID 语义的严格限制，扩展性非常有限。目前最先进的并行数据库 Oracle 在理论上的扩展能力也只有 100 台左右。

9. 数据规模

Hive 建立在集群上并可以利用 MapReduce 进行并行计算，可以支持很大规模的数据；数据库可以支持的数据规模较小。

10.1.4 Hive 下载与安装

Hive 安装依赖 Hadoop 的集群，它是运行在 Hadoop 的基础上。所以在安装 Hive 之前，保证 Hadoop 集群能够成功运行。Hive 是 Apache 的一个顶级开源项目[⊖]，本书使用的是 hive-0.9.0.tar.gz 版本。

1. 安装与配置

1）把 hive-0.9.0 移动到 /opt 目录下并解压，操作命令如下。

```
jayliu@Ubuntu1:cp /home/jayliu/Desktop/hive-0.9.0.tar.gz /opt
jayliu@Ubuntu1:~$ cd /opt
jayliu@Ubuntu1:~$ tar -zxvf hive-0.9.0.tar.gz
```

2）添加 hive-0.9.0 环境变量到 PATH 下面，操作命令如下。

```
vi /etc/profile
HIVE_HOME=/opt/hive-0.9.0
HADOOP_HOME=/opt/hadoop-1.0.3
PATH=$JAVA_HOME/bin:$HIVE_HOME/bin:$HADOOP_HOME/bin:$PATH
export HADOOP_HOME
export HIVE_HOME
```

3）配置 hive-default.xml 和 hive-site.xml。

“/opt/hive-0.9.0/conf” 目录下没有这两个文件，只有一个 “hive-default.xml.template”，所以要复制两个 “hive-default.xml.template”，分别命名为 “hive-default.xml” 和 “hive-site.xml”。

“/opt/hive-0.9.0/conf” 目录下有一个 “hive-env.sh.template” 文件，复制并命名为 “hive-env.sh”，然后修改 hive-site.sh 权限。修改 hive-site.sh 权限命令如下。

```
chmod u+x hive-env.sh
```

备注 hive-default.xml 用于保留默认配置，hive-site.xml 用于个性化配置，可覆盖默认配置。

2. 运行和测试

在终端输入 “hive”，显示如图 10-2 所示。

⊖ 下载地址为 http://www.apache.org/dyn/closer.cgi/hive/。

```
jayliu@Ubuntu1:/opt/hive-0.9.0$ hive
WARNING: org.apache.hadoop.metrics.jvm.EventCounter is deprecated. Please use
 org.apache.hadoop.log.metrics.EventCounter in all the log4j.properties files
.
Logging initialized using configuration in jar:file:/opt/hive-0.9.0/lib/hive-
common-0.9.0.jar!/hive-log4j.properties
Hive history file=/tmp/jayliu/hive_job_log_jayliu_201305062302_820322557.txt
```

图 10-2 Hive 运行

警告（WARNING）是没有使用 org.apache.hadoop.log.metrics 导致的，解决办法是，在 hive-log4j.properties 文件（/opt/hive-0.9.0/conf 下）中将 log4j.appender.EventCounter 的值修改为以下就不会报。

```
org.apache.hadoop.log. metrics.EventCounter
```

1）测试 Hive 是否安装成功。

```
hive> show tables;
OK
Time taken: 24.06 seconds
```

2）创建表。

```
  hive> create table test_table (id int,name string,no int);
OK
Time taken: 6.808 seconds
```

3）查询表。

```
hive> select * from test_table;
OK
Time taken: 3.12 seconds
```

如果创建表和查询表运行没有出错，就说明 Hive 安装成功。

10.2 Hive 架构

Hive 架构可以分为四部分。

- **用户接口。**主要有三个 CLI、Client 和 WUI。其中最常用的是 CLI 接口，CLI 启动的时候，会同时启动一个 Hive 副本。Client 是 Hive 的客户端，用户连接至 Hive Server。在启动 Client 模式的时候，需要指出 Hive Server 所在节点，并且在该节点启动 Hive Server。WUI 是通过浏览器访问 Hive。
- **元数据存储。**Hive 将元数据存储在数据库中，如 MySQL、Derby。Hive 中的元数据包括表的名字、表的列和分区及其属性、表的属性（是否为外部表等）、表的数据所在目录等。
- **解释器、编译器、优化器。**完成 HQL 查询语句从词法分析、语法分析、编译、优化以及查询计划的生成。生成的查询计划存储在 HDFS 中，并在随后由 MapReduce 调用执行。

❑ **数据存储**。Hive 的数据存储在 HDFS 中，大部分的查询都被解释成 MapReduce 任务来执行，只有少部分直接读取文件，如“SELECT * FROM tableName”。

Hive 的主要部件如图 10-3 所示。当前系统有一个命令行的接口和基于 Web 的 GUI Driver 接受 Query 的组件，该组件实现 session 的概念，以处理和提供基于 JDBC/ODBC 执行以及 API 接口。

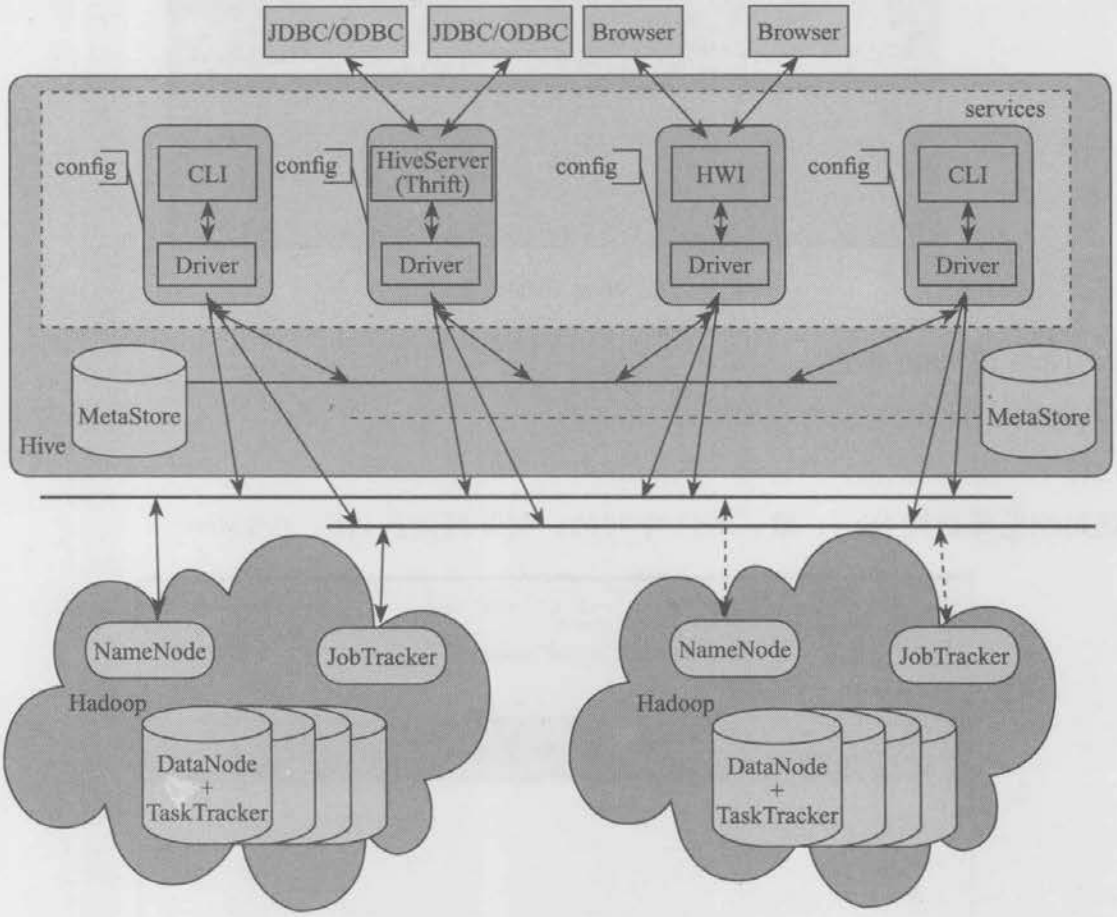


图 10-3 Hive 架构

10.2.1 Hive 用户接口

Hive 对外提供了三种服务模式，即 Hive 命令行模式（CLI）、Hive 的 Web 模式（WUI）、Hive 的远程服务（Client）。下面介绍这些服务的用法。

1. Hive 命令行模式

Hive 命令行模式启动有两种方式。执行这条命令的前提是要配置 Hive 的环境变量。

1）进入 /opt/hive-0.9.0/bin/ 目录，执行如下命令。

```
./hive
```

2）直接执行命令。

```
hive  --service cli
```

Hive 命令行模式用于 Linux 平台命令行查询，查询语句基本跟 MySQL 查询语句类似，运行如图 10-4 所示。

```
hive> show tables;
OK
Time taken: 24.06 seconds
hive> create table test_table (id int,name string,no int);
OK
Time taken: 6.808 seconds
hive> select  * from test_table;
OK
Time taken: 3.12 seconds
hive> exit;
jayliu@Ubuntu1:/opt/hive-0.9.0$ ls
```

图 10-4　Hive 命令行模式运行

2. Hive 的 Web 模式

Hive Web 界面的启动命令如下。

```
hive  --service hwi
```

通过浏览器访问 Hive，默认端口为 9999，浏览器运行如图 10-5 所示。

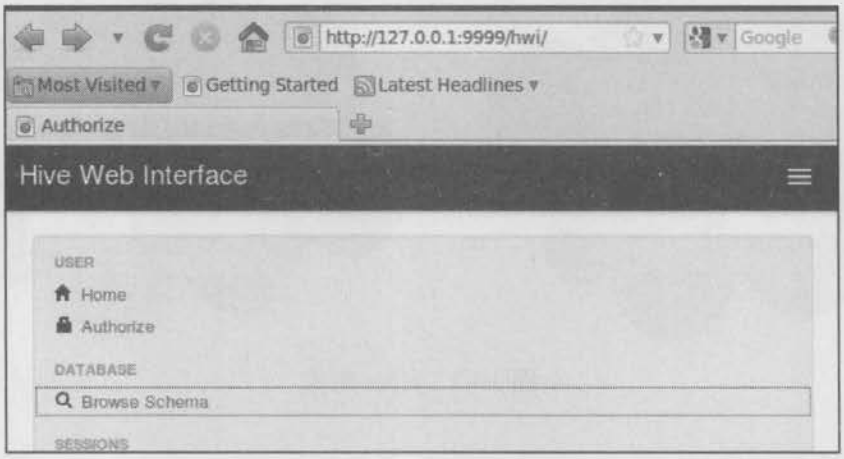


图 10-5　Hive 的 Web 模式运行效果

3. Hive 的远程服务

远程服务（端口号 10000）启动方式命令如下，“nohup...&”是 Linux 命令，表示命令在后台运行。

```
nohup  hive  --service hiveserver  &
```

Hive 远程服务通过 JDBC 等访问来连接 Hive，这是程序员最需要的方式。

启动 hive service 命令如下。

```
$HIVE_HOME/bin/hive --service hiveserver 10001 >/dev/null 2>/dev/null &
```

Hive 的 JDBC 连接和 MySQL 类似，如下所示。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
public class HiveJdbcClient {
    private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";
    /**
     * @param args
     * @throws SQLException
     */
    public static void main(String[] args) throws SQLException {
        try {
           Class.forName(driverName);
        } catch (ClassNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
        System.exit(1);
      }
      Connection con = DriverManager.getConnection
                      ("jdbc:hive://192.168.153.100:10000/default", "", "");
      System.out.printf(conn);
    }
}
```

10.2.2 Hive 元数据库

Hive 将元数据存储在 RDBMS 中，一般常用 MySQL 和 Derby。Hive 默认存储元数据信息在 Derby。

1. 存储元数据库 Derby

启动 Hive 的元数据库（Derby），进入 Hive 的安装目录，操作步骤如下所示。

1）启动 Derby 数据库（下载 Derby 安装包并解压），操作命令如下。

```
cd  /opt / db-derby-10.4.1.3-bin
start  NetworkServer -h 0.0.0.0
```

2）连接 Derby 数据库进行测试。查看 /opt/hive-0.9.0 /conf/hive-default.xml。连接 Derby 数据库信息如下。

```
<property>
    <name>javax.jdo.option.ConnectionURL</name>
    <value>jdbc:derby://Ubuntu1:1527/metastore_db;create=true</value>
    <description>JDBC connect string for a JDBC metastore</description>
</property>
```

3）进入 Derby 安装目录，操作命令如下。

```
cd  /opt/db-derby-10.4.1.3-bin/bin
```

4）输入“./ij”命令，显示如下信息。

```
Connect 'jdbc:derby://hadoop1:1527/metastore_db;create=true';
```

Hive 的元数据字典信息如表 10-3 所示。

表 10-3 Hive 的元数据字典信息列表

表 名	功 能 描 述
BUCKETING_COLS	Hive 表 CLUSTERED BY 字段信息（字段名、字段序号）
COLUMNS	Hive 表字段信息（字段注释、字段名、字段类型、字段序号）
DBS	元数据库信息，存放 HDFS 路径信息
NUCLEUS_TABLES	元数据表和 Hive 中 class 类的对应关系
PARTITIONS	Hive 表分区信息（创建时间、具体的分区）
PARTITION_KEYS	Hive 分区表分区键（名称、类型、comment、序号）
PARTITION_KEY_VALS	Hive 表分区名（键值、序号）
SDS	所有 Hive 表、表分区所对应的 hdfs 数据目录和数据格式
SD_PARAMS	序列化反序列化信息，如行分隔符、列分隔符、NULL 的表示字符等
SEQUENCE_TABLE	Hive 对象的下一个可用 ID
SERDES	Hive 表序列化反序列化使用的类库信息
SERDE_PARAMS	序列化反序列化信息，如行分隔符、列分隔符、NULL 的表示字符等
SORT_COLS	Hive 表 SORTED BY 字段信息（字段名、sort 类型、字段序号）
TABLE_PARAMS	表级属性，如是否外部表、表注释等
TBLS	所有 Hive 表的基本信息

从表的内容来看，Hive 创建表的整个过程如下。

解析用户提交 Hive 语句→对其进行解析→分解为表、字段、分区等 Hive 对象。

根据解析到的信息构建对应的表、字段、分区等对象，从 SEQUENCE_TABLE 中获取构建对象的最新 ID，与构建对象信息（名称、类型等）一同通过 DAO 方法写入元数据表中，成功后将 SEQUENCE_TABLE 中对应的最新 ID+5。实际上常见的 RDBMS 都是通过这种方法进行组织的（如 PostgreSQL），其系统表中和 Hive 元数据一样显示了这些 ID 信息（oid、cid 等），而 Oracle 等商业化的系统则隐藏了这些具体的 ID。通过这些元数据可以很容易地读到数据，如创建一个表的数据字典信息、导出建表语名等。

2. 存储元数据库 MySQL

默认情况下，Hive 元数据保存在内嵌的 Derby 数据库中，只能允许一个会话连接，只适合简单的测试。实际生产环境中不适用，为了支持多用户多会话，则需要一个独立的元数据库，使用 MySQL 作为元数据库，Hive 内部对 MySQL 提供了很好的支持，配置一个独立的元数据库需要增加以下步骤。

1）安装 MySQL 数据库。

```
sudo apt-get install mysql-server
```

```
sudo apt-get install mysql-client
```

2）MySQL 的启动、停止和重启的命令如下。

```
sudo /etc/init.d/mysql  restart   # 重启 mysql
sudo /etc/init.d/mysql  stop      # 停止 mysql
sudo /etc/init.d/mysql  start     # 启动 mysql
```

3）为 Hive 建立相应的 MySQL 账户，并赋予足够的权限。

```
jayliu@Ubuntu1:~$ mysql -u root -p root
mysql> CREATE USER 'hive' IDENTIFIED BY 'mysql';
mysql> GRANT ALL PRIVILEGES ON *.* TO 'hive'@'%' WITH GRANT OPTION;
mysql> flush privileges;
```

默认情况下 MySQL 只允许本地登录，所以需要修改配置文件将地址绑定注释。

```
jayliu@Ubuntu1: sudo gedit /etc/mysql/my.cnf
```

找到如下内容。

```
# Instead of skip-networking the default is now to listen only on
# localhost which is more compatible and is not less secure.
# bind-address            = 127.0.0.1  <--- 注释这一行就可以远程登录了
```

重启 mysql 服务。

```
sudo service mysql restart
```

4）建立 Hive 专用的元数据库，记得用刚才创建的“hive”账号登录，命令如下。

```
mysql> exit;
jayliu@Ubuntu1: mysql -u hive -p mysql
mysql> create database hive;
```

5）在 Hive 的 conf 目录下的 hive-site.xml 文件中增加如下配置。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
    <property>
        <name>hive.metastore.local</name>
        <value>true</value>
    </property>
    <property>
        <name>javax.jdo.option.ConnectionURL</name>
        <value>jdbc:mysql://192.168.153.100:3306/hive?characterEncoding=UTF-8</value>
    </property>
    <property>
        <name>javax.jdo.option.ConnectionDriverName</name>
        <value>com.mysql.jdbc.Driver</value>
    </property>
    <property>
        <name>javax.jdo.option.ConnectionUserName</name>
        <value>hive</value>
    </property>
    <property>
```

```
        <name>javax.jdo.option.ConnectionPassword</name>
        <value>mysql</value>
    </property>
    </configuration>
```

提示 “hive-site.xml”是“hive-default.xml.template”的一个副本，其中的配置参数非常多，但并不是都需要的。Hive 系统会加载两个配置文件：默认配置文件 hive-default.xml，以及用户自定义文件 hive-site.xml。当 hive-site.xml 中的配置参数值与 hive-default.xml 文件中不一致时，以用户自定义的为准。所以把不需要的参数都删除，只保留上面所示的内容。

6）把 MySQL 的 JDBC 驱动包复制到 Hive 的 lib 目录下，JDBC 驱动包的版本为 mysql-connector-java-5.1.18-bin.jar。

7）启动 Hive Shell。

```
jayliu@Ubuntu1:~$ hive
Logging initialized using configuration in
        file:/opt/hive-0.9.0/conf/hive-log4j.properties
Hive history file=/tmp/hadoop/hive_job_log_hadoop_201303041631_450140463.txt
hive> show tables;
OK
Time taken: 1.988 seconds
Hive>
```

8）查看 MySQL 数据库。

```
mysql> show TBLS;
```

图 10-6 说明 MySQL 的元数据存储配置成功。

Field	Type	Null	Key	Default	Extra
TBL_ID	bigint(20)	NO	PRI	NULL	
CREATE_TIME	int(11)	NO		NULL	
DB_ID	bigint(20)	YES	MUL	NULL	
LAST_ACCESS_TIME	int(11)	NO		NULL	
OWNER	varchar(767)	YES		NULL	
RETENTION	int(11)	NO		NULL	
SD_ID	bigint(20)	YES	MUL	NULL	
TBL_NAME	varchar(128)	YES	MUL	NULL	
VIEW_EXPANDED_TEXT	mediumtext	YES		NULL	
VIEW_ORIGINAL_TEXT	mediumtext	YES		NULL	
TBL_TYPE	varchar(128)	YES		NULL	

图 10-6 Hive 元数据列表

10.2.3 Hive 的数据存储

首先，Hive 没有专门的数据存储格式，也没有为数据建立索引，用户可以自由地组

织 Hive 中的表，只需要在创建表时告诉 Hive 数据中的列分隔符和行分隔符，Hive 就可以解析数据。其次，Hive 中所有的数据都存储在 HDFS 中，Hive 中的数据模型包括 Table、External、Table、Partition、Bucket。

Hive 中的 Table 和数据库中的 Table 在概念上类似，每一个 Table 在 Hive 中都有一个相应的目录存储数据。例如，一个表 jayliu，它在 HDFS 中的路径为 / warehouse /jayliu，其中，warehouse 是在 hive-site.xml 中由 ${hive.metastore.warehouse.dir} 指定的数据仓库的目录，所有 Table 数据（不包括 External Table）都保存在这个目录中。

Partition 对应于数据库中 Partition 列的密集索引，但是 Hive 中 Partition 的组织方式和数据库中的不相同。在 Hive 中，表中的一个 Partition 对应于表下的一个目录，所有 Partition 的数据都存储在对应的目录中。例如 jayliu 表中包含 dt 和 city 两个 Partition：

- 对应于 dt = 20120801, ctry = US 的 HDFS 子目录为：
  /warehouse /jayliu/dt=20120801/ctry=US；
- 对应于 dt = 20120801, ctry = CA 的 HDFS 子目录为：
  / warehouse /jayliu/dt=20120801/ctry=CA。

Buckets 对指定列计算 hash，根据 hash 值切分数据，目的是并行，每个 Bucket 对应一个文件。将 user 列分散至 32 个 bucket，首先对 user 列的值计算 hash，对应：

- hash 值为 0 的 HDFS 目录为：
  / warehouse /jayliu/dt=20120801/ctry=US/part-00000；
- hash 值为 20 的 HDFS 目录为：
  / warehouse/jayliu/dt =20120801/ctry=US/part-00020。

External Table 指向已经在 HDFS 中存在的数据，可以创建 Partition。它和 Table 在元数据的组织上是相同的，而实际数据的存储则有较大的差异。

Table 的创建过程和数据加载过程（这两个过程可以在同一个语句中完成），在加载数据的过程中，实际数据会被移动到数据仓库目录中；之后对数据对访问将会直接在数据仓库目录中完成。删除表时，表中的数据和元数据将会被同时删除。External Table 只有一个过程，加载数据和创建表同时完成（CREATE EXTERNALTABLE …LOCATION），实际数据是存储在 LOCATION 后面指定的 HDFS 路径中，并不会移动到数据仓库目录中。删除一个 External Table 时，不删除元数据。

10.2.4 Hive 解释器

Driver 调用解释器（Compiler）处理 HiveQL 字串，这些字串可能是一条 DDL、DML 或查询语句。编译器将字符串转化为策略（plan）。策略仅由元数据操作和 HDFS 操作组成，元数据操作只包含 DDL 语句，HDFS 操作只包含 LOAD 语句。对插入和查询而言，策略由 MapReduce 任务中的具有方向的非循环图（directedacyclic graph，DAG）组成，具体流程如下。

1）**解析器**（parser）：将查询字串转换为解析树表达式。

2）**语义分析器**（semantic analyzer）：将解析树表达式转换为基于块（block-based）的内部查询表达式，将输入表的模式（schema）信息从 metastore 中进行恢复。用这些信息验证列名，展开 SELECT * 以及类型检查（固定类型转换也包含在此检查中）。

3）**逻辑策略生成器**（logical plan generator）：将内部查询表达式转换为逻辑策略，这些策略由逻辑操作树组成。

4）**优化器**（optimizer）：通过逻辑策略构造多途径并以不同方式重写。

优化器的功能如下。

- 将多 multiple join 合并为一个 multi-way join；
- 对 join、group-by 和自定义的 map-reduce 操作重新进行划分；
- 消减不必要的列；
- 在表扫描操作中推行使用断言（predicate）；
- 对于已分区的表，消减不必要的分区；
- 在抽样（sampling）查询中，消减不必要的桶。

此外，优化器还能增加局部聚合操作用于处理大分组聚合（grouped aggregations）和增加再分区操作用于处理不对称（skew）的分组聚合。

10.3 Hive 文件格式

Hive 支持四种文件格式：TextFile、SequenceFile、RCFile 和自定义。

TextFile 为默认格式，建表时不指定默认为这个格式，导入数据时会直接把数据文件复制到 HDFS 上不进行处理。

SequenceFile 和 RCFile 格式的表不能直接从本地文件导入数据，数据要先导入到 TextFile 格式的表中，然后再从 TextFile 表中用 INSERT 导入到 SequenceFile 和 RCFile 表中。

当用户的数据文件格式不能被当前 Hive 所识别的时候，可以自定义文件格式。用户可以通过实现 InputFormat 和 OutputFormat 来自定义输入、输出格式，表 10-4 为 Hive 三种文件格式的比较。

表 10-4 Hive 文件格式比较

比较属性 \ 文件格式	TextFile	SequenceFile	RCFile
Data type	Text Only	Text/Binary	Text/Binary
Internal Storage Order	Row-based	Row-based	Column-based
Compression	File Based	Block Based	Block Based
Splitable	YES	YES	YES
Splitable After Compression	No	YES	YES

Hive 的三种文件格式中，TextFile 和 SequenceFile 的存储格式都是基于行存储的，TextFile 是基于行列混合的思想，先按行把数据划分成 N 个 row group，在 row group 中对

每个列分别进行存储。

基于 HDFS 的行存储具备快速数据加载和动态负载的高适应能力，因为行存储保证了相同记录的所有域都在同一个集群节点。但是它不能满足快速的查询响应时间的要求，因为当查询仅仅针对所有列中的少数几列时，它就不能跳过不需要的列，直接定位到所需列；同时在存储空间利用上，它也存在一些瓶颈，由于数据表中包含不同类型，不同数据值的列，行存储不易获得一个较高的压缩比。TextFile 是基于 SequenceFile 实现的列存储格式。除了满足快速数据加载和动态负载高适应的需求外，也解决了 SequenceFile 的一些瓶颈。

接下来从压缩比、数据加载、查询响应角度对 Hive 的三种文件格式进行详解。

10.3.1 TextFile 格式

TextFile 是默认格式，数据不做压缩，磁盘开销大，数据解析开销大。可结合 Gzip、Bzip2 使用（系统自动检查，执行查询时自动解压）。但使用这种方式，Hive 不会对数据进行切分，从而无法对数据进行并行操作。

例如创建一个表，名为“text_file”，文件格式为“TextFile”，命令如下。

```
hive> CREATE TABLE text_file(str STRING)
hive> STORED AS TEXTFILE;
OK
Time taken: 0.786 seconds
```

写脚本生成一个随机字符串文件，导入文件，命令如下。

```
hive> LOAD DATA LOCAL INPATH '/home/work/data/test.txt' INTO TABLE text_file;
Copying data from file:/home/work/data/test.txt
Copying file: file:/home/work/data/test.txt
Loading data to table default. text_file
OK
Time taken: 0.243 seconds
```

10.3.2 SequenceFile 格式

SequenceFile 是 Hadoop API 提供的一种二进制文件支持，其具有使用方便、可分割、可压缩的特点。SequenceFile 支持三种压缩选择 NONE、RECORD、BLOCK。 Record 压缩率低，一般建议使用 BLOCK 压缩。

例如创建一个表，名为“sequence _file”，文件格式为“SequenceFile”，命令如下。

```
hive> create table sequence_file (str STRING)
hive> STORED AS SEQUENCEFILE;
OK
Time taken: 5.526 seconds
hive> SET hive.exec.compress.output=true;
hive> SET io.seqfile.compression.type=BLOCK;
hive> INSERT OVERWRITE TABLE sequence_file1 SELECT * FROM sequence_file;
```

10.3.3 RCFile 文件格式

RCFile 是 Facebook 开发的一个集行存储和列存储的优点于一身，压缩比更高，读取列

更快，它在 MapReduce 大规模数据处理中扮演着重要的角色。

RCFile 是一种行列存储相结合的存储方式。首先，将数据按行分块，保证同一个 Record 在一个块上，避免读一个记录需要读取多个 Block。

Facebook 曾在 2010 ICDE（IEEE International Conference on Data Engineering）会议上介绍了数据仓库 Hive。Hive 在 Hadoop 系统中存储海量数据，提供了一套类数据库的数据存储和处理机制。它采用类 SQL 语言对数据进行自动化管理和处理，经过语句解析和转换，最终生成基于 Hadoop 的 MapReduce 任务，通过执行这些任务完成数据处理。

1. 特点

基于 MapReduce 的数据仓库在超大规模数据分析中扮演了重要角色，对于典型的 Web 服务供应商，这些分析有助于它们快速理解动态的用户行为及变化的用户需求。数据存储结构是影响数据仓库性能的关键因素之一。Hadoop 系统中常用的文件存储格式有支持文本的 TextFile 和支持二进制的 SequenceFile 等，它们都属于行存储方式[⊖]。文章中介绍了一种高效的数据存储结构——RCFile（Record Columnar File），并将其应用于 Facebook 的数据仓库 Hive 中。与传统数据库的数据存储结构相比，RCFile 更有效地满足了基于 MapReduce 的数据仓库的四个关键需求。

（1）Fast data loading

对于 Facebook 的产品数据仓库而言，快速加载数据（写数据）是非常关键的。每天大约有超过 20TB 的数据上传到 Facebook 的数据仓库，由于数据加载期间网络和磁盘流量会干扰正常的查询执行，因此缩短数据加载时间是非常必要的。

（2）Fast query processing

为了满足实时性的网站请求和支持高并发用户提交查询的大量读负载，查询响应时间是非常关键的，这要求底层存储结构能够随着查询数量的增加而保持高速的查询处理。

（3）Highly efficient storage space utilization

高速增长的用户活动总是需要可扩展的存储容量和计算能力，有限的磁盘空间需要合理管理海量数据的存储。实际上，该问题的解决方案就是最大化磁盘空间利用率。

（4）Strong adaptivity to highly dynamic workload patterns

同一份数据集会供给不同应用的用户，通过各种方式来分析。某些数据分析是例行过程，按照某种固定模式周期性执行；而另一些则是从中间平台发起的查询。大多数负载不遵循任何规则模式，这需要底层系统在存储空间有限的前提下，对数据处理中不可预知的动态数据具备高度的适应性，而不是专注于某种特殊的负载模式。

2. RCFile 的设计与实现

RCFile 存储结构遵循“先水平划分，再垂直划分”的设计理念，这个想法来源于 PAX。

⊖ Facebook 工程师发表的 *RCFile: A Fast and Spaceefficient Data Placement Structure in MapReducebased Warehouse Systems*。

它结合了行存储和列存储的优点。首先，RCFile 保证同一行的数据位于同一节点，因此元组重构的开销很低；其次，像列存储一样，RCFile 能够利用列维度的数据压缩，并且能跳过不必要的列读取。图 10-7 是一个 HDFS 块内 RCFile 方式存储的例子。

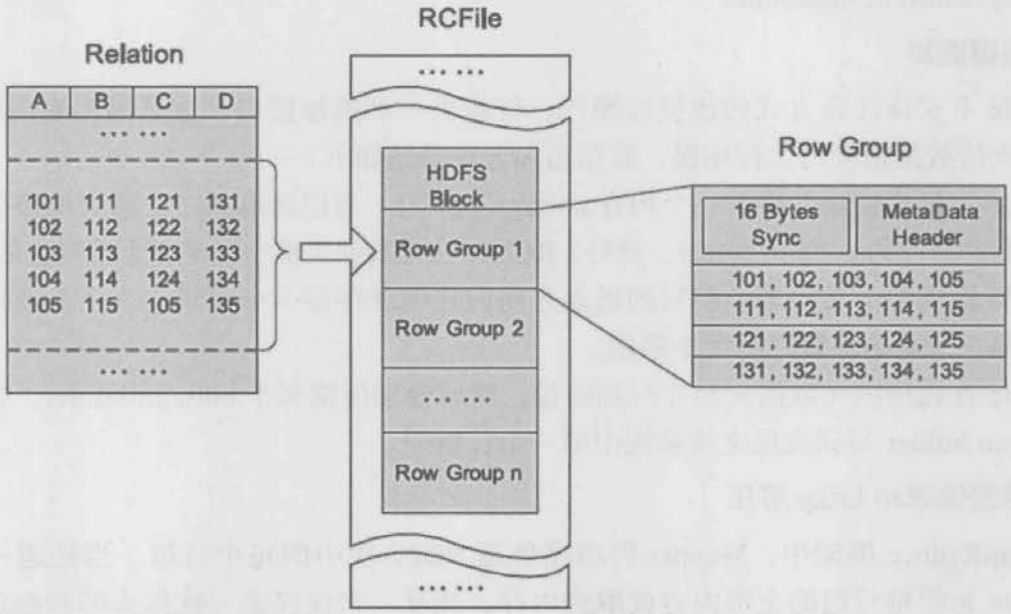


图 10-7 HDFS 块内 RCFile 方式存储

3. 数据格式

RCFile 在 HDFS 分布式文件系统之上设计并实现，RCFile 按照如下所示数据格式来存储一张表。

RCFile 基于 HDFS 架构，表格占用多个 HDFS 块。每个 HDFS 块中，RCFile 以行组为基本单位来组织记录。也就是说，存储在一个 HDFS 块中的所有记录被划分为多个行组。对于一张表，所有行组大小都相同。一个 HDFS 块会有一个或多个行组。

一个行组包括三个部分。第一部分是行组头部的同步标识，主要用于分隔 HDFS 块中的两个连续行组；第二部分是行组的元数据头部，用于存储行组单元的信息，包括行组中的记录数、每个列的字节数、列中每个域的字节数；第三部分是表格数据段，即实际的列存储数据。在该部分中，同一列的所有域顺序存储。

4. 压缩方式

RCFile 的每个行组中，元数据头部和表格数据段分别进行压缩。对于所有元数据头部，RCFile 使用 RLE（Run Length Encoding）算法来压缩数据。由于同一列中所有域的长度值都顺序存储在该部分，RLE 算法能够找到重复值的长序列（尤其对于固定的域长度）。

表格数据段不会作为整个单元来压缩；相反每个列被独立压缩（使用 Gzip 压缩算法）。RCFile 使用重量级的 Gzip 压缩算法，为了获得较好的压缩比，不使用 RLE 算法的原因在于此时列数据非排序。此外，由于 Lazy 压缩策略，当处理一个行组时，RCFile 不需要解压

所有列。因此，使用 Gzip 解压开销可以减少。

尽管 RCFile 对表格数据的所有列使用同样的压缩算法，但是使用不同的算法来压缩不同列或许效果会更好。RCFile 将来的工作之一可能就是根据每列的数据类型和数据分布，来自适应选择最好的压缩算法。

5. 数据追加

RCFile 不支持任意方式的数据写操作，仅提供一种追加接口，这是因为底层的 HDFS 当前仅仅支持数据追加写文件尾部。数据追加方法描述如下。

RCFile 为每列创建并维护一个内存 column holder，当记录追加时，所有域被分发，每个域追加到其对应的 column holder。此外，RCFile 在元数据头部中记录每个域对应的元数据。

RCFile 提供两个参数来控制写到磁盘之前内存中缓存多少个记录。一个参数是记录数的限制，另一个是内存缓存的大小限制。

RCFile 首先压缩元数据头部并写到磁盘，然后分别压缩每个 column holder，并将压缩后的 column holder 写到底层文件系统中的一个行组中。

6. 数据读取和 Lazy 解压

在 MapReduce 框架中，Mapper 将顺序处理 HDFS 块中的每个行组。当处理一个行组时，RCFile 无须将行组的全部内容读取到内存。相反，它仅仅读元数据头部和给定查询需要的列。因此，它可以跳过不必要的列以获得列存储的 I/O 优势。

例如，表 table1(c1, c2, c3, c4) 有 4 个列，做一次查询如下所示。

```
SELECT c1 FROM table1 WHERE c4 = 1
```

对每个行组，RCFile 仅仅读取 c1 和 c4 列的内容。在元数据头部和需要的列数据加载到内存中之后，它们需要解压。元数据头部总会解压并在内存中维护，直到 RCFile 处理下一个行组。然而，RCFile 不会解压所有加载的列，它使用 Lazy 解压技术。

Lazy 解压意味着列将不会在内存解压，直到 RCFile 决定列中数据真正对查询执行有用。由于查询使用各种 WHERE 条件，Lazy 解压非常有用。如果一个 WHERE 条件不能被行组中的所有记录满足，RCFile 将不会解压 WHERE 条件中不满足的列。例如，在上述查询中，所有行组中的列 c4 都解压了。然而，对于一个行组，如果列 c4 中没有值为 1 的域，就无须解压列 c1。

7. 行组大小

I/O 性能是 RCFile 关注的重点，因此 RCFile 需要行组够大并且大小可变。行组大小和下面几个因素相关。

行组大，数据压缩效率会比行组小时更有效。根据对 Facebook 日常应用的观察，当行组大小达到一个阈值后，增加行组大小并不能进一步增加 Gzip 算法下的压缩比。

行组变大能够提升数据压缩效率并减少存储量。因此，如果对缩减存储空间方面有强烈需求，不建议选择使用小行组。需要注意的是，当行组的大小超过 4MB，数据的压缩比

将趋于一致。尽管行组变大有助于减少表格的存储规模，但是可能会损害数据的读性能，因为这样减少了 Lazy 解压带来的性能提升。而且行组变大会占用更多的内存，这会影响并发执行的其他 MapReduce 作业。考虑到存储空间和查询效率两个方面，Facebook 选择 4MB 作为默认的行组大小，当然也允许用户自行选择参数进行配置。

RCFile 操作示例如下。

```
hive> CREATE TABLE rc_file(str STRING)
hive> STORED AS RCFILE;
OK
Time taken: 0.184 seconds
hive> INSERT OVERWRITE TABLE rc_file SELECT * FROM sequence_file;
```

10.3.4 自定义文件格式

当用户的数据文件格式不能被当前 Hive 所识别的时候，可以自定义文件格式。通过实现 InputFormat 和 OutputFormat 自定义输入 / 输出格式[⊖]。

创建表命令如下。

```
hive> CREATE  TABLE user_def(str STRING)
hive> STORED AS
hive> INPUTFORMAT  'org.apache.hadoop.hive.contrib.fileformat.base64.Base64TextInputFormat'
hive> OUTPUTFORMAT 'org.apache.hadoop.hive.contrib.fileformat.base64.Base64TextOutputFormat';
```

base64.txt 文件是用 Base64 加密过的文件，文件内容如下。

```
$ cat base64.txt
aGVsbG8saGl2ZQ==
aGVsbG8sd29ybGQ=
aGVsbG8saGFkb29w
```

base64.txt 文件解码后的内容如下。

```
hello,hive
hello,world
hello,hadoop
```

Load 数据并查询。

```
hive> LOAD DATA LOCAL INPATH '/home/work/ base-file.txt' INTO TABLE user_def;
Copying data from file:/home/work/ base-file.txt
Copying file: file:/home/work/ base-file.txt
Loading data to table default. base-file
    OK
    Time taken: 4.742 seconds
hive> SELECT * FROM user_def;
    OK
    hello,hive
    hello,world
```

⊖ 参考代码地址为 .\hive-0.9.0\src\contrib \src\java\ org\ apache\hadoop\hive\contrib\fileformat\base64。

```
    hello,hadoop
Time taken: 1.953 seconds
```

10.4 Hive 操作

10.4.1 表操作

Hive 和 MySQL 的表操作语句类似，如果熟悉 MySQL，学习 Hive 的表操作就非常容易了，下面对 Hive 的表操作进行深入讲解。

1. 创建表

Hive 创建表操作命令如下。

```
Create [EXTERNAL] TABLE [IF NOT EXISTS] table_name
[(col_name data_type [COMMENT col_comment], ...)]
[COMMENT table_comment]
[PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
[CLUSTERED BY (col_name, col_name, ...) [SORTED BY (col_name [ASC|DESC], ...)]
INTO num_buckets BUCKETS]
[ROW FORMAT row_format]
[STORED AS file_format]
[LOCATION hdfs_path]
[LIKE existing_table_name]
[SERDE serde_name]
```

- CREATE TABLE：创建一个指定名字的表。如果相同名字的表已经存在，则抛出异常；用户可以用 IF NOT EXIST 选项来忽略这个异常。
- EXTERNAL：关键字可以让用户创建一个外部表，在建表的同时指定一个指向实际数据的路径（LOCATION），Hive 创建内部表时，会将数据移动到数据仓库指向的路径；若创建外部表，仅记录数据所在的路径，不对数据的位置做任何改变。在删除表的时候，内部表的元数据和数据会被一起删除，而外部表只删除元数据，不删除数据。
- COMMENT：字段的注释。
- LIKE：允许用户复制现有的表结构，但是不复制数据。

注意 如果文件数据是纯文本，可以使用 STORED AS TEXTFILE。如果数据需要压缩，使用 STORED AS SEQUENCE 。有分区的表可以在创建的时候使用 PARTITIONED BY 语句。一个表可以拥有一个或者多个分区，每一个分区单独存在一个目录下。而且，表和分区都可以对某个列进行 CLUSTERED BY 操作，将若干个列放入一个桶（bucket）中。也可以利用 SORTED BY 对数据进行排序。这样可以为特定应用提高性能。

用户在建表的时候可以自定义 SerDe 或者使用自带的 SerDe。如果没有指定 ROW FORMAT 或者 ROW FORMAT DELIMITED，将会使用自带的 SerDe。在建表的时候，用

户还需要为表指定列，用户在指定表的列的同时也会指定自定义的 SerDe，Hive 通过 SerDe 确定表的具体列的数据。表名和列名不区分大小写，SerDe 和属性名区分大小写。表和列的注释是字符串。

Hive 的数据表分为两种，内部表和外部表。

- **内部表**：Hive 创建并通过 LOAD DATA INPATH 进数据库的表，这种表可以理解为数据和表结构都保存在一起的数据表。当通过 DROP TABLE table_name 删除元数据中表结构的同时，表中的数据也同样会从 HDFS 中被删除。
- **外部表**：在表结构创建以前，数据已经保存在 HDFS 中，通过创建表结构，将数据格式化到表的结构里。当进行 DROP TABLE table_name 操作的时候，Hive 仅仅删除元数据的表结构，而不删除 HDFS 上的文件，所以，相比内部表，外部表可以更放心大胆地使用。

下面详细介绍对表操作的命令及使用方法。

1）创建内部表使用 CREATE TABLE 命令。与 MySQL 创建表的命令一样，COMMENT 是对字段的注释。例如：

```
CREATE TABLE table1 IF NOT EXISTS table1(
  id  INT COMMENT 'comment1',
  name  STRING COMMENT 'comment2'
no  INT COMMENT 'comment3'
);
```

2）创建外部表使用 EXTERNAL 关键字。IF NOT EXISTS 表示如果 table2 表不存在就创建，存在就不创建。例如：

```
CREATE EXTERNAL TABLE table2 IF NOT EXISTS table2 (
    id  INT COMMENT 'comment1',
    name  STRING COMMENT 'comment2',
no  INT COMMENT 'comment3'
)
```

3）删除表。数据表在删除的时候，内部表会连数据一起删除，而外部表只删除表结构，数据还是保留的。删除表的命令如下。

```
DROP TABLE tableName;
```

4）改表结构。例如对 table2 表添加两个字段 data_time 和 password，操作命令如下。

```
ALTER TABLE  table2 ADD COLUMNS(
data_time STRING COMMENT 'comment1',
password STRING COMMENT 'comment2'
);
```

5）修改表名。例如把 table2 表重命名为 table3，操作命令如下。

```
ALTER TABLE  table2 RENAME TO  table3;
```

这个命令可以让用户为表更名，数据所在的位置和分区名并不改变。换而言之，旧的

表名并未“释放”，对旧表的更改会改变新表的数据。

6）创建与已知表相同结构的表。例如创建一个与 table2 表结构相同的表，表名为 copy_table2，这里要用到 LIKE 关键字，操作命令如下。

```
CREATE  TABLE  copy_table2 LIKE table2;
```

2. 表查询

Hive 的查询语句与标准 SQL 语句类似，具体的语法如下。

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
  [
CLUSTER BY col_list
  | [DISTRIBUTE BY col_list]
[SORT BY col_list]
  ]
[LIMIT number]
```

一个 SELECT 语句可以是一个 union 查询或一个子查询的一部分；table_reference 是查询的输入，可以是一个普通表、视图、join 或子查询。看以下几个例子。

1）查询 table1 表的所有内容，查询语句如下：

```
hive> SELECT * FROM table1;
OK
 01     liugang     1000
02      lisi        1001
03      wangwu      1002
Time taken: 5.002 seconds
```

SELECT * 查询没有开启 MapReduce 任务，这是 Hive 查询语句中唯一没有把 Hive 查询语句解释为 MapReduce 任务执行。

2）查询 table1 表的 name 属性，查询语句如下：

```
hive> SELECT name FROM table1;
Total MapReduce jobs = 1
...
Total MapReduce CPU Time Spent: 460 msec
OK
    liugang
    lisi
    wangwu
Time taken: 20.902 seconds
```

“SELECT name FROM table1;”被解释成了一个 MapReduce 任务执行。

(1) WHERE 查询

WHERE 查询是一个布尔表达式。例如，下面的查询语句只返回销售记录大于 10，且

归属地属于美国的销售代表。Hive 不支持在 WHERE 子句中的 IN、EXIST 或子查询。

```
SELECT * FROM sales WHERE amount > 10 AND region = "beijing" ;
```

（2）ALL 和 DISTINCT 选项

使用 ALL 和 DISTINCT 选项区分对重复记录的处理。默认是 ALL，表示查询所有记录，DISTINCT 表示去掉重复的记录。

```
SELECT age, grade FROM table1;
```

1) 查询 table1 表中的所有 age、grade 的内容，等同于如下操作。

```
SELECT ALL age, grade FROM table1;
```

2) 查询去掉 age 和 grade 重复的记录，操作如下。

```
SELECT DISTINCT age, grade FROM table1;
```

3) 查询去掉 age 重复的记录，操作如下。

```
SELECT DISTINCT age FROM table1;
```

（3）基于 Partition 的查询

一般 SELECT 查询会扫描整个表（除非是为了抽样查询）。但是如果一个表使用 PARTITIONED BY 子句建表，查询就可以利用**分区剪枝**（input pruning）的特性，只扫描一个表中它关心的那一部分。Hive 当前的实现是，只有分区断言出现在离 FROM 子句最近的那个 WHERE 子句中，才会启用分区剪枝。例如，page_views 表使用 date 列分区，以下语句只会读取分区为‘2013-08-01’的数据。

```
SELECT page_views.* FROM page_views WHERE page_views.date >= '2013-08-01' AND
page_views.date <= '2013-08-31';
```

（4）HAVING 子句转换

Hive 不支持 HAVING 子句，可以将 HAVING 子句转化为一个子查询。例如以下这条语句 Hive 不支持。

```
SELECT col1 FROM table1 GROUP BY col1 HAVING SUM(col2) > 10;
```

可以用以下查询来表达。

```
SELECT col1 FROM (SELECT col1, SUM(col2) AS col2 sum FROM table1 GROUP BY col1)
table2 WHERE table2.col2sum > 10 ;
```

（5）LIMIT 限制查询

LIMIT 可以限制查询的记录数，查询的结果是随机选择的。下面的语句用来从 t1 表中随机查询 5 条记录。

```
SELECT * FROM table1 LIMIT 5 ;
```

下面的语句查询销售记录前 5 个销售代表。

```
SET mapred.reduce.tasks = 1 // 设置 Reduce 的个数，默认为 1
SELECT * FROM sales SORT BY amount DESC LIMIT 5 ;
```

（6）正则表达式查询

SELECT 语句可以使用正则表达式做列选择，下面的语句用来查询除了 datetime 和 month 之外的所有列。

```
SELECT `(datetime|month)?+.+` FROM sales;
```

（7）GROUP BY 分组查询

GROUP BY 分组查询在数据统计时比较常用，接下来讲解 GROUP BY 的使用。

1）创建一个表 group_test，表的内容如下。

```
hive> desc group_test;
OK
userid  string
gender string
ip string
```

2）计算表的行数命令如下。

```
SELECT COUNT(*) FROM group_test;
```

3）根据性别计算去重用户数。

```
INSERT OVERWRITE TABLE group_gender_sum
SELECT group_test.gender, count (DISTINCT group_test.userid)
FROM group_test
GROUP BY group_test.gender;
```

同时可以做多个聚合操作，但是不能有两个聚合操作有不同的 DISTINCT 列。下面的语句是合法的。

```
INSERT OVERWRITE TABLE group_gender_agg
SELECT group_test.gender, count(DISTINCT group_test.userid),
                          count(*), sum(DISTINCT group_test.userid)
FROM group_test
GROUP BY group_test.gender;
```

但是，不允许在同一个查询内有多个 DISTINCT 表达式。下面的查询是不允许的。

```
INSERT OVERWRITE TABLE group_gender_agg
SELECT group_test.gender, count(DISTINCT group_test.userid), count(DISTINCT
group_test.ip)
FROM group_test
GROUP BY group_test.gender;
```

这条查询语句是不合法的，因为 DISTINCT group_test.userid 和 DISTINCT group_test.ip 操作了 userid 和 ip 两个不同的列。

（8）ORDER BY 排序查询

ORDER BY 会对输入做全局排序，因此只有一个 Reduce（多个 Reduce 无法保证全局有序）会导致当输入规模较大时，需要较长的计算时间。使用 ORDER BY 查询的时候，为了优化查询的速度，使用 hive.mapred.mode 属性。

```
hive.mapred.mode=nonstrict; (default value / 默认值)
hive.mapred.mode=strict;
```

与数据库中 ORDER BY 的区别在于，在 hive.mapred.mode = strict 模式下必须指定 limit，否则执行会报错。

```
hive>hive.mapred.mode= strict;
hive> SELECT * FROM group_test ORDER BY userid;
FAILED: Error in semantic analysis: 1:28 In strict mode, if ORDER BY is specified,
LIMIT must also be specified. Error encountered near token 'id'
```

原因是，ORDER BY 状态下，所有数据会到一台服务器进行 Reduce 操作，即只有一个 Reduce，如果在数据量大的情况下会出现无法输出结果的情况，如果进行 limit n ，只有 n * map number 条记录而已，只有一个 Reduce 也可以处理。正确的操作命令如下所示。

```
hive>hive.mapred.mode= strict;
hive> SELECT * FROM group_test ORDER BY userid LIMIT 2;
Total MapReduce jobs = 1
...
Total MapReduce CPU Time Spent: 430 msec
OK
  0001      female     127.0.0.1
  0002      male       127.0.0.1
Time taken: 10.802 seconds
```

(9) SORT BY 查询

SORT BY 不受 hive.mapred.mode 的值是否为 strict 和 nostrict 的影响。SORT BY 的数据只能保证在同一个 Reduce 中的数据可以按指定字段排序。

使用 SORT BY 可以指定执行的 Reduce 个数（set mapred.reduce.tasks=<number>）这样可以输出更多的数据。对输出的数据再执行归并排序，即可以得到全部结果。

```
hive>hive.mapred.mode= strict;
hive> SELECT * FROM group_test SORT BY userid;
Total MapReduce jobs = 1
...
Total MapReduce CPU Time Spent: 430 msec
OK
  0001      female     127.0.0.1
  0002      male       127.0.0.1
  0003      female     192.168.1.2
  0004      male       192.168.1.2
Time taken: 7.642 seconds
```

(10) DISTRIBUTE BY 查询

按照指定的字段对数据划分到不同的输出 Reduce 文件中。操作如下。

```
INSERT OVERWRITE LOCAL DIRECTORY '/home/hadoop/test.txt'  SELECT * FROM test
ORDER BY name DISTRIBUTE BY length(name);
```

此方法根据 name 的长度划分到不同的 Reduce 中，最终输出到不同的文件中。 length

是内建函数，也可以指定其他的函数或者使用自定义函数。

（11）CLUSTER BY 查询

CLUSTER BY 除了具有 DISTRIBUTE BY 的功能外还兼具 SORT BY 的功能。

3. 数据加载

Hive 装载数据没有做任何转换，加载到表中的数据只是进入相应的配置单元表的位置移动数据文件。语法如下。

```
LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE tablename
[PARTITION (partcol1=val1, partcol2=val2 ...)]
```

LOAD 操作只是单纯地复制/移动操作，将数据文件移动到 Hive 表对应的位置。filepath 可以是以下类型。

- 相对路径，例如 project/data1；
- 绝对路径，例如 /user/hive/project/data1；
- 包含模式的完整 URI，例如 hdfs://namenode:9000/user/hive/project/data1。

加载的目标可以是一个表或者分区。如果表包含分区，必须指定每一个分区的分区名。filepath 可以引用一个文件（这种情况下，Hive 会将文件移动到表所对应的目录中）或者是一个目录（在这种情况下，Hive 会将目录中的所有文件移动至表所对应的目录中）。

1）指定 LOCAL 的情况下：

- LOAD 命令会查找本地文件系统中的 filepath。如果发现是相对路径，则路径会被解释为相对于当前用户的当前路径。用户也可以为本地文件指定一个完整的 URI，比如 file:///user/hive/project/data1。
- LOAD 命令会将 filepath 中的文件复制到目标文件系统中。目标文件系统由表的位置属性决定。被复制的数据文件移动到表的数据对应的位置。

2）没有指定 LOCAL 的情况下，如果 filepath 指向的是一个完整 URI，Hive 会直接使用这个 URI。否则：

- 如果没有指定 Schema 或者 authority，Hive 会使用在 Hadoop 配置文件中定义的 Schema 和 authority，fs.default.name 指定了 NameNode 的 URI。
- 如果路径不是绝对的，Hive 相对于 /user/ 进行解释。
- Hive 会将 filepath 中指定的文件内容移动到 table（或者 Partition）所指定的路径中。

注意 如果使用 OVERWRITE 关键字，则目标表（或者分区）中的内容（如果有）会被删除，然后再将 filepath 指向的文件/目录中的内容添加到表/分区中。

如果目标表（分区）已经有一个文件，并且文件名和 filepath 中的文件名冲突，现有的文件会被新文件所替代，从本地导入数据到表格并追加原表。

（1）加载本地数据

加载本地数据使用 LOCAL 关键字。例如，文件 pv_2012-06-08_us.txt 内容为如下。

```
20120608123445, AutoNavi1, 5
20120608123446, AutoNavi2, 6
20120608123447, AutoNavi3, 7
```

pv_2012-06-08_us.txt 文件放在本地 /tmp 目录下，把 pv_2012-06-08_us.txt 文件的数据导入表 table2 中，操作命令如下。

```
LOAD DATA LOCAL INPATH '/tmp/pv_2012-06-08_us.txt'  INTO TABLE table2
```

如果还有一个 pv_2012-06-09_us.txt 文件，与 pv_2012-06-08_us.txt 文件内容相似，也存放在 /tmp 目录下。执行如下命令，结果发现 table2 里原来的数据还在，没有被覆盖。

```
LOAD DATA LOCAL INPATH '/tmp/pv_2012-06-09_us.txt ' INTO TABLE table2;
```

如果执行如下命令，table2 表中原来 pv_2012-06-08_us.txt 文件的内容就会被删除，table2 表中只存了 pv_2012-06-09_us.txt 的内容。

```
LOAD DATA LOCAL INPATH '/tmp/pv_2012-06-09_us.txt ' OVERWRITE  INTO TABLE table2;
```

（2）加载 HDFS 中的文件

加载 HDFS 中的文件不使用 LOCAL 关键字，操作如下。

```
LOAD DATA INPATH '/user/hive/input/ pv_2012-06-09_us.txt ' INTO TABLE table2;
```

如果加上 OVERWRITE 关键字，table2 中的原始数据会被删除。OVERWRITE 关键字在开发中还是很少用到的。

4. 表插入

Hive 表插入的基本语法如下。

```
INSERT OVERWRITE TABLE tablename [PARTITON(partcol1=val1,partclo2=val2)]
                    select_statement FROM from_statement
```

（1）单表插入

创建一个表 insert_table，表结构和 table2 的结构相同，把 table2 表中的数据插入到新建的表 insert_table 中，代码如下。

```
INSERT OVERWRITE TABLE insert_table  SELECT * FROM table2;
```

OVERWRITE 关键字表示如果 insert_table 表中有数据就删除。

（2）多表插入

多表插入的语法如下。

```
FROM table_name
INSERT OVERWRITE TABLE tablename1 [PARTITON(partcol1=val1,partclo2=val2)]
                    select_statement1
INSERT OVERWRITE TABLE tablename2 [PARTITON(partcol1=val1,partclo2=val2)]
                    select_statement2
```

也可以把 FROM 放在后面，语法如下。

```
INSERT OVERWRITE TABLE tablename1 [PARTITON(partcol1=val1,partclo2=val2)]
                    select_statement1 FROM table_name
```

```
INSERT OVERWRITE TABLE tablename2 [PARTITON(partcol1=val1,partclo2=val2)]
                    select_statement2 FROM table_name
```

例如，表 test_table 有 key 和 value 两个字段，查询字段 key 并插入 test_insert1 表，查询字段 value 并插入 test_insert2 表。操作命令如下。

```
FROM test_table
INSERT OVERWRITE TABLE test_insert1
SELECT key
INSERT OVERWRITE TABLE test_insert2
SELECT value;
```

INSERT 操作的时候，FROM 子句既可以放在 SELECT 子句后面，也可以放在 INSERT 子句前面。

Hive 不支持用 INSERT 语句一条一条地进行插入操作，也不支持 UPDATE 操作。数据是以 Load 的方式加载到建立好的表中。数据一旦导入就不可以修改。

通过查询将数据保存到 HDFS，directory 为 HDFS 文件系统的目录。操作命令如下。

```
INSERT OVERWRITE [LOCAL] DIRECTORY directory SELECT... FROM ...
```

导入数据到本地目录，操作命令如下。

```
INSERT OVERWRITE LOCAL DIRECTORY '/home/jayliu/hive'  SELECT * FROM test_insert1;
```

产生的文件会覆盖指定目录中的其他文件，即将目录中已经存在的文件进行删除。

同一个查询结果可以同时插入到多个表或者多个目录中，命令如下。

```
FROM  test_insert1
INSERT OVERWRITE LOCAL DIRECTORY '/home/jayliu/hive' SELECT *
INSERT OVERWRITE  DIRECTORY '/user/hive/test' SELECT value;
```

- SELECT * 表示把 test_insert1 表中的所有数据复制到本地 /home/jayliu/hive 目录下面。
- SELECT value 表示把 test_insert1 的 value 字段内容复制到 HDFS 文件系统的 /user/hive/test 目录下。

10.4.2 视图操作

Hive 0.6 及以上版本支持视图，Hive VIEW 具有以下特点。

- VIEW 是逻辑存在，Hive 暂不支持物化视图；
- VIEW 只读，不支持 LOAD/INSERT/ALTER。可以使用 ALTER VIEW 改变 VIEW 定义；
- VIEW 内可能包含 ORDER BY/LIMIT 语句，假如一个针对 VIEW 的查询也包含这些语句，则 VIEW 中的语句优先级高。例如，定义 VIEW 数据为 limit 10，针对 VIEW 的查询 limit 20，则最多返回 10 条数据；
- Hive 支持迭代视图。

创建 VIEW 语句：

```
CREATE VIEW [IF NOT EXISTS] view_name [(column_name [COMMENT column_comment], ...) ]
```

```
[COMMENT view_comment]
[TBLPROPERTIES (property_name = property_value, ...)]
AS SELECT ...
```

删除 VIEW 语句：

```
DROP VIEW [IF EXISTS] view_name
```

修改 VIEW 语句：

```
ALTER VIEW view_name SET TBLPROPERTIES table_properties
table_properties:
    : (property_name = property_value, property_name = property_value, ...)
```

接下来通过一个 test 表演示视图的创建、删除和修改操作。

1）创建一个测试表。

```
hive> CREATE TABLE test (id int, name string);
OK
Time taken: 0.19 seconds
hive> desc test;
OK
id      int
name    string
Time taken: 0.16 seconds
```

2）基于表 test 创建一个 test_view 视图。

```
hive> CREATE VIEW test_view (id, name_length)  AS SELECT id, length(name) FROM test;
OK
Time taken: 0.424 seconds
```

3）查看 test_view 视图属性。

```
hive> DESC test_view;
OK
id      int
name_length     int
Time taken: 0.071 seconds
```

4）查看视图结果。

```
hive> SELECT * FROM  test;
1       zhangsan
2       lisi
3       wangwu
Time taken: 0.176 seconds
hive> SELECT * FROM  test_view;
Total MapReduce jobs = 1
...
Total MapReduce CPU Time Spent: 680 msec
OK
1       zhangsan
2       lisi
3       wangwu
Time taken: 20.592 seconds
```

10.4.3 索引操作

Hive 的查询跟普通的 Hadoop MapReduce 相比没有大的区别，都是对原始数据的暴力扫描，如果能够像数据库那样使用索引，数据扫描的速度将会大幅度提升。前面在 MapReduce 上使用了索引，这次在此基础上拓展到 Hive 中（实际上也是一个特殊的 InputFormat）。

索引是标准的数据库技术，Hive 0.7 版本之后支持索引。Hive 索引采用的不是 'one size fites all' 的索引实现方式，而是提供插入式接口，并且提供一个具体的索引实现作为参考。

Hive 索引具有以下特点。

- 索引 Key 冗余存储，提供基于 Key 的数据视图；
- 存储设计以优化查询和检索性能；
- 对于某些查询减少 I/O，从而提高性能。

Hive 索引操作的方法如下。

1）Hive 创建索引。

```
CREATE INDEX index_name
ON TABLE base_table_name (col_name, ...)
AS 'index.handler.class.name'
[WITH DEFERRED REBUILD]
[IDXPROPERTIES (property_name=property_value, ...)]
[IN TABLE index_table_name]
[PARTITIONED BY (col_name, ...)]
[
  [ ROW FORMAT ...] STORED AS ...
  | STORED BY ...
]
[LOCATION hdfs_path]
[TBLPROPERTIES (...)]
[COMMENT "index comment"]
```

注意 INDEX 的 Partition 默认和数据表一致；视图上不能创建 INDEX；INDEX 可以通过 STROED AS 配置存储格式。

2）更新数据。

```
ALTER INDEX index_name ON table_name [PARTITION (...)] REBUILD
```

假如在创建索引是使用“ WITH DEFERRED REBUILD”语句，则索引创建是为空，可以通过“ALTER INDEX ... REBUILD”在一个 Partition 上或所有 Partition 上构建索引。

注意 当 Hive 数据更新时，必须调用该语句更新索引。Index rebuild 操作是一个原子操作，因此当 rebuild 失败时，先前构建的索引也无法使用。

3）删除索引。

```
DROP INDEX  index_name ON table_name;
```

4）查看索引。

```
SHOW  INDEX ON table_name;
```

5）创建表和索引案例。

```
hive> CREATE TBALE index_test(id INT, name STRING)
    > PARTITIONED  BY (dt STRING)
    > ROW FORMAT DELIMITED FILEDS TERMINATED BY ',';
OK
Time taken: 32.446 seconds
```

创建一个索引测试表 index_test，dt 作为分区属性，“ROW FORMAT DELIMITED FILEDS TERMINATED BY ‘,’ ” 表示用逗号分割字符串，默认为 ‘\001’。

6）创建一个临时索引表 index_tmp。

```
hive> CREATE TBALE index_tmp(id INT, name STRING, dt STRING)
    > ROW FORMAT DELIMITED FILEDS TERMINATED BY ',';
OK
Time taken: 0.157 seconds
```

7）加载本地数据到 index_tmp 表中。

```
hive> LOAD DATA LOCAL INPATH '/home/work/data/alter_test.txt' INTO TABLE index_tmp;
Copying data from file:/home/work/data/alter_test.txt
Copying file: file:/home/work/data/alter_test.txt
Loading data to table default.index_tmp
OK
Time taken: 1.587 seconds
```

设置 Hive 的索引属性来优化索引查询，命令如下。

```
hive> set hive.exec.dynamic.partition.mode=nonstrict;
hive> set hive.exec.dynamic.partition=true;
```

设置所有列为 dynamic partition 列时，需要以下设置。

```
hive.exec.dynamic.partition.mode=nonstrict
```

使用动态分区需要设置 hive.exec.dynamic.partition 参数值为 true。这两个属性可以优化 Hive 索引的查询。

8）查询 index_tmp 表中的数据，插入 table_test 表中。

```
hive> INSERT OVERWIRTE TABLE index_test  PARTITION(dt)
    > SELECT id,name,dt
    > FROM index_tmp;
Total MapReduce jobs = 2
...
OK
Time taken: 56.103 seconds
```

9）使用 index_test 表，在属性 id 上创建一个索引 index1_index_test。

```
hive> CREATE INDEX index1_index_test ON TABLE index_test(id) AS
'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler' WITH DEFERRED REBUILD ;
OK
Time taken: 0.764 seconds
```

创建索引的时候，Hive 首先调用接口的 usesIndexTable 方法，判断索引是否是以 Hive Table 的方式存储（默认的实现是存储在 Hive 中的）。然后调用 analyzeIndexDefinition 分析索引创建语句是否合法，如果没有问题将在元数据标 IDXS 中添加索引表，否则抛出异常。如果索引创建语句中使用 WITH DEFERRED REBUILD，在执行以下语句时将调用 generateIndexBuildTaskList 获取 Index 的 MapReduce，并执行为索引填充数据。

```
ALTER INDEX  table1_index on table1 rebuild
```

10）填充索引数据。

```
hive> ALTER INDEX index1_index_test ON index_test REBUILD;
Total MapReduce jobs = 3
...
OK
Time taken: 138.589 seconds
```

11）查看创建的索引。

```
hive> SHOW INDEX ON index_test;
OK
index1_index_test  index_test  id  default__index_test_index1_index_test__ compact
Time taken: 0.163 seconds
```

12）查看分区信息。

```
hive> SHOW PARTITIONS index_test;
OK
dt=2013-08-10
dt=2013-08-11
dt=2013-08-12
Time taken: 0.17 seconds
```

13）查看索引数据。

```
$ hadoop fs -ls /user/hive/warehouse/default__index_test_index1_index_test__
Found 3 items
drwxr-xr-x   - work supergroup          0 2013-08-16 18:27 /user/hive/warehouse/
default__index_test_index1_index_test__/dt=2013-08-10
drwxr-xr-x   - work supergroup          0 2013-08-16 18:27 /user/hive/warehouse/
default__index_test_index1_index_test__/dt=2013-08-11
drwxr-xr-x   - work supergroup          0 2013-08-16 18:28 /user/hive/warehouse/
default__index_test_index1_index_test__/dt=2013-08-12
$ hadoop fs -cat /user/hive/warehouse/default__index_test_index1_index_test__/
dt=2013-08-10/000000_0
5,hdfs://localhost:9000/user/hive/warehouse/index_test/dt=2013-08-10/000000_0,0
6,hdfs://localhost:9000/user/hive/warehouse/index_test/dt=2013-08-10/000000_0,10
7,hdfs://localhost:9000/user/hive/warehouse/index_test/dt=2013-08-10/000000_0,20
```

14）删除索引。

```
hive> DROP INDEX index1_index_test ON index_test;
OK
Time taken: 0.761 seconds
hive> SHOW INDEX ON index_test;
OK
Time taken: 0.095 seconds
```

15）索引数据也被删除。

```
$ hadoop fs -ls /user/hive/warehouse/default__index_test_index1_index_test__
ls: Cannot access /user/hive/warehouse/default__index_test_index1_index_test__:
No such file or directory.
```

删除索引的同时，索引数据也被删除。

16）修改配置信息。

```
<property>
  <name>hive.optimize.index.filter</name>
  <value>true</value>
  <description>Whether to enable automatic use of indexes</description>
</property>
<property>
  <name>hive.optimize.index.groupby</name>
  <value>true</value>
  <description>Whether to enable optimization of group-by queries using Aggregate
indexes.</description>
</property>
<property>
  <name>hive.optimize.index.filter.compact.minsize</name>
  <value>5120</value>
  <description>Minimum size (in bytes) of the inputs on which a compact index is
automatically used. default=5368709120</description>
</property>
```

hive.optimize.index.filter 和 hive.optimize.index.groupby 参数默认是 false。使用索引的时候必须把这两个参数开启，才能起到作用。

hive.optimize.index.filter.compact.minsize 参数为输入一个紧凑的索引将被自动采用最小尺寸、默认 5368709120（以字节为单位）。

10.4.4 分区操作

Hive 的分区通过在创建表时启用 PARTITION BY 实现，用来分区的维度并不是实际数据的某一列，具体分区的标志是由插入内容时给定的。当要查询某一分区的内容时可以采用 WHERE 语句，例如使用“WHERE tablename.partition_key > a”创建含分区的表。

创建分区的语法如下。

```
CREATE TABLE table_name(
...
)
PARTITIONED BY(dt STRING, country STRING)
```

1. 创建分区

Hive 中创建分区表没有什么复杂的分区类型（范围分区、列表分区、hash 分区、混合分区等）。分区列也不是表中的一个实际的字段，而是一个或者多个伪列。意思是说，在表的数据文件中实际并不保存分区列的信息与数据。

例如，创建一个简单的分区表。

```
CREATE TABLE partition_test(
member_id string,
name string
)
PARTITIONED BY (stat_date string,province string)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
```

这个例子中创建了 stat_date 和 province 两个字段作为分区列。通常情况下需要预先创建好分区，然后才能使用该分区。例如：

```
ALTER TABLE partition_test ADD PARTITION (stat_date='20130528',province='beijing');
```

这样就创建了一个分区。这时会看到 Hive 在 HDFS 存储中创建了一个相应的文件夹。

```
$ hadoop fs -ls /user/hive/warehouse/partition_test/stat_date=20130528
Found 1 items
drwxr-xr-x  -  admin  supergroup  0  2013-05-29 09:53
/user/hive/warehouse/partition_test/stat_date=20130528/province=beijing
```

每一个分区都会有一个独立的文件夹，下面是该分区所有的数据文件。在这个例子中 stat_date 是主层次，province 是副层次，所有 stat_date='20130528'，而 province 不同分区都会在 /user/hive/warehouse/partition_test/stat_date=20130528 下，而 stat_date 不同分区都会在 /user/hive/warehouse/partition_test/ 下。例如：

```
$ hadoop fs -ls /user/hive/warehouse/partition_test/
Found 2 items
drwxr-xr-x  -  admin  supergroup  0  2013-05-28 19:46
/user/hive/warehouse/partition_test/stat_date=20130528
drwxr-xr-x  -  admin  supergroup  0  2013-05-29 09:53
/user/hive/warehouse/partition_test/stat_date=20130529
```

注意，因为分区列的值要转化为文件夹的存储路径，所以如果分区列的值中包含特殊值（如 '%'、':'、'/'、'#'），它将会被使用 % 加 2 字节的 ASCII 码进行转义。例如：

```
hive> ALTER TABLE partition_test ADD PARTITION (stat_date='2013/05/28',province='beijing');
OK
Time taken: 4.644 seconds
```

结果如下。

```
$hadoop fs -ls /user/hive/warehouse/partition_test/
Found 3 items
drwxr-xr-x  -  admin supergroup  0  2013-05-29 10:06
/user/hive/warehouse/partition_test/stat_date=2013% 2F05% 2F28
```

```
drwxr-xr-x  -  admin  supergroup  0  2013-05-28 19:46
/user/hive/warehouse/partition_test/stat_date=20130526
drwxr-xr-x  -  admin  supergroup  0  2013-05-29 09:53
/user/hive/warehouse/partition_test/stat_date=20130528
```

2. 插入数据

使用一个辅助的非分区表 partition_test_input 准备向 partition_test 中插入数据，实现步骤如下。

1）查看 partition_test_input 表的结构，命令如下。

```
hive> desc partition_test_input;
OK
stat_date string
member_id string
name string
province string
```

2）查看 partition_test_input 的数据，命令如下。

```
hive> select * from partition_test_input。
OK
20130526 1 jayliu jiangsu
20130526 2 zhangsan hubei
20130528 3 lisi sichuan
20130528 4 wanwu beijing
```

3）向 partition_test 的分区中插入数据。

```
hive> INSERT OVERWRITE TABLE partition_test partition(stat_date='20130528',
province=' jiangsu') SELECT member_id,name FROM  partition_test_input WHERE stat_
date='20130528'and province=' jiangsu';
Total MapReduce jobs = 2
...
1 Rows loaded to partition_test
OK
```

还可以同时向多个分区插入数据，0.7 版本以后不存在的分区会自动创建，0.6 之前的版本官方文档上说必须要预先创建好分区，向多个分区插入数据命令如下。

```
hive>
> from partition_test_input
> insert overwrite table partition_test partition (stat_date='20130526',province
='jiangsu')
> select member_id,name where stat_date='20130526' and province='jiangsu'
> insert overwrite table partition_test partition (stat_date='20130528',province
='sichuan')
> select member_id,name where stat_date='20130528' and province='sichuan'
> insert overwrite table partition_test partition (stat_date='20130528',province
='beijing')
> select member_id,name where stat_date='20130528' and province='beijing';
```

```
Total MapReduce jobs = 4
...
3 Rows loaded to partition_test
OK
```

特别注意，在其他数据库中，一般向分区表中插入数据时系统会校验数据是否符合该分区，如果不符合会报错。而在 Hive 中，向某个分区中插入什么样的数据完全是手动控制的，因为分区键是伪列，不实际存储在文件中。例如：

```
hive> INSERT OVERWRITE TABLE partition_test PARTITION(stat_date='20130527',
province='liaoning') SELECT member_id,name FROM partition_test_input;
Total MapReduce jobs = 2
...
5 Rows loaded to partition_test
OK
```

stat_date 和 province 的值不存在，执行的时候没有报错，而是成功执行，接下来看执行结果。

```
hive> SELECT * FROM partition_test WHERE stat_date='20110527' AND province=
'liaoning';
OK
1 jayliu 20130527 liaoning
2 zhangsan 20130527 liaoning
3 lisi 20130527 liaoning
4 wangwu 20130527 liaoning
```

可以看到，partition_test_input 中的 4 条数据有着不同的 stat_date 和 province，但是在插入到 partition(stat_date='20130527', province='liaoning') 分区后，4 条数据的 stat_date 和 province 都变成相同，因为这两列数据是根据文件夹名字读取来的，而不是实际从数据文件中读取来的。

```
$ hadoop  fs  -cat
/user/hive/warehouse/partition_test/stat_date=20130527/province=liaoning/000000_0
1,jayliu
2,zhangsan
3,lisi
4,wangwu
```

3. 动态分区

下面介绍动态分区，因为按照上面的方法向分区表中插入数据，如果源数据量很大，针对一个分区就要写一个 INSERT，非常麻烦。况且在之前的版本中，必须先手动创建好所有的分区后才能插入，这就更麻烦了，你必须先要知道源数据中都有什么样的数据才能创建分区。

使用动态分区可以很好地解决上述问题。动态分区可以根据查询得到的数据自动匹配到相应的分区中去。

先设置 hive.exec.dynamic.partition 参数值为 true，默认值为 false，即不允许使用。

```
hive> set hive.exec.dynamic.partition;
hive.exec.dynamic.partition=false
hive> set hive.exec.dynamic.partition=true;
hive> set hive.exec.dynamic.partition;
hive.exec.dynamic.partition=true
```

动态分区的使用方法很简单，假设向 stat_date='20130528' 这个分区下插入数据，至于 province 插到哪个子分区下让数据库自己来判断，可以这样写。

```
hive> INSERT OVERWRITE TABLE partition_test PARTITION(stat_date='20130528',province)
> SELECT member_id,name,province FROM partition_test_input WHERE
stat_date='20130528';
Total MapReduce jobs = 2
...
3 Rows loaded to partition_test
OK
```

stat_date 叫做静态分区列，province 叫做动态分区列。SELECT 子句中需要把动态分区列按照分区的顺序写出来，静态分区列不用写出来。这样 stat_date='20130528' 的所有数据，会根据 province 的不同，分别插入 /user/hive/warehouse/partition_test/stat_date=20130528/ 下不同的子文件夹下，如果源数据对应的 province 子分区不存在，则会自动创建，非常方便，而且避免了手动控制插入数据与分区的映射关系存在的潜在风险。

注意，动态分区不允许主分区采用动态列而副分区采用静态列，这样将导致所有的主分区都要创建副分区静态列所定义的分区，命令如下。

```
hive> INSERT OVERWRITE TABLE partition_test partition(stat_date,province='liaoning')
> SELECT member_id,name,province FROM partition_test_input WHERE
province='liaoning';
FAILED: Error in semantic analysis: Line 1:48 Dynamic partition cannot be the
parent of a static partition 'liaoning'
```

创建失败，错误信息“Dynamic partition cannot be the parent of a static partition 'liaoning'”提示动态分区不允许主分区采用动态列而副分区采用静态列。

动态分区可以允许所有的分区列都是动态分区列，但是要首先设置一个参数 hive.exec.dynamic.partition.mode，命令如下。

```
hive> set hive.exec.dynamic.partition.mode;
hive.exec.dynamic.partition.mode=strict
```

它的默认值是 strick，即不允许分区列全部是动态的，这是为了防止用户只在子分区内创建动态分区，但是由于疏忽忘记为主分区列指定值了，这将导致一个 DML 语句在短时间内创建大量的新的分区（对应大量新文件夹），对系统性能带来影响。所以要进行以下设置。

```
hive> set hive.exec.dynamic.partition.mode=nostrick;
```

介绍 3 个参数。

- hive.exec.max.dynamic.partitions.pernode：每一个 MapReduce Job 允许创建的分区的最大数量，如果超过这个数量就会报错（默认值 100）。

- hive.exec.max.dynamic.partitions：一个 dml 语句允许创建的所有分区的最大数量（默认值 1000）。
- hive.exec.max.created.files：所有 MapReduce Job 允许创建的文件的最大数量（默认值 100000）。

当源表数据量很大时，单独一个 MapReduce Job 中生成的数据在分区列上可能很分散，举个简单的例子，比如下面的表要用 3 个 Map。

```
1
1
1
2
2
2
3
3
3
```

如果数据这样分布，每个 MapReduce 只需要创建 1 个分区就可以了。

```
          |1
map1 -->  |1
          |1
_____________________________________________________
          |2
map2 -->  |2
          |2
_____________________________________________________
          |3
map3 -->  |3
          |3
```

但是如果数据按下面这样分布，第一个 MapReduce 就要创建 3 个分区。

```
          |1
map1 -->  |2
          |3
_____________________________________________________
          |1
map2 -->  |2
          |3
_____________________________________________________
          |1
map3 -->  |2
          |3
```

尽量让分区列的值相同的数据在同一个 MapReduce 中，这样每一个 MapReduce 可以尽量少地产生新的文件夹，可以通过 DISTRIBUTE BY 将分区列值相同的数据放到一起，命令如下。

```
hive> INSERT OVERWRITE table partition_test PARTITION(stat_date,province)
> SELECT member_id,name,stat_date,province FROM partition_test_input DISTRIBUTE
BY stat_date,province;
Total MapReduce jobs = 1
...
18 Rows loaded to partition_test
OK
```

10.4.5 桶操作

Hive 中 table 可以拆分成 Partition table 和桶（BUCKET），桶操作是通过 Partition 的 CLUSTERED BY 实现的，BUCKET 中的数据可以通过 SORT BY 排序。

BUCKET 主要作用如下。

- 数据 sampling；
- 提升某些查询操作效率，例如 Map-Side Join 。

需要特别注意的是，CLUSTERED BY 和 SORT BY 不会影响数据的导入，这意味着，用户必须自己负责数据的导入，包括数据的分桶和排序。

'set hive.enforce.bucketing = true' 可以自动控制上一轮 Reduce 的数量从而适配 BUCKET 的个数，当然，用户也可以自主设置 mapred.reduce.tasks 去适配 BUCKET 个数，推荐使用：

```
set hive.enforce.bucketing = true
```

操作示例如下。

1）创建临时表 student_tmp，并导入数据。

```
hive> desc student_tmp;
OK
id      int
age     int
name    string
stat_date       string
Time taken: 0.106 seconds
hive> select * from student_tmp;
OK
1       20      jayliu      20130601
2       21      liugang     20160601
3       19      zhangsan    20130601
4       18      lisi        20130601
5       22      wangwu      20130601
Time taken: 0.123 seconds
```

2）创建 student 表。

```
hive>CREATE TABLE student(id INT, age INT, name STRING)
      >PARTITIONED BY(stat_date STRING)
      > CLUSTERED BY (id) SORTED BY(age) into 2 bucket
      > ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
```

3）设置环境变量。

```
hive> set hive.enforce.bucketing = true;
```

4）插入数据。

```
hive >FROM student_tmp
     >INSERT OVERWRITE TABLE student PARTITION(stat_date="20130602")
     > SELECT id,age,name WHERE stat_date="20130601" SORT BY age;
```

5）查看文件目录。

```
$ hadoop fs -ls /user/hive/warehouse/student/stat_date=20130602/
Found 2 items
-rw-r--r--   1 work supergroup          31 2013-05-31 19:52 /user/hive/warehouse/
student/stat_date=20130602/000000_0
-rw-r--r--   1 work supergroup          39 2013-05-31 19:52 /user/hive/warehouse/
student/stat_date=20130602/000001_0
```

6）查看 sampling 数据。

```
hive> SELECT * FROM student  TABLESAMPLE (bucket 1 out of 2 on id);
Total MapReduce jobs = 1
Launching Job 1 out of 1
...
OK
4       18      lisi    20130602
2       21      liugang     20130602
Time taken: 20.608 seconds
```

TABLESAMPLE 是抽样语句，语法如下。

```
TABLESAMPLE(BUCKET x OUT OF y)
```

y 必须是 table 中 BUCKET 总数的倍数或者因子。Hive 根据 y 的大小，决定抽样的比例。例如，table 总共分了 64 份，当 y=32 时，抽取 2 (64/32) 个 BUCKET 的数据，当 y=128 时，抽取 1/2 (64/128) 个 BUCKET 的数据。x 表示从哪个 BUCKET 开始抽取。例如，table 的 BUCKET 总数为 32，table sample(bucket 3 out of 16) 表示总共抽取 2（32/16）个 BUCKET 的数据，分别为第 3 个 BUCKET 和第 19（3+16）个 BUCKET 的数据。

10.5 Hive 复合类型

Hive 提供了复合数据类型。

- Struct：内部的数据可以通过 DOT（.）来存取。
  例如，表中一列 c 的类型为 STRUCT{a INT; b INT}，可以通过 c.a 来访问域 a。
- Array：array 中的数据为相同类型。
  例如，array A 中元素 ['a', 'b', 'c']，则 A[1] 的值为 'b'。
- Map（Key-Value 对）：可以通过 [" 指定域名称 "] 访问指定域。
  例如，Map M 包含一个 group->gid 的 Key-Value 对，gid 的值可以通过 M['group'] 来获取。

10.5.1 Struct 类型

Struct 是 Hive 自带的一个复合类型，和 C 语言中的 struct 结构体类似。例如有一个 student 对象，包含 name 和 age 两个属性，用 Struct 结构表示为：

```
student STRUCT<name:STRING,age:INT>
```

接下来介绍 Struct 复合结构在 Hive 里的使用。创建 Struct 表如下所示。

```
hive> CREATE TABLE struct_test(id INT, info struct<name:STRING, age:INT>)
    > ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
    > COLLECTION ITEMS TERMINATED BY ':';
OK
Time taken: 0.446 seconds
```

- FIELDS TERMINATED BY：字段与字段之间的分隔符。
- COLLECTION ITEMS TERMINATED BY：一个字段各个 item 的分隔符。

导入数据的步骤如下。

1）准备一个 struct_test.txt 文件，内容如下。

```
$ cat struct_test.txt
1,liu:30
2,zhang:30
3,li:20
4,wang:80
```

2）把 struct_test.txt 文件的内容加载到 struct_test 表中，命令如下。

```
hive> LOAD DATA LOCAL INPATH '/home/work/data/ struct_test ' INTO TABLE student_test;
Copying data from file:/home/work/data/ struct_test.txt
Copying file: file:/home/work/data/ struct_test.txt
Loading data to table default.student_test
OK
Time taken: 0.35 seconds
```

3）查询结果。

```
hive> SELECT info.age FROM student_test;
Total MapReduce jobs = 1
...
Total MapReduce CPU Time Spent: 490 msec
OK
30
30
20
80
Time taken: 21.677 seconds
```

提示 Hive 的 Struct 结构可以通过 info.age 来访问 info 的 age 属性；同理，如果访问 name 属性，操作命令为 info.name。

10.5.2 Array 类型

Array 是 Hive 自带的一个复合数据结构，与 Java 中的 Array 类型结构很类似，例如存放一系列的 Interge 类型的数据。

```
Array<Interge> array = new ArrayList< Interge >();
```

创建 Array 结构的表。

```
  hive> CREATE TABLE array_test(name STRING, student_id_list array<INT>)
      > ROW FORMAT DELIMITED
      > FIELDS TERMINATED BY ','
      > COLLECTION ITEMS TERMINATED BY ':';
OK
  Time taken: 0.099 seconds
```

导入数据步骤如下。

1）准备一个 array_test.txt 文件，内容如下。

```
$ cat array_test.txt
liu,1:2:3:4
zhang,5:6
li,7:8:9:10
```

2）导入 array_test.txt 文件内容到 array_test 表里，操作命令如下。

```
hive>  LOAD DATA LOCAL INPATH '/home/work/data/ array_test.txt' INTO TABLE class_test;
Copying data from file:/home/work/data/ array_test.txt
Copying file: file:/home/work/data/ array_test.txt
Loading data to table default. array_test
OK
Time taken: 0.198 seconds
```

3）查询导入的结果。

```
hive> SELECT student_id_list[3] FROM array_test;
Total MapReduce jobs = 1
...
Total MapReduce CPU Time Spent: 480 msec
OK
4
NULL
10
Time taken: 21.574 seconds
```

注意 Array 复合类型的下标从 1 开始，这点与 Java 的 Array 类型不同，Java 的 Array 类型下标是从 0 开始。结果中的 NULL 是因为 zhang, 5:6 条数据，Array 类型只有两个数据 5 和 6，没有第三个数据，所以 student_id_list[3] 为 NULL，这里不是为 0，请读者注意。

10.5.3 Map 类型

Map 是 Hive 自带的一个复合结构类型，与 Java 中的 Map 结构很类似，用来存放 Key-Value 数据类型。创建 Map 结构的表。

```
hive> CREATE TABLE map_test(id STRING, perf map<STRING, INT>)
    > ROW FORMAT DELIMITED
    > FIELDS TERMINATED BY '\t'
    > COLLECTION ITEMS TERMINATED BY ','
    > MAP KEYS TERMINATED BY ':';
OK
Time taken: 0.144 seconds
```

MAP KEYS TERMINATED BY 表示 Key -Value 分隔符。

导入数据的步骤如下。

1）准备一个 map_test.txt 文件，内容如下。

```
 $ cat map_test.txt
1       job:80,team:60,person:70
2       job:60,team:80
3       job:90,team:70,person:100
```

2）导入 map_test.txt 文件内容到 map_test 表里，操作命令如下。

```
hive>  LOAD DATA LOCAL INPATH '/home/work/data/ map_test.txt' INTO TABLE map_test ;
Copying data from file:/home/work/data/ map _test.txt
Copying file: file:/home/work/data/ map _test.txt
Loading data to table default. map _test
OK
Time taken: 0.198 seconds
```

3）查询结果操作。

```
hive> SELECT perf['person'] FROM map_test;
Total MapReduce jobs = 1
...
Total MapReduce CPU Time Spent: 460 msec
OK
70
NULL
100
Time taken: 20.902 seconds
hive> SELECT perf['person'] FROM map_test WHERE perf['person'] IS NOT NULL;
Total MapReduce jobs = 1
...
Total MapReduce CPU Time Spent: 610 msec
OK
70
100
Time taken: 21.989 seconds
hive>
```

10.6 Hive 的 JOIN 详解

Hive 的版本不同，对 JOIN 的支持也不一样，下面的讲解是基于 Hive-0.9.0 版本。Hive-0.9.0 支持的 JOIN 操作包括：支持 LEFT SEMI JOIN、OUTER JOIN 和 RIGHT JOIN；只支持等值条件；支持多表 JOIN。

10.6.1 JOIN 操作语法

JOIN 的操作语法如下。

```
join_table:
table_reference JOIN table_factor [join_condition]
| table_reference {LEFT|RIGHT|FULL} [OUTER]
JOIN table_reference join_condition
| table_reference LEFT SEMI JOIN table_reference join_condition
 table_reference:
table_factor
 | join_table
table_factor:
tbl_name [alias]
| table_subquery alias
| ( table_references )
join_condition:
ON equality_expression ( AND equality_expression )*
equality_expression:
expression = expression
```

10.6.2 JOIN 原理

Hive 执行引擎会将 HQL “翻译” 成为 MapReduce 任务，如果多张表使用同一列做 JOIN，将被 “翻译” 成一个 MapReduce 任务，否则将被 “翻译” 成多个 MapReduce 任务。

例如，以下将被 “翻译” 成 1 个 MapReduce 任务。

```
SELECT table1.val, table2.val, table3.val FROM table1 JOIN table2 ON
      (table1.key = table2.key1) JOIN table3 ON (table3.key = table2.key1);
```

以下将被 “翻译” 成 2 个 MapReduce 任务。

```
SELECT table1.val, table2.val, table3.val FROM table1 JOIN table2 ON
```

（table1.key = table2.key1）JOIN table3 ON (table3.key = table2.key2)；很好理解，一般来说（map side join 除外，后面会介绍），Map 过程负责分发数据，具体的 JOIN 操作在 Reduce 完成，因此，如果多表基于不同的列做 JOIN，则无法在一轮 MapReduce 任务中将所有相关数据 Shuffle 到同一个 Reduce。

对于多表 JOIN，Hive 会将前面的表缓存在 Reduce 的内存中，然后后面的表会流式的进入 Reduce 和 Reduce 内存中其他的表做 JOIN。例如：

```
SELECT table1.val, table2.val, table3.val FROM a JOIN b ON
    (table1.key = table2.key1) JOIN c ON (table3.key = table2.key1)
```

在 Reduce 中，table1、table2 表等待 JOIN 的数据会放在内存中。这会引起一些问题，如果 Reduce 个数不足或者 table1、table2 表数据过大，可能导致“Out Of Memory”。

因此，需要将数据量最大的表放到最后，或者通过“STREAMTABLE”显示指定 Reduce 流式读入的表。例如：

```
SELECT  STREAMTABLE(table1) table1.val, table2.val, table3.val FROM table1 JOIN table2 ON
      (table1.key = table2.key1) JOIN table3 ON (table3.key = table2.key1)
```

此时，table2、table3 表数据在 Reduce 将放在内存中。

10.6.3 外部 JOIN

外部 JOIN 包括 LEFT、RIGHT、FULL OUTER JOIN，其目的是针对不匹配的情况做一些控制。LEFT OUTER JOIN 操作如下。

```
SELECT table1.val, table2.val  FROM table1 LEFT OUTER JOIN table2 ON
      (table1.key=table2.key)LEFT OUTER JOIN;
```

如果 table1.key 中找不到对应的 table2.key，则输出“table1.val，NULL”。

LEFT OUTER JOIN 可以用来代替 NOT IN 操作，NOT IN 在 Hive 0.8 才支持。

例如，以下是过滤 table2.key 为 NULL 的值。

```
SELECT table1.key FROM table1 LEFT OUTER JOIN table2 ON
      table1.key=table2.key WHERE table2.key1 IS NOT NULL;
```

下面列举几个实例操作。

1）查看表 table1 的内容，命令如下。

```
hive> SELECT * FROM table1 ;
OK
key     value
1       a
2       b
3       c
Time taken: 0.155 seconds
```

2）查看表 table2 的内容，命令如下。

```
hive> SELECT * FROM table2;
OK
key     value
1       d
2       e
4       f
```

3）对表 table1 和表 table2 进行左连接，操作如下。

```
hive> SELECT table1.value, table2.value FROM table1 LEFT OUTER JOIN table2 ON
(table1.key=table2.key);
```

```
OK
value   value
a       d
b       e
c       NULL
```

4）对表 table1 和表 table2 进行右连接，操作如下。

```
hive> SELECT table1.value, table2.value FROM table1 RIGHT OUTER JOIN table2 ON
(table1.key=table2.key);
OK
value   value
a       d
b       e
NULL    f
```

5）对表 table1 和表 table2 进行全连接，操作如下。

```
hive> SELECT table1.value, table2.value FROM table1 FULL OUTER JOIN table2 ON
(table1.key=table2.key);
OK
value   value
a       d
b       e
c       NULL
NULL    f
```

10.6.4 Map 端 JOIN

假如 JOIN 两张表，其中有一张表特别小（可以放到内存中），可以使用 Map-Side JOIN。Map-Side JOIN 是在 Mapper 中做 JOIN，原理是将其中一张 JOIN 表放到每个 Mapper 任务的内存中，从而不用 Reducer 任务，在 Mapper 中就完成 JOIN。Map-Side JOIN 不适合 FULL/RIGHT OUTER JOIN。示例如下。

```
SELECT MAPJOIN(b) table1.key, table1.value FROM a join b on table1.key = table2.key;
```

10.6.5 JOIN 中处理 NULL 值的语义区别

SQL 标准中，任何对 NULL 的操作（如数值比较、字符串操作等）结果都为 NULL。Hive 对 NULL 值处理的逻辑和标准基本一致，除了 JOIN 时的特殊逻辑。

这里的特殊逻辑是指，Hive 的 JOIN 中作为 JOIN Key 的字段比较，NULL=NULL 是有意义的，且返回值为 true。例如：

```
SELECT user.uid, count(user.uid)
FROM user  JOIN class ON (class.uid = user.uid) GROUP BY user.uid;
```

查询中，user 表中 uid 为空的记录将和 class 表中 uid 为空的记录连接，即以下成立。

```
class.uid = user.uid=NULL
```

如果需要与标准一致的语义，需要改写查询手动过滤 NULL 值的情况，操作如下。

```
SELECT user.uid, count(user.uid)
FROM class JOIN user
ON (class.uid = user.uid and class.uid IS NOT NULL and user.uid IS NOT NULL)
GROUP BY user.uid;
```

提示 实践中，这一语义区别也是经常导致数据倾斜的原因之一。

10.7 Hive 优化策略

Hive 针对不同的查询进行了优化，优化可以通过配置进行控制，下面将介绍部分优化的策略以及优化控制选项。

10.7.1 列裁剪

在读数据的时候，只读取查询中需要用到的列，而忽略其他列。例如，对于查询：

```
SELECT a,b FROM T WHERE e < 10;
```

其中，T 包含 5 个列 (a, b, c, d, e)，列 c 和 d 将会被忽略，只会读取 a, b, e 列。这个选项默认为真（hive.optimize.cp = true）。

10.7.2 Map Join 操作

Join 操作在 Map 阶段完成，不再需要 Reduce，前提条件是需要的数据在 Map 过程中可以访问。比如查询：

```
INSERT OVERWRITE TABLE pv_users SELECT /*+ MAPJOIN(pv) */ pv.pageid, u.age FROM page_view pv JOIN user u ON (pv.userid = u.userid);
```

可以在 Map 阶段完成 Join，如图 10-8 所示。

page_view

pageid	userid	time
1	111	9:08:01
2	111	9:08:03
1	222	9:08:14

Hash table

key	value
111	<1, 2>
222	<2>

user

userid	age	gender
111	25	female
222	32	male

pv_users

Pageid	age
1	25
2	25
1	32

图 10-8 Map Join 图解

相关的参数如下。

- hive.join.emit.interval = 1000
- hive.mapjoin.size.key = 10000
- hive.mapjoin.cache.numrows = 10000

10.7.3 Group By 操作

Group By 操作对 Hive 的性能影响非常大，特别是处理海量数据的情况下。接下来介绍 Group By 优化的两个案例，Map 端部分聚合，以及有数据倾斜的时候进行负载均衡。

1. Map 端部分聚合

并不是所有的聚合操作都需要在 Reduce 端完成，很多聚合操作都可以先在 Map 端进行部分聚合，最后在 Reduce 端得出最终结果。注意，Map 端部分聚合基于 Hash。参数包括：

- hive.map.aggr = true：是否在 Map 端进行聚合，默认为 true。
- hive.groupby.mapaggr.checkinterval = 100000：在 Map 端进行聚合操作的条目数目。

2. 有数据倾斜的时候进行负载均衡

数据倾斜的时候进行负载均衡控制的参数为：hive.groupby.skewindata = false，默认为 false。当选项设定为 true，生成的查询计划会有两个 MR Job。

第一个 MR Job 中，Map 的输出结果集合会随机分布到 Reduce 中，每个 Reduce 做部分聚合操作，并输出结果，这样处理的结果是相同的，Group By Key 有可能被分发到不同的 Reduce 中，从而达到负载均衡的目的。第二个 MR Job 再根据预处理的数据结果按照 Group By Key 分布到 Reduce 中（这个过程可以保证相同的 Group By Key 被分布到同一个 Reduce 中），最后完成最终的聚合操作。

10.7.4 合并小文件

文件数目过多，会给 HDFS 带来压力，并且会影响处理效率，可以通过合并 Map 和 Reduce 的结果文件来消除这样的影响。

- hive.merge.mapfiles = true：是否合并 Map 输出文件，默认为 true。
- hive.merge.mapredfiles = false：是否合并 Reduce 输出文件，默认为 false。
- hive.merge.size.per.task = 256*1000*1000：合并文件的大小。

10.8 Hive 内置操作符与函数

Hive 内置许多操作符与函数，用户可以直接使用，如关系运算、逻辑运算、数值运算、条件函数、日期函数等。更多的内容可以参考附录 A。本节主要讲解几个重点的内置函数和复合类型操作：字符串函数、集合统计函数、复合类型构建。

10.8.1 字符串函数

Hive 内置的字符串函数如表 10-5 所示，基本涵盖了字符串的所有常用操作功能，如统计字符串的长度、截取字符串等操作。

表 10-5 Hive 内置字符串函数

函数名	功能描述	函数名	功能描述
length	字符串长度函数	regexp_replace	正则表达式替换函数
reverse	字符串反转函数	regexp_extract	正则表达式解析函数
concat	字符串连接函数	parse_url	URL 解析函数
concat_ws	带分隔符字符串连接函数	get_json_object	JSON 解析函数
substr,substring	字符串截取函数	space	空格字符串函数
substr,substring	字符串截取函数	repeat	重复字符串函数
upper,ucase	字符串转大写函数	ascii	首字符 ASCII 函数
lower,lcase	字符串转小写函数	lpad	左补足函数
trim	去空格函数	rpad	右补足函数
ltrim	左边去空格函数	split	分割字符串函数
rtrim	右边去空格函数	find_in_set	集合查找函数

下面举两个例子说明字符串函数的使用，更多的操作请参考附录 A。

1）字符串长度函数 length，返回字符串 str 的长度。返回值 int。语法：

```
length(string str)
```

SQL 语句：

```
hive> SELECT length(user.uid) FROM user;
OK
7
Time taken: 0.155 seconds
```

2）字符串反转函数 reverse，返回字符串 str 的反转结果。返回值 string。语法：

```
reverse(string str)
```

SQL 语句：

```
hive> SELECT reverse(user.uid) FROM user;
OK
1000000
Time taken: 0.102 seconds
```

10.8.2 集合统计函数

Hive 内置的集合统计函数如表 10-6 所示，集合统计函数是用得最多的内置函数，如统计总数、平均数、最大值、最小值等操作。

表 10-6 Hive 内置的集合统计函数

函数名	功能描述	函数名	功能描述
count	个数统计函数	min	最小值统计函数
sum	总和统计函数	max	最大值统计函数
avg	平均值统计函数		

Hive 内置的集合统计函数案例如下。

1）个数统计函数 count。返回值 int。语法：

```
count(*), count(expr), count(DISTINCT expr[, expr_.])
```

❑ count(*) 统计检索出行的个数，包括 NULL 值的行；

❑ count(expr) 返回指定字段的非空值的个数；

❑ count(DISTINCT expr[, expr_.]) 返回指定字段的不同的非空值的个数。

SQL 语句：

```
hive> SELECT count(*) FROM user;
OK
20
Time taken: 0.095 seconds
hive> SELECT count(distinct age) FROM user;
OK
8
Time taken: 0.065 seconds
```

2）总和统计函数 sum。返回值 double。语法：

```
sum(col), sum(DISTINCT col)
```

❑ sum(col) 统计结果集中 col 的相加的结果；

❑ sum(DISTINCT col) 统计结果中 col 不同值相加的结果。

SQL 语句：

```
hive> SELECT sum(age) FROM user;
OK
412
Time taken: 0.085 seconds
hive> SELECT sum(distinct age) FROM user;
OK
165
Time taken: 0.025 seconds
```

3）平均值统计函数 avg。返回值 double。语法：

```
avg(col), avg(DISTINCT col)
```

❑ avg(col) 统计结果集中 col 的平均值；

❑ avg(DISTINCT col) 统计结果中 col 不同值相加的平均值。

举例：

```
hive> SELECT avg(mark) FROM user;
OK
86
Time taken: 0.075 seconds
hive> SELECT avg(distinct mark) FROM user;
OK
78
Time taken: 0.035 seconds
```

4）最小值统计函数 min。统计结果集中 col 字段的最小值。返回值 double。语法：

```
min(col)
```

SQL 语句：

```
hive> SELECT count(mark) FROM user;
OK
64
Time taken: 0.055 seconds
```

5）最大值统计函数 max。统计结果集中 col 字段的最大值。返回值 double。语法：

```
max(col)
```

SQL 语句：

```
hive> SELECT count(mark) FROM user;
OK
98
Time taken: 0.025 seconds
```

10.8.3 复合类型操作

Hive 内置的复合类型操作如表 10-7 所示，复合类型是满足一些复杂的数据结构设计的。

表 10-7 Hive 复合类型

复合类型	功能描述
map	Map 类型构建
struct	Struct 类型构建
array	Array 类型构建

Hive 复合类型操作案例如下。

1）Map 类型构建 。根据输入的 Key-Value 对构建 Map 类型。语法：

```
map (key1, value1, key2, value2, …)
```

SQL 语句：

```
hive> CREATE TABLE  map_test AS SELECT map('100','jay','200','liu') FROM student;
hive> describe map_test;
   map_test        map
hive> SELECT map_test FROM student;
  OK
  {"100":"jay","200":"liu"}
  Time taken: 0.025 seconds
```

2）Struct 类型构建。根据输入的参数构建结构体 Struct 类型。语法：

```
struct(val1, val2, val3, …)
```

SQL 语句：

```
hive> CREATE TABLE struct_test AS SELECT struct('jay','liu','gang') FROM student;
hive> describe struct_test;
   struct_test        struct
hive> SELECT struct_test FROM student;
 OK
{"col1":"jay","col2":"liu","col3":"gang"}
Time taken: 0.035 seconds
```

3）Array 类型构建。根据输入的参数构建数组 Array 类型。语法：

```
array(val1, val2, …)
```

SQL 语句：

```
hive> CREATE TABLE array_test AS SELECT array("jay","liu","gang") FROM student;
hive> describe array_test;
   array_test       array
hive> SELECT array_test FROM array_test;
OK
["jay","liu","gang"]
Time taken: 0.025 seconds
```

10.9 Hive 用户自定义函数接口

Hive 提供了用户自定义函数的接口，在自带的内嵌函数不能满足开发需求的时候，用户自定义函数是一个不错的选择，例如时间格式的转换等。

10.9.1 用户自定义函数 UDF

Hive 是基于 Hadoop 的 MapReduce，提供 HQL 查询的数据仓库。Hive 是一个开放的系统，很多内容都支持用户定制。

- 文件格式：Text File，Sequence File；
- 内存中的数据格式：Java Integer/String 和 Hadoop IntWritable/Text；
- 用户提供的 MapReduce 脚本：不管什么语言，利用 stdin/stdout 传输数据；
- 用户自定义函数 : Substr, Trim, 1 – 1；
- 用户自定义聚合函数 : Sum, Average…n – 1；

UDF（User Defined Function，用户自定义函数）对数据进行处理。UDF 函数可以直接应用于 select 语句，对查询结构做格式化处理后，再输出内容。

编写 UDF 函数的时候需要注意以下几点。

- 自定义 UDF 需要继承 org.apache.hadoop.hive.ql.UDF。
- 需要实现 evaluate 函。

❑ evaluate 函数支持重载。

以下示例是两个数求和函数的 UDF。evaluate 函数代表两个整型数据相加，两个浮点型数据相加，可变长数据相加。如代码清单 10-1 所示。

代码清单 10-1 自定义 UDF 函数实现两个或者多个数的相加 hive\udf\ AddUDF.java

```
package hive.udf;
import org.apache.hadoop.hive.ql.exec.UDF;
public final class AddUDF extends UDF {
    public Integer evaluate(Integer a, Integer b) {
      if (null == a || null == b) {
          return null;
      }
      return a + b;
    }
    public Double evaluate(Double a, Double b) {
      if (a == null || b == null)
          return null;
      return a + b;
    }
    public Integer evaluate(Integer... a) {
      int total = 0;
      for (int i = 0; i < a.length; i++)
        if (a[i] != null)
            total += a[i];
      return total;
    }
}
```

运行步骤如下。

1）把程序打包放到目标机器上。

2）进入 Hive 客户端，添加 JAR 包（hive>add jar /run/jar/udf_test.jar）。

3）创建临时函数。

```
hive>CREATE TEMPORARY FUNCTION add_example AS 'hive.udf.AddUDF';
```

4）查询 HQL 语句。

```
hive>SELECT add_example(5, 6) FROM user;
       OK
       11
     Time taken: 0.035 seconds
hive>SELECT add_example(6, 7, 6, 3.2) FROM user;
       OK
       22.2
     Time taken: 0.045 seconds
```

5）销毁临时函数。

```
hive> DROP TEMPORARY FUNCTION add_example;
```

6）在使用 UDF 的时候，会自动进行类型转换。例如：

```
hive>SELECT add_example(3,5.1) FROM user;
    OK
    8.1
   Time taken: 0.045 seconds
```

10.9.2 用户自定义聚合函数 UDAF

Hive 查询数据时，有些聚类函数在 HQL 没有自带，需要用户自定义实现，即 UDAF（User Defined Aggregation Funcation，用户自定义聚合函数）。

首先，导入以下两个包：

- import org.apache.hadoop.hive.ql.exec.UDAF
- org.apache.hadoop.hive.ql.exec.UDAFEvaluator

函数类需要继承 UDAF 类，内部类 Evaluator 是 UDAFEvaluator 接口。Evaluator 需要实现 init、iterate、terminatePartial、merge、terminate 这几个函数。函数的功能如下。

- init 函数实现接口 UDAFEvaluator 的 init 函数。
- iterate 函数用于接收传入的参数，并进行内部的轮转，其返回类型为 boolean。
- terminatePartial 函数无参数，用于 iterate 函数轮转结束后，返回轮转数据，类似 Hadoop 的 Combiner。
- merge 函数用于接收 terminatePartial 的返回结果，进行数据 merge 操作，其返回类型为 boolean。
- terminate 函数返回最终的聚集函数结果。

看一个示例，自定义 UDAF 函数实现求平均值，如代码清单 10-2 所示。

代码清单 10-2 hive\udaf\ AvgUDAF.java

```
package hive.udaf;
import org.apache.hadoop.hive.ql.exec.UDAF;
import org.apache.hadoop.hive.ql.exec.UDAFEvaluator;
public class AvgUDAF extends UDAF {
   public static class AvgState {
     private long mCount;
     private double mSum;
   }
   public static class AvgEvaluator implements UDAFEvaluator {
     AvgState state;
     public AvgEvaluator() {
        super();
        state = new AvgState();
        init();
     }
     /**
      * init 函数类似于构造函数，用于 UDAF 的初始化
      */
```

```
    public void init() {
        state.mSum = 0;
        state.mCount = 0;
    }
    /**
     * iterate 接收传入的参数，并进行内部的轮转。其返回类型为 boolean
     *
     * @param o
     * @return
     */
     public boolean iterate(Double o) {
        if (o != null) {
            state.mSum += o;
            state.mCount++;
        }
        return true;
    }
    /**
     * terminatePartial 无参数，其为 iterate 函数轮转结束后，返回轮转数据
     * terminatePartial 类似于 hadoop 的 Combiner
     *
     * @return
     */
    public AvgState terminatePartial() {// combiner
        return state.mCount == 0 ? null : state;
    }
    /**
     * merge 接收 terminatePartial 的返回结果，进行数据 merge 操作，其返回类型为 boolean
     *
     * @param o
     * @return
     */
    public boolean merge(AvgState o) {
        if (o != null) {
            state.mCount += o.mCount;
            state.mSum += o.mSum;
        }
        return true;
    }
    /**
     * terminate 返回最终的聚集函数结果
     *
     * @return
     */
    public Double terminate() {
        return state.mCount == 0 ? null : Double.valueOf(state.mSum / state.mCount);
    }
}
}
```

运行自定义 UDAF 函数的步骤如下。

1）将 Java 文件编译成 Avg_test.jar。

2）进入 Hive 客户端添加 JAR 包。

```
hive>add jar /run/jar/Avg_test.jar;
```

3）创建临时函数。

```
hive> CREATE TEMPORARY FUNCTION avg_test 'hive.udaf. AvgUDAF ';
```

4）查询语句。

```
hive>SELECT avg_test(scores.math) FROM scores;
```

5）销毁临时函数。

```
hive>DROP TEMPORARY FUNCTION avg_test;
```

10.10 Hive 的权限控制

Hive 从 0.10 版本（包含 0.10 版本）开始可以通过元数据来控制权限，Hive-0.10 之前的版本对权限的控制主要是通过 Linux 的用户和用户组来控制，不能对 Hive 表的 CREATE、SELECT、DROP 等操作进行控制。当然，Hive 基于元数据来控制权限也不是完全安全的，目的就是防止用户不小心做了不该做的操作。

在使用 Hive 的元数据配置权限之前，必须先在 hive-site.xml 中配置两个参数。

```
<property>
   <name>hive.security.authorization.enabled</name>
   <value>true</value>
   <description>enable or disable the hive client authorization</description>
 </property>
 <property>
   <name>hive.security.authorization.createtable.owner.grants</name>
   <value>ALL</value>
   <description>the privileges automatically granted to the owner whenever a
   table gets created. An example like "select,drop" will grant select and drop
   privilege to the owner of the table</description>
 </property>
```

- hive.security.authorization.enabled 参数是开启权限验证，默认为 false。
- hive.security.authorization.createtable.owner.grants 参数是指表的创建者对表拥有所有权限，例如创建一个表 table1，该用户对表 table1 拥有 SELECT、DROP 等操作权限。还有个值是 NULL，表示表的创建者无法访问该表，这个肯定是不合理的。

Hive 授权的核心就是用户（User）、组（Group）、角色（Role），Hive 中的角色和平常认知的角色是有区别的。Hive 中的角色可以理解为一部分有一些相同“属性”的用户或组或角色的集合。这里有个递归的概念，就是一个角色可以是一些角色的集合。

举个例子，有三个用户分别属于 group_db1、group_db2、group _bothdb，表示该组用

户可以访问数据库 1、数据库 2 两个数据库。

```
用户      组
张三    group_db1
李四    group_db2
王五    group_bothdb
```

现在可以创建 role_db1 和 role_db2，分别授予访问数据库 1 和数据库 2 的权限。只要将 role_db1 赋给 group_db1（或者该组的所有用户），将 role_db2 赋给 group_db2，就可以实现指定用户访问指定数据库。最后创建 role_bothdb 指向 role_db1、role_db2（role_bothdb 不需要指定访问那个数据库），然后 role_bothdb 授予 group_bothdb，则 group_bothdb 中的用户可以访问两个数据库。

注意 用户和组使用的是 Linux 机器上的用户和组，而角色必须自己创建。如果有一个属于组 group1 的用户 jayliu，他通过 cli 连接到远程的 Server 上执行操作，而远程的 Server 上有一个用户 jayliu 属于 group2 组，则在权限控制中 jayliu 是对应 group2 组的。

10.10.1 角色的创建和删除

Hive 中的角色定义与关系型数据库中角色的定义类似，它是一种机制，给予那些没有适当权限的用户分配一定的权限。下面介绍 Hive 中角色的应用。

1）创建角色。

```
CREATE ROLE ROLE_NAME
```

例如创建一个 role_test1 角色，命令如下。

```
hive> create role role_test1;
OK
Time taken: 0.106 seconds
```

2）删除角色。

```
DROP ROLE ROLE_NAME
```

例如删除 role_test1 角色，命令如下。

```
hive> DROP ROLE role_test1;
OK
Time taken: 6.483 seconds
```

10.10.2 角色的授权和撤销

角色的授权（GRANT）就是给角色授予创建表、查询表等操作，撤销（REVOKE）反之。语法如下。

```
GRANT ROLE role_name [, role_name] ... TO principal_specification
                     [, principal_specification] ...
REVOKE ROLE role_name [, role_name] ... FROM principal_specification
                      [, principal_specification] ...
```

```
principal_specification   :
USER user   | GROUP group   | ROLE role
```

看下面的示例。

1）把 role_test1 角色授权给 jayliu 用户，命令如下。

```
hive> grant role role_test1 to user jayliu;
OK
Time taken: 0.102 seconds
```

2）查看 jayliu 用户被授权的角色，命令如下。

```
hive> SHOW ROLE GRANT user jayliu;
OK
role name:role_test1
role name:role_test1
Time taken: 7.913 seconds, Fetched: 2 row(s)
```

3）取消 jayliu 用户的 role_test1 角色，命令如下。

```
hive> revoke role role_test1 from user jayliu;
OK
Time taken: 0.189 seconds
```

Hive 支持的权限控制如表 10-8 所示。

表 10-8 Hive 权限控制

操作名称	功能描述
ALL	所有权限
ALTER	允许修改元数据，即表信息数据
UPDATE	允许修改物理数据，即实际数据
CREATE	允许进行 create 操作
DROP	允许进行 drop 操作
INDEX	允许建索引（目前还没有实现）
LOCK	当出现并发的时候允许用户进行 lock 和 unlock 操作
SELECT	允许用户进行 select 操作
SHOW_DATABASE	允许用户查看可用的数据库

看下面的实例。

1）把 select 权限授权给 jayliu 用户，命令如下。

```
hive> grant select on database default to user jayliu;
OK
Time taken: 0.188 seconds
```

2）查看 jayliu 被授予哪些操作权限，命令如下。

```
hive> show grant user jayliu on database default;
OK
database   default
principalName     jayliu
```

```
principalType     USER
privilege  Select
grantTime  Thu Jul 11 18:17:39 CST 2013
grantor    root
Time taken: 7.615 seconds, Fetched: 7 row(s)
```

3）收回 jayliu 的 select 权限，操作如下。

```
hive> revoke select on database default from user jayliu;
OK
Time taken: 0.147 seconds
```

4）查看 jayliu 用户拥有哪些权限，命令如下。

```
hive> show grant user jayliu on database default;
OK
Time taken: 0.032 seconds
```

10.10.3 超级管理员权限

Hive 本身有权限管理功能，需要通过配置开启。配置文件内容如下。

```
<property>
    <name>hive.metastore.authorization.storage.checks</name>
    <value>true</value>
</property>
<property>
    <name>hive.metastore.execute.setugi</name>
    <value>false</value>
</property>
<property>
    <name>hive.security.authorization.enabled</name>
    <value>true</value>
</property>
<property>
    <name>hive.security.authorization.createtable.owner.grants</name>
    <value>ALL</value>
</property>
```

其中，将 hive.security.authorization.createtable.owner.grants 设置成 ALL，表示用户对自己创建的表是有所有权限。

开启权限控制有 Hive 的权限功能还有一个需要完善的地方，那就是“超级管理员”。Hive 中没有超级管理员，任何用户都可以进行 Grant/Revoke 操作，为了完善“超级管理员”，必须添加 hive.semantic.analyzer.hook 配置，并实现自己的权限控制类。实现“超级管理员”的步骤如下。

1）编写权限控制类，如代码清单 10-3 所示。

代码清单 10-3 hive\udaf\ AvgUDAF.java

```
package com.autoNavi.hive;
importorg.apache.hadoop.hive.ql.parse.ASTNode;
importorg.apache.hadoop.hive.ql.parse.AbstractSemanticAnalyzerHook;
```

```
importorg.apache.hadoop.hive.ql.parse.HiveParser;
import org.apache.hadoop.hive.ql.parse.HiveSemanticAnalyzerHookContext;
import org.apache.hadoop.hive.ql.parse.SemanticException;
import org.apache.hadoop.hive.ql.session.SessionState;

public class  HiveAdmin extends Abstract SemanticAnalyzerHook {
  privatestatic String admin = "admin";
   @Override
  public ASTNodepreAnalyze(HiveSemanticAnalyzerHookContext context,
         ASTNodeast) throws SemanticException {
    switch (ast.getToken().getType()) {
      caseHiveParser.TOK_CREATEDATABASE:
      caseHiveParser.TOK_DROPDATABASE:
      caseHiveParser.TOK_CREATEROLE:
      caseHiveParser.TOK_DROPROLE:
      caseHiveParser.TOK_GRANT:
      caseHiveParser.TOK_REVOKE:
      caseHiveParser.TOK_GRANT_ROLE:
      caseHiveParser.TOK_REVOKE_ROLE:
      String userName = null;
      if (SessionState.get() != null&&
                  SessionState.get().getAuthenticator() != null) {
          userName =SessionState.get().getAuthenticator().getUserName();
      }
      if (!admin.equalsIgnoreCase(userName)) {
         thrownewSemanticException(userName
                    + " can't use ADMIN options, except " + admin + ".");
      }
      break;
      default:
      break;
      }
       returnast;
   }
}
```

2）添加控制类 HiveAdmin 到 Hive 的配置文件（hive-site.xml）中，代码如下所示。

```
<property>
    <name>hive.semantic.analyzer.hook</name>
    <value>com.autonavi.hive.HiveAdmin</value>
</property>
```

添加了配置文件以后，如果启动 Hive Server，必须关闭重启功能才能生效。

至此，只有 admin 用户可以进行 Grant/Revoke 操作。例如，jayliu 用户启动 Hive 的 Shell 命令，把 default 数据库中的所有表的查询（SELECT）授权给用户 hadoop，命令如下。

```
hive> GRANT SELECT ON DATABASE default TO USER hadoop;
FAILED: SemanticExceptionjayliu can't use ADMIN options, except admin.
```

提示 Hive 的权限控制在 Hive 的生产环境中使用非常多，在 0.10 版本之前 Hive 没有提供这个功能。读者使用这个功能的时候一定要确保 Hive 的版本在 0.10 以后。

10.11 应用实例：使用 JDBC 开发 Hive 程序

在使用 JDBC 开发 Hive 程序时，必须首先开启 Hive 的远程服务接口。使用下面命令进行开启。

```
hive --service hiveserver &
```

10.11.1 准备测试数据

准备一个 hiveJDBC.txt 文件，每行数据之间用 Tab 键隔开。文件内容如下。

```
1    hadoop
2    hive
3    Hbase
```

数据准备好了，接下来使用 Hive 的 JDBC 实现对 Hive 表的数据加载、表结构信息和数据查询等操作。

10.11.2 代码实现

Hive 的 JDBC 操作实现表的数据加载、表结构信息和数据查询，如代码清单 10-4 所示。

代码清单 10-4 hive\udaf\ HiveJdbcClient.java

```
package com.hive.jdbc;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import org.apache.log4j.Logger;
public class HiveJdbcClient {
   private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";
   private static String url = "jdbc:hive://192.168.153.100:10000/default";
   private static String user = "hive";
   private static String password = "hive";
   private static String sql = "";
   private static ResultSet res;
   private static final Logger log = Logger.getLogger(HiveJdbcClient.class);
   public static void main(String[] args) {
      try {
          Class.forName(driverName);
          Connection conn = DriverManager.getConnection(url, user, password);
          Statement stmt = conn.createStatement();
          // 创建的表名
          String tableName = "testHiveJDBC";
          /** 第一步：存在就先删除 **/
          sql = "drop table " + tableName;
          stmt.executeQuery(sql);
```

```
            /** 第二步：不存在就创建 **/
            sql = "create table " + tableName + " (key int, value string)
                        row format delimited fields terminated by '\t'";
            stmt.executeQuery(sql);
            // 执行 "showtables" 操作
            sql = "show tables '" + tableName + "'";
            System.out.println("Running:" + sql);
            res = stmt.executeQuery(sql);
            System.out.println("执行 "showtables" 运行结果：");
            if (res.next()) {
                System.out.println(res.getString(1));
            }
            // 执行 "describetable" 操作
            sql = "describe " + tableName;
            System.out.println("Running:" + sql);
            res = stmt.executeQuery(sql);
            System.out.println("执行 "describetable" 运行结果：");
            while (res.next()) {
               System.out.println(res.getString(1) + "\t" + res.getString(2));
            }

            // 执行 "load data into table" 操作
            String filepath = "/home/hadoop/ziliao/userinfo.txt";
            sql = "load data local inpath '" + filepath + "' into table " + tableName;
            System.out.println("Running:" + sql);
            res = stmt.executeQuery(sql);
            // 执行 "select * query" 操作
            sql = "select * from " + tableName;
            System.out.println("Running:" + sql);
            res = stmt.executeQuery(sql);
            System.out.println("执行 "select * query" 运行结果：");
            while (res.next()) {
               System.out.println(res.getInt(1) + "\t" + res.getString(2));
            }
            // 执行 "regular hive query" 操作
            sql = "select count(1) from " + tableName;
            System.out.println("Running:" + sql);
            res = stmt.executeQuery(sql);
            System.out.println("执行 "regular hive query" 运行结果：");
            while (res.next()) {
                System.out.println(res.getString(1));
            }
            conn.close();
            conn = null;
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            log.error(driverName + " not found!", e);
            System.exit(1);
        } catch (SQLException e) {
            e.printStackTrace();
```

```
            log.error("Connection error!", e);
            System.exit(1);
        }
    }
}
```

代码实现完成，接下来运行这个代码，操作为“Run AS->Java Aplication”，Eclipse 的控制台会出现如下结果。

```
Running:show tables 'testHiveJDBC'
执行 "show tables" 运行结果：
'testHiveJDBC'
Running:describe 'testHiveJDBC'
执行 "describe table" 运行结果：
key     int
value    string
Running:load data local inpath '/user/hadoop/hive/hiveJDBC.txt' into table
testHiveJDBC
Running:select * from 'testHiveJDBC'
执行 "select * query" 运行结果：
    hadoop
    hive
    hbase
Running:select count(1) from testHiveJDBC
执行 "regular hive query" 运行结果：
```

Hive 的 JDBC 接口在实际开发中经常用到，和 MySQL 的 JDBC 接口非常类似。给开发人员提供了远程操作 Hive 的接口。

10.12 本章小结

本章对 Hive 的知识点和使用做详细介绍，Hive 是基于 Hadoop 的一个数据仓库，它支持类似 SQL 语句来替代 MapReduce，这使不会写代码的数据分析师也能利用 Hadoop 平台对海量数据进行分析，如 DBA。Hive 工具现在已经非常成熟，用的人也越来越多。

第11章　开源数据库 HBase

HBase 是一个分布式的、面向列的开源数据库，该技术来源于 Google 论文“Bigtable：一个结构化数据的分布式存储系统”。就像 Bigtable 利用 Google 文件系统（File System）所提供的分布式数据存储一样，HBase 在 Hadoop 之上提供了类似 Bigtable 的能力。HBase 是 Apache 的 Hadoop 项目的子项目。HBase 不同于一般的非关系数据库，它是一个适合于非结构化数据存储的数据库。另一个不同点是，Hbase 是基于列的而不是基于行的模式。

11.1　认识 HBase

HBase 是一个高可靠、高性能、面向列、可伸缩的分布式存储系统，利用 HBase 技术可在廉价 PC Server 上搭建大规模结构化存储集群。

HBase 是 Google Bigtable 的开源实现，与 Google Bigtable 利用 GFS 作为其文件存储系统类似，HBase 利用 Hadoop HDFS 作为其文件存储系统；Google 运行 MapReduce 来处理 Bigtable 中的海量数据，HBase 同样利用 Hadoop MapReduce 来处理 HBase 中的海量数据；Google Bigtable 利用 Chubby 作为协同服务，HBase 利用 ZooKeeper 作为对应。

11.1.1　HBase 的特点

HBase 中的表一般有以下特点。

- 大：一个表可以有上亿行，上百万列。
- 面向列：面向列（族）的存储和权限控制，列（族）独立检索。
- 稀疏：对于为空（NULL）的列，并不占用存储空间，因此，表可以设计的非常稀疏。

此外，Pig 和 Hive 还为 HBase 提供了高层语言支持，在 HBase 上进行数据统计处理变得非常简单。Sqoop 为 HBase 提供了 RDBMS 数据导入功能，使得传统数据库数据向 HBase 中迁移变得非常方便。

11.1.2　HBase 访问接口

HBase 支持很多种访问，访问 HBase 的常见接口如下。

- Native Java API：最常规和高效的访问方式，适合 Hadoop MapReduce Job 并行批处理 HBase 表数据。
- HBase Shell：HBase 的命令行工具，最简单的接口，适合 HBase 管理使用。
- Thrift Gateway：利用 Thrift 序列化技术，支持 C++、PHP、Python 等多种语言，适合其他异构系统在线访问 HBase 表数据。

- REST Gateway：支持 REST 风格的 HTTP API 访问 HBase，解除了语言限制。
- Pig：可以使用 Pig Latin 流式编程语言来操作 HBase 中的数据，和 Hive 类似，本质上最终也是编译成 MapReduce Job 来处理 HBase 表数据，适合做数据统计。
- Hive：Hive 0.7.0 的 Release 版本之前尚没有加入对 HBase 的支持，Hive 0.8.0 以后支持 HBase，可以使用类似 SQL 语言来访问 HBase。

11.1.3 HBase 存储结构

HBase 的存储结构基于 Hadoop 的 HDFS 存储，所以 HBase 的存储具备 HDFS 的特性，如图 11-1 所示。

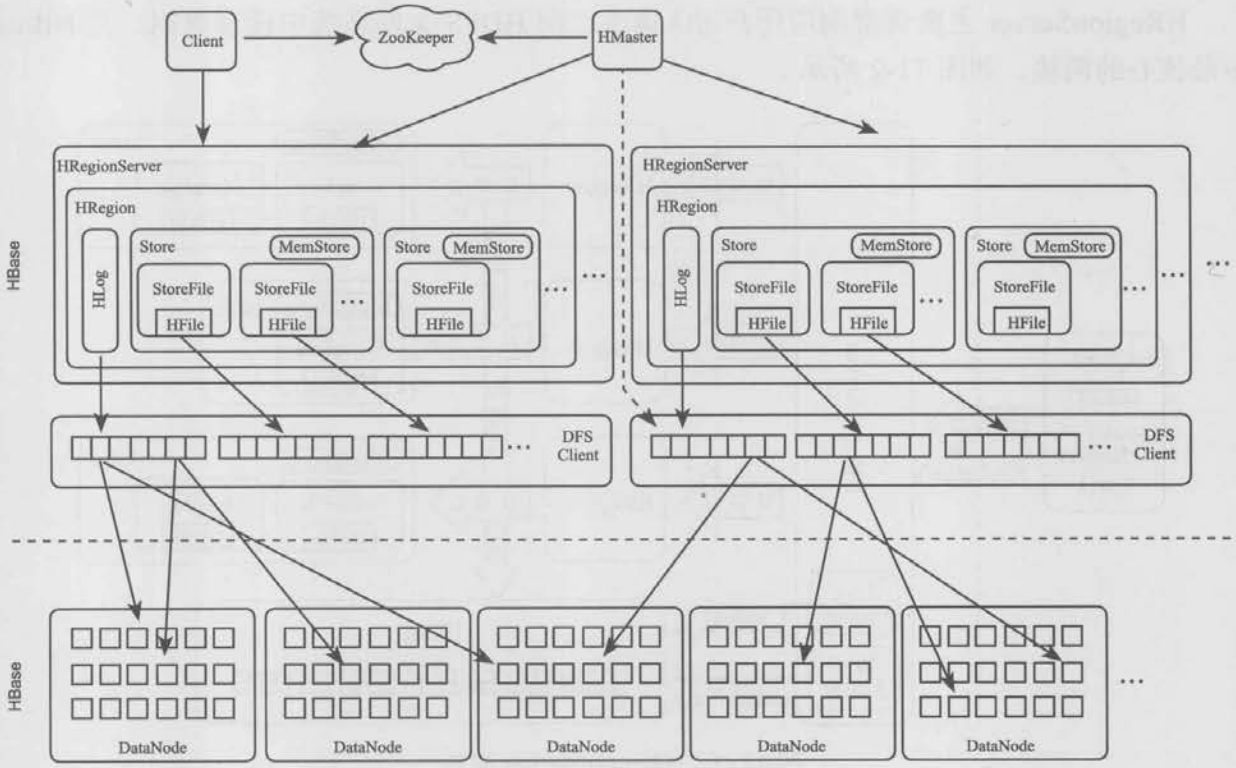


图 11-1 HBase 架构

1. Client

HBase Client 使用 HBase 的 RPC 机制与 HMaster 和 HRegionServer 进行通信，对于管理类操作，Client 与 HMaster 进行 RPC；对于数据读写类操作，Client 与 HRegionServer 进行 RPC。

2. ZooKeeper

在 ZooKeeper Quorum 中，除了存储 -ROOT- 表的地址和 HMaster 的地址外，HRegionServer 也会以 Ephemeral 方式把自己注册到 ZooKeeper 中，使得 HMaster 可以随时感知到各个 HRegionServer 的健康状态。此外，ZooKeeper 也避免了 HMaster 的单点问题。

3. HMaster

HMaster 没有单点问题，HBase 中可以启动多个 HMaster，通过 ZooKeeper 的 Master Election 机制保证总有一个 Master 运行。HMaster 主要负责 Table 和 Region 的管理工作，具体包括以下功能。

- 管理用户对 Table 的增、删、改、查操作；
- 管理 HRegionServer 的负载均衡，调整 Region 分布；
- 在 Region Split 后，负责新 Region 的分配；
- 在 HRegionServer 停机后，负责失效 HRegionServer 上的 Region 迁移。

4. HRegionServer

HRegionServer 主要负责响应用户 I/O 请求，向 HDFS 文件系统中读写数据，是 HBase 中最核心的模块，如图 11-2 所示。

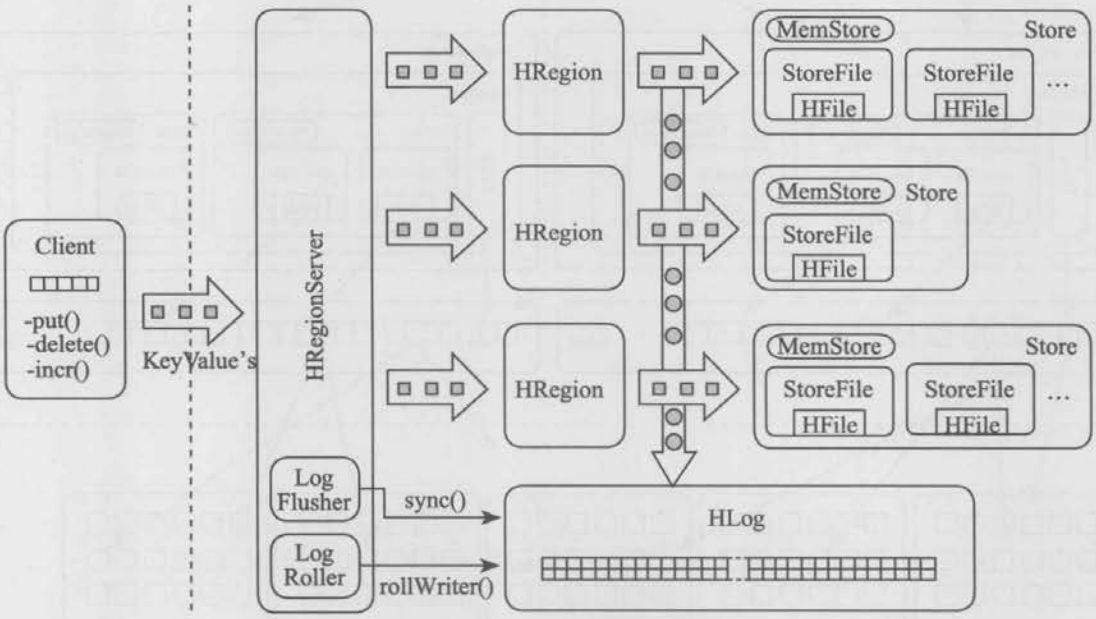


图 11-2 HRegionServer 原理

HRegionServer 内部管理了一系列 HRegion 对象，每个 HRegion 对应 Table 中的一个 Region，HRegion 由多个 HStore 组成。每个 HStore 对应 Table 中的一个 Column Family 的存储，可以看出每个 Column Family 其实就是一个集中的存储单元，因此最好将具备共同 I/O 特性的 Column 放在一个 Column Family 中，这样最高效。

HStore 存储是 HBase 存储的核心，其中由两部分组成，一部分是 MemStore，一部分是 StoreFile。MemStore 是 Sorted Memory Buffer，用户写入的数据首先会放入 MemStore，当 MemStore 满了以后会执行 Flush 操作变成一个 StoreFile（底层实现是 HFile），当 StoreFile 文件数量增长到一定阈值时，会触发 Compact 合并操作，将多个 StoreFile 合并成一个 StoreFile，合并过程中会进行版本合并和数据删除。

因此可以看出，HBase 其实只有增加数据，所有的更新和删除操作都是在后续的 Compact 过程中进行的，这使得用户的写操作只要进入内存中就可以立即返回，保证了 HBase I/O 的高性能。StoreFile 在完成 Compact 合并操作后，会逐步形成越来越大的 StoreFile，当单个 StoreFile 大小超过一定阈值后，会触发 Split 操作，同时把当前 Region 分裂成 2 个 Region，父 Region 会下线，新分裂出的 2 个孩子 Region 会被 HMaster 分配到相应的 HRegionServer 上，使得原先 1 个 Region 的压力得以分流到 2 个 Region 上。图 11-3 描述了 Compaction 和 Split 的过程。

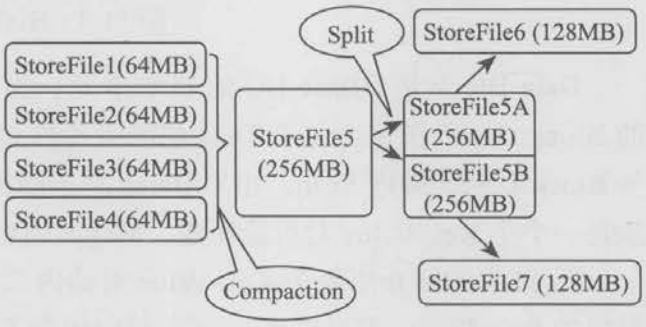


图 11-3 Compaction 和 Split 操作过程

理解上述 HStore 基本原理后，还必须了解 HLog 的功能，因为上述的 HStore 在系统正常工作的前提下是没有问题的，但是在分布式系统环境中，无法避免系统出错或者宕机，因此一旦 HRegionServer 意外退出，MemStore 中的内存数据将会丢失，这就需要引入 HLog 了。每个 HRegionServer 中都有一个 HLog 对象，HLog 是一个实现 Write Ahead Log 的类，在每次用户操作写入 MemStore 的同时，也会写一份数据到 HLog 文件中（HLog 文件格式见后续章节），HLog 文件定期会滚动出新的，并删除旧的文件（已持久化到 StoreFile 中的数据）。当 HRegionServer 意外终止后，HMaster 会通过 ZooKeeper 感知到，HMaster 首先会处理遗留的 HLog 文件，将其中不同 Region 的 Log 数据进行拆分，分别放到相应 Region 的目录下，然后再将失效的 Region 重新分配，领取这些 Region 的 HRegionServer 在 Load Region 的过程中，会发现有历史 HLog 需要处理，因此会重新加载 HLog 中的数据到 MemStore 中，然后保存到 StoreFile，完成数据恢复。

11.1.4 HBase 存储格式

HBase 中的所有数据文件都存储在 Hadoop HDFS 文件系统上，主要包括两种文件类型。

- HFile：HBase 中 Key-Value 数据的存储格式，HFile 是 Hadoop 的二进制格式文件，实际上 StoreFile 就是对 HFile 做了轻量级包装，即 StoreFile 底层就是 HFile。
- HLog File：HBase 中 WAL（Write Ahead Log）的存储格式，物理上是 Hadoop 的 Sequence File。

1. HFile 存储格式

HFile 文件是不定长的，长度固定的只有其中的两块 Trailer 和 File Info。如图 11-4 所示，Trailer 中有指针指向其他数据块的起始点。File Info 中记录了文件的一些 Meta 信息，如 AVG_KEY_LEN、AVG_VALUE_LEN、LAST_KEY、COMPARATOR、MAX_SEQ_ID_KEY 等。Data Index 和 Meta Index 块记录了每个 Data 块和 Meta 块的起始点。

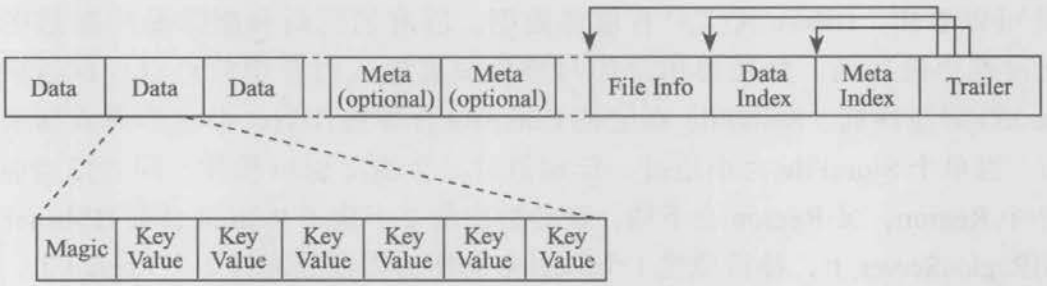


图 11-4 HFile 存储格式

Data Block 是 HBase I/O 的基本单元。为了提高效率，HRegionServer 中有基于 LRU 的 Block Cache 机制。每个 Data 块的大小可以在创建一个 Table 的时候通过参数指定，大号 Block 有利于顺序 Scan，小号 Block 利于随机查询。每个 Data 块除了开头的 Magic 以外，就是一个个 Key-Value 对拼接而成，Magic 内容就是一些随机数字，目的是防止数据损坏。

图 11-5 详细介绍每个 Key-Value 对的内部构造。HFile 中的每个 Key-Value 对就是一个简单的 byte 数组。但是这个 byte 数组中包含了很多项，并且有固定的结构。

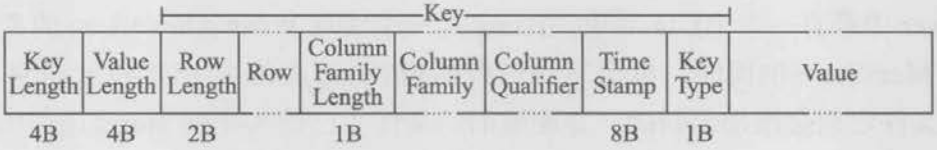


图 11-5 Key-Value 对的内部构造

开始是两个固定长度的数值，分别表示 Key 的长度和 Value 的长度。紧接着是 Key，开始是固定长度的数值，表示 RowKey 的长度，紧接着是 RowKey，然后是固定长度的数值，表示 Family 的长度，后面是 Family，接着是 Qualifier，再后面是两个固定长度的数值，表示 Time Stamp 和 Key Type（Put/Delete）。Value 部分没有这么复杂的结构，就是纯粹的二进制数据。

HFile 分为六个部分。

- Data Block：保存表中的数据，这部分可以被压缩。
- Meta Block：保存用户自定义的 Key-Value 对，可以被压缩。
- File Info：HFile 的元信息，不被压缩，用户也可以在这一部分添加自己的元信息。
- Data Block Index：Data Block 的索引。每条索引的 Key 是被索引 Block 的第一条记录的 Key。
- Meta Block Index：Meta Block 的索引。
- Trailer：这一段是定长的。保存了每一段的偏移量，读取一个 HFile 时，会先读取 Trailer，Trailer 保存了每个段的起始位置（段的 Magic Number 用来做安全 check），然后，Data Block Index 会被读取到内存中。这样，当检索某个 Key 时，不需要扫描整个 HFile，而只需从内存中找到 Key 所在的 Block，通过一次磁盘 I/O 将整个

Block 读取到内存中，再找到需要的 Key。Data Block Index 采用 LRU 机制淘汰。

HFile 的 Data Block、Meta Block 通常采用压缩方式存储，压缩之后可以大大减少网络 I/O 和磁盘 I/O，随之而来的开销当然是需要花费 CPU 进行压缩和解压缩。HFile 的压缩支持两种方式：Gzip 和 LZO。

2. HLogFile 存储格式

HLogFile 存储格式如图 11-6 所示。

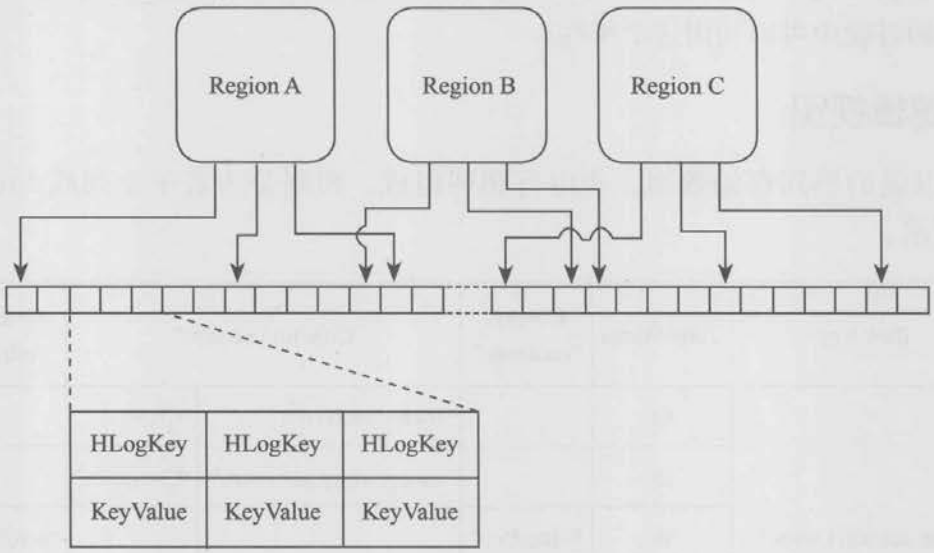


图 11-6 HLogFile 存储

WAL（Write Ahead Log）类似 MySQL 中的 bin log，用来做灾难恢复，HLogFile 记录数据的所有变更，一旦数据修改，就可以从 log 中进行恢复。

每个 Region Server 维护一个 HLog，这样不同 Region（来自不同 Table）的日志会混在一起，这样做的目的是不断追加单个文件，相对于同时写多个文件而言，可以减少磁盘寻址次数，可以提高对 Table 写的能力。带来的麻烦是，如果一台 Region Server 下线，为了恢复其上的 Region，需要将 Region Server 上的 log 进行拆分，然后分发到其他 Region Server 上进行恢复。

HLogFile 就是一个普通的 Hadoop Sequence File。Sequence File 的 Key 是 HLogKey 对象，HLogKey 中记录了写入数据的归属信息，除了 Table 和 Region 名字外，同时还包括 sequence number 和 timestamp，timestamp 是写入时间，sequence number 的起始值为 0，或者是最近一次存入文件系统中的 sequence number。HLog Sequece File 的 Value 是 HBase 的 KeyValue 对象，即对应 HFile 中的 KeyValue。

11.2 HBase 设计

HBase 中的每一张表就是所谓的 BigTable。BigTable 会存储一系列的行记录，行记录

有三个基本类型的定义：Row Key、Time Stamp、Column。

- Row Key 是行在 BigTable 中的唯一标识。
- Time Stamp 是每次数据操作对应关联的时间戳，可以看做 SVN 的版本。
- Column 定义为 <family>:<label>，通过这两部分可以指定唯一的数据的存储列，family 的定义和修改需要对 HBase 进行类似于 DB 的 DDL 操作，而 label，不需要定义直接可以使用，这也为动态定制列提供了一种手段。family 另一个作用体现在物理存储优化读写操作上，同 family 的数据物理上保存的会比较临近，因此在业务设计的过程中可以利用这个特性。

11.2.1 逻辑视图

HBase 以表的形式存储数据。表由行和列组成。列划分为若干个列族（row family），如图 11-7 所示。

Row Key	Time Stamp	Column "contents:"	Column "anchor:"		Column "mime:"
"com.autonavi.www"	t9		"anchor:auto.com"	"CNN"	
	t8		"anchor:amap.auto.com"	"CNN.com"	
	t6	"<html>..."			"text/html"
	t5	"<html>..."			
	t3	"<html>..."			

图 11-7 HBase 数据存储

1. Row Key

与 NoSQL 数据库一样，Row Key 是用来检索记录的主键。访问 HBase table 中的行，只有三种方式：通过单个 Row Key 访问；通过 Row Key 的 range 全表扫描；Row Key 可以是任意字符串（最大长度是 64KB，实际应用中长度一般为 10 ～ 100bytes），在 HBase 内部，Row Key 保存为字节数组。

在存储时，数据按照 Row Key 的字典序（byte order）排序存储。设计 Key 时，要充分排序存储这个特性，将经常一起读取的行存储放到一起（位置相关性）。

注意 字典序对 int 排序的结果是 1,10,100,11,12,13,14,15,16,17,18,19,2,20,21,…,9,91,92,93,94,95,96,97,98,99。要保持整形的自然序，Row Key 必须用 0 进行左填充。

行的一次读写是原子操作（不论一次读写多少列）。这个设计决策能够使用户很容易理解程序在对同一个行进行并发更新操作时的行为。

2. 列族

HBase 表中的每个列都归属于某个列族。列族是表的 Schema 的一部分（而列不是），必须在使用表之前定义。列名都以列族作为前缀，例如 courses:history、courses:math 都属于 courses 这个列族。

访问控制、磁盘和内存的使用统计都是在列族层面进行的。在实际应用中，列族上的控制权限能帮助我们管理不同类型的应用，例如，允许一些应用可以添加新的基本数据、一些应用可以读取基本数据并创建继承的列族、一些应用则只允许浏览数据（甚至可能因为隐私的原因不能浏览所有数据）。

3. 时间戳

HBase 中通过 Row 和 Columns 确定的一个存贮单元称为 Cell。每个 Cell 都保存着同一份数据的多个版本。版本通过时间戳来索引，时间戳的类型是 64 位整型。时间戳可以由 HBase（在数据写入时自动）赋值，此时时间戳是精确到毫秒的当前系统时间。时间戳也可以由客户显式赋值。如果应用程序要避免数据版本冲突，就必须自己生成具有唯一性的时间戳。每个 Cell 中，不同版本的数据按照时间倒序排序，即最新的数据排在最前面。

为了避免数据存在过多版本造成的管理（包括存贮和索引）负担，HBase 提供了两种数据版本回收方式。一是保存数据的最后 n 个版本，二是保存最近一段时间内的版本（比如最近七天）。用户可以针对每个列族进行设置。

4. Cell

Cell 是由 {row key, column(=<family> + <label>), version} 唯一确定的单元。Cell 中的数据是没有类型的，全部是字节码形式存贮。

11.2.2 框架结构及流程

HBase 的核心组件包括 HBase Master Server、HRegion Server、HRegion、HMemcache、HLog、HStore，它们之间的关系如图 11-8 所示。

HBase 依托 Hadoop 的 HDFS 作为存储基础，因此结构也类似 Hadoop 的 Master-Slave 模式，HBase Master Server 负责管理所有的 HRegion Server，但 HBase Master Server 本身并不存储 HBase 中的任何数据。HBase 逻辑上的 Table 被定义为一个 Region 存储在某一台 HRegion Server 上，HRegion Server 与 Region 的对应关系是一对多的关系。每一个 HRegion 在物理上被分为三个部分：HMemcache、HLog、HStore，分别代表了缓存、日志、持久层。图 11-9 描述了一次更新流程的步骤。

由图 11-9 可以看出，提交更新操作将会写入两部分实体 HMemcache 和 HLog 中：HMemcache 就是为了提高效率而在内存中建立的缓存，保证部分最近操作过的数据能够快速地被读取和修改；HLog 作为同步 HMemcache 和 HStore 的事务日志，在 HRegion Server 周期性地发起 Flush Cache 命令的时候，就会将 HMemcache 中的数据持久化到 HStore 中，同时会清空 HMemcache 中的数据。这里采用的是比较简单的策略来做数据缓存和同步，更复杂的策略是参照 Java 的垃圾收集机制来操作。

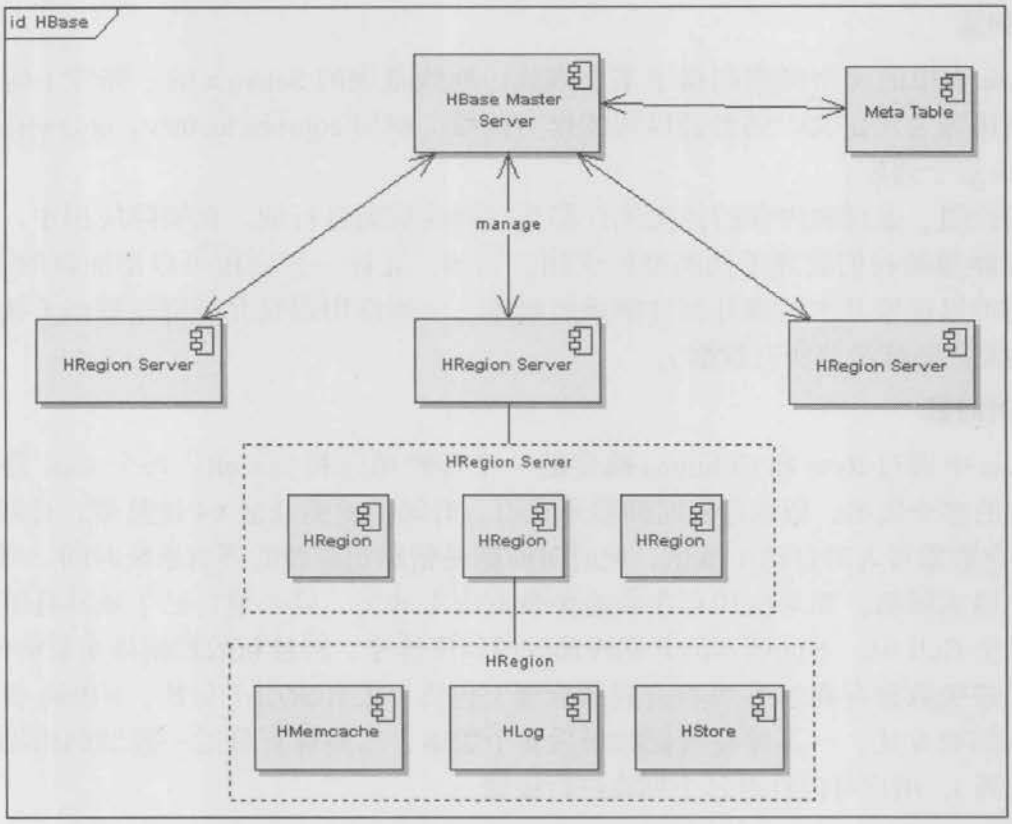


图 11-8 框架结构图

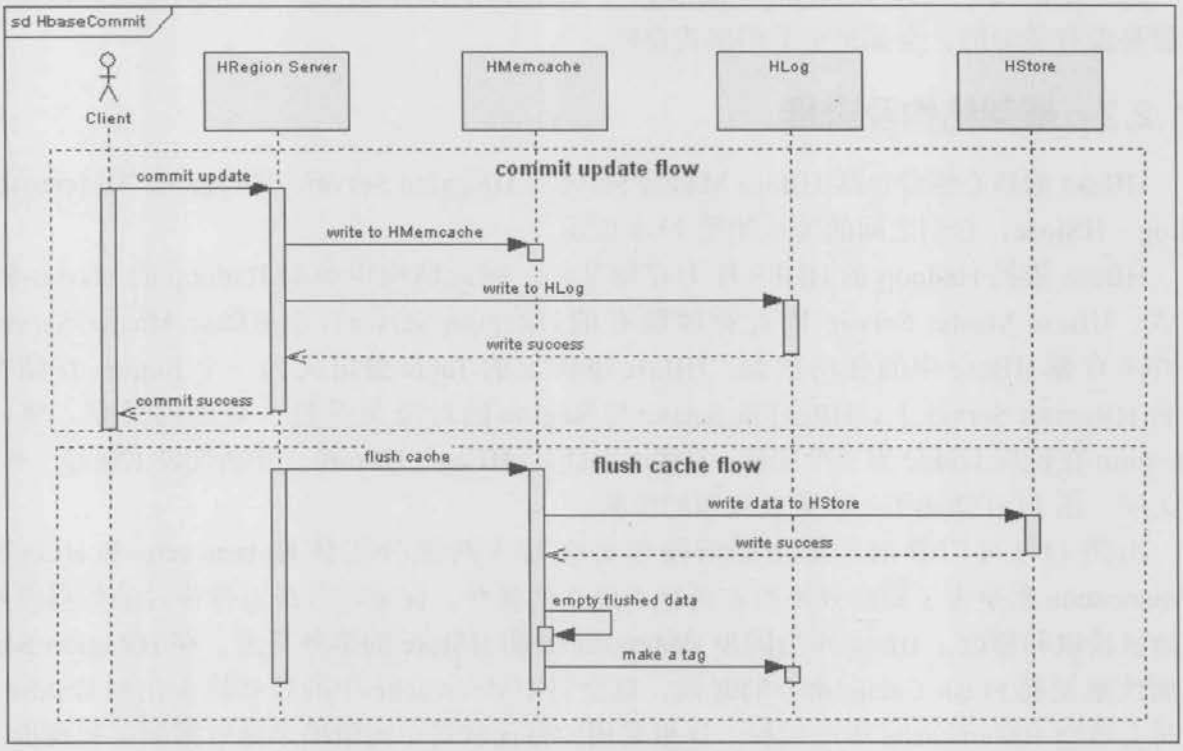


图 11-9 提交更新以及刷新 Cache 流程

在读取 Region 信息的时候，优先读取 HMemcache 中的内容，如果未读取到，再去读取 HStore 中的数据。注意几个细节：

- 由于每一次执行 Flash Cache 都会产生一个 HStore File，在 HStore 中存储的文件会越来越多，对性能也会产生一定影响，因此达到设置文件数量阀值的时候就会合并（Merge）这些文件为一个大文件。
- 设置 Cache 大小以及 Flush 的时间间隔，需要考虑内存消耗以及对性能的影响。
- HRegion Server 每次重新启动的时候，会将 HLog 中没有被 Flush 到 HStore 中的数据再次载入 HMemcache，因此 HMemcache 过大对于启动的速度也有直接影响。
- HStore File 中存储数据采用 B-Tree 算法，因此也支持前面提到的对于 Column 和 Family 数据操作的快速定位获取；
- HRegion 可以进行 Merge 操作也可以进行 Split 操作，根据 HRegion 的大小决定。不过操作的时候，HRegion 都会被锁定，不可使用；
- HBase Master Server 通过 Meta-info Table 来获取 HRegion Server 的信息以及 Region 的信息，Meta 最顶部的 Region 是虚拟的，叫做 Root Region，通过 Root Region 可以找到下面各个实际的 Region；
- 客户端通过 HBase Master Server 获得 Region 所在的 Region Server，然后就直接和 Region Server 进行交互，而对于 Region Server 相互之间不通信，只和 HBase Master Server 交互，受到 Master Server 的监控和管理。

11.2.3 Table 和 Region 的关系

Table 随着记录数不断增加而变大后，会逐渐分裂成多份 Split，每一个 Split 成为 Region，一个 Region 由 [startkey, endkey) 表示，不同的 Region 会被 Master 分配给相应的 Region Server 进行管理，如图 11-10 所示。

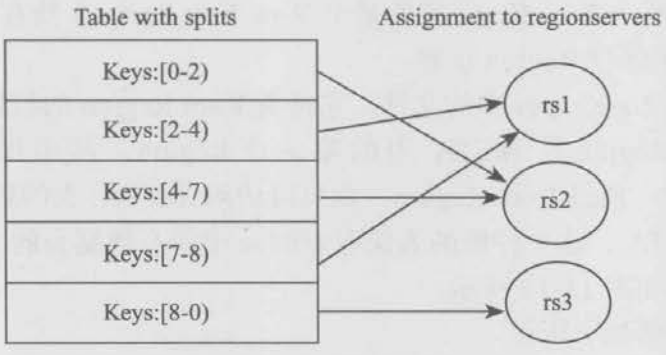


图 11-10 Split 存储

11.2.4 -ROOT- 表和 .META. 表

HBase 中有两张特殊的表，-ROOT- 和 .META.，它们的功能如下。

- .META. 记录用户表的 Region 信息，可以有多个 Region。
- -ROOT- 记录 .META. 表的 Region 信息，只有一个 Region。

ZooKeeper 中记录 -ROOT- 表的 Location，.META. 和 -ROOT- 的关系如图 11-11 所示。

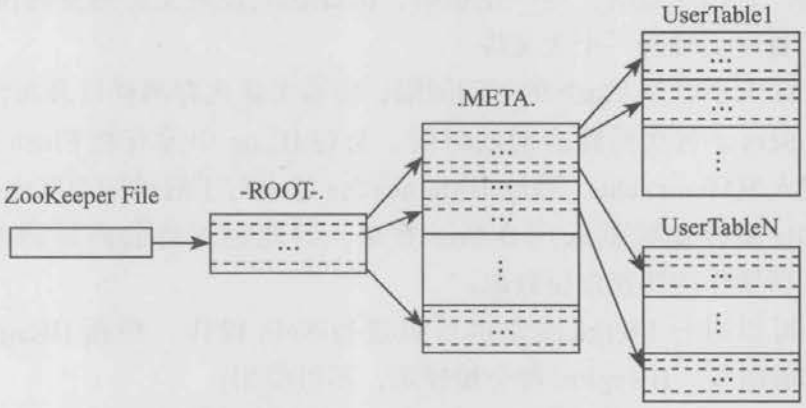


图 11-11 .META. 和 -ROOT- 的关系

Client 访问用户数据之前需要先访问 ZooKeeper，然后访问 -ROOT- 表，接着访问 .META. 表，最后才能找到用户数据的位置去访问，中间需要多次网络操作，不过 Client 端会执行 Cache 缓存。

11.3 关键算法和流程

HBase 中包含了很多优秀的算法，如 Region 定位、Region 分配、Region Server 上线和下线、Master 上线和下线，接下来介绍这些算法的实现。

11.3.1 Region 定位

HBase 如何找到某个 Row Key（或者某个 Row Key Range）所在的 Region 呢？使用三层类似 B+ 树的结构来保存 Region 位置。

- 第一层是保存 ZooKeeper 中的文件，它持有 Root Region 的位置。
- 第二层 Root Region 是 .META. 表的第一个 Region，其中保存了 .META. 表其他 Region 的位置。通过 Root Region，就可以访问 .META. 表的数据。
- 第三层是 .META.，这个特殊的表保存 HBase 中所有数据表的 Region 位置信息。

Region 定位过程如图 11-12 所示。

Region 定位的策略如下所示。

1）Root Region 永远不会被 Split，保证最多需要三次跳转，就能定位到任意 Region 。

2）.META. 表每行保存一个 Region 的位置信息，Row Key 采用“表名 + 表的最后一行”编码而成。假设 .META. 表的一行在内存中大约占用 1KB，并且每个 Region 限制为 128MB，上面的三层结构可以存储的 Region 数目为：

（128MB/1KB）×（128MB/1KB）= 2^{34} 个 Region。

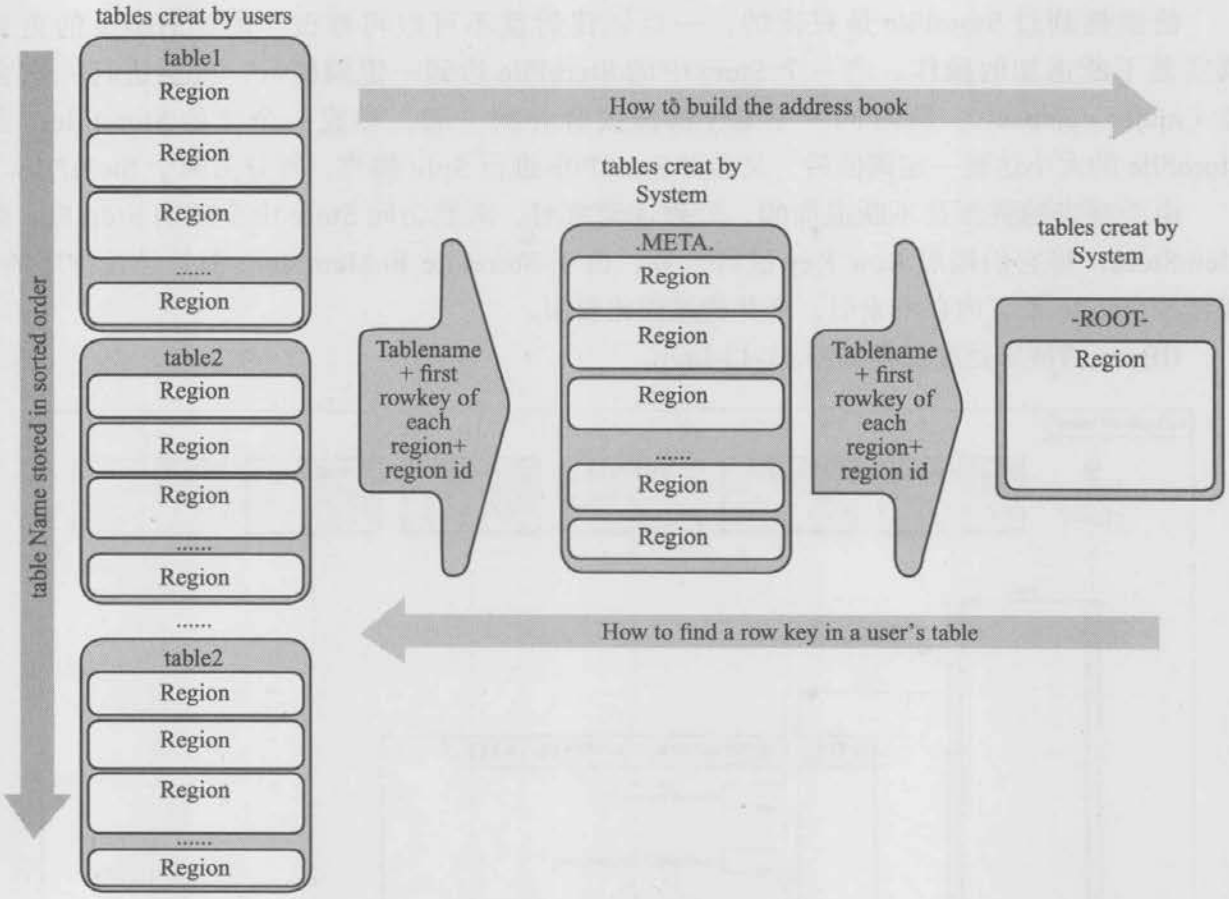


图 11-12 Region 定位

3）为了加快访问，.META. 表的全部 Region 都保存在内存中。

假设 .META. 表的一行在内存中大约占用 1KB，并且每个 Region 限制为 128MB。上面的三层结构可以保存的 Region 数目为（128MB/1KB）×（128MB/1KB）= = 2^{34}。

Client 会将查询过的位置信息保存缓存起来，缓存不会主动失效，因此如果 Client 上的缓存全部失效，则需要进行 6 次网络来回，才能定位到正确的 Region（其中三次用来发现缓存失效，另外三次用来获取位置信息）。

11.3.2 读写过程

HBase 使用 MemStore 和 StoreFile 存储对表的更新。数据在更新时首先写入 Log（WALLog）和内存（MemStore）中，MemStore 中的数据是排序的，当 MemStore 累计到一定阈值时，就会创建一个新的 MemStore，并且将老的 MemStore 添加到保存队列，由单独的线程保存到磁盘上，成为一个 StoreFile。同时，系统会在 ZooKeeper 中记录一个 redo point，表示这个时刻之前的变更已经持久化了。

系统出现意外时，可能导致内存（MemStore）中的数据丢失，此时使用 Log（WALLog）来恢复 check point 之后的数据。

前面提到过 StoreFile 是只读的，一旦创建后就不可以再修改。因此 HBase 的更新其实是不断追加的操作。当一个 Store 中的 StoreFile 达到一定阈值后，就会进行一次合并（major compact），将对同一个 Key 的修改合并到一起，形成一个大的 StoreFile，当 StoreFile 的大小达到一定阈值后，又会对 StoreFile 进行 Split 操作，等分为两个 StoreFile。

由于对表的更新是不断追加的，处理读请求时，需要访问 Store 中全部的 StoreFile 和 MemStore，将它们按照 Row Key 进行合并，由于 StoreFile 和 MemStore 都是经过排序的，并且 StoreFile 带有内存中索引，合并的过程比较快。

HBase 的请求处理过程如图 11-13 所示。

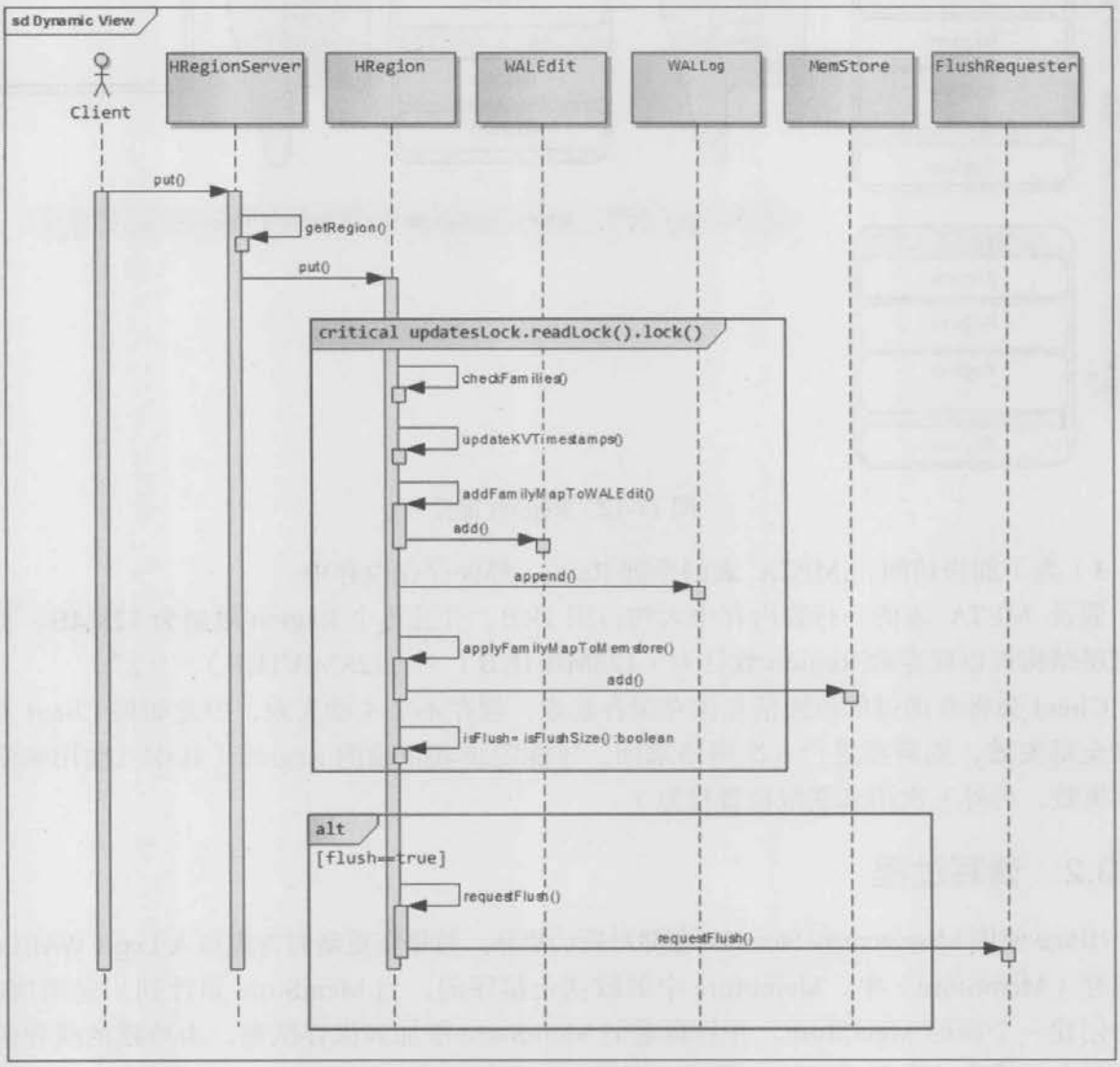


图 11-13 HBase 的请求处理流程

请求处理流程解释如下。

1）Client 向 Region Server 提交写请求；

2）Region Server 找到目标 Region；
3）Region 检查数据是否与 Schema 一致；
4）如果客户端没有指定版本，则获取当前系统时间作为数据版本；
5）将更新写入 WALLog；
6）将更新写入 MemStore；
7）判断 MemStore 是否需要保存为 Store 文件。

11.3.3 Region 分配

任何时刻，一个 Region 只能分配给一个 Region Server。Master 记录了当前有哪些可用的 Region Server，当前哪些 Region 分配给哪些 Region Server，哪些 Region 还没有分配。当存在未分配的 Region，并且一个 Region Server 上有可用空间时，Master 就向这个 Region Server 发送一个装载请求，把 Region 分配给这个 Region Server。Region Server 得到请求后，就开始对此 Region 提供服务。

11.3.4 Region Server 上线和下线

Master 使用 ZooKeeper 跟踪 Region Server 状态。当某个 Region Server 启动时，会首先在 ZooKeeper 上的 Server 目录下建立代表自己的文件，并获得该文件的独占锁。由于 Master 订阅了 Server 目录上的变更消息，当 Server 目录下的文件出现新增或删除操作时，Master 可以得到来自 ZooKeeper 的实时通知。因此一旦 Region Server 上线，Master 能马上得到消息。

当 Region Server 下线时，它和 ZooKeeper 的会话断开，ZooKeeper 自动释放代表这台 Server 的文件上的独占锁。而 Master 不断轮询 Server 目录下文件的锁状态。如果 Master 发现某个 Region Server 丢失自己的独占锁，或者 Master 连续几次和 Region Server 通信都无法成功，Master 就尝试去获取代表这个 Region Server 的读写锁。一旦获取成功，就可以确定 Region Server 和 ZooKeeper 之间的网络断开或 Region Server 挂了。无论哪种情况发生，Region Server 都无法继续为它的 Region 提供服务，此时 Master 会删除 Server 目录下代表这台 Region Server 的文件，并将这台 Region Server 的 Region 分配给其他还活着的节点。

如果网络短暂出现问题导致 Region Server 丢失它的锁，Region Server 重新连接到 ZooKeeper 之后，只要代表它的文件还在，它就会不断尝试获取这个文件上的锁，一旦获取到，就可以继续提供服务。

11.3.5 Master 上线和下线

Master 启动进行以下步骤。
1）从 ZooKeeper 上获取唯一代表 Master 的锁，用来阻止 Standby Master 成为 Master。
2）扫描 ZooKeeper 上的 Server 目录，获得当前可用的 Region Server 列表。

3）与步骤 2）中获取的所有 Region Server 进行通信，获得当前已分配的 Region 和 Region Server 的对应关系。

4）扫描 .META. 表中 Region 的集合，计算得到当前还未分配的 Region，将它们放入待分配 Region 列表。

由于 Master 只维护表和 Region 的元数据，而不参与表数据 I/O 的过程，Master 下线仅导致所有元数据的修改被冻结（无法创建删除表、修改表的 Schema、进行 Region 负载均衡、处理 Region 上下线、进行 Region 合并，唯一例外的是 Region 的 Split 可以正常进行，因为只有 Region Server 参与），表的数据读写还可以正常进行。因此 Master 下线短时间内对整个 HBase 集群没有影响。

从上线过程可以看到，Master 保存的信息全是冗余信息（都可以从系统其他地方收集到或计算出来），因此，一般 HBase 集群中总是有一个 Master 在提供服务，还有一个以上的 Master 在等待时机抢占它的位置。

11.4 HBase 安装

安装 HBase 有两种方式，单机安装和分布式安装。单机安装在一个节点上配置即可，分布式安装要用到 ZooKeeper 工具，hbase-0.92.2 自带 ZooKeeper，默认是被注释掉的。也可以自己下载 ZooKeeper 进行配置，本章使用 HBase 默认的 ZooKeeper 来搭建 HBase 的分布式环境。

11.4.1 HBase 单机安装

HBase 需要在 Hadoop 环境运行，因此安装 HBase 的前提是必须安装 Hadoop 环境。Hadoop 环境的安装可以参考第 2 章的内容。从 Apache 官网下载 hbase-0.92.2.tar.gz 软件包。

HBase 的安装步骤如下所示。

步骤 1 解压 hbase-0.92.2.tar.gz 到指定的目录下（这里是在 /opt），代码如下。

```
jayliu@Ubuntu1:/opt$ tar -zxvf hbase-0.92.2.tar.gz
```

执行上面的操作会在 /opt 下发现一个 hbase-0.92.2 文件夹。

步骤 2 配置 HBase 的环境变量到 /etc/profile 文件中，代码如下。

```
    jayliu@Ubuntu1:/opt$ sudo vi /etc/profile
  export HBASE_HOME=/opt/hbase-0.92.2
export  PATH=$HADOOP_HOME/bin:$HIVE_HOME/bin: $HBASE_HOME/bin:$PATH
```

步骤 3 修改 conf/hbase-env.sh。

1）去掉 JAVA_HOME 前的“#”，并将其修改成自己安装的 Java 路径。

2）去掉 HBASE_MANAGES_ZK 前的“#”，并设置其值为 true（HBase 管理自己的 ZooKeeper，这样就不需要安装 ZooKeeper）。有些旧版本的 HBase 未自带 ZooKeeper，这种情况需要自行安装 ZooKeeper。修改后的代码如下。

```
# The java implementation to use.  Java 1.6 required.
export JAVA_HOME=/usr/jdk1.6.0_26
# Tell HBase whether it should manage it's own instance of ZooKeeper or not.
export HBASE_MANAGES_ZK=true
```

步骤 4 打开 conf/hbase-site.xml，添加如下内容。

```
<configuration>
<property>
<name>hbase.rootdir</name>
<value>hdfs://Ubuntu1:9000/hbase</value>
</property>
<property>
<name>hbase.cluster.distributed</name>
<value>true</value>
</property>
<property>
<name>hbase.ZooKeeper.quorum</name>
<value>Ubuntu1</value>
</property>
<property>
<name>ZooKeeper.session.timeout</name>
<value>60000</value>
</property>
<property>
<name>hbase.ZooKeeper.property.clientPort</name>
<value>2181</value>
</property>
<property>
<name>hbase.tmp.dir</name>
<value>/opt/hbase-0.94.1/tmp</value>
<description>Temporary directory on the local filesystem.</description>
</property>
<property>
<name>hbase.client.keyvalue.maxsize</name>
<value>10485760</value>
<description>keyvalue 的大小，默认 10MB</description>
</property>
</configuration>
```

hbase.rootdir 需要与之前安装的 Hadoop 目录下 conf/core-site.xml 文件中的 fs.default.name 属性值对应。例如：

- fs.default.name 设置为 hdfs://Ubuntu1:9000/；
- hbase.rootdir 设置为 hdfs:// Ubuntu1:9000/hbase；
- hbase.ZooKeeper.quorum 设置为 Ubuntu1；
- hbase.tmp.dir 设置为之前创建的 tmp 目录（本节创建在 /opt/hbase-0.92.2/ 下）。

步骤 5 复制 hadoop 下的 hadoop-core-1.0.3.jar 到 hbase/lib 目录下。

步骤 6 启动 HBase，命令如下。

```
jayliu@Ubuntu1:/opt/hbase-0.92.2/conf$ start-hbase.sh
```

因为刚才修改了 /etc/profile 文件，所以重新加载才能生效。

步骤 7 若出现如图 11-14 所示的信息，说明 HBase 运行成功。

```
jayliu@Ubuntu1:/opt/hbase-0.92.2/conf$ jps
2526 TaskTracker
2323 JobTracker
14064 Jps
2232 SecondaryNameNode
2036 DataNode
1845 NameNode
13533 HQuorumPeer
13588 HMaster
13841 HRegionServer
```

图 11-14 HBase 运行图

通过 HBase 的 WebUI 查看 HBase 是否安装成功。打开浏览器，输入网址 http://192.168.192.200:60010。192.168.192.200 是虚拟机的 IP，可以在 Windows 系统中访问，访问结果如图 11-15 所示。到这里 HBase 单机版已经安装成功了。

Local logs, Thread Dump, Log Level, Debug dump

Attributes

Attribute Name	Value	Description
HBase Version	0.92.2, r1379292	HBase version and revision
HBase Compiled	Fri Aug 31 13:13:53 UTC 2012, jenkins	When HBase version was compiled and by whom
Hadoop Version	1.0.3, r1335192	Hadoop version and revision
Hadoop Compiled	Wed May 9 00:11:02 UTC 2012, hortonfo	When Hadoop version was compiled and by whom
HBase Root Directory	hdfs://Ubuntu1:9000/hbase	Location of HBase home directory
HBase Cluster ID	764e32ed-73ff-4046-b498-3872b77854ee	Unique identifier generated for each HBase cluster
Load average	2	Average number of regions per regionserver. Naive computation.
Zookeeper Quorum	Ubuntu1:2181	Addresses of all registered ZK servers. For more, see zk dump.
Coprocessors	[]	Coprocessors currently loaded loaded by the master
HMaster Start Time	Sun May 12 00:59:03 CST 2013	Date stamp of when this HMaster was started
HMaster Active Time	Sun May 12 00:59:03 CST 2013	Date stamp of when this HMaster became active

Tasks

Show All Monitored Tasks Show non-RPC Tasks Show All RPC Handler Tasks Show Active RPC Calls Show Client Operations View as JSON

No tasks currently running on this node.

图 11-15 HBase 的 WebUI

11.4.2 HBase 分布式安装

使用三个节点实现 HBase 分布式安装，这三个节点的部署信息如表 11-1 所示。

表 11-1 HBase 分布式启动进程

主机名	Hadoop 角色	IP	Hadoop jps 命令结果
Ubuntu1	Master Slave	192.168.195.140	NameNode DataNode JobTracker TaskTracker SecondaryNameNode HMaster、HQuorumPeer、HRegionServer

（续）

主机名	Hadoop 角色	IP	Hadoop jps 命令结果
Ubuntu2	Slave	192.168.195.141	DataNode TaskTracker HQuorumPeer、HRegionServer
Ubuntu3	Slave	192.168.195.142	DataNode TaskTracker HQuorumPeer、HRegionServer

因为 HBase 是运行在 Hadoop 集群之上的，所以部署 HBase 之前确保 Hadoop 集群能够正确运行，这里使用第 2 章安装的 Hadoop 集群。

HBase 的安装和 Hive 不一样，Hive 只要在一个节点上安装就可以了，而 HBase 要在所有 Hadoop 节点上安装和配置，首先把 hbase-0.92.2.tar.gz 分发到所有 Hadoop 节点上，然后解压。

1. 安装步骤

HBase 分布式安装可基于 11.4.1 节单机版安装进行修改，具体的安装步骤如下所示。

1）任意选一台 Hadoop 节点，这里选 Master 节点，修改 hbase-site.xml 文件如下。

```
<configuration>
  <property>
    <name>hbase.master</name>
    <value>Ubuntu1:60000</value>
  </property>
  <property>
    <name>hbase.master.port</name>
    <value>60000</value>
    </property>
    <property>
    <name>hbase.rootdir</name>
    <value>hdfs://Ubuntu1:9000/hbase</value>
  </property>
<property>
    <name>hbase.cluster.distributed</name>
    <value>true</value>
  </property>
    <property>
    <name>hbase.ZooKeeper.property.clientPort</name>
    <value>2181</value>
  </property>
    <property>
    <name>dfs.replication</name>
    <value>3</value>
  </property>
    <property>
```

```
    <name>hbase.ZooKeeper.quorum</name>
    <value>Ubuntu1,Ubuntu2,Ubuntu3</value>
  </property>
  <property>
    <name>hbase.ZooKeeper.property.dataDir</name>
    <value>/hadoop/storage/ZooKeeper</value>
  </property>
  <property>
    <name>ZooKeeper.session.timeout</name>
    <value>1200000</value>
  </property>
  <property>
    <name>hbase.ZooKeeper.property.tickTime</name>
    <value>6000</value>
  </property>
</configuration>
```

hbase.rootdir 属性的值需要与之前安装的 Hadoop 目录下 conf/core-site.xml 文件中的 fs.default.name 属性值对应。例如：

- fs.default.name 设置为 hdfs://Ubuntu1:9000/；
- hbase.rootdir 设置为 hdfs:// Ubuntu1:9000/hbase；
- hbase.master.port 属性值为 HBase 的 Master 主机端口；
- hbase.cluster.distributed 属性值为 true，表示采用完全分布式；
- hbase.zookeeper.property.dataDir 属性值为 ZooKeeper 的存放目录；
- hbase.zookeeper.quorum 属性值为 ZooKeeper 集群节点的 IP 地址或主机，建议 3, 5, 7 个。

zookeeper.session.timeout 属性值表示 ZooKeeper 的 Session 超时时间，默认值是 3（单位是分钟）。这意味着一旦一个 Server 宕掉了，Master 至少需要 3 分钟才能察觉到宕机，开始恢复。可能希望将这个超时调短，这样 Master 就能更快察觉到了。在调用这个值之前，需要确认 JVM 的 GC 参数，否则一个长时间的 GC 操作就可能导致超时。当一个 RegionServer 在运行一个长时间的 GC 的时候，可能想要重启并恢复它。

2）编辑 regionservers 文件，添加如下内容。

```
jayliu@Ubuntu1:/opt/hbase-0.92.2/conf$ vi regionservers
Ubuntu1
Ubuntu2
Ubuntu3
```

3）复制 Hadoop 下的 hadoop-core-1.0.3.jar 到 $HBASE_HOME/lib 目录下。

4）复制配置好的 HBase 到另外两台节点上，操作代码如下。

```
sudo scp -r /opt/hbase-0.92.2 root@192.168.195.141:/opt/
sudo scp -r /opt/hbase-0.92.2 root@192.168.195.141:/opt/
cd /opt/
sudo chown -R jayliu:jayliu hbase-0.92.2
```

注意 scp 命令只有 root 权限用户才能操作，所以 scp 复制过去的 hbase-0.92.2 文件夹属性是“root：root”，要手动修改成为“jayliu：jayliu”，如果 Hadoop 和 HBase 安装用户具有 root 权限，就不用执行 chown 操作了，实际生产环境中很少使用 root 用户。

2. 测试安装是否成功

到这里 HBase 的完全分布式安装就全部结束了。下面测试一下安装是否成功。

HBase 的启动和 Hadoop 有点类似，必须在 Master 上面启动，因为配置了 Master 到 Slave 的无密钥（ssh）登录，SSH 具体配置请看第 2 章 Hadoop 安装的内容。HBase 启动分两步，操作如下。

1）启动 HBase 集群。

```
jayliu@Ubuntu1:/opt/hbase-0.92.2/conf$ start-hbase.sh
```

2）启动 HBase Shell。

```
jayliu@Ubuntu1:/opt/hbase-0.92.2$ hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.92.2, r1379292, Fri Aug 31 13:13:53 UTC 2012
```

3）创建一个 member 表。

```
hbase(main):002:0>  create 'member','member_id','address','info'
0 row(s) in 7.8290 seconds
```

4）显示表内容。

```
hbase(main):003:0> list
TABLE
member
2 row(s) in 0.7350 seconds
```

5）查看 member 表结构。

```
hbase(main):004:0> describe 'member'
DESCRIPTION                                          ENABLED
  {NAME => 'member', FAMILIES => [{NAME => 'address true',
    BLOOMFILTER => 'NONE', REPLICATION_SCOPE => '0',
    VERSIONS => '3', COMPRESSION => 'NONE', MIN_VE RSIONS => '0',
    TTL => '2147483647', BLOCKSIZE =>'65536',
    IN_MEMORY => 'false', BLOCKCACHE => 'true'},
    {NAME => 'info', BLOOMFILTER => 'NONE', REPL ICATION_SCOPE => '0',
     VERSIONS => '3', COMPRESSION => 'NONE', MIN_VERSIONS => '0',
     TTL => '2147483 647', BLOCKSIZE => '65536',
     IN_MEMORY => 'false', BLOCKCACHE => 'true'},
     {NAME => 'member_id', BLO OMFILTER => 'NONE',
     REPLICATION_SCOPE => '0', VER SIONS => '3', COMPRESSION => 'NONE',
     MIN_VERSIONS => '0', TTL => '2147483647', BLOCKSIZE => '65536',
     IN_MEMORY => 'false', BLOCKCACHE => 'true'}]}
1 row(s) in 0.8870 seconds
```

如果上面的操作都成功，说明 HBase 分布式安装也成功了。

11.5 HBase 的 Shell 操作

HBase Shell 基本操作命令如表 11-2 所示。

表 11-2 HBase 的 Shell 命令

命 令	命令表达式
创建表	create '表名称','列名称 1','列名称 2','列名称 N'
添加记录	put '表名称','行名称','列名称 :','值'
查看记录	get '表名称','行名称'
查看表中的记录总数	count '表名称'
删除记录	delete '表名','行名称','列名称'
删除一张表	先要屏蔽该表，才能对该表进行删除，第一步 disable '表名称'，第二步 drop '表名称'
查看所有记录	scan " 表名称 "
查看某个表某个列中所有数据	scan " 表名称 ",['列名称 :']
更新记录	就是重写一遍进行覆盖

11.5.1 一般操作

如果想知道 HBase Shell 提供哪些功能操作，可以输入 help 命令查看。

```
hbase(main):005:0> help
HBase Shell, version 0.92.2, r1379292, Fri Aug 31 13:13:53 UTC 2012
Type 'help "COMMAND"', (e.g. 'help "get"' -- the quotes are necessary) for help
on a specific command.
Commands are grouped. Type 'help "COMMAND_GROUP"', (e.g. 'help "general"') for
help on a command group.
COMMAND GROUPS:
  Group name: general
  Commands: status, version, whoami
  Group name: ddl
  Commands: alter, alter_async, alter_status, create, describe, disable,
            disable_all, drop, drop_all, enable, enable_all, exists,
            is_disabled, is_enabled, list, show_filters
  Group name: dml
  Commands: count, delete, deleteall, get, get_counter, incr, put, scan, truncate
  Group name: tools
  Commands: assign, balance_switch, balancer, close_region, compact, flush,
            hlog_roll, major_compact, move, split, unassign, zk_dump
  Group name: replication
  Commands: add_peer, disable_peer, enable_peer, list_peers, remove_peer,
            start_replication, stop_replication
  Group name: security
```

```
  Commands: grant, revoke, user_permission
SHELL USAGE:
Quote all names in HBase Shell such as table and column names.  Commas delimit
command parameters.  Type <RETURN> after entering a command to run it.
Dictionaries of configuration used in the creation and alteration of tables are
Ruby Hashes. They look like this:
  {'key1' => 'value1', 'key2' => 'value2', ...}
and are opened and closed with curley-braces.  Key/values are delimited by the '=>'
character combination.  Usually keys are predefined constants such as
NAME, VERSIONS, COMPRESSION, etc.  Constants do not need to be quoted.
Type 'Object.constants' to see a (messy) list of all constants in the environment.
If you are using binary keys or values and need to enter them in the shell,
use double-quote'd hexadecimal representation. For example:
  hbase> get 't1', "key\x03\x3f\xcd"
  hbase> get 't1', "key\003\023\011"
  hbase> put 't1', "test\xef\xff", 'f1:', "\x01\x33\x40"
The HBase shell is the (J)Ruby IRB with the above HBase-specific commands added.
```

1）查看 HBase 服务器状态。

```
hbase(main):006:0> status
servers, 0 dead, 4.0000 average load
```

2）查看 HBase 版本。

```
hbase(main):007:0> version
0.92.2, r1379292, Fri Aug 31 13:13:53 UTC 2012
```

11.5.2 DDL 操作

使用个人信息为例演示 HBase 的用法。创建一个 member 表，其结构如表 11-3 所示。

表 11-3 member 表结构

Row Key	address			info		
	contry	province	city	age	birthday	company
jayliu	Chia	Beijing	Beijing	27	1986-08-23	Autonavi
…						
…						

这里 address 和 info 对于表来说是一个有三个列的列族：address 列族由三个列组成 contry、province 和 city；info 列族由三个列组成 age、birthday 和 company。当然可以根据需要在 address 和 info 中建立更多的列族，如 name、telephone 等相应的列添加入 info 列族。

1）创建一个表 member。

```
hbase(main):011:0>create 'member','member_id','address','info'
0 row(s) in 1.2210seconds
```

member 是表的名字，随便起名，member_id 这里与 Row Key 对应。

2）查看所有的表。

```
hbase(main):012:0>list
TABLE
member
1 row(s) in 0.0160seconds
```

3）查看表结构。

```
hbase(main):006:0>describe 'member'
DESCRIPTION                                    ENABLED
 {NAME => 'member', FAMILIES => [{NAME=> 'address', BLOOMFILTER => 'NONE',
  REPLICATION_SCOPE => '0', true
  VERSIONS => '3', COMPRESSION => 'NONE',TTL => '2147483647',
  BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'},
  {NAME =>'info',BLOOMFILTER => 'NONE', REPLICATION_SCOPE => '0',
  VERSI ONS => '3', COMPRESSION => 'NONE', TTL=> '2147483647',
  BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}]}
1 row(s) in 0.0230seconds
```

4）删除一个列族。删除列族有两步，首先 disable 表，最后 enable 表。

之前建了 3 个列族，但是发现 member_id 这个列族是多余的，因为它就是主键，所以要将其删除。

```
hbase(main):003:0>alter 'member',{NAME=>'member_id',METHOD=>'delete'}
ERROR: Table memberis enabled. Disable it first before altering.
```

报错，删除列族的时候必须先将表给 disable 掉。

```
hbase(main):004:0>disable 'member'
0 row(s) in 2.0390seconds
```

删除 member_id 列。

```
hbase(main):005:0>alter 'member',{NAME=>'member_id',METHOD=>'delete'}
0 row(s) in 0.0560seconds
```

查看是否删除成功。

```
hbase(main):006:0>describe 'member'
DESCRIPTION                                    ENABLED
 {NAME => 'member', FAMILIES => [{NAME=> 'address', BLOOMFILTER => 'NONE',
  REPLICATION_SCOPE => '0',false
  VERSIONS => '3', COMPRESSION => 'NONE',TTL => '2147483647', BLOCKSIZE => '65536',
  IN_MEMORY => 'false', BLOCKCACHE => 'true'},
  {NAME =>'info', BLOOMFILTER => 'NONE', REPLICATION_SCOPE => '0',
  VERSI ONS => '3', COMPRESSION => 'NONE', TTL=> '2147483647',
  BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}]}
1 row(s) in 0.0230seconds
```

该列族已经删除，继续对表进行 enable 操作。

```
hbase(main):008:0> enable 'member'
0 row(s) in 2.0420seconds
```

5）列出所有的表。

```
hbase(main):028:0>list
    TABLE
    member
    temp_table
2 row(s) in 0.0150seconds
```

6）删除表 temp_table。

```
hbase(main):029:0>disable 'temp_table'
0 row(s) in 2.0590seconds
hbase(main):030:0>drop 'temp_table'
0 row(s) in 1.1070seconds
```

删除一个表，先 disable 再 drop，否则报错。

7）查询表是否存在。

```
hbase(main):021:0>exists 'member'
Table member doesexist
0 row(s) in 0.1610seconds
```

8）判断表是否 enable。

```
hbase(main):034:0>is_enabled 'member'
true
0 row(s) in 0.0110seconds
```

9）判断表是否 disable。

```
hbase(main):032:0>is_disabled 'member'
false
0 row(s) in 0.0110seconds
```

11.5.3 DML 操作

1）向表 member 插入几条记录，命令如下。

```
put 'member','jayliu','info:age','27'
put 'member','jayliu','info:birthday','1986-08-23'
put 'member','jayliu','info:company','autonavi'
put 'member','jayliu','address:contry','china'
put 'member','jayliu','address:province','beijing'
put 'member','jayliu','address:city','beijing''
```

2）获取一条数据。

获取一个 ID 的所有数据，命令如下。

```
hbase(main):001:0>get 'member','jayliu'
COLUMN                    CELL
 address:city          timestamp=1368299129738, value=beijing
 address:contry        timestamp=1368299082123, value=china
 address:province      timestamp=1368299082289, value=beijing
```

```
 info:age              timestamp=1368299081654, value=27
 info:birthday         timestamp=1368299081991, value=1986-08-23
 info:company          timestamp=1368299082066, value=autonavi
6 row(s) in 6.2580 seconds
```

获取一个 ID，一个列族的所有数据。

```
hbase(main):002:0>get 'member','jayliu','info'
COLUMN                  CELL
  info:age              timestamp=1368299081654, value=27
  info:birthday         timestamp=1368299081991, value=1986-08-23
  info:company          timestamp=1368299082066, value=autonavi
3 row(s) in 0.4910 seconds
```

获取一个 ID，一个列族中一个列的所有数据。

```
hbase(main):002:0>get 'member','jayliu','info:age'
COLUMN                    CELL
 info:age               timestamp=1368299081654, value=27
1 row(s) in 0.4520 seconds
```

3）更新一条记录。

将 jayliu 的年龄改成 120，命令如下。

```
hbase(main):004:0>put 'member','jayliu','info:age' ,'120'
0 row(s) in 0.0210seconds
hbase(main):005:0>get 'member','jayliu','info:age'
COLUMN                        CELL
  info:age              timestamp=1368299539768, value=120
1 row(s) in 0.9620 seconds
```

4）通过 timestamp 获取两个版本的数据。

```
hbase(main):010:0>get 'member','jayliu',{COLUMN=>'info:age',TIMESTAMP=>1368299081654}

COLUMN                        CELL
  info:age              timestamp=1368299081654, value=27
1 row(s) in 0.0140seconds
hbase(main):011:0>get 'member','jayliu',{COLUMN=>'info:age',TIMESTAMP=>1368299539768}
COLUMN                        CELL
  info:age              timestamp=1368299539768, value=120
1 row(s) in 0.0180seconds
```

5）全表扫描。

```
hbase(main):013:0>scan 'member'
ROW                   COLUMN+CELL
 jayliu               column=address:city, timestamp=1368299129738, value=beijing
 jayliu               column=address:contry, timestamp=1368299082123, value=china
 jayliu               column=address:province, timestamp=1368299082289, value=beijing
 jayliu               column=info:age, timestamp=1368299539768, value=120
 jayliu               column=info:birthday, timestamp=1368299081991, value=1986-08-23
 jayliu               column=info:company, timestamp=1368299082066, value=autonavi
1 row(s) in 1.4210 seconds
```

6）删除 ID 为 "jayliu" 的值的 'info:age' 字段。

```
hbase(main):016:0>delete 'member','jayliu','info:age'
0 row(s) in 0.0150seconds
hbase(main):018:0>get 'member','jayliu'
COLUMN                  CELL
 address:city           timestamp=1368299129738, value=beijing
 address:contry         timestamp=1368299082123, value=china
 address:province       timestamp=1368299082289, value=beijing
 info:birthday          timestamp=1368299081991, value=1986-08-23
 info:company           timestamp=1368299082066, value=autonavi
5 row(s) in 0.1310 seconds
```

7）查询表中有多少行。

```
hbase(main):019:0>count 'member'
1 row(s) in 0.0160seconds
```

8）给 'jayliu' 这个 ID 增加 'info:age' 字段，并使用 counter 实现递增。

```
hbase(main):057:0*incr 'member','xiaofeng','info:age'
COUNTER VALUE = 1
hbase(main):058:0>get 'member','jayliu','info:age'
COLUMN                  CELL
 info:age       timestamp=1321590997648, value=\x00\x00\x00\x00\x00\x00\x00\x01
1 row(s) in 0.0140seconds
hbase(main):059:0>incr 'member','jayliu','info:age'
COUNTER VALUE = 2
hbase(main):060:0>get 'member', 'jayliu','info:age'
COLUMN                  CELL
 info:age       timestamp=1321591025110, value=\x00\x00\x00\x00\x00\x00\x00\x02
1 row(s) in 0.0160seconds
```

获取当前 counter 的值：

```
hbase(main):069:0>get_counter 'member', 'jayliu','info:age'
COUNTER VALUE = 2
```

9）将整张表清空。

```
hbase(main):035:0>truncate 'member'
Truncating 'member'table (it may take a while):
 - Disabling table...
 - Dropping table...
 - Creating table...
0 row(s) in 4.3430seconds
```

可以看出，HBase 是通过先对表执行 disable，然后再执行 drop 操作后重建表来实现 truncate 的功能的。

11.5.4 HBase Shell 脚本

既然是 Shell 命令，当然也可以把所有的 HBase Shell 命令写入一个文件内，像 Linux

Shell 脚本程序那样顺序地执行所有命令。如同写 Linux Shell，把所有 HBase Shell 命令书写在一个文件内，然后执行如下命令。例如有个 test.sh 文件，执行命令如下。

```
jayliu@Ubuntu1:~$  hbase shell test.sh
```

11.6 HBase 客户端

与 HBase 集群进行交互，有很多种不同的方式可以选择，如 Java API、REST、Thrift 等。下面分别介绍它们的用法。

11.6.1 Java API 交互

HBase 与 Hadoop 一样，都是用 Java 编写的，所以 HBase 对 Java 支持是必需的，下面看看怎么使用 Java 语言对 HBase 进行操作。Java API 核心类介绍如下。

1. HBaseConfiguration 类

HBaseConfiguration 是每一个 HBase Client 都会使用到的对象，它代表 HBase 配置信息。有两种构造方式。

```
public HBaseConfiguration()
public HBaseConfiguration(final Configuration c)
```

默认构造方式会尝试从 hbase-default.xml 和 hbase-site.xml 文件中读取配置。如果 CLASSPATH 没有这两个文件，就需要自己配置。

```
Configuration HBASE_CONFIG = new Configuration();
HBASE_CONFIG.set("hbase.ZooKeeper.quorum", "zkServer");
HBASE_CONFIG.set("hbase.ZooKeeper.property.clientPort", "2181" );
HBaseConfiguration cfg = new HBaseConfiguration(HBASE_CONFIG);
```

2. 创建表

创建表通过 HBaseAdmin 对象操作。HBaseAdmin 负责 META 表信息的处理。HBaseAdmin 提供了 createTable 方法。

```
public void createTable(HTableDescriptor desc)
```

HTableDescriptor 表示表的 Schema，提供的常用方法有以下两个。

- ❑ setMaxFileSize：指定最大 Region 的大小。
- ❑ setMemStoreFlushSize：指定 MemStore Flush 到 HDFS 上的文件大小。

3. 增加 Family

使用 addFamily 方法实现 Family 的添加。

```
public void addFamily(final HColumnDescriptor family)
```

HColumnDescriptor 代表 Column 的 Schema，提供的常用方法有以下几个。

- setTimeToLive：指定最大的 TTL（单位是 ms），过期数据会被自动删除。
- setInMemory：指定是否放在内存中，对小表有用，可用于提高效率。默认关闭。
- setBloomFilter：指定是否使用 BloomFilter，可提高随机查询效率。默认关闭。
- setCompressionType：设定数据压缩类型。默认无压缩。
- setMaxVersions：指定数据最大保存的版本个数。默认为 3。

举个简单的例子，创建 4 个 Family 表，命令如下。

```
HBaseAdmin hAdmin = new HBaseAdmin(hbaseConfig);
HTableDescriptor table = new HTableDescriptor(tableName);
table.addFamily(new HColumnDescriptor("f1"));
table.addFamily(new HColumnDescriptor("f2"));
table.addFamily(new HColumnDescriptor("f3"));
table.addFamily(new HColumnDescriptor("f4"));
hAdmin.createTable(table);
```

4. 删除表

删除表也是通过 HBaseAdmin 来操作，删除表之前首先要 disable 表。这是一个非常耗时的操作，所以不建议频繁删除表。

disableTable 和 deleteTable 分别用来执行 disable 和 delete 操作。使用方法如下。

```
HBaseAdmin hAdmin = new HBaseAdmin(hbaseConfig);
if (hAdmin.tableExists(tableName)) {
        hAdmin.disableTable(tableName);
        hAdmin.deleteTable(tableName);
}
```

5. 查询数据

查询分为单条随机查询和批量查询。单条查询通过 Row Key 在 Table 中查询某一行的数据，HTable 提供了 get 方法完成单条查询。批量查询通过制定一段 Row Key 的范围来查询，HTable 提供了 getScanner 方法完成批量查询。

```
public Result get(final Get get)
public ResultScanner getScanner(final Scan scan)
```

Get 对象包含一个 Get 查询需要的信息，它的构造方法有两种。

```
public Get(byte [] row)
public Get(byte [] row, RowLock rowLock)
```

Row Lock 为了保证读写的原子性，可以传递一个已经存在 Row Lock，否则 HBase 会自动生成一个新的 Row Lock。

Scan 对象提供了默认构造函数，一般使用默认构造函数。

1）Get 和 Scan 的常用方法有以下几个。

- addFamily/addColumn：指定需要的 Family 或者 Column，如果没有调用任何 Family 或者 Column，会返回所有的 Column。

- setMaxVersions：指定最大的版本个数。如果不带任何参数调用 setMaxVersions，表示取所有的版本。如果不调用 setMaxVersions，只会取到最新的版本。
- setTimeRange：指定最大的时间戳和最小的时间戳，只有在此范围内的 Cell 才能被获取。
- setTimeStamp：指定时间戳。
- setFilter：指定 Filter 过滤不需要的信息。

2）Scan 特有的方法如下。

- setStartRow：指定开始的行。如果不调用，从表头开始。
- setStopRow：指定结束的行（不含此行）。
- setBatch：指定最多返回的 Cell 数目。防止一行中有过多的数据，导致 OOM 错误。

3）Result 代表是一行的数据。常用方法有以下几个。

- getRow：返回 Row Key。
- raw：返回所有的 KeyValue 数组。
- getValue：按照 Column 来获取 Cell 的值。

ResultScanner 是 Result 的一个容器，每次调用 ResultScanner 的 next 方法会返回 Result。

```
public Result next() throws IOException;
public Result [] next(int nbRows) throws IOException;
```

示例代码如下。

```
Scan scan = new Scan();
scan.setMaxVersions();
ResultScanner ss = table.getScanner(scan);
for(Result r:ss){
    System.out.println(new String(r.getRow()));
    for(KeyValue kv:r.raw()){
        System.out.println(new String(kv.getColumn()));
    }
}
```

6. 插入数据

HTable 通过 put 方法插入数据，可以传递单个 put 对象或 List put 对象分别实现单条插入和批量插入。

```
public void put(final Put put) throws IOException
public void put(final List<Put> puts) throws IOException
```

Put 提供 3 种构造方式。

```
public Put(byte [] row)
public Put(byte [] row, RowLock rowLock)
public Put(Put putToCopy)
```

Put 常用的方法有以下几个。

- add：增加一个 Cell。

- setTimeStamp：指定所有 Cell 默认的 timestamp，如果一个 Cell 没有指定 timestamp，就会用到这个值。如果没有调用，HBase 会将当前时间作为未指定 timestamp 的 Cell 的 timestamp。
- setWriteToWAL：WAL 是 Write Ahead Log 的缩写，指的是 HBase 在插入操作前是否写 Log。默认是打开，关掉会提高性能，但是如果系统出现故障（负责插入的 Region Server 挂掉），数据可能会丢失。

另外 HTable 也有两个方法会影响插入的性能。

- setAutoFlash：AutoFlush 指的是在每次调用 HBase 的 Put 操作，是否提交到 HBase Server。默认是 true，每次会提交。如果此时是单条插入，就会有更多的 I/O，从而降低其性能。
- setWriteBufferSize：Write Buffer Size 在 AutoFlush 为 false 的时候起作用，默认是 2MB，也就是插入数据超过 2MB, 就会自动提交到 Server 。

示例代码如下。

```
HTable table = new HTable(hbaseConfig, tableName);
table.setAutoFlush(autoFlush);
List<Put> lp = new ArrayList<Put>();
int count = 10000;
byte[] buffer = new byte[1024];
Random r = new Random();
for (int i = 1; i <= count; ++i) {
   Put p = new Put(String.format("row%09d",i).getBytes());
   r.nextBytes(buffer);
   p.add("f1".getBytes(), null, buffer);
   p.add("f2".getBytes(), null, buffer);
   p.add("f3".getBytes(), null, buffer);
   p.add("f4".getBytes(), null, buffer);
   p.setWriteToWAL(wal);
   lp.add(p);
   if(i%1000==0){
     table.put(lp);
     lp.clear();
   }
}
```

7. 删除数据

HTable 通过 delete 方法删除数据。

```
public void delete(final Delete delete)
```

Delete 构造方法如下。

```
public Delete(byte [] row)
public Delete(byte [] row, long timestamp, RowLock rowLock)
public Delete(final Delete d)
```

Delete 常用方法有 deleteFamily/deleteColumns，用来指定要删除的 Family 或者 Column 的数据。如果不调用任何这样的方法，将会删除整行。

注意 如果某个 Cell 的 timestamp 高于当前时间，这个 Cell 将不会被删除，仍然可以查出来。

示例代码如下。

```
HTable table = new HTable(hbaseConfig, "mytest");
Delete d = new Delete("row1" .getBytes());
table.delete(d)
```

8. 切分表

HBaseAdmin 提供 split 方法将 table 进行切分。

```
public void split(final String tableNameOrRegionName)
```

如果提供的是 tableName，会将 table 所有 Region 进行切分；如果提供的是 RegionName，只会切分这个 Region。Split 是一个异步操作，因此它并不能确切控制 Region 的个数。

示例代码如下。

```
public void split(String tableName,int number,int timeout) throws Exception {
    Configuration HBASE_CONFIG = new Configuration();
    HBASE_CONFIG.set("hbase.ZooKeeper.quorum", GlobalConf.ZOOKEEPER_QUORUM);
    HBASE_CONFIG.set("hbase.ZooKeeper.property.clientPort", GlobalConf.ZOOKEEPER_PORT);
    HBaseConfiguration cfg = new HBaseConfiguration(HBASE_CONFIG);
    HBaseAdmin hAdmin = new HBaseAdmin(cfg);
    HTable hTable = new HTable(cfg,tableName);
    int oldsize = 0;
    long time = System.currentTimeMillis();
    while(true){
        int size = hTable.getRegionsInfo().size();
        logger.info("the region number="+size);
        if(size>=number ) break;
        if(size!=oldsize){
            hAdmin.split(hTable.getTableName());
            oldsize = size;
        }       else if(System.currentTimeMillis()-time>timeout){
            break;
        }
        Thread.sleep(1000*10);
    }
}
```

11.6.2 MapReduce 操作 HBase

在 HBase 系统上运行批处理运算，最方便和实用的模型依然是 MapReduce，如图 11-16 所示。

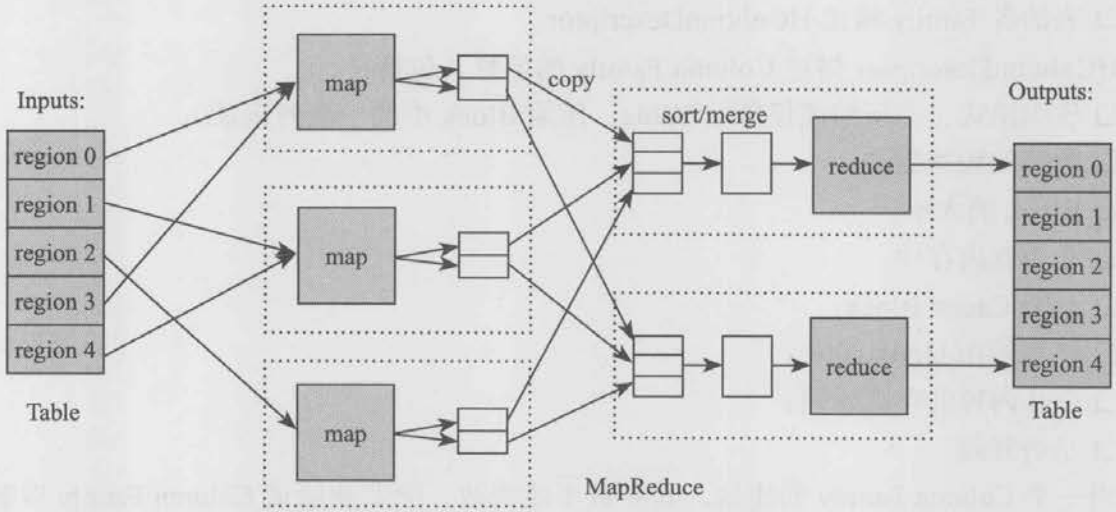


图 11-16 MapReduce 操作 HBase

HBase Table 和 Region 的关系类似 HDFS File 和 Block 的关系，HBase 提供配套的 TableInputFormat 和 TableOutputFormat API，可以方便地将 HBase Table 作为 Hadoop MapReduce 的 Source 和 Sink。对于 MapReduce Job 应用开发人员来说，基本不需要关注 HBase 系统自身的细节。

org.apache.hadoop.hbase.mapreduce 包中的类和工具用来将 HBase 作为 MapReduce 作业的输出。

1. HbaseConfiguration 类

org.apache.hadoop.hbase.HbaseConfiguration 继承了 org.apache.hadoop.conf.Configuration 类，创建一个 HbaseConfiguration 对象实例，会返回读入 CLASSPATH 下 hbase-site.xml 文件和 hbase-default.xml 文件中 HBase 配置信息的一个 Configuration，该 Configuration 接下来会用于创建 HbaseAdmin 和 HTable 实例。

HbaseAdmin 和 HTable 两个类在 org.apache.hadoop.hbase.client 包中，HBase Admin 用于管理 HBase 集群、添加和删除表，HTable 用于访问指定的表，Configuration 实例指向了执行这些代码的集群上的这些类。

2. HTableDescriptor 类和 HColumnDescriptor 类

HBase 中表结构由 HTableDescriptor 描述（包括 HcolumnDescriptor），对表的新增、修改、删除操作在接口 HMasterInterface 中定义，而该接口由 HMaster 实现。

HTableDescriptor 包含：

- 表名、byte[] 和 String 格式。
- 表的元信息，以 Key-Value 形式存储。存储信息包括文件最大的大小（默认 256MB）、是否只读、Flush 时内存占用大小（默认 64MB）、是否 root 或 Meta Region、DEFERRED_LOG_FLUSH。

❑ 表的各 Family 描述 HColumnDescriptor。

HColumnDescriptor 描述 Column Family 的信息，包括：

❑ 压缩格式（不压缩或仅压缩 Value、压缩 Block 中的一系列记录）；

❑ 数据的版本数量；

❑ Block 的大小；

❑ 是否在内存中；

❑ 是否 Cache Block；

❑ 是否使用 bloomfilter；

❑ Cell 内容的存活时间；

❑ 是否复制。

当一个 Column Family 创建后，其参数不能修改，除非删除该 Column Family 后新建一个，但删除 Column Family 也会删除该 Column Family 下的数据。另外，HTableDescriptor 中包含 ROOT_TABLEDESC 和 META_TABLEDESC 两个实例以描述 -ROOT- 和 .META. 表。

❑ ROOT_TABLEDESC 包含一个 info 的 Column Family。

❑ META_TABLEDESC 包含一个 info 和 historian，两个 Column Family。

3. CreateTable 方法

如果指定 Split Key 为该 Table 按指定键初始创建多个 Region，否则仅创建一个 Region，过程如下。

1）为 Table 创建 HRegionInfo；

2）判断是不是所有的 Meta Region 都 online（由 RegionManager 的 MetaScanner 扫描线程分配 Meta Region）。

3）判断 ServerManager 是否有足够 Region Server 来创建 table。

从 RegionManager 的 online MetaRegion 查找该 HRegionInfo 应放入哪一个 MetaRegion 中，在 onlineMetaRegion 中查找仅比 RegionName 小的 MetaRegion，而 RegionName 由 tableName、起始 Key 和 regionId（root 为 0，meta 为 1，user 当前时间）组成，同时 Master 的 ServerConnection 获取 HRegionInterface 代理连接到该 MetaRegion，并查找对应该 Table 为 Key 的记录是否存在，若存在则报错该表已存在，由 RegionManager 根据 HRegionInfo 创建新的 user。Region 在 rootDir 目录下新建以 tableName 为名的目录，在 tableName 目录下新建一个 Region 的目录（经编码后的 regionName），并新建一个 HRegion 对象。

disable、enable 和 delete 等操作封装在继承自 TableOperation 的类中，该类先获得要操作表的所有 MetaRegion，扫描这些 MetaRegion 中所有该表的 user Region 信息并做相应处理，最后处理 MetaRegion。

4. TableInputFormat 类

通过设置 conf.set(TableInputFormat.INPUT_TABLE, "tableName") 设定 HBase 的输入表，tableName 为表名。

设置 conf.set(TableInputFormat.SCAN, TableMRUtil.convertScanToString(scan))，设定对 HBase 输入表的 scan 方式。

```
setHTable(new HTable(new Configuration(conf),tableName));
```

通过 TableInputFormat.setConf(Configuration conf) 方法初始化 scan 对象；scan 对象是从 Job 中设置的对象，以字符串的形式传给 TableInputFormat，在 TableInputFormat 内部将 scan 字符串转换为 scan 对象，操作如下。

```
scan = TableMapReduceUtil.convertStringtoScan(conf.get(SCAN));
```

TableInputFormat 继承 TableInputFormatBase 实现了 InputFormat 抽象类的两个抽象方法 getSplits() 和 createRecordReader()。

getSplits() 断定输入对象的切分原则，原则是对于 TableInputFormatBase，会遍历 HBase 相应表的所有 HRegion，每一个 HRegion 都会被切分成一个 Split，所以切分的块数与表中 HRegion 的数目是相同的，代码如下。

```
InputSplit split = new TableSplit(table.getTableName(),
                   splitStart, splitStop, regionLocation);
```

在 Split 中只会记载 HRegion 的起始 Row Key 和结束 Row Key，具体去读取这片区域的数据是 createRecordReader() 实现的。

对于一个 Map 任务，JobTracker 会考虑 TaskTracker 的网络位置，并选取一个距离其输入分片文件最近的 TaskTracker。在理想情况下，任务是数据本地化的（data-local），也就是任务运行在输入分片所在的节点上。同样，任务也可能是机器本地化的：任务和输入分片在同一个机架，但不在同一个节点上。

对于 Reduce 任务，JobTracker 简单地从待运行的 Reduce 任务列表中选取下一个来运行，用不着考虑数据段本地化。

createRecordReader() 按照必然格式读取响应数据，接收 Split 块，返回读取记录的结果，操作代码如下。

```
public RecordReader<ImmutableBytesWritable, Result>
            createRecordReader(InputSplit split, TaskAttemptContext context){
    Scan scan = new Scan(this.scan);
    scan.setStartRow(tableSplit.getStartRow());
    scan.setStopRow(tableSplit.getEndRow());
    tableRecorderReader.setScan(scan);
    tableRecorderReader.setHTable(table);
    tableRecorderReader.init();
    return tableRecorderReader;
}
```

tableRecorderReader.init() 返回的是这个分块的起始 Row Key 的记录；RecordReader 将一个 Split 解析成 <Key, Value> 的形式提供给 map 函数，Key 就是 Row Key，Value 就是对应的一行数据。

RecordReader 用于在划分中读取 <Key, Value> 对。RecordReader 有 5 个虚方法，下面

分别进行介绍。

- initialize：初始化，输入参数包括该 Reader 工作的数据划分 InputSplit 和 Job 的上下文 context。
- nextKey：得到输入的下一个 Key，如果数据划分已经没有新的记录，返回空。
- nextValue：得到 Key 对应的 Value，必须在调用 nextKey 后调用。
- getProgress：得到现在的进度。
- close：来自 java.io 的 Closeable 接口，用于清理 RecordReader。

在 MapReduce 驱动中调用 TableInputFormat 的类：

```
job.setInputFormatClass(TableInputFormat.class);
```

使用以下方法就不需要再单独定义。

```
tableMapReduceUtil.initTableReducerJob("daily_result", DailyReduce.class, job);
```

initTableReducerJob() 方法完成一系列操作。

1）job.setOutputFormatClass(TableOutputFormat.class)；设置输出格式。

2）conf.set(TableOutputFormat.OUTPUT_TABLE, table)；设置输出表。

3）初始化 partition。

11.6.3 向 HBase 中写入数据

本节介绍利用 MapReduce 操作 HBase，首先介绍如何上传数据，借助最熟悉的 WordCount 案例，将 WordCount 的结果存储到 HBase 而不是 HDFS。

输入文件 test.txt 的内容为：

```
hello hadoop
hadoop is easy
```

1）编写 Mapper 函数。这里和 WordCount 的 Mapper 函数一样，没有任何的变化，如代码清单 11-1 所示。

代码清单 11-1 mapred\hbase\ WordCountMapperHbase .java

```
public class WordCountMapperHbase extends Mapper<Object, Text, Text, IntWritable>{
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();
  //map 函数没有改变
  public void map(Object key, Text value, Context context
                  ) throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, one);
    }
}
```

2）实现 Reducer 类，如代码清单 11-2 所示。

代码清单 11-2 mapred\hbase\ WordCountReducerHbase.java

```
public  class WordCountReducerHbase extends TableReducer <Text,IntWritable,
        ImmutableBytesWritable> {
  private IntWritable result = new IntWritable();
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
                        throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    Put put = new Put(key.getBytes());  //put 实例化，每一个词存一行
    //列族为 content，列修饰符为 count，列值为数目
    put.add(Bytes.toBytes("content"), Bytes.toBytes("count"),
          Bytes.toBytes(String.valueOf(sum)));
    context.write(new ImmutableBytesWritable(key.getBytes()), put);
  }
}
```

由上面可知，IntSumReducer 继承自 TableReduce，在 Hadoop 中 TableReducer 继承 Reducer 类，它的原型为 TableReducer<KeyIn, Values, KeyOut>，可以看出，从 HBase 中读出的 Key 类型是 ImmutableBytesWritable，意为不可变类型，因为 HBase 中所有数据都是用字符串存储的。

3）编写驱动类，如代码清单 11-3 所示。

代码清单 11-3 mapred\hbase\ MapReduceHbaseDerver.java

```
public  class  MapReduceHbaseDerver{
    public static void main(String[] args) throws Exception {
         String tablename  = "wordcount";
         //实例化 Configuration，注意不能用 new HBaseConfiguration() 了
         Configuration conf = HBaseConfiguration.create();
         HBaseAdmin admin = new HBaseAdmin(conf);
         if(admin.tableExists(tablename)){
              System.out.println("table exists! recreating ...");
              admin.disableTable(tablename);
              admin.deleteTable(tablename);
         }
         HTableDescriptor htd = new HTableDescriptor(tablename);
         HColumnDescriptor hcd = new HColumnDescriptor("content");
         htd.addFamily(hcd);          //创建列族
         admin.createTable(htd);      //创建表

         String[] otherArgs =
                        new GenericOptionsParser(conf, args).getRemainingArgs();
         if (otherArgs.length != 1) {
           System.err.println("Usage: wordcount <in> <out>"+otherArgs.length);
```

```
            System.exit(2);
        }
        Job job = new Job(conf, "word count");
        job.setJarByClass(MapReduceHbaseDerver.class);
        job.setMapperClass(WordCountMapperHbase .class);
        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
        // 注意此处的 TableMapReduceUtil 要用 hadoop.hbase.mapreduce 包中的
        // 而不是 hadoop.hbase.mapred 包中的
        TableMapReduceUtil.initTableReducerJob(tablename,
                WordCountReducerHbase.class, job);
        //key 和 value 的类型设定最好放在 initTableReducerJob 函数后面，否则会报错
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

在 Job 配置的时候没有设置 job.setReduceClass()，而是用以下方法执行 Reduce 类。

```
TableMapReduceUtil.initTableReducerJob(tablename, IntSumReducer.class, job);
```

提示 需要注意的是，此处 TableMapReduceUtil 是 hadoop.hbase.mapreduce 包中的，而不是 hadoop.hbase.mapred 包中的，否则会报错。

4）运行程序。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$  hadoop  jar  hbase-mapreduce.jar
            mapred.hbase. MapReduceHbaseDerver
```

5）查看结果。

```
hbase(main):013:0>scan  'wordcount'
    ROW                  COLUMN+CELL
     hello              column= content:count, timestamp=1368299129738, value=1
     hadoop             column= content:count, timestamp=1368299082123, value=2
     is                 column= content:count, timestamp=1368299082289, value=1
     easy               column= content:count, timestamp=1368299539768, value=1
    1 row(s) in 1.4210 seconds
```

HBase 提供了 API 接口，可以把 MapReduce 分析出来的结果直接存到 HBase 里面，默认是存到 HDFS 下面。

11.6.4 读取 HBase 中的数据

下面介绍如何读取 HBase 中的数据。读取数据比较简单，编写 Mapper 函数，读取 <Key, Value> 值，通过 Reducer 函数直接输出得到的结果就行了。

1）编写 Mapper 函数，如代码清单 11-4 所示。

代码清单 11-4 mapred\hbase\ WordCountMapperHbaseReader.java

```
public  class WordCountMapperHbaseReader
```

```
        extends TableMapper<Text, Text>{
    public void map(ImmutableBytesWritable row, Result values, Context context)
                throws IOException, InterruptedException {
        StringBuffer sb = new StringBuffer("");
        for(java.util.Map.Entry<byte[],byte[]> value : values.getFamilyMap(
                "content".getBytes()).entrySet()){
            String str = new String(value.getValue());
            //将字节数组转换成 String 类型，需要 new String()
            if(str != null){
                sb.append(new String(value.getKey()));
                sb.append(":");
                sb.append(str);
            }
            context.write(new Text(row.get()), new Text(new String(sb)));
        }
    }
}
```

编写 Mapper 实现继承 TableMapper<Text, Text> 抽象类，TableMapper 类专门用于完成 MapReduce 与 HBase 表之间的操作。

2）实现 Reducer 类。该 Reducer 类与 WordCount 的 Reduce 类似，如代码清单 11-5 所示。

代码清单 11-5　mapred\hbase\ WordCountReduceHbaseReader.java

```
public  class WordCountReduceHbaseReader
        extends Reducer <Text,Text,Text,Text> {
    private Text result = new Text();

    public void reduce(Text key, Iterable<Text> values,
                       Context context
                       ) throws IOException, InterruptedException {
        for (Text val : values) {
            result.set(val);
            context.write(key,result);
        }
    }
}
```

3）编写驱动类，如代码清单 11-6 所示。

代码清单 11-6　mapred\hbase\ MapReduceReaderHbaseDerver.java

```
public  class  MapReduceReaderHbaseDerver{
    public static void main(String[] args) throws Exception {
     String tablename  = "wordcount";
     //实例化 Configuration，注意不能用 new HBaseConfiguration()
     Configuration conf = HBaseConfiguration.create();
      String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
      if (otherArgs.length != 2) {
```

```
            System.err.println("Usage: wordcount <in> <out>"+otherArgs.length);
            System.exit(2);
        }
        Job job = new Job(conf, "word count");
        job.setJarByClass(MapReduceReaderHbaseDerver.class);
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
        job.setReducerClass(WordCountReduceHbaseReader.class);

        // 此处的 TableMapReduceUtil 注意要用 hadoop.hbase.mapreduce 包中的
        // 而不是 hadoop.hbase.mapred 包中的
        Scan scan = new Scan(args[0].getBytes());
        TableMapReduceUtil.initTableMapperJob(tablename, scan,
                WordCountMapperHbaseReader.class, Text.class, Text.class, job);
        System.exit(job.waitForCompletion(true) ? 0 : 1);
        }
}
```

4）运行代码。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop jar hbase-mapreduce-reader.jar mapred.
hbase. MapReduceReaderHbaseDerver
/hbase/output/
```

5）查看结果。

```
jayliu@Ubuntu1:/opt/hadoop-1.0.3$ hadoop fs -text /hbase/output/paert-r-0000
    easy    1
    hadoop  2
    hello   1
      is    1
```

查看的结果与 11.6.3 节写入 HBase 里的如果一样，说明对 HBase 的读取数据成功了。

11.6.5 Avro、REST 和 Thrift 接口

HBase 提供了 Avro、REST 和 Thrift 接口。在使用 Java 以外的编程语言和 HBase 交互时，会用到这些接口。在所有情况下，Java 服务器上都会运行一个 HBase 客户端实例，它负责协调 Avro、REST 和 Thrift 的请求与 HBase 集群之间的双向交互。由于需要代理这些请求和响应工作，因此使用这些接口比直接使用 Java 接口要慢很多。

1. REST

启动一个 stargate 实例（stargate 是 HBase REST 服务的名称）可以运行下面的命令。

```
% hbase-daemon.sh start rest
```

这将启动一个服务器实例，默认情况下使用端口号 8080，在后台运行，并且捕捉 HBase logs 目录下日志文件中服务器的任务状态。

客户端可以要求响应以 JSON 格式、Google 的 protobuf 格式或 XML 格式的输出，采用哪种格式取决于客户端的 HTTP Accept 头的设置。

关闭 REST 服务命令如下。

```
% hbase-daemon.sh stop rest
```

2. Thrift

启动 Thrift 服务使用如下命令。

```
% hbase-daemon.sh start thrift
```

这将启动一个服务器实例，默认情况下使用端口号 9090，在后台运行，并且捕捉 HBase logs 目录下日志文件中服务器的任务状态。

要停止 Thrift 服务使用如下命令。

```
% hbase-daemon.sh stop thrift
```

3. Avro

启动 Avro 服务使用如下命令。

```
% hbase-daemon.sh start avro
```

这将启动一个服务器实例，默认情况下使用端口号 9090，在后台运行，并且捕捉 HBase logs 目录下日志文件中服务器任务状态。

要停止 Avro 服务使用如下命令。

```
% hbase-daemon.sh stop avro
```

11.7 本章小结

本章介绍了 HBase 的安装、HBase 的 Shell 操作、MapReduce 操作 HBase 等，HBase 是一个基于列存储的 NoSQL 数据库，不同于一般的 NoSQL 数据库，它是一个适合于非结构化数据存储的数据库。

第12章　Mahout 算法

Mahout 是 ASF（Apache Software Foundation）旗下的一个开源项目，提供一些可扩展的机器学习领域经典算法的实现，旨在帮助开发人员更加方便快捷地创建智能应用程序。Mahout 具有很强的扩展性，其主要部分是协同过滤（CF）、推荐引擎（recommender）、聚类（clustering）和分类（classification）。此外，通过使用 Apache Hadoop 库，Mahout 可以有效地扩展到云中。Mahout 备受关注还有一个原因，就是它收购了 Taste 框架，本章会对此框架进行简单介绍。

12.1　Mahout 的使用

Mahout 实现是用 Java 语言编写的，有些部分是建立在 Apache Hadoop 分布式计算项目上的。它是一个 Java Library，不提供用户接口。

12.1.1　安装 Mahout

Mahout 是运行在 Hadoop 服务之上的算法工具，安装 Mahout 的前提是 Hadoop 集群服务运行正常。Mahout 的安装非常简单，下面进行具体介绍。

1）下载 Mahout。Mahout 目前的版本为 0.8（本书使用 0.7 版本）。Mahout 的下载地址为 https://cwiki.apache.org/confluence/display/MAHOUT/Downloads，下载的文件名称为 mahout-distribution-0.7.tar.gz。

2）解压。把下载的 mahout-distribution-0.7.tar.gz 文件复制到 /opt 目录下并且解压，操作命令如下。

```
jayliu@Ubuntu1:/opt$ tar -zxvf mahout-distribution-0.7.tar.gz
```

解压成功以后，会在 /opt 目录下生成一个文件夹，名为 mahout-distribution-0.7。

3）配置环境变量。在 /etc/profile 文件中添加如下内容。

```
MAHOUT_HOME=/opt/mahout-distribution-0.7
PATH=$MAHOUT_HOME/bin:$PATH
CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$MAHOUT_HOME/lib:$JAVA_HOME/lib/tools.jar
export MAHOUT_HOME
```

12.1.2　运行一个 Mahout 案例

本节运行 Mahout 自带的 Kmeans 算法，同时验证安装的 Mahout 是否能够正常运行。具体步骤如下。

1）准备测试数据。下载文件 synthetic_control.data[⊖]并保存在 $MAHOUT_HOME 目录下。在这里打开网站直接显示文件的内容，解决办法是手动创建一个 synthetic_control.data 文件，复制一些数据到 synthetic_control.data 文件中。

2）启动 hadoop 集群。

```
jayliu@Ubuntu1:/opt$ start-all.sh
```

3）创建测试目录 testdata，并把数据文件 synthetic_control.data 导入这个 tastdata 目录中。注意，这里的目录名字只能是 testdata。

```
jayliu@Ubuntu1:/opt$ hadoop fs -mkdir testdata  #
jayliu@Ubuntu1:/opt$ hadoop fs - put /opt/mahout-distribution-0.7/synthetic_
control.datatestdata
```

4）运行 Kmeans 算法。

```
jayliu@Ubuntu1:/opt$hadoop jar /opt
/mahout-distribution-0.7/mahout-examples-0.7-job.jar org.apache.mahout.
clustering.syntheticcontrol.kmeans.Job
```

这个算法需要运行 5 分钟左右，启动 10 个 MapReduce Job 任务。运行生成的结果比较多，下面截取一小部分。

```
1.0: [35.844, 27.312, 25.803, 33.267, 23.822, 23.450, 23.362, 27.733, 28.103,
20.647, 23.134, 26.835, 22.623, 29.043, 24.388, 21.914, 23.800, 22.042, 20.880,
19.068, 21.175, 25.578, 17.327, 15.291, 20.405, 14.693, 22.309, 21.272, 18.508,
13.380, 20.993, 12.381, 14.007, 17.523, 21.281, 20.327, 16.246, 16.580, 8.345,
18.392, 12.150, 10.579, 9.443, 17.270, 11.130, 9.278, 7.688, 12.391, 6.051, 6.346,
15.007, 7.232, 7.100, 11.662, 6.824, 6.193, 10.506, 2.222, 4.652, 8.999]
1.0: [32.737, 34.652, 33.941, 28.376, 31.975, 22.705, 24.932, 23.720, 30.326,
25.785, 27.381, 27.536, 19.481, 18.693, 20.573, 23.884, 22.825, 22.634, 21.796,
25.042, 21.671, 21.172, 15.079, 16.058, 14.822, 17.689, 20.970, 17.672, 11.787,
15.136, 14.409, 15.313, 18.687, 17.696, 17.492, 13.232, 9.704, 15.655, 14.007, 9.606,
15.320, 4.980, 12.540, 7.290, 7.326, 2.875, 1.293, 12.469, 0.641, 2.545, 1.080,
3.530, 3.801, 8.919, 7.789, 2.897, -2.516, 2.294, 6.691, 5.528]
1.0: [28.027, 33.812, 23.806, 32.976, 32.005, 30.910, 27.652, 32.209, 28.175,
21.076, 26.416, 29.944, 25.300, 23.878, 22.361, 22.991, 28.164, 21.284, 26.487,
17.396, 15.812, 18.417, 22.165, 24.718, 14.396, 25.079, 22.057, 19.129, 22.816,
21.612, 15.630, 18.155, 12.663, 14.733, 19.729, 10.677, 13.345, 18.290, 17.492,
15.353, 10.545, 16.460, 10.094, 17.122, 17.863, 9.039, 14.747, 4.801, 13.672, 3.828,
4.813, 13.870, 6.790, 7.633, 10.502, 12.385, 7.230, 1.212, 3.007, 3.409]
```

5）查看结果。

```
jayliu@Ubuntu1:/opt/mahout-distribution-0.7$ n-0.7$ fs -ls output
Found 14 items
-rw-r--r-- 1 jayliusupergroup 194 2013-08-20 15:54 /user/jayliu/output/_policy
drwxr-xr-x - jayliusupergroup 0 2013-08-20 15:54 /user/jayliu/output/clusteredPoints
drwxr-xr-x - jayliusupergroup 0 2013-08-20 15:44 /user/jayliu/output/clusters-0
drwxr-xr-x - jayliusupergroup 0 2013-08-20 15:45 /user/jayliu/output/clusters-1
```

⊖ 下载地址为 http://archive.ics.uci.edu/ml/databases/synthetic_control/synthetic_control.data。

```
drwxr-xr-x - jayliusupergroup 0 2013-08-20 15:54 /user/jayliu/output/clusters-10-final
drwxr-xr-x - jayliusupergroup 0 2013-08-20 15:46 /user/jayliu/output/clusters-2
drwxr-xr-x - jayliusupergroup 0 2013-08-20 15:48 /user/jayliu/output/clusters-3
drwxr-xr-x - jayliusupergroup 0 2013-08-20 15:49 /user/jayliu/output/clusters-4
drwxr-xr-x - jayliusupergroup 0 2013-08-20 15:50 /user/jayliu/output/clusters-5
drwxr-xr-x - jayliusupergroup 0 2013-08-20 15:51 /user/jayliu/output/clusters-6
drwxr-xr-x - jayliusupergroup 0 2013-08-20 15:51 /user/jayliu/output/clusters-7
drwxr-xr-x - jayliusupergroup 0 2013-08-20 15:52 /user/jayliu/output/clusters-8
drwxr-xr-x - jayliusupergroup 0 2013-08-20 15:53 /user/jayliu/output/clusters-9
drwxr-xr-x - jayliusupergroup 0 2013-08-20 15:44 /user/jayliu/output/data
```

到这里 Mahout 的安装和第一个使用案例就介绍完了。Mahout 是一个非常优秀的算法工具，对开发者来说非常方便。接下来介绍 Mahout 中的一些经典算法。

12.2 Mahout 数据表示

推荐数据的处理是大规模的，在集群环境下一次要处理的数据可能是数 GB 级甚至 TB 级，所以 Mahout 针对推荐数据进行了优化。Mahout 数据表示常用的两种形式为偏好（Perference）和数据模型（Data Model）。

12.2.1 偏好 Perference 类

偏好是一个整数（Long），这个值越大说明用户越喜欢。用户的偏好被抽象为一个 Preference，包含 userID、itemID 和偏好值（user 对 item 的偏好）。Preference 是一个接口，它有一个通用的实现类 GenericPreference。类关系如图 12-1 所示。

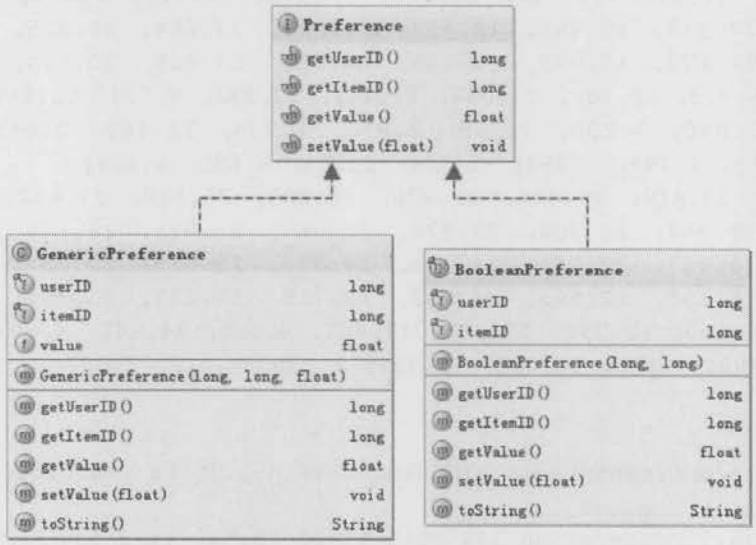


图 12-1 Preference 类图

因为用户的偏好数据是大规模的，所以通常会选择把它放入集合或者数组。但是由于 Java 对象的内存消耗机制，在大数据量下使用 Collection<Preference> 和 Preference[ ] 是非常低

效的。Mahout 封装了一个 PreferenceArray 接口，用于保存一组用户偏好数据。为了优化性能，Mahout 提供两个实现类 GenericUserPreferenceArray 和 GenericItemPreferenceArray，分别按照用户和物品本身对用户偏好进行组装，这样就可以压缩用户 ID 或者物品 ID 的空间。PreferenceArray 接口及实现类如图 12-2 所示。

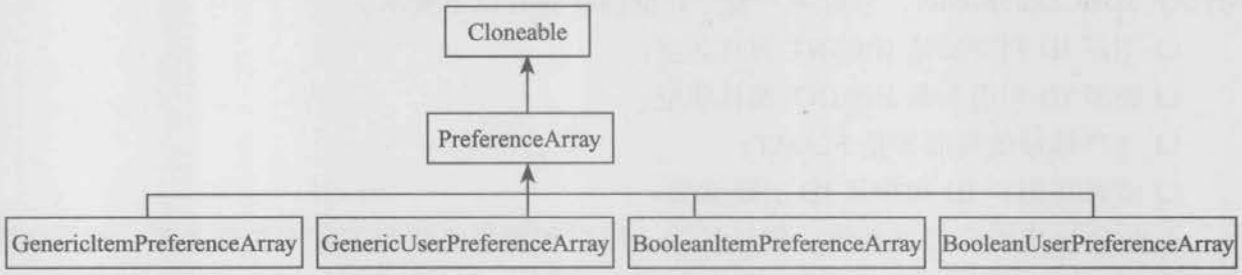


图 12-2 PreferenceArray 类关系

GenericUserPreferenceArray 包含一个 userID、一个 itemID 的数组 long[]、一个用户的偏好评分数据 float[]，而不是一个 Preference 对象的集合，它只有较少的对象需要被创建。

除了 PreferenceArray，Mahout 中还大量使用 Map 和 Set 等典型数据结构，但是 Mahout 没有直接使用 HashMap 和 TreeSet 等常用的 Java 集合实现，取而代之是专门为 Mahout 推荐的需要实现两个 API：FastByIDMap 和 FastIDSet。专门封装这两个数据结构，主要目的是减少内存开销，提高性能。

12.2.2 数据模型 DataModel 类

Mahout 推荐引擎实际接受的输入是 DataModel，它是对用户偏好数据的压缩表示。DataModel 的具体实现支持从任意类型的数据源抽取用户偏好信息，可以很容易地返回输入的偏好数据中关联到一个物品的用户 ID 列表和 count 计数，以及输入数据中所有用户和物品的数量。具体实现包括内存版的 GenericDataModel，支持文件读取的 FileDataModel 和支持数据库读取的 JDBCDataModel。DataModel 类关系如图 12-3 所示。

GenericDataModel 是 DataModel 的内存版实现，适用于在内存中构造推荐数据，它仅作为推荐引擎的输入接受用户的偏好数据，保存着一个按照用户 ID 和物品 ID 进行散列的 PreferenceArray，而 PreferenceArray 中对应保存该用户 ID 或物品 ID 的所有用户偏好数据。

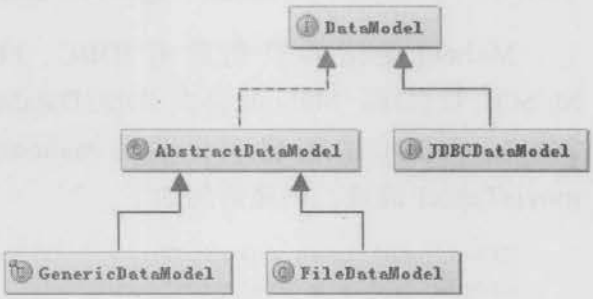


图 12-3 DataModel 类关系

FileDataModel 支持文件的读取。Mahout 对文件的格式没有太多严格的要求，只要满足以下格式就可以。

- 每一行包含一个用户 ID、物品 ID、用户偏好；
- 逗号隔开或 Tab 隔开；

❑ *.zip 和 *.gz 文件会自动解压缩（Mahout 建议在数据量过大时采用压缩的数据存储）。

FileDataModel 从文件中读取数据，然后将数据以 GenericDataModel 的形式载入内存，具体可以查看 FileDataModel 中的 buildModel 方法。

JDBCDataModel 支持对数据库的读取操作。Mahout 提供了对 MySQL 的默认支持 MySQLJDBCDataModel，它对用户偏好数据的存储有以下要求。

❑ 用户 ID 列需要是 BIGINT 而且非空；

❑ 物品 ID 列需要是 BIGINT 而且非空；

❑ 用户偏好值列需要是 FLOAT；

❑ 建议在用户 ID 和物品 ID 上建索引。

有时会忽略用户的偏好值，仅仅只关心用户和物品之间存不存在关联关系，这种关联关系在 Mahout 中叫做“boolean preference”。之所以有这类偏好，因为用户和物品的关联要么存在，要么不存在，记住只是表示关联关系存不存在，不代表喜欢和不喜欢。实际上一条“boolean preference”可有三个状态：喜欢、不喜欢、没有任何关系。

在偏好数据中有大量噪声数据的情况下，这种特殊的偏好评定方式是有意义的。同时 Mahout 为“boolean preference”提供一个内存版的实现类 GenericBooleanPrefDataModel，方法如图 12-4 所示。

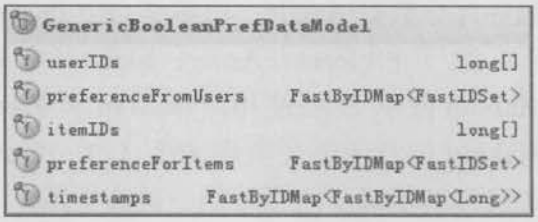


图 12-4 GenericBooleanPrefDataModel 方法

可以看到，GenericBooleanPrefDataModel 没有对偏好值进行存储，仅仅存储了关联的 userID 和 itemID。注意和 GenericDataModel 的差别，GenericBooleanPrefDataModel 采用 FastIDSet，只有关联的 ID，没有偏好值。因此它的一些方法（继承自 DataModel）如 getItemIDsForUser() 有更好的执行速度，而 getPreferencesFromUser() 的执行速度会更差，因为 GenericBooleanPrefDataModel 本来就没存储偏好值，默认用户对物品的偏好值都是 1.0。

12.2.3 Mahout 链接 MySQL 数据库

Mahout 支持的数据源有 JDBC、MongoDB 和 Cassandra，本节介绍 Mahout 操作 MySQL 数据库。Mahout 提供 JDBCDataModel 接口用来操作 MySQL 数据库。

在 MySQL 中创建一个名为 mahoutDB 的数据库，在该数据库中创建一个名为 movieTable1 的表。创建语句如下。

```
Create database mahoutDB;
create table movietable1(
    user_idvarchar(100) not null ,
    item_idvarchar(100) not null,
    preferencevarchar(100) not null,
    time_datevarchar(100) default null
);
```

查看 movieTable1 表的数据如图 12-5 所示。

```
select * from movietable1 where user_id=1;
```

user_id	item_id	preference	time_date
1	101	5.0	20130823
1	102	3.0	20130823
1	103	2.5	20130823

图 12-5 movieTable1 表的数据

Mahout 使用 JDBCDataModel 数据源链接 MySQL 数据，并使用 movieTable1 表里的数据进行推荐分析。实现代码如代码清单 12-1 所示。

代码清单 12-1 mahout\ MahoutJdbcMysql.java

```
packagecom mahout;
importjava.util.List;
importorg.apache.mahout.cf.taste.common.TasteException;
import org.apache.mahout.cf.taste.impl.neighborhood.NearestNUserNeighborhood;
import org.apache.mahout.cf.taste.impl.recommender.GenericUserBasedRecommender;
import org.apache.mahout.cf.taste.impl.similarity.PearsonCorrelationSimilarity;
importorg.apache.mahout.cf.taste.model.DataModel;
importorg.apache.mahout.cf.taste.model.JDBCDataModel;
importorg.apache.mahout.cf.taste.neighborhood.UserNeighborhood;
importorg.apache.mahout.cf.taste.recommender.RecommendedItem;
importorg.apache.mahout.cf.taste.recommender.Recommender;
importorg.apache.mahout.cf.taste.similarity.UserSimilarity;

import com.mysql.jdbc.jdbc2.optional.MysqlDataSource;
public class MahoutJdbcMysql {
public static  void main(String[] args) throws TasteException {
    // TODO Auto-generated method stub
    long t1=System.currentTimeMillis();
    MysqlDataSource dataSource=new MysqlDataSource(); ----------------------(1)
    dataSource.setServerName("localhost");
    dataSource.setUser("root");
    dataSource.setPassword("root");
    dataSource.setDatabaseName("mahoutDB");
    JDBCDataModel dataModel=new MySQLJDBCDataModel(dataSource,"movieTable1",
                    "user_id","item_id","preference","time_date");-----(2)
    DataModel model=dataModel;  ----------------------------------------(3)
    UserSimilarity similarity=new PearsonCorrelationSimilarity(model);
    UserNeighborhood neighborhood=new NearestNUserNeighborhood(2,similarity,model);
    Recommender recommender=new GenericUserBasedRecommender(model,neighborhood,
                    similarity);
```

```
        List<RecommendedItem> recommendations=recommender.recommend(1, 2);
        for(RecommendedItemrecommendation:recommendations){
            System.out.println(recommendation);
            }
        }
    }
```

代码分析如下：

1）创建一个 MySQLDataSource 的对象，设置用户名和密码来链接 MySQL 数据库。

2）使用 MySQLJDBCDataModel 数据源来读取 MySQL 里面的数据。

3）把 MySQLJDBCDataModel 对象赋值给 DataModel。

12.3 认识 Taste 框架

Apache Mahout 中提供一个协同过滤算法的高效实现，协同过滤算法的框架为 Taste。Taste 是一个基于 Java 实现的可扩展的高效推荐引擎。利用用户对已接触内容的评价值来推测用户尚未接触的内容的评价值，并以此作为推荐的依据。Taste 提供一组丰富的组件和算法让你根据项目的需要进行组合，从而设置满足项目的推荐功能。虽然基于 Java 开发，但是并不仅限于应用在基于 Java 开发的应用系统中，还可以通过 Web Services 以及 HTTP 等向不同的业务系统提供推荐逻辑。图 12-6 给出了 Taste 中协同过滤推荐实现的组件。

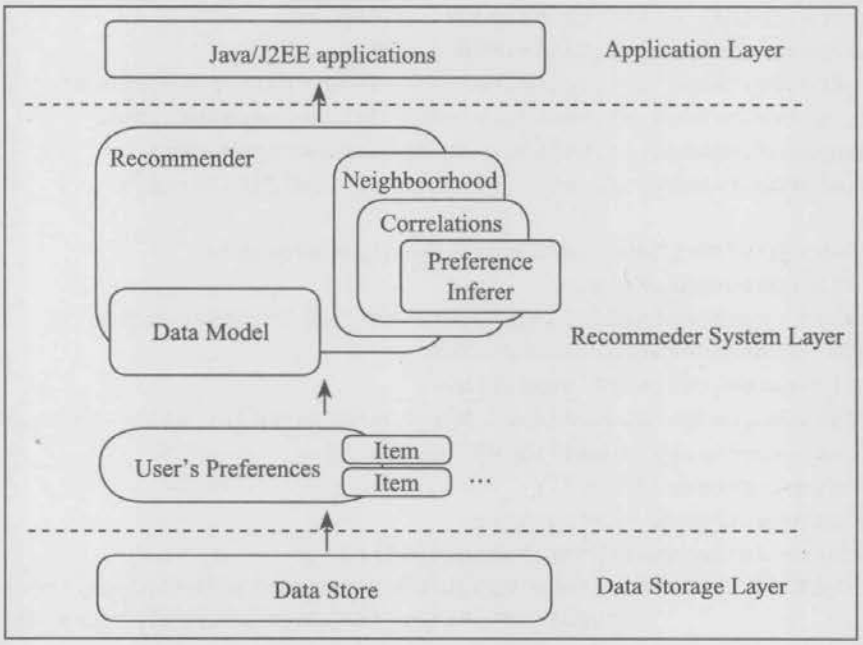


图 12-6 Taste 架构组件

Taste 推荐算法组件分三层：数据层（Data Storage Layer）、推荐系统层（Recommender System Layer）和应用层（Application Layer）。推荐系统层又包含以下组件。

- Recommender：推荐引擎的抽象接口，Taste 中的核心组件。在程序中，为它提供一个 DataModel，可以计算出对不同用户的推荐内容。实际应用中，主要使用它的实现类 GenericUserBasedRecommender 或者 GenericItemBasedRecommender，分别实现基于用户相似度的推荐引擎或者基于内容的推荐引擎。
- DataModel：用户偏好信息的抽象接口，具体实现支持从任意类型的数据源抽取用户偏好信息。Taste 默认提供 JDBCDataModel 和 FileDataModel，分别支持从数据库和文件中读取用户的偏好信息。
- User's Preferences：用户偏好。

12.4 Mahout 推荐器

Mahout 算法框架提供了很多推荐器供开发者使用，Mahout 自带的推荐器如下所示。

- GenericUserBasedRecommender：基于用户的推荐器，用户数量相对较少时速度较快。
- GenericItemBasedRecommender：基于商品的推荐器，商品数量较少时速度较快，外部提供了商品相似度数据后会更加有效率。
- SlopeOneRecommender：基于 slope-one 算法的推荐器，在线推荐或更新比较快，需要先大量的预处理运算。物品数量相对较少时使用比较合适。
- SVDRecommender：效果不错，和 slope-one 一样，事先需要大量的预处理运算。
- KnnItemBasedRecommender：基于最近邻算法的推荐器，物品数量较小时表现良好。
- TreeClusteringRecommender：基于聚类的推荐器，在线推荐较快，同时也需要事先大量预处理运算，用户数量相对较少时表现良好。

接下来介绍常用的三个推荐器。

12.4.1 基于用户的推荐器

GenericUserBasedRecommender 是基于用户（user-based）的简单推荐器实现类，按照传入的 DataModel 和 UserNeighborhood 进行推荐。其推荐流程分成三步。

1）应用 UserNeighborhood 获取与指定用户 U_i 最类似的 K 个用户 $\{U_1 \cdots U_k\}$；

2）从 $\{U_1 \cdots U_k\}$ 偏好的 item 集合中去掉 U_i 偏好的 item，获得一个 item 集合 $\{Item_0 \cdots Item_m\}$；

3）对 $\{Item_0 \cdots Item_m\}$ 每个 $Item_j$ 计算 U_i 可能偏好的程度值 $perf(U_i, Item_j)$，并把 item 按这个数值从高到低排序，把前 N 个 item 推荐给 U_i。

$perf(U_i, Item_j)$ 计算公式如下所示。其中 $p(U_l, Item_j)$ 是用户 U_l 对 $Item_j$ 的偏好值。

$$perf(U_i, Item_j) = \frac{\sum_{l=1}^{k} sim(U_l, U_i) \times p(U_l, Item_j)}{\sum_{l=1}^{k} sim(U_l, U_i)}$$

接下来介绍使用 GenericUserBasedRecommender 推荐器实现一个物品推荐的案例。创

建一个 userbase.csv 文件，文件内容包含 user、item、rating 三个属性，如代码清单 12-2 所示。

代码清单 12-2 mahout\ GenericUserBasedRecommenderInfo.java

```
package mahout;
import org.apache.mahout.cf.taste.impl.model.file.*;
import org.apache.mahout.cf.taste.impl.neighborhood.*;
import org.apache.mahout.cf.taste.impl.recommender.*;
import org.apache.mahout.cf.taste.impl.similarity.*;
import org.apache.mahout.cf.taste.model.*;
import org.apache.mahout.cf.taste.neighborhood.*;
import org.apache.mahout.cf.taste.recommender.*;
import org.apache.mahout.cf.taste.similarity.*;
import java.io.*;
import java.util.*;
public class GenericUserBasedRecommenderInfo{
public static void main(String[] args) throws Exception {
  DataModel model = new FileDataModel(new File("userbase.csv"));   ----------(1)
  UserSimilarity similarity = new PearsonCorrelationSimilarity(model);  ------(2)
  UserNeighborhood neighborhood =
        new NearestNUserNeighborhood(2, similarity, model);
  Recommender recommender =
        new GenericUserBasedRecommender( model, neighborhood, similarity); ------(3)
  List<RecommendedItem> recommendations = recommender.recommend(1, 1);  ------(4)
  for (RecommendedItem recommendation : recommendations) {        ----------(5)
      System.out.println(recommendation);
  }
 }
}
```

下面对代码进行解释。

1）使用 DataModel 数据模型加载 userbase.csv 里的数据，加载完成以后 DataModel 包含 user、item、rating 的数据源。

2）使用 UserSimilarity 构建一个用户相似度，接受一个 DataModel 数据源。用户相似度包含相似性度量和邻居的参数。

3）一旦确定相邻用户，一个普通的 user-based 推荐器就可以构建起来，而构建一个 GenericUserBasedRecommender 推荐器需要数据源（DataModel）、用户相似性（UserSimilarity）和相邻用户相似度（UserNeighborhood）。

4）向用户 1 推荐 1 个商品，recommender.recommend(1, 1) 方法的参数表示用户 ID 和推荐商品的个数。

5）输出推荐结果。如 RecommendedItem[item:104, value:4.257081]。

12.4.2 基于项目的推荐器

GenericItemBasedRecommender 是基于项目（item-based）的简单推荐器，根据传入的 DateModel 和 ItemSimilarity 推荐。与 user-based 相比，item-based 的相似度计算有个好处，

item 数量较少，计算量也就少；另外 item 之间的相似度比较固定，相似度可以事先算好，这样可以大幅提高推荐的速度。

Generic ItemBasedRecommender 的推荐流程可以分成三步。

1）获取用户 U_i 偏好的 item 集合 $\{It_1 \cdots It_m\}$。

2）使用 MostSimilarItemsCandidateItemsStrategy（功能类似 UserNeighborhood）获取与用户偏好集合中每个 item 最相似的其他 item 构成集合 $\{Item_1 \cdots Item_k\}$；

3）对 $\{Item_1 \cdots Item_k\}$ 中的每个 $item_j$ 计算 U_i 可能喜欢的程度值 $perf(U_i, Item_j)$，并把 item 按这个数值从高到低排序，把前 N 个 item 推荐给 U_i。

$perf(U_i, Item_j)$ 的计算公式如下所示。其中 $p(U_i, It_l)$ 是用户 U_i 对 $Item_l$ 的偏好值。

$$perf(U_i, Item_j) = \frac{\sum_{l=1}^{m} sim(It_l, Item_j) \times p(U_i, It_l)}{\sum_{l=1}^{m} sim(It_l, Item_j)}$$

接下来分析一个 GenericItemBasedRecommender 项目推荐案例，如代码清单 12-3 所示。

代码清单 12-3 mahout\ GenericItemBasedRecommender.java

```
package mahout;
import org.apache.mahout.cf.taste.impl.model.file.*;
import org.apache.mahout.cf.taste.impl.recommender.*;
import org.apache.mahout.cf.taste.impl.similarity.*;
import org.apache.mahout.cf.taste.model.*;
import org.apache.mahout.cf.taste.neighborhood.*;
import org.apache.mahout.cf.taste.recommender.*;
import org.apache.mahout.cf.taste.similarity.*;
import java.io.*;
import java.util.*;
public class GenericItemBasedRecommender  {
  public static void main(String[] args) throws Exception{
    DataModel model;
    model = new FileDataModel(new File("itembase.csv"));
    ItemSimilarity itemsimilarity =new PearsonCorrelationSimilarity(model);
    Recommender recommender= new GenericItemBasedRecommender(model,itemsimilarity);
    List<RecommendedItem> recommendations =recommender.recommend(1, 1);
    for(RecommendedItem recommendation :recommendations){
          System.out.println(recommendation);
    }
  }
}
```

基于项目推荐的实现和基于用户推荐非常类似，只是所用的推荐器不一样，基于项目推荐使用的是 GenericItemBasedRecommender 推荐器，而基于用户推荐使用的是 GenericUserBasedRecommender 推荐器。

12.4.3 Slope One 推荐策略

前面介绍的 GenericUserBasedRecommender 和 GenericItemBasedRecommender 是最容

易理解的两种推荐策略，但在数据量较大时，计算量会很大，导致推荐效率较差。因此 Mahout 还提供了一种更加轻量级的推荐策略 Slope One（Daniel Lemire 和 Anna Maclachlan 在 2005 年提出的基于评分协同过滤推荐引擎的改进方法）。

假设系统对物品 A，物品 B 和物品 C 的平均评分分别是 3，4 和 4（如图 12-7 所示）。

用户 / 物品	物品 A	物品 B	物品 C
平均打分	3	4	4
用户 A 打分	4	?	?
用户 B 打分	2	?	4

Slope One 推理：
对物品 B 的评分 = 对物品 A 的评分 +1
对物品 B 的评分 = 对物品 C 的评分

用户 / 物品	物品 A	物品 B	物品 C
平均打分	3	4	4
用户 A 打分	4	5	5
用户 B 打分	2	3.5	4

图 12-7 Slope One 推荐策略示例

基于 Slope One 的方法会得到以下规律。

- 用户对物品 B 的评分 = 用户对物品 A 的评分 + 1；
- 用户对物品 B 的评分 = 用户对物品 C 的评分。

基于以上规律，可以对用户 A 和用户 B 的打分进行预测。

- 用户 A，给物品 A 打分 4，可以推测对物品 B 的评分是 5，对物品 C 的打分也是 5。
- 用户 B，给物品 A 打分 2，给物品 C 打分 4，根据第一条规律，可以推断他对物品 B 的评分是 3；而根据第二条规律，推断评分是 4。当出现冲突时，可以对各种规则得到的推断进行取平均值，所以给出的推断是 3.5。

这就是 Slope One 推荐的基本原理，将用户评分之间的关系看做简单的线性关系，$Y = mX + b$，当 $m = 1$ 时就是 Slope One，也就是刚刚展示的例子。

Slope One 的核心优势是，对于大规模数据依然能保证良好的计算速度和推荐效果。Mahout 提供了 Slope One 推荐方法的基本实现，很简单，参考代码清单 12-4。

代码清单 12-4 mahout\ SlopeOneRecommenderInfo.java

```
package mahout;
import java.io.File;
import java.io.IOException;
import org.apache.mahout.cf.taste.common.TasteException;
import org.apache.mahout.cf.taste.common.Weighting;
import org.apache.mahout.cf.taste.eval.RecommenderBuilder;
import org.apache.mahout.cf.taste.eval.RecommenderEvaluator;
```

```
import  org.apache.mahout.cf.taste.impl.eval.
          AverageAbsoluteDifferenceRecommenderEvaluator;
import org.apache.mahout.cf.taste.impl.model.file.FileDataModel;
import org.apache.mahout.cf.taste.impl.recommender.slopeone.MemoryDiffStorage;
import org.apache.mahout.cf.taste.impl.recommender.slopeone.SlopeOneRecommender;
import org.apache.mahout.cf.taste.model.DataModel;
import org.apache.mahout.cf.taste.recommender.Recommender;
import org.apache.mahout.cf.taste.recommender.slopeone.DiffStorage;
import org.apache.mahout.common.RandomUtils;
public class SlopeOneRecommenderInfo{
  public static void recommenderModelEvaluation(DataModel model)
       throws TasteException {
  RecommenderEvaluator evaluator = new AverageAbsoluteDifferenceRecommenderEvaluator();
  RandomUtils.useTestSeed();
  RecommenderBuilder builder = new RecommenderBuilder() {
    long diffStorageNb = 100000;
    @Override
  public Recommender buildRecommender(DataModel dm) throws TasteException {
      DiffStorage diffStorage = new MemoryDiffStorage(dm, Weighting.WEIGHTED,
              diffStorageNb);
      return new SlopeOneRecommender(dm, Weighting.WEIGHTED, Weighting.WEIGHTED,
              diffStorage);
    }
  };
}
public static void main(String[] args) throws IOException, TasteException {
  DataModel model = new FileDataModel(new File("data/dating/ratings.dat"));
  recommenderModelEvaluation(model);
 }
}
```

Slope One 是一种非常快速简单的基于项目的推荐方法，只需要使用用户的评分信息。具体的实现，只需要在推荐引擎中包含一个 SlopeOneRecommender 实例，创建实例的过程如下。

1）根据 DataModel 创建数据之间线性关系的模型 DiffStorage。

2）基于 DataModel 和 DiffStorage 创建 SlopeOneRecommender，实现 Slope One 推荐策略。

12.5 推荐系统

推荐系统是指根据分析用户的行为来实现个性化。用户的行为包含很多，如用户性别、年龄段、偏好、职业等。

12.5.1 个性化推荐

个性化推荐是指根据用户的兴趣特点和购买行为，向用户推荐用户感兴趣的信息和商

品。随着电子商务规模的不断扩大，商品个数和种类快速增长，顾客需要花费大量的时间才能找到自己想买的商品。这种浏览大量无关的信息和产品的过程无疑会使淹没在信息过载问题中的消费者不断流失。为了解决这些问题，个性化推荐系统应运而生。

个性化推荐系统是建立在海量数据挖掘基础上的一种高级商务智能平台，以帮助电子商务网站为其顾客购物提供完全个性化的决策支持和信息服务。例如，登录购物网站的时候，购物网站会根据你以前浏览或购买的商品，推荐你可能喜欢的商品列表；登录某 SNS 社区网站时，会根据注册信息（如学校、专业等）推荐你可能认识的人。随着互联网的快速发展，推荐系统在海量信息中把信息定制化地反馈给用户。

电子商务推荐系统的界面表现形式有以下几种。

- 浏览：客户提出对特定商品的查询要求，推荐引擎根据查询要求返回高质量的推荐。
- 相似商品：推荐引擎根据客户购物篮中的商品和客户可能感兴趣的商品，推荐类似的商品。
- E-mail：推荐系统通过电子邮件的方式通知客户可能感兴趣的商品信息。
- 评论：推荐系统向客户提供其他客户对相应产品的评论信息。

12.5.2 商品推荐系统案例

举个买书的例子，user 偏好包含一个 user ID 和一个 item ID，在通常情况下，一个数值代表 user 对 item 偏好的强度。Mahout 中 ID 都是数值，实际上是整数。user 偏好值可以是任何值，值越大代表正向偏好越大。例如，这些值是 1 ～ 5，1 表示 user 不喜欢这个商品，5 表示非常喜欢。接下来详细分析这个案例。

1. 准备数据

创建一个 useritem.txt 文件，文件内容每行有三个属性：userID、itemID 和 perfValue。

userID 表示 user 的 ID, itemID 表示一本书的编号，perfValue 表示 user 对这本书的偏好值。文件数据如下。

```
userID,itemID,perfValue
1,101,5.0
1,102,3.0
1,103,2.5
2,101,2.0
2,102,2.5
2,103,5.0
2,104,2.0
3,101,2.5
3,104,4.0
3,105,4.5
3,107,5.0
4,101,5.0
```

```
4,103,3.0
4,104,4.5
4,106,4.0
5,101,4.0
5,102,3.0
5,103,2.0
5,104,4.0
5,105,3.5
5,106,4.0
```

第一列为 userID，第二列为 itemID，第三列为 prefValue 即评分。

2. 分析关系

经过分析之后，趋势就显现出来。user 1 和 user 5 具有相同的兴趣，他们都喜欢 101 这本书。同理，user 1 和 user 4 具有相同的兴趣，他们都喜欢 101，没有信息显示 user 4 喜欢 102。另一方面，user 1 和 user 2 的兴趣正好相反，user 1 喜欢 101，但 user 2 不喜欢 101，user 1 喜欢 103 而 user 2 相反。user 1 和 user 3 的交集很少，只有 101 这本书显示他们的兴趣相同。图 12-8 是 user 和 item 之间的关系，可能是正的也可能是负的。

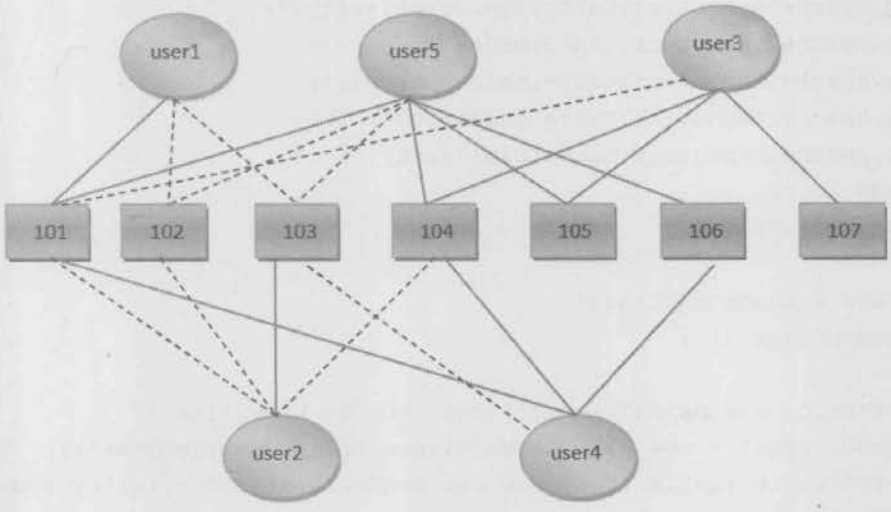


图 12-8 user 和 item 之间的关系

3. 代码实现

应该向 user 1 推荐哪两本书呢？不是 101、102 或者 103，因为 user 1 已经知道自己对它们感兴趣，推荐系统需要发现新的事物。直觉告诉我们，user 4、user 5 与 user 1 类似，所以推荐一些 user 4 和 user 5 喜欢的书籍给 user 1 是不错的。这样使得 104、105 和 106 成为可能的推荐。整体上看，104 是最有可能的一个推荐，这基于 item 104 的偏好打分 4.5 和 4.0。实现步骤如下。

1）用 Eclipse 创建一个 Mahout 工程。

详细的创建过程前面已经讲解，这里主要了解需要的第三方 JAR 文件，如图 12-9 所示。

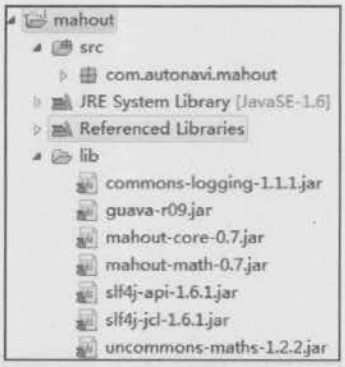


图 12-9 Mahout 工程

2）编写推荐类 RecommenderClass.java，如代码清单 12-5 所示。

代码清单 12-5 mahout\RecommenderClass.java

```
package mahout;
importorg.apache.mahout.cf.taste.impl.model.file.*;
importorg.apache.mahout.cf.taste.impl.neighborhood.*;
importorg.apache.mahout.cf.taste.impl.recommender.*;
importorg.apache.mahout.cf.taste.impl.similarity.*;
importorg.apache.mahout.cf.taste.model.*;
importorg.apache.mahout.cf.taste.neighborhood.*;
importorg.apache.mahout.cf.taste.recommender.*;
importorg.apache.mahout.cf.taste.similarity.*;
import java.io.*;
importjava.util.*;

public class RecommenderClass{
  RecommenderClass () {
  }
  public static void main(String[] args) throws Exception {
    DataModel model = new FileDataModel(new File("useritem.txt"));
    UserSimilarity similarity = new PearsonCorrelationSimilarity(model);
    UserNeighborhood neighborhood =
                    new NearestNUserNeighborhood(2,similarity, model);
    Recommender recommender =
                    new GenericUserBasedRecommender(model,neighborhood, similarity);
    List<RecommendedItem> recommendations = recommender.recommend(1, 2);
    // 为用户 1 推荐两个 ItemID
    for (RecommendedItem recommendation : recommendations) {
        System.out.println(recommendation);
    }
  }
}
```

3）把项目保存为 JAR 文件，如图 12-10 所示。

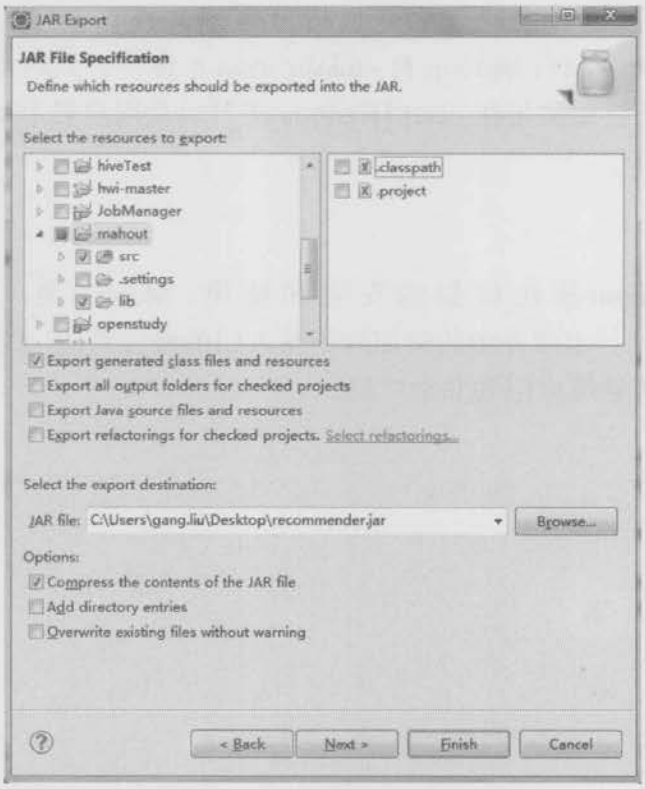


图 12-10 把 Mahout 工程保存为 JAR 文件

这里要注意，必须把 lib 目录下的 JAR 依赖文件也保存到 recommender.jar 中。

4）把 recommender.jar 文件复制到 $MAHOUT_HOME 下以后，运行代码。

```
    jayliu@Ubuntu1:/opt/mahout-distribution-0.7$ hadoop jar recommender.jar
com.autonavi.mahout.RecommenderIntro /user/hadoop/output21
    13/08/20 19:03:43 INFO file.FileDataModel: Creating FileDataModel for file
useritem.txt
    13/08/20 19:03:44 INFO file.FileDataModel: Reading file info...
    13/08/20 19:03:44 INFO file.FileDataModel: Read lines: 21
    13/08/20 19:03:44 INFO model.GenericDataModel: Processed 5 users
    RecommendedItem[item:104, value:4.257081]
    RecommendedItem[item:106, value:4.0]
```

通过运行结果可以看出，为 user 1 推荐的两本书分别为 104 和 106，至此一个完整的用户推荐案例就全部实现了。

以下代码是读取文件的操作。

```
    DataModel model = new FileDataModel(new File("useritem.txt"));
```

其中 useritem.txt 放在 HDFS 的 /user/jayliu/ 下。文件读取的路径要写成以下形式。

```
    new File("useritem.txt")
```

而不能写成以下形式。

```
    new File("/user/jayliu/useritem.txt")
```

还有个问题，有的读者会问怎么知道把分析的文件 useritem.txt 放到 /user/jayliu/ 下呢？这里介绍一个技巧，执行 hadoop fs –mkdir aaaa # 命令，然后去 HDFS 中查找 aaaa 目录，看它在哪个目录下。通常是在 /user/[Hadoop 集群启动用户名]/aaaa。

12.6 本章小结

本章介绍了 Mahout 算法框架的安装和使用，Mahout 是一个非常优秀的基于 MapReduce 的算法框架。本章介绍的常用知识是入门的最佳实战。Mahout 工具在商品推荐、电影推荐、广告推荐等领域应用的非常广泛。

附录A　Hive 内置操作符与函数

A.1　关系运算

Hive 支持的关系运算符，我们使用 Hive 自带的 dual 表，Hive 中也存在和 MySQL、Oracle 类似的 dual 虚拟表。常见的关系运算符如下。

1. 等值比较

语　法	A = B
操作类型	所有基本类型
说　明	如果表达式 A 与表达式 B 相等，则为 TRUE；否则为 FALSE
示　例	`hive> SELECT 1 FROM dual WHERE 1=1;`

2. 不等值比较

语　法	A <> B
操作类型	所有基本类型
说　明	如果表达式 A 为 NULL，或者表达式 B 为 NULL，返回 NULL；如果表达式 A 与表达式 B 不相等，则为 TRUE；否则为 FALSE
示　例	`hive> SELECT 1 FROM dual WHERE 1 < > 2;`

3. 小于比较

语　法	A < B
操作类型	所有基本类型
说　明	如果表达式 A 为 NULL，或者表达式 B 为 NULL，返回 NULL；如果表达式 A 小于表达式 B，则为 TRUE；否则为 FALSE
示　例	`hive> SELECT 1 FROM dual WHERE 1 < 2;`

4. 小于等于比较

语　法	A <= B
操作类型	所有基本类型
说　明	如果表达式 A 为 NULL，或者表达式 B 为 NULL，返回 NULL；如果表达式 A 小于或者等于表达式 B，则为 TRUE；否则为 FALSE
示　例	`hive> SELECT 1 FROM dual WHERE 1 <= 1;`

5. 大于比较

语　法	A > B
操作类型	所有基本类型

（续）

说　明	如果表达式 A 为 NULL，或者表达式 B 为 NULL，返回 NULL；如果表达式 A 大于表达式 B，则为 TRUE；否则为 FALSE
示　例	hive> SELECT 1 FROM dual WHERE 1 > 1;

6. 大于等于比较

语　法	A >= B
操作类型	所有基本类型
说　明	如果表达式 A 为 NULL，或者表达式 B 为 NULL，返回 NULL；如果表达式 A 大于或者等于表达式 B，则为 TRUE；否则为 FALSE
示　例	hive> SELECT 1 FROM dual WHERE 1 >= 1;

7. 空值判断

语　法	A IS NULL
操作类型	所有类型
说　明	如果表达式 A 的值为 NULL，则为 TRUE；否则为 FALSE
示　例	hive> SELECT 1 FROM dual WHERE NULL is NULL;

8. 非空判断

语　法	A IS NOT NULL
操作类型	所有类型
说　明	如果表达式 A 的值为 NULL，则为 FALSE；否则为 TRUE
示　例	hive> SELECT 1 FROM dual WHERE 1 IS NOT NULL;

9. LIKE 比较

语　法	A LIKE B
操作类型	strings
说　明	如果字符串 A 或者字符串 B 为 NULL，则返回 NULL；如果字符串 A 符合表达式 B 的正则语法，则为 TRUE；否则为 FALSE。B 中字符“_”表示任意单个字符，而字符“%”表示任意数量的字符
示　例	hive> SELECT 1 FROM dual WHERE 'key' LIKE 'foot%'; hive> SELECT 1 FROM dual WHERE 'key ' LIKE 'foot____'; hive> SELECT 1 FROM dual WHERE NOT 'key ' LIKE 'fff%';
注意事项	否定比较时候用 NOT A LIKE B，例如： hive> SELECT 1 FROM dual WHERE NOT 'key ' LIKE 'fff%';

10. Java 的 LIKE 操作

语　法	A RLIKE B
操作类型	strings
说　明	如果字符串 A 或者字符串 B 为 NULL，则返回 NULL；如果字符串 A 符合 Java 正则表达式 B 的正则语法，则为 TRUE；否则为 FALSE
示　例	hive> SELECT 1 FROM dual WHERE 'footbar' RLIKE '^f.*r$'; 判断一个字符串是否全为数字： hive> SELECT 1 FROM dual WHERE '123456' RLIKE '^\\d+$';

11. REGEXP 操作

语　法	A REGEXP B
操作类型	strings
说　明	功能与 RLIKE 相同
示　例	`hive> SELECT 1 FROM dual WHERE 'key' REGEXP '^f.*r$';`

A.2 逻辑运算与数学运算

Hive 支持数学运算，常见的数学运算符如下。

1. 加法操作

语　法	A + B
操作类型	所有数值类型
说　明	返回 A 与 B 相加的结果。结果的数值类型等于 A 的类型和 B 的类型的最小父类型。比如，int + int 一般结果为 int 类型，而 int + double 一般结果为 double 类型
示　例	`hive> SELECT 1 + 9 FROM dual;` `10`

2. 减法操作

语　法	A – B
操作类型	所有数值类型
说　明	返回 A 与 B 相减的结果。结果的数值类型等于 A 的类型和 B 的类型的最小父类型。比如，int – int 一般结果为 int 类型，而 int – double 一般结果为 double 类型
示　例	`hive> SELECT 10 - 5 FROM dual;` `5`

3. 乘法操作

语　法	A * B
操作类型	所有数值类型
说　明	返回 A 与 B 相乘的结果。结果的数值类型等于 A 的类型和 B 的类型的最小父类型
示　例	`hive> SELECT 40 * 5 FROM dual;` `200`
注　意	如果 A 乘以 B 的结果超过默认结果类型的数值范围，则需要通过 cast 将结果转换成范围更大的数值类型

4. 除法操作

语　法	A / B
操作类型	所有数值类型
说　明	返回 A 除以 B 的结果。结果的数值类型为 double
示　例	`hive> SELECT 40 / 5 FROM dual;` `8.0`
注　意	Hive 中最高精度的数据类型是 double，只精确到小数点后 16 位，在做除法运算的时候要特别注意

（续）

注　意	hive>SELECT ceil(28.0/6.99999999999999999)FROM dual LIMIT 1;
	结果为 4
	hive>select ceil(28.0/6.99999999999999)from dual limit 1;
	结果为 5

5. 取余操作

语　法	A % B
操作类型	所有数值类型
说　明	返回 A 除以 B 的余数。结果的数值类型等于 A 的类型和 B 的类型的最小父类型
示　例	hive> SELECT 41 % 5 FROM dual; 1 hive> SELECT 8.4 % 4 FROM dual; 0.40000000000000036
注　意	精度在 Hive 中是个很大的问题，类似这样的操作最好通过 round 指定精度。例如： hive> SELECT round(8.4 % 4 , 2)FROM dual; 0.4

6. 位与操作

语　法	A & B
操作类型	所有数值类型
说　明	返回 A 和 B 按位进行与操作的结果。结果的数值类型等于 A 的类型和 B 的类型的最小父类型
示　例	hive> SELECT 4 & 8 FROM dual; 0 hive> SELECT 6 & 4 FROM dual; 4

7. 位或操作

语　法	A | B
操作类型	所有数值类型
说　明	返回 A 和 B 按位进行或操作的结果。结果的数值类型等于 A 的类型和 B 的类型的最小父类型
示　例	hive> SELECT 4 | 8 FROM dual; 12 hive> SELECT 6 | 8 FROM dual; 14

8. 位异或操作

语　法	A ^ B
操作类型	所有数值类型
说　明	返回 A 和 B 按位进行异或操作的结果。结果的数值类型等于 A 的类型和 B 的类型的最小父类型
示　例	hive> SELECT 4 ^ 8 FROM dual; 12 hive> SELECT 6 ^ 4 FROM dual; 2

9. 位取反操作

语　法	~ A
操作类型	所有数值类型
说　明	返回 A 按位取反操作的结果。结果的数值类型等于 A 的类型
示　例	hive> SELECT ~6 FROM dual; -7 hive> SELECT ~4 FROM dual; -5

A.3 Hive 逻辑运算

Hive 支持的逻辑算如下。

1. 逻辑与操作

语　法	A AND B
操作类型	boolean
说　明	如果 A 和 B 均为 TRUE，则为 TRUE；否则为 FALSE。如果 A 为 NULL 或 B 为 NULL，则为 NULL
示　例	hive> SELECT 1 FROM dual WHERE 1=1 AND 2=2; 1

2. 逻辑或操作

语　法	A OR B
操作类型	boolean
说　明	如果 A 为 TRUE，或者 B 为 TRUE，或者 A 和 B 均为 TRUE，则为 TRUE；否则为 FALSE
示　例	hive> SELECT 1 FROM dual WHERE 1=2 OR 2=2; 1

3. 逻辑非操作

语　法	NOT A
操作类型	boolean
说　明	如果 A 为 FALSE，或者 A 为 NULL，则为 TRUE；否则为 FALSE
示　例	hive> SELECT 1 FROM dual WHERE NOT 1=2;

A.4 数值运算

1. 取整函数

语　法	round(double a)
返 回 值	BIGINT
说　明	返回 double 类型的整数值部分 (遵循四舍五入)

（续）

示　例	hive> select round(3.1415926)from dual;
	3
	hive> select round(3.5)from dual;
	4
	hive> create table dual as select round(9542.158)from dual;
	hive> describe dual;
	_c0　　bigint

2. 指定精度取整函数

语　法	round(double a, int d)
返 回 值	double
说　明	返回指定精度 d 的 double 类型
示　例	hive> select round(3.1415926,4)from dual; 3.1416

3. 向下取整函数

语　法	floor(double a)
返 回 值	BIGINT
说　明	返回等于或者小于该 double 变量的最大的整数

示　例	hive> select floor(3.1415926)from dual;
	3
	hive> select floor(25)from dual;
	25

4. 向上取整函数

语　法	ceil(double a)
返 回 值	BIGINT
说　明	返回等于或者大于该 double 变量的最小的整数

示　例	hive> select ceil(3.1415926)from dual;
	4
	hive> select ceil(46)from dual;
	46

5. 向上取整函数

语　法	ceiling(double a)
返 回 值	BIGINT
说　明	与 ceil 功能相同

示　例	hive> select ceiling(3.1415926)from dual;
	4
	hive> select ceiling(46)from dual;
	46

6. 取随机数函数

语　法	rand()，rand(int seed)
返回值	double
说　明	返回一个 0 到 1 的随机数。如果指定种子 seed，则会等到一个稳定的随机数序列
示　例	hive> select rand()from dual; 0.5577432776034763

7. 自然指数函数

语　法	exp(double a)
返回值	double
说　明	返回自然对数 e 的 a 次方
示　例	hive> select exp(2)from dual; 7.38905609893065

8. 自然对数函数

语　法	ln(double a)
返回值	double
说　明	返回 a 的自然对数

9. 以 10 为底对数函数

语　法	log10(double a)
返回值	double
说　明	返回以 10 为底 a 的对数
示　例	hive> select log10(100)from dual; 2.0

10. 以 2 为底对数函数

语　法	log2(double a)
返回值	double
说　明	返回以 2 为底 a 的对数
示　例	hive> select log2(8)from dual; 3.0

11. 对数函数

语　法	log(double base, double a)
返回值	double
说　明	返回以 base 为底 a 的对数
示　例	hive> select log(4,256)from dual; 4.0

12. 幂运算函数

语　法	pow(double a, double p)
返回值	double

（续）

说　明	返回 a 的 p 次幂
示　例	hive> select pow(2,4)from dual; 16.0

13. 幂运算函数

语　法	power(double a, double p)
返 回 值	double
说　明	返回 a 的 p 次幂，与 pow 功能相同
示　例	hive> select power(2,4)from dual; 16.0

14. 开平方函数

语　法	sqrt(double a)
返 回 值	double
说　明	返回 a 的平方根
示　例	hive> select sqrt(16)from dual; 4.0

15. 二进制函数

语　法	bin(BIGINT a)
返 回 值	string
说　明	返回 a 的二进制代码表示
示　例	hive> select bin(7)from dual; 111

16. 十六进制函数

语　法	hex(BIGINT a)
返 回 值	string
说　明	如果变量是 int 类型，返回 a 的十六进制表示；如果变量是 string 类型，返回该字符串的十六进制表示

示　例	hive> select hex(17)from dual;
	11
	hive> select hex('abc')from dual;
	616263

17. 反转十六进制函数

语　法	unhex(string a)
返 回 值	string
说　明	返回该十六进制字符串所代码的字符串

示　例	hive> select unhex('616263')from dual;
	abc
	hive> select unhex('11')from dual;
	-
	hive> select unhex(616263)from dual;
	abc

18. 进制转换函数

语　法	conv(BIGINT num, int from_base, int to_base)
返回值	string
说　明	将数值 num 从 from_base 进制转化到 to_base 进制

示　例	hive> select conv(17,10,16)from dual;
	11
	hive> select conv(17,10,2)from dual;
	10001

19. 绝对值函数

语　法	abs(double a)，abs(int a)
返回值	double， int
说　明	返回数值 a 的绝对值

示　例	hive> select abs(-3.9)from dual;
	3.9
	hive> select abs(10.9)from dual;
	10.9

20. 正取余函数

语　法	pmod(int a, int b)，pmod(double a, double b)
返回值	Int，double
说　明	返回正的 a 除以 b 的余数

示　例	hive> select pmod(9,4)from dual;
	1
	hive> select pmod(-9,4)from dual;
	3

21. 正弦函数

语　法	sin(double a)
返回值	double
说　明	返回 a 的正弦值
示　例	hive> select sin(0.8)from dual; 0.7173560908995228

22. 反正弦函数

语　法	asin(double a)
返回值	double
说　明	返回 a 的反正弦值
示　例	hive> select asin(0.7173560908995228)from dual; 0.8

23. 余弦函数

语　法	cos(double a)

（续）

返 回 值	double
说　　明	返回 a 的余弦值
示　　例	hive> select cos(0.9)from dual; 0.6216099682706644

24. 反余弦函数

语　　法	acos(double a)
返 回 值	double
说　　明	返回 a 的反余弦值
示　　例	hive> select acos(0.6216099682706644)from dual; 0.9

25. positive 函数

语　　法	positive(int a), positive(double a)
返 回 值	int double
说　　明	返回 a

示　　例	hive> select positive(-10)from dual;
	-10
	hive> select positive(12)from dual;
	12

26. negative 函数

语　　法	negative(int a), negative(double a)
返 回 值	int double
说　　明	返回 -a

示　　例	hive> select negative(-5)from dual;
	5
	hive> select negative(8)from dual;
	-8

A.5 日期函数

1. UNIX 时间戳转日期函数

语　　法	from_unixtime(bigint unixtime[, string format])
返 回 值	string
说　　明	转化 UNIX 时间戳（从 1970-01-01 00:00:00 UTC 到指定时间的秒数）到当前时区的时间格式
示　　例	hive> select from_unixtime(1323308943,'yyyyMMdd')from dual; 20111208

2. 获取当前 UNIX 时间戳函数

语　　法	unix_timestamp()

（续）

返 回 值	bigint
说　明	获得当前时区的 UNIX 时间戳
示　例	hive> select unix_timestamp()from dual; 1323309615

3. 日期转 UNIX 时间戳函数

语　法	unix_timestamp(string date)
返 回 值	bigint
说　明	转换格式为“yyyy-MM-dd HH:mm:ss”的日期到 UNIX 时间戳。如果转化失败，返回 0
示　例	hive> select unix_timestamp('2011-12-07 13:01:03')from dual; 1323234063

4. 指定格式日期转 UNIX 时间戳函数

语　法	unix_timestamp(string date, string pattern)
返 回 值	bigint
说　明	转换 pattern 格式的日期到 UNIX 时间戳。如果转化失败，返回 0
示　例	hive> select unix_timestamp('20111207 13:01:03','yyyyMMdd HH:mm:ss') from dual; 1323234063

5. 日期时间转日期函数

语　法	to_date(string timestamp)
返 回 值	string
说　明	返回日期时间字段中的日期部分
示　例	hive> select to_date('2011-12-08 10:03:01')from dual; 2011-12-08

6. 日期转年函数

语　法	year(string date)
返 回 值	int
说　明	返回日期中的年

示　例	hive> select year('2011-12-08 10:03:01')from dual;
	2011
	hive> select year('2012-12-08')from dual;
	2012

7. 日期转月函数

语　法	month (string date)
返 回 值	int
说　明	返回日期中的月份

示　例	hive> select month('2011-12-08 10:03:01')from dual;
	12
	hive> select month('2011-08-08')from dual;
	8

8. 日期转天函数

语　法	day (string date)
返回值	int
说　明	返回日期中的天

示　例	hive> select day('2011-12-08 10:03:01')from dual;
	8
	hive> select day('2011-12-24')from dual;
	24

9. 日期转小时函数

语　法	hour (string date)
返回值	int
说　明	返回日期中的小时
示　例	hive> select hour('2011-12-08 10:03:01')from dual; 10

10. 日期转分钟函数

语　法	minute (string date)
返回值	int
说　明	返回日期中的分钟
示　例	hive> select minute('2011-12-08 10:03:01')from dual; 3

11. 日期转秒函数

语　法	second (string date)
返回值	int
说　明	返回日期中的秒
示　例	hive> select second('2011-12-08 10:03:01')from dual; 1

12. 日期转周函数

语　法	weekofyear (string date)
返回值	int
说　明	返回日期在当前的周数
示　例	hive> select weekofyear('2011-12-08 10:03:01')from dual; 49

13. 日期比较函数

语　法	datediff(string enddate, string startdate)
返回值	int
说　明	返回结束日期减去开始日期的天数
示　例	hive> select datediff('2012-12-08','2012-05-09')from dual; 213

14. 日期增加函数

语　法	date_add(string startdate, int days)
返回值	string
说　明	返回开始日期 startdate 增加 days 天后的日期
示　例	hive> select date_add('2012-12-08',10)from dual; 2012-12-18

15. 日期减少函数

语　法	date_sub (string startdate, int days)
返回值	string
说　明	返回开始日期 startdate 减少 days 天后的日期
示　例	hive> select date_sub('2012-12-08',10)from dual; 2012-11-28

A.6 条件函数

1. If 函数

语　法	if(boolean testCondition, T valueTrue, T valueFalseOrNull)
返回值	T
说　明	当条件 testCondition 为 TRUE 时，返回 valueTrue；否则返回 valueFalseOrNull

示　例	hive> select if(1=2,100,200)from dual;
	200
	hive> select if(1=1,100,200)from dual;
	100

2. 非空查找函数

语　法	COALESCE(T v1, T v2, …)
返回值	T
说　明	返回参数中的第一个非空值；如果所有值都为 NULL，那么返回 NULL
示　例	hive> select COALESCE(null,'100','50')from dual; 100

3. 条件判断函数

语　法	CASE a WHEN b THEN c [WHEN d THEN e]* [ELSE f] END
返回值	T
说　明	如果 a 等于 b，返回 c；如果 a 等于 d，返回 e；否则返回 f
示　例	hive> Select case 100 when 50 then 'tom' when 100 then 'mary' else 'tim' end from dual; mary

A.7 字符串函数

1. 字符串长度函数

语　法	length(string A)
返 回 值	int
说　明	返回字符串 A 的长度
示　例	hive> select length('abcedfg')from dual; 7

2. 字符串反转函数

语　法	reverse(string A)
返 回 值	string
说　明	返回字符串 A 的反转结果
示　例	hive> select reverse(abcedfg')from dual; gfdecba

3. 字符串连接函数

语　法	concat(string A, string B…)
返 回 值	string
说　明	返回输入字符串连接后的结果，支持任意个输入字符串
示　例	hive> select concat('abc','def','gh')from dual; abcdefgh

4. 带分隔符字符串连接函数

语　法	concat_ws(string SEP, string A, string B…)
返 回 值	string
说　明	返回输入字符串连接后的结果，SEP 表示各个字符串间的分隔符
示　例	hive> select concat_ws(',','abc','def','gh')from dual; abc,def,gh

5. 字符串截取函数

语　法	substr(string A, int start),substring(string A, int start)
返 回 值	string
说　明	返回字符串 A 从 start 位置到结尾的字符串

示　例	hive> select substr('abcde',3)from dual;
	cde
	hive> select substring('abcde',3)from dual;
	cde
	hive> select substr('abcde',-1)from dual; （和 ORACLE 相同）
	e

6. 字符串截取函数

语　法	substr(string A, int start, int len),substring(string A, int start, int len)
返 回 值	string

（续）

说　明	返回字符串 A 从 start 位置开始，长度为 len 的字符串

示　例	hive> select substr('abcde',3,2)from dual;
	cd
	hive> select substring('abcde',3,2)from dual;
	cd
	hive>select substring('abcde',-2,2)from dual;
	de

7. 字符串转大写函数

语　法	upper(string A)ucase(string A)
返 回 值	string
说　明	返回字符串 A 的大写格式

示　例	hive> select upper('abSEd')from dual;
	ABSED
	hive> select ucase('abSEd')from dual;
	ABSED

8. 字符串转小写函数

语　法	lower(string A)lcase(string A)
返 回 值	string
说　明	返回字符串 A 的小写格式

示　例	hive> select lower('abSEd')from dual;
	absed
	hive> select lcase('abSEd')from dual;
	absed

9. 去空格函数

语　法	trim(string A)
返 回 值	string
说　明	去除字符串两边的空格
示　例	hive> select trim('abc ')from dual; abc

10. 左边去空格函数

语　法	ltrim(string A)
返 回 值	string
说　明	去除字符串左边的空格
示　例	hive> select ltrim(' abc ')from dual; abc

11. 右边去空格函数

语　法	rtrim(string A)
返 回 值	string

（续）

说　明	去除字符串右边的空格
示　例	hive> select rtrim(' abc ')from dual; abc

12. 正则表达式替换函数

语　法	regexp_replace(string A, string B, string C)
返回值	string
说　明	将字符串 A 中的符合 java 正则表达式 B 的部分替换为 C
示　例	hive> select regexp_replace('foobar', 'oo|ar', '')from dual; fb
注　意	有些情况下要使用转义字符，类似 oracle 中的 regexp_replace 函数

13. 正则表达式解析函数

语　法	regexp_extract(string subject, string pattern, int index)
返回值	string
说　明	将字符串 subject 按照 pattern 正则表达式的规则拆分，返回 index 指定的字符

示　例	hive> select regexp_extract('foothebar', 'foo(.*?)(bar)', 1)from dual;
	the
	hive> select regexp_extract('foothebar', 'foo(.*?)(bar)', 2)from dual;
	bar
	hive> select regexp_extract('foothebar', 'foo(.*?)(bar)', 0)from dual;
	foothebar

注　意	有些情况下要使用转义字符，下面的等号要用双竖线转义，这是 Java 正则表达式的规则： select data_field, regexp_extract(data_field,'.*?bgStart\\=([^&]+)',1)as aaa, regexp_extract(data_field,'.*?contentLoaded_headStart\\=([^&]+)',1)as bbb, regexp_extract(data_field,'.*?AppLoad2Req\\=([^&]+)',1)as ccc from pt_nginx_loginlog_st where pt = '2012-03-26' limit 2;

14. URL 解析函数

语　法	parse_url(string urlString, string partToExtract [, string keyToExtract])
返回值	string
说　明	返回 URL 中指定的部分。partToExtract 的有效值包括 HOST、PATH、QUERY、REF、PROTOCOL、AUTHORITY、FILE 和 USERINFO

示　例	hive> select parse_url('http://facebook.com/path1/
	p.php?k1=v1&k2=v2#Ref1', 'HOST')from dual;
	facebook.com
	hive> select parse_url('http://facebook.com/path1/
	p.php?k1=v1&k2=v2#Ref1', 'QUERY', 'k1')from dual;
	v1

15. JSON 解析函数

语　法	get_json_object(string json_string, string path)
返回值	string
说　明	解析 JSON 的字符串 json_string，返回 path 指定的内容。如果输入的 JSON 字符串无效，返回 NULL
示　例	hive> select get_json_object('{"store": > {"fruit":\[{"weight":8, "type":"apple"},{"weight":9,"type":"pear"}], > "bicycle":{"price":19.95,"color":"red"} > }, > "email":"amy@only_for_json_udf_test.net", > "owner":"amy" > } > ','$.owner')from dual; amy

16. 空格字符串函数

语　法	space(int n)
返回值	string
说　明	返回长度为 n 的字符串

示　例	hive> select space(10)from dual;
	hive> select length(space(10))from dual;
	10

17. 重复字符串函数

语　法	repeat(string str, int n)
返回值	string
说　明	返回重复 n 次后的 str 字符串
示　例	hive> select repeat('abc',5)from dual; abcabcabcabcabc

18. 首字符 ASCII 函数

语　法	ascii(string str)
返回值	int
说　明	返回字符串 str 第一个字符的 ASCII 码
示　例	hive> select ascii('abcde')from dual; 97

19. 左补足函数

语　法	lpad(string str, int len, string pad)
返回值	string
说　明	将 str 进行用 pad 进行左补足到 len 位
示　例	hive> select lpad('abc',10,'td')from dual; tdtdtdtabc
注　意	与 GP 、ORACLE 不同，pad 不能默认

20. 右补足函数

语　法	rpad(string str, int len, string pad)
返回值	string
说　明	将 str 进行用 pad 进行右补足到 len 位
示　例	hive> select rpad('abc',10,'td')from dual; abctdtdtdt

21. 分割字符串函数

语　法	split(string str, string pat)
返回值	array
说　明	按照 pat 字符串分割 str，会返回分割后的字符串数组
示　例	hive> select split('abtcdtef','t')from dual; ["ab","cd","ef"]

22. 集合查找函数

语　法	find_in_set(string str, string strList)
返回值	int
说　明	返回 str 在 strlist 第一次出现的位置，strlist 是用逗号分割的字符串。如果没有找该 str 字符，则返回 0

示　例	hive> select find_in_set('ab','ef,ab,de')from dual;
	2
	hive> select find_in_set('at','ef,ab,de')from dual;
	0

A.8 集合统计函数

1. 个数统计函数

语　法	count(*), count(expr), count(DISTINCT expr[, expr_.])
返回值	int
说　明	count(*) 统计检索行的个数，包括 NULL 值的行；count(expr) 返回指定字段的非空值的个数；count(DISTINCT expr[, expr_.]) 返回指定字段不同非空值的个数

示　例	hive> select count(*)from dual;
	20
	hive> select count(distinct t)from dual;
	10

2. 总和统计函数

语　法	sum(col), sum(DISTINCT col)
返回值	double
说　明	sum(col) 统计结果集中 col 相加的结果；sum(DISTINCT col) 统计结果中 col 不同值相加的结果

示　例	hive> select sum(t)from dual;
	100
	hive> select sum(distinct t)from dual;
	70

3. 平均值统计函数

语　法	avg(col), avg(DISTINCT col)
返回值	double
说　明	avg(col) 统计结果集中 col 的平均值；avg(DISTINCT col) 统计结果中 col 不同值相加的平均值
示　例	hive> select avg(t)from dual; 50 hive> select avg (distinct t)from dual; 30

4. 最小值统计函数

语　法	min(col)
返回值	double
说　明	统计结果集中 col 字段的最小值
示　例	hive> select min(t)from dual; 20

5. 最大值统计函数

语　法	maxcol)
返回值	double
说　明	统计结果集中 col 字段的最大值
示　例	hive> select max(t)from dual; 120

A.9 符合类型构建操作

1. Map 类型构建

语　法	map (key1, value1, key2, value2, …)
说　明	根据输入的 key 和 value 构建 Map 类型
示　例	hive> Create table alex_test as select map('100','tom','200','mary')as t from dual; hive> describe alex_test; t　　map hive> select t from alex_test; {"100":"tom","200":"mary"}

2. Struct 类型构建

语　法	struct(val1, val2, val3, …)
说　明	根据输入的参数构建结构体 Struct 类型
示　例	hive> create table alex_test as select struct('tom','mary','tim')as t from dual; hive> describe alex_test; t　　struct hive> select t from alex_test; {"col1":"tom","col2":"mary","col3":"tim"}

3. Array 类型构建

语　法	array(val1, val2, …)
说　明	根据输入的参数构建数组 Array 类型
示　例	hive> create table alex_test as select array("tom","mary","tim")as t from dual; hive> describe alex_test; t　　array hive> select t from alex_test; ["tom","mary","tim"]

A.10 复杂类型访问操作

1. Array 类型访问

语　法	A[n]
操作类型	A 为 Array 类型，n 为 int 类型
说　明	返回数组 A 中的第 n 个变量值。数组的起始下标为 0。比如，A 是个值为 ['foo', 'bar'] 的数组类型，那么 A[0] 将返回 'foo', 而 A[1] 将返回 'bar'
示　例	hive> create table alex_test as select array("tom","mary","tim")as t from dual; hive> select t[0],t[1],t[2] from alex_test; tom　　mary　　tim

2. Map 类型访问

语　法	M[key]
操作类型	M 为 Map 类型，key 为 Map 中的 key 值
说　明	返回 Map 类型 M 中，key 值为指定值的 value 值。比如，M 是值为 {'f' -> 'foo', 'b' -> 'bar', 'all' -> 'foobar'} 的 Map 类型，那么 M['all'] 将会返回 'foobar'
示　例	hive> Create table alex_test as select map('100','tom','200','mary')as t from dual; hive> select t['200'],t['100'] from alex_test; mary　　tom

3. Struct 类型访问

语　法	S.x
操作类型	S 为 Struct 类型
说　明	返回结构体 S 中的 x 字段。比如，对于结构体 struct foobar {int foo, int bar}，foobar.foo 返回结构体中的 foo 字段
示　例	hive> create table alex_test as select struct('tom','mary','tim')as t from dual; hive> describe alex_test; t　　struct hive> select t.col1,t.col3 from alex_test; tom　　tim

A.11 复杂类型长度统计函数

1. Map 类型长度函数

语　法	size(Map)
返回值	int
说　明	返回 Map 类型的长度
示　例	hive> select size(map('100','tom','101','mary'))from dual; 2

2. Array 类型长度函数

语　法	size(Array)
返回值	int
说　明	返回 Array 类型的长度
示　例	hive> select size(array('100','101','102','103'))from dual; 4

3. 类型转换函数

语　法	cast(expr as)
返回值	Expected "=" to follow "type"
说　明	返回 Array 类型的长度
示　例	hive> select cast(1 as bigint)from dual; 1

附录B　HBase 默认配置解释

	配　　置	功能描述	默　　认
1	hbase. rootdir	这个目录是 region server 的共享目录，用来持久化 HBase。URL 需要是'完全正确'的，还要包含文件系统的 scheme。例如，要表示 HDFS 中的'/hbase'目录，NameNode 运行在 namenode. example. org 的 9090 端口。则需要设置为 hdfs://namenode. example. org:9000/ hbase。默认情况下 HBase 是写到 /tmp 的。不改这个配置，数据会在重启的时候丢失	file:///tmp/hbase-${user.name}/hbase
2	hbase. master. port	HBase 的 Master 端口	60000
3	hbase. cluster. distributed	HBase 的运行模式。false 是单机模式，true 是分布式模式。若为 false，HBase 和 ZooKeeper 会运行在同一个 JVM 里	false
4	hbase. tmp. dir	本地文件系统的临时文件夹。可以修改到一个更为持久的目录上 (/tmp 会在重启时明确)	/tmp/hbase-${user.name}
5	hbase. master. info. port	HBase Master Web 界面端口。 设置为 -1 意味着不想让它运行	60010
6	hbase. master. info. bindAddress	HBase Master Web 界面绑定的端口	0.0.0.0
7	hbase. client. write. buffer	HTable 客户端的写缓冲的默认大小。这个值越大，需要消耗的内存越大。因为缓冲在客户端和服务端都有实例，所以需要消耗客户端和服务端两个地方的内存。得到的好处是，可以减少 RPC 的次数。可以这样估算服务器端被占用的内存 hbase. client. write. buffer * hbase. regionserver. handler. count	2097152
8	hbase. regionserver. port	HBase RegionServer 绑定的端口	60020
9	hbase. regionserver. info. port	HBase RegionServer Web 界面绑定的端口设置为 -1 意味着不想运行 RegionServer 界面	60030
10	hbase. regionserver. info. port. auto	Master 或 RegionServer 是否要动态搜一个可以用的端口来绑定界面。当 HBase. regionserver. info. port 已经被占用的时候，可以搜一个空闲的端口绑定。这个功能在测试的时候很有用。默认关闭	false
11	hbase. regionserver. info. bindAddress	HBase RegionServer Web 界面的 IP 地址	0.0.0.0
12	hbase. regionserver. class	RegionServer 使用的接口。客户端打开代理来连接 region server 的时候会使用到	org. apache. hadoop. hbase. ipc. HRegionInterface
13	hbase. client. pause	通常的客户端暂停时间。最多的用法是客户端在重试前的等待时间。比如失败的 get 操作和 region 查询操作等都很可能用到	1000

（续）

	配　置	功能描述	默　认
14	hbase. client. retries. number	最大重试次数。例如 region 查询、Get 操作、Update 操作等都可能发生错误，需要重试。这是最大重试错误的值	10
15	hbase. client. scanner. caching	当调用 Scanner 的 next 方法，而值又不在缓存里的时候，从服务端一次获取的行数。越大的值意味着 Scanner 会快一些，但是会占用更多的内存。当缓冲被占满的时候，next 方法调用会越来越慢。慢到一定程度，可能会导致超时。例如超过了 hbase. regionserver. lease. period	1
16	hbase. client. keyvalue. maxsize	一个 KeyValue 实例的最大 size。这个是用来设置存储文件中的单个 entry 的大小上界。因为一个 KeyValue 是不能分割的，所以可以避免因为数据过大导致 region 不可分割。明智的做法是把它设为可以被最大 region size 整除的数。如果设置为 0 或者更小，就会禁用这个检查。默认 10MB	10485760
17	hbase. regionserver. lease. period	客户端租用 HRegion Server 期限，即超时阈值。单位是毫秒。默认情况下，客户端必须在这个时间内发一条信息，否则视为死掉	60000
18	hbase. regionserver. handler. count	RegionServers 受理的 RPC Server 实例数量。对于 Master 来说，这个属性是 Master 受理的 handler 数量	10
19	hbase. regionserver. msginterval	RegionServer 发消息给 Master 时间间隔，单位是毫秒	3000
20	hbase. regionserver. optionallogflushinterv al	将 Hlog 同步到 HDFS 的间隔。如果 Hlog 没有积累到一定的数量，到了时间，也会触发同步。默认 1000，单位是毫秒	1000
21	hbase. regionserver. regionSplitLimit	region 的数量到达这个值后就不会分裂了。这不是一个 region 数量的硬性限制，但是起到一定指导性的作用，到了这个值就该停止分裂了。默认是 MAX_INT，即不阻止分裂	2147483647
22	hbase. regionserver. logroll. period	提交 commit log 的间隔，不管有没有写足够的值	3600000
23	hbase. regionserver. hlog. reader. impl	HLog file reader 的实现	org. apache. hadoop. hbase. regionserver. wal.SequenceFileLog Reader
24	hbase. regionserver. hlog. writer. impl	HLog file writer 的实现	org. apache. hadoop. hbase. regionserver. wal. SequenceFileLog Writer
25	hbase. regionserver. thread. splitcompactcheck-fre quency	RegionServer 多久执行一次 split/compaction 检查	20000
26	hbase. regionserver. nbreservationblocks	储备的内存 Block 的数量。当发生 out of memory 异常的时候，可以用这些内存在 Region Server 停止之前做清理操作	4
27	hbase. ZooKeeper. dns. interface	当使用 DNS 的时候，ZooKeeper 用来上报的 IP 地址的网络接口名字	default

（续）

	配　置	功能描述	默　认
28	hbase. ZooKeeper. dns. nameserver	当使用 DNS 的时候，ZooKeeper 使用的 DNS 域名或者 IP 地址，ZooKeeper 用它来确定和 Master 进行通信的域名	default
29	hbase. regionserver. dns. interface	当使用 DNS 的时候，RegionServer 用来上报的 IP 地址的网络接口名字	default
30	hbase. regionserver. dns. nameserver	当使用 DNS 的时候，Region Server 使用的 DNS 域名或者 IP 地址，Region Server 用它来确定和 Master 进行通信的域名	default
31	hbase. master. dns. interface	当使用 DNS 的时候，Master 用来上报的 IP 地址的网络接口名字	default
32	hbase. master. dns. nameserver	当使用 DNS 的时候，Region Server 使用的 DNS 域名或者 IP 地址，Master 用它来确定进行通信的域名	default
33	hbase. balancer. period	Master 执行 region balancer 的间隔	300000
34	hbase. regions. slop	当任意一个 Region Server 有 average + (average * slop) 个 region 时，执行 Rebalance	0
35	hbase. master. logcleaner. ttl	HLog 存在于 oldlogdir 文件夹的最长时间，超过就会被 Master 的线程清理	600000
36	hbase. master. logcleaner. plugins	LogsCleaner 服务会执行的一组 LogCleanerDelegat。值用逗号间隔的文本表示。这些 WAL/HLog cleaners 会按顺序调用。可以把先调用的放在前面。可以实现自己的 LogCleanerDelegat，加到 Classpath 下，然后在这里写类的全称。一般加在默认值的前面	org. apache. hadoop. hbase. master. TimeToLiveLogCleaner
37	hbase. regionserver. global. memstore. upperLimit	单个 Region Server 的全部 memtores 的最大值。超过这个值，一个新的 update 操作会被挂起，强制执行 flush 操作	0.4
38	hbase. regionserver. global. memstore. lowerLimit	强制执行 flush 操作时，当低于这个值的时候，flush 会停止。默认是堆大小的 35%。如果这个值和 hbase. regionserver. global. memstore. upperLimit 相同，意味着 update 操作因为内存限制被挂起时，会尽量少地执行 flush(一旦执行 flush，值就会比下限要低，不再执行)	0.35
39	hbase. server. thread. wakefrequency	service 工作的 sleep 间隔，单位毫秒。可以作为 service 线程的 sleep 间隔，比如 log roller	10000
40	hbase. hregion. memstore. flush. size	当 memstore 的大小超过这个值的时候，会 flush 到磁盘。这个值被一个线程每隔 hbase. server. thread. wakefrequency 检查一次	67108864
41	hbase. hregion. preclose. flush. size	当一个 region 中 memstore 的大小大于这个值的时候，触发 close，会先运行 "pre-flush" 操作，清理这个需要关闭的 memstore，然后将这个 region 下线。当一个 region 下线了，无法再进行任何写操作。如果一个 memstore 很大，flush 操作会消耗很多时间。"pre-flush" 操作意味着在 region 下线之前，会先把 memstore 清空。这样在最终执行 close 操作的时候，flush 操作会很快	5242880

（续）

	配　置	功能描述	默　认
42	hbase. hregion. memstore. block. multiplier	如果 memstore 有 hbase. hregion. memstore. block. multiplier 倍数的 hbase. hregion. flush. size 的大小，就会阻塞 update 操作。这是为了预防在 update 高峰期导致失控。如果不设上界，flush 的时候会花很长时间来合并或者分割，最坏的情况就是引发 out of memory 异常	2
43	hbase. hregion. memstore. mslab. enabled	体验特性，启用 memstore 分配本地缓冲区。这个特性是为了防止在大量写负载时堆的碎片过多。可以减少 GC 操作的频率	false
44	hbase. hregion. max. filesize	最大 HstoreFile 尺寸。若某个 Column families 的 HStoreFile 增长达到这个值，这个 Hegion 会被切割成两个。 默认值是 256M	268435456
45	hbase. hstore. compactionThreshold	当一个 HStore 含有多于这个值的 HStoreFiles(每一个 memstore flush 产生一个 HStoreFile) 的时候，会执行一个合并操作，把 HStoreFiles 写成一个。这个值越大，需要合并的时间就越长。	3
46	hbase. hstore. blockingStoreFiles	当一个 HStore 含有多于这个值的 HStoreFiles(每一个 memstore flush 产生一个 HStoreFile) 的时候，会执行一个合并操作，update 会阻塞直到合并完成，直到超过 hbase. hstore. blockingWaitTime 的值	7
47	hbase. hstore. blockingWaitTime	hbase. hstore. blockingStoreFiles 所限制的 StoreFile 数量会导致 update 阻塞，这个时间是用来限制阻塞时间的。当超过这个时间，HRegion 会停止阻塞 update 操作，不过合并还没有完成。默认为 90s	90000
48	hbase. hstore. compaction. max	每个 " 小 " 合并的 HStoreFiles 最大数量	10
49	hbase. hregion. majorcompaction	一个 Region 中的所有 HStoreFile 的 major compactions 时间间隔。默认是 1 天。设置为 0 就是禁用这个功能	86400000
50	hbase. mapreduce. hfileoutputformat. blocksize	MapReduce 中 HFileOutputFormat 可以写 storefiles/hfiles。该值是 HFile 的 blocksize 的最小值。通常在 HBase 写 HFile 时，bloocksize 是由 table schema(HColumnDescriptor) 决定的，但是在 MapReduce 写的时候，无法获取 schema 中 blocksize。这个值越小，索引就越大，随机访问需要获取的数据就越小。如果 cell 很小，而且需要更快随机访问，可以把这个值调低	65536
51	hfile. block. cache. size	分配给 HFile/StoreFile 的 block cache 占最大堆 (-Xmx setting) 的比例。默认是 20%，设置为 0 就是不分配	0. 2
52	hbase. hash. type	哈希函数使用的哈希算法。可以选择 murmur (MurmurHash) 和 jenkins (JenkinsHash) 两个值。这是给 bloom filters 用的	murmur
53	hbase. master. keytab. file	HMaster server 验证登录使用的 kerberos keytab 文件路径	
54	hbase. master. kerberos. principal	例如，"hbase/_HOST@EXAMPLE.COM"。HMaster 运行需要使用 kerberos principal name. principal name 可以在 user/hostname@DOMAIN 中获取。如果 "_HOST" 被用做 hostname portion，需要使用实际运行的 hostname 来替代它	

（续）

	配　置	功能描述	默　认
55	hbase. regionserver. keytab. file	HRegionServer 验证登录使用的 kerberos keytab 文件路径	
56	hbase. regionserver. kerberos. principal	例 如，"hbase/_HOST@EXAMPLE.COM "。HRegionServer 运行需要使用 kerberos principal name。 principal name 可以在 user/hostname@DOMAIN 中获取。 如果 "_HOST" 被用做 hostname portion，需要使用实际运行的 hostname 来替代它。在这个文件中必须要有一个 entry 来描述 hbase. regionserver. keytab. file	
57	ZooKeeper. session. timeout	ZooKeeper 会话超时。HBase 把这个值传递改 zk 集群，推荐一个会话的最大超时时间㊀。单位是毫秒	180000
58	ZooKeeper. znode. parent	ZooKeeper 中 HBase 的根 ZNode。所有 HBase 的 ZooKeeper 会用这个目录配置相对路径。默认情况下，所有 HBase 的 ZooKeeper 文件使用相对路径，所以都在这个目录下面	/hbase
59	ZooKeeper. znode. rootserver	ZNode 保存的根 region 的路径。 这个值由 Master 来写，由 client 和 regionserver 来读。如果设为一个相对地址，父目录就是 ${ZooKeeper. znode. parent}。默认情况下，意味着根 region 的路径存储在 /hbase/ root-region-server	root-region-server
60	hbase. ZooKeeper. quorum	ZooKeeper 集群的地址列表，用逗号分割。例如，"host1. mydomain. com, host2. mydomain. com, host3. mydomain. com"。默认是 localhost，是给伪分布式用的。要修改才能在完全分布式的情况下使用。如果在 hbase-env.sh 设置了 HBASE_MANAGES_ZK，这些 ZooKeeper 节点就会和 HBase 一起启动	localhost
61	hbase. ZooKeeper. peerport	ZooKeeper 节点使用的端口㊁	2888
62	hbase. ZooKeeper. leaderport	ZooKeeper 用来选择 Leader 的端口㊂	3888
63	hbase. ZooKeeper. property. initLimit	ZooKeeper 的 zoo.conf 中 的 配 置。初 始 化 synchronization 阶段的 ticks 数量限制	10
64	hbase. ZooKeeper. property. syncLimit	ZooKeeper 的 zoo.conf 中的配置。发送一个请求到获得承认之间的 ticks 的数量限制	5
65	hbase. ZooKeeper. property. dataDir	ZooKeeper 的 zoo.conf 中的配置。快照的存储位置	${hbase.tmp.dir}/ ZooKeeper

㊀ 参见 http://hadoop. apache. org/ ZooKeeper/ docs/ current/ ZooKeeperProgrammers. html#ch_zkSessions “The client sends a requested timeout，the server responds with the timeout that it can give the client.”。

㊁ 参见 http://hadoop. apache. org/ ZooKeeper/ docs/ r3.1.1/ ZooKeeperStarted. html#sc_RunningReplicatedZooKeeper。

㊂ 参见 http://hadoop. apache. org/ ZooKeeper/ docs/ r3.1.1/ ZooKeeperStarted. html#sc_RunningReplicatedZooKeeper。

（续）

	配　置	功能描述	默　认
66	hbase. ZooKeeper. property. clientPort	ZooKeeper 的 zoo.conf 中的配置。客户端连接的端口	2181
67	hbase. ZooKeeper. property. maxClientCnxns	ZooKeeper 的 zoo.conf 中的配置。 ZooKeeper 集群中的单个节点接受的单个 Client(以 IP 区分) 请求的并发数。这个值可以调高一点，防止在单机和伪分布式模式中出问题	2000
68	hbase.rest.port	HBase REST server 的端口	8080
69	hbase.rest. readonly	定义 REST server 的运行模式。可以设置成 false，表示所有的 HTTP 请求都是被允许的 (-GET/PUT/POST/DELETE) ；true 表示只有 GET 请求是被允许的	false

附录C　Hadoop 三个配置文件的参数含义说明

配置 Hadoop，主要是配置 core-site.xml、hdfs-site.xml、mapred-site.xml 三个配置文件，默认情况下，这些配置文件都是空的，所以很难知道这些配置文件有哪些配置可以生效，上网找的配置可能因为 Hadoop 版本不同，导致无法生效。

浏览更多的配置，有两个方法。

1）选择相应版本的 Hadoop，下载并解压后，搜索 "*.xml"，找到 core-default.xml、hdfs-default.xml、mapred-default.xml，这些就是默认配置，可以参考这些配置的说明和 key，配置 Hadoop 集群。

2）浏览 Apache 官网，三个配置文件链接如下：

- http://hadoop.apache.org/common/docs/current/core-default.html
- http://hadoop.apache.org/common/docs/current/hdfs-default.html
- http://hadoop.apache.org/common/docs/current/mapred-default.html

这里是浏览 Hadoop 当前版本号的默认配置文件，其他版本号需另外去官网找。其中第一个方法找到默认的配置是最好的，因为每个属性都有说明，可以直接使用。

另外，core-site.xml 是全局配置，hdfs-site.xml 和 mapred-site.xml 分别是 hdfs 和 mapred 的局部配置。

C.1　常用的端口配置

1. HDFS 端口

参数名称	功能描述	默认	配置文件	例子值
fs.default.name namenode	NameNode RPC 交互端口	8020	core-site.xml	hdfs://master:8020/
dfs.http.address	NameNode Web 管理端口	50070	hdfs- site.xml	0.0.0.0:50070
dfs.datanode.address	DataNode　控制端口	50010	hdfs -site.xml	0.0.0.0:50010
dfs.datanode.ipc.address	DataNode 的 RPC 服务器地址和端口	50020	hdfs-site.xml	0.0.0.0:50020
dfs.datanode.http.address	DataNode 的 HTTP 服务器和端口	50075	hdfs-site.xml	0.0.0.0:50075

2. MR 端口

参数名称	功能描述	默认	配置文件	例子值
mapred.job.tracker	JobTracker 交互端口	8021	mapred-site.xml	hdfs://master:8021/
job	Tracker 的 Web 管理端口	50030	mapred-site.xml	0.0.0.0:50030
mapred.task.tracker.http.address	TaskTracker 的 HTTP 端口	50060	mapred-site.xml	0.0.0.0:50060

3. 其他端口

	参数名称	参数值	功能描述
1	hadoop.tmp.dir	/tmp/hadoop-${user.name}	临时目录设定
2	hadoop.native.lib	true	使用本地 Hadoop 库标识。
3	hadoop.http.filter.initializers		HTTP 服务器过滤链设置
4	hadoop.security.group.mapping	org.apache.hadoop.security.ShellBasedUnixGroupsMapping	组内用户的列表的类设定
5	hadoop.security.authorization	false	服务端认证开启
6	hadoop.security.authentication	simple	无认证或认证设置
7	hadoop.security.token.service.use_ip	true	是否开启使用 IP 地址作为连接的开关
8	hadoop.logfile.size	10000000	日志文件最大为 10MB
9	hadoop.logfile.count	10	日志文件数量为 10 个
10	io.file.buffer.size	4096	流文件的缓冲区为 4KB
11	io.bytes.per.checksum	512	校验位数为 512 字节
12	io.skip.checksum.errors	false	校验出错后是抛出异常还是略过标识。True 则略过
13	io.compression.codecs	org.apache.hadoop.io.compress.DefaultCodec，org.apache.hadoop.io.compress.GzipCodec，org.apache.hadoop.io.compress.BZip2Codec，org.apache.hadoop.io.compress.SnappyCodec	压缩和解压的方式设置
14	io.serializations	org.apache.hadoop.io.serializer.WritableSerialization	序例化和反序列化的类设定
15	fs.default.name	file:///	默认的文件 URI 标识设定
16	fs.trash.interval	0	文件废弃标识设定，0 为禁止此功能
17	fs.file.impl	org.apache.hadoop.fs.LocalFileSystem	本地文件操作类设置
18	fs.hdfs.impl	org.apache.hadoop.hdfs.DistributedFileSystem	HDFS 文件操作类设置
19	fs.s3.impl	org.apache.hadoop.fs.s3.S3FileSystem	S3 文件操作类设置
20	fs.s3n.impl	org.apache.hadoop.fs.s3native.NativeS3FileSystem	S3 文件本地操作类设置
21	fs.kfs.impl	org.apache.hadoop.fs.kfs.KosmosFileSystem	KFS 文件操作类设置
22	fs.hftp.impl	org.apache.hadoop.hdfs.HftpFileSystem	HTTP 方式操作文件设置
23	fs.hsftp.impl	org.apache.hadoop.hdfs.HsftpFileSystem	HTTPS 方式操作文件设置
24	fs.webhdfs.impl	org.apache.hadoop.hdfs.web.WebHdfsFileSystem	Web 方式操作文件类设置
25	fs.ftp.impl	org.apache.hadoop.fs.ftp.FTPFileSystem	FTP 文件操作类设置
26	fs.ramfs.impl	org.apache.hadoop.fs.InMemoryFileSystem	内存文件操作类设置
27	fs.har.impl	org.apache.hadoop.fs.HarFileSystem	压缩文件操作类设置 .

（续）

	参数名称	参数值	功能描述
28	fs.har.impl.disable.cache	true	是否缓存 HAR 文件的标识设定
29	fs.checkpoint.dir	${hadoop.tmp.dir}/dfs/namesecondary	备份名称节点的存放目前录设置
30	fs.checkpoint.edits.dir	${fs.checkpoint.dir}	备份名称节点日志文件的存放目前录设置
31	fs.checkpoint.period	3600	动态检查的间隔时间设置
32	fs.checkpoint.size	67108864	日志文件大小为 64MB
33	fs.s3.block.size	67108864	写 S3 文件系统的块的大小为 64MB
34	fs.s3.buffer.dir	${hadoop.tmp.dir}/s3	S3 文件数据的本地存放目录
35	fs.s3.maxRetries	4	S3 文件数据的尝试读写次数
36	fs.s3.sleepTimeSeconds	10	S3 文件尝试的间隔
37	local.cache.size	10737418240	缓存大小设置为 10GB
38	io.seqfile.compress.blocksize	1000000	压缩流式文件中的最小块数为 100 万
39	io.seqfile.lazydecompress	true	块是否需要压缩标识设定
40	io.seqfile.sorter.recordlimit	1000000	内存中排序记录块类最小为 100 万
41	io.mapfile.bloom.size	1048576	BloomMapFiler 过滤量为 1MB
42	io.mapfile.bloom.error.rate	0.005	
43	hadoop.util.hash.type	murmur	缺少 hash 方法为 murmur
44	ipc.client.idlethreshold	4000	连接数据最小阈值为 4000
45	ipc.client.kill.max	10	一个客户端连接数最大值为 10
46	ipc.client.connection.maxidletime	10000	断开与服务器连接的时间最大为 10 秒
47	ipc.client.connect.max.retries	10	建立与服务器连接的重试次数为 10 次
48	ipc.server.listen.queue.size	128	接收客户连接的监听队列的长度为 128
49	ipc.server.tcpnodelay	false	开启或关闭服务器端 TCP 连接算法
50	ipc.client.tcpnodelay	false	开启或关闭客户端 TCP 连接算法
51	webinterface.private.actions	false	Web 交互的行为设定
52	hadoop.rpc.socket.factory.class.default	org.apache.hadoop.net.StandardSocketFactory	默认的 socket 工厂类设置

（续）

	参数名称	参数值	功能描述
53	hadoop.rpc.socket.factory.class.ClientProtocol		与dfs连接时的默认socket工厂类
54	hadoop.socks.server		服务端的工厂类默认设置为SocksSocketFactory
55	topology.node.switch.mapping.impl	org.apache.hadoop.net.ScriptBasedMapping	
56	topology.script.file.name		
57	topology.script.number.args	100	参数数量最多为100
58	hadoop.security.uid.cache.secs	14400	

C.2 集群目录配置

1. 核心配置属性

参数名称	功能描述	默认	配置文件	例子值
dfs.name.dir	NameNode的元数据。以逗号隔开，HDFS会把元数据冗余复制到这些目录，一般这些目录是不同的块设备，不存在的目录会被忽略掉	{hadoop.tmp.dir}/dfs/name	hdfs-site.xm	/hadoop/hdfs/name
dfs.name.edits.dir	NodeNode的事务文件存储的目录。以逗号隔开，HDFS会把事务文件冗余复制到这些目录，一般这些目录是不同的块设备，不存在的目录会被忽略掉	${dfs.name.dir}	hdfs-site.xm	${dfs.name.dir}
fs.checkpoint.dir	Secondary NameNode的元数据。以逗号隔开，HDFS会把元数据冗余复制到这些目录，一般这些目录是不同的块设备，不存在的目录会被忽略掉	${hadoop.tmp.dir}/dfs/namesecondary	core-site.xml	/hadoop/hdfs/namesecondary
fs.checkpoint.edits.dir	Secondary NameNode的事务文件存储的目录。以逗号隔开，HDFS会把事务文件冗余复制到这些目录	${fs.checkpoint.dir}	core-site.xml	${fs.checkpoint.dir}
hadoop.tmp.dir	临时目录，其他临时目录的父目录	/tmp/hadoop-${user.name}	core-site.xml	/hadoop/tmp/hadoop-${user.name}
dfs.data.dir	DataNode的数据目录。以逗号隔开，HDFS会把数据存在这些目录下，一般这些目录是不同的块设备，不存在的目录会被忽略掉	${hadoop.tmp.dir}/dfs/data	hdfs-site.xm	/hadoop/hdfs/data1/data,/hadoop/hdfs/data2/data

（续）

参数名称	功能描述	默认	配置文件	例子值
mapred.local.dir	MapReduce 产生的中间数据存放目录。以逗号隔开，HDFS 会把数据存在这些目录下，一般这些目录是不同的块设备，不存在的目录会被忽略掉	${hadoop.tmp.dir}/mapred/local	mapred-site.xml	/hadoop/hdfs/data1/mapred/local，/hadoop/hdfs/data2/mapred/local
mapred.system.dir	MapReduce 的控制文件	${hadoop.tmp.dir}/mapred/system	mapred-site.xml	/hadoop/hdfs/data1/system

2. HDFS 属性配置

序号	参数名称	参数值	功能描述
1	dfs.namenode.logging.level	info	输出日志类型
2	dfs.secondary.http.address	0.0.0.0:50090	备份名字节点的 HTTP 协议访问地址与端口
3	dfs.datanode.address	0.0.0.0:50010	数据节点的 TCP 管理服务地址和端口
4	dfs.datanode.http.address	0.0.0.0:50075	数据节点的 HTTP 协议访问地址和端口
5	dfs.datanode.ipc.address	0.0.0.0:50020	数据节点的 IPC 服务访问地址和端口
6	dfs.datanode.handler.count	3	数据节点的服务连接处理线程数
7	dfs.http.address	0.0.0.0:50070	名字节点的 HTTP 协议访问地址与端口
8	dfs.https.enable	false	支持 HTTPS 访问方式标识
9	dfs.https.need.client.auth	false	客户端指定 HTTPS 访问标识
10	dfs.https.server.keystore.resource	ssl-server.xml	SSL 密钥服务端的配置文件
11	dfs.https.client.keystore.resource	ssl-client.xml	SSL 密钥客户端的配置文件
12	dfs.datanode.https.address	0.0.0.0:50475	数据节点的 HTTPS 协议访问地址和端口
13	dfs.https.address	0.0.0.0:50470	名字节点的 HTTPS 协议访问地址和端口
14	dfs.datanode.dns.interface	default	数据节点采用 IP 地址标识
15	dfs.datanode.dns.nameserver	default	指定 DNS 的 IP 地址
16	dfs.replication.considerLoad	true	加载目标或不加载的标识
17	dfs.default.chunk.view.size	32768	浏览时的文件块大小设置为 32KB
18	dfs.datanode.du.reserved	0	每个卷预留的空闲空间数量
19	dfs.name.dir	${hadoop.tmp.dir}/dfs/name	存贮在本地的名字节点数据镜像的目录，作为名字节点的冗余备份
20	dfs.name.edits.dir	${dfs.name.dir}	存贮文件操作过程信息的存贮目录
21	dfs.web.ugi	webuser，webgroup	Web 接口访问的用户名和组的账户设定
22	dfs.permissions	true	文件操作时的权限检查标识
23	dfs.permissions.supergroup	supergroup	超级用户的组名定义
24	dfs.block.access.token.enable	false	数据节点访问令牌标识
25	dfs.block.access.key.update.interval	600	升级访问钥匙的间隔时间
26	dfs.block.access.token.lifetime	600	访问令牌的有效时间

（续）

序号	参数名称	参数值	功能描述
27	dfs.data.dir	${hadoop.tmp.dir}/dfs/data	数据节点的块本地存放目录
28	dfs.datanode.data.dir.perm	755	数据节点的存贮块的目录访问权限设置
29	dfs.replication	3	默认的块复制数量
30	dfs.replication.max	512	块复制的最大数量
31	dfs.replication.min	1	块复制的最小数量
32	dfs.block.size	67108864	默认的文件块大小为 64MB
33	dfs.df.interval	60000	磁盘空间统计间隔为 6 秒
34	dfs.client.block.write.retries	3	块写入出错时的重试次数
35	dfs.blockrepo.t.intervalMsec	3600000	块的报告间隔时为 1 小时
36	dfs.blockreport.initialDelay	0	块顺序报告的间隔时间
37	dfs.heartbeat.interval	3	数据节点的心跳检测间隔时间
38	dfs.namenode.handler.count	10	名字节点的连接处理的线程数量
39	dfs.safemode.threshold.pct	0.999f	启动安全模式的阀值设定
40	dfs.safemode.extension	30000	当阈值达到量值后扩展的时限
41	dfs.balance.bandwidthPerSec	1048576	启动负载均衡的数据节点可利用带宽最大值为 1MB
42	dfs.hosts		可与名字节点连接的主机地址文件指定
43	dfs.hosts.exclude		不允许与名字节点连接的主机地址文件设定
44	dfs.max.objects	0	文件数、目录数、块数的最大数量
45	dfs.namenode.decommission.interval	30	名字节点解除命令执行时的监测时间周期
46	dfs.namenode.decommission.nodes.per.interval	5	名字节点解除命令执行是否完检测次数
47	dfs.replication.interval	3	名字节点计算数据节点的复制工作的周期数
48	dfs.access.time.precision	3600000	允许访问文件的时间精确到 1 小时
49	dfs.support.append	false	是否允许链接文件指定
50	dfs.namenode.delegation.key.update-interval	86400000	名字节点上的代理令牌的主 key 的更新间隔时间为 24 小时
51	dfs.namenode.delegation.token.max-lifetime	604800000	代理令牌的有效时间最大值为 7 天
52	dfs.namenode.delegation.token.renew-interval	86400000	代理令牌的更新时间为 24 小时
53	dfs.datanode.failed.volumes.tolerated	0	决定停止数据节点提供服务允许卷的出错次数。0 次则任何卷出错都要停止数据节点

3. MapReduce 配置属性

序号	参数名称	参数值	功能描述
1	hadoop.job.history.location		作业跟踪管理器的静态历史文件的存放目录
2	hadoop.job.history.user.location		可以指定具体某个作业的跟踪管理器的历史文件存放目录
3	mapred.job.tracker.history.completed.location		已完成作业的历史文件的存放目录
4	io.sort.factor	10	排完序的文件的合并时的打开文件句柄数
5	io.sort.mb	100	排序文件的内存缓存大小为 100MB
6	io.sort.record.percent	0.05	排序线程阻塞的内存缓存剩余比率
7	io.sort.spill.percent	0.80	当缓冲占用量为该值时，线程需要将内容先备份到磁盘中
8	io.map.index.skip	0	索引条目的间隔设定
9	mapred.job.tracker	local	作业跟踪管理器是否和 MR 任务在一个进程中
10	mapred.job.tracker.http.address	0.0.0.0:50030	作业跟踪管理器的 HTTP 服务器访问端口和地址
11	mapred.job.tracker.handler.count	10	作业跟踪管理器的管理线程数，线程数比例是任务管理跟踪器数量的 0.04
12	mapred.task.tracker.report.address	127.0.0.1:0	任务管理跟踪器的主机地址和端口地址
13	mapred.local.dir	${hadoop.tmp.dir}/mapred/local	MR 的中介数据文件存放目录
14	mapred.system.dir	${hadoop.tmp.dir}/mapred/system	MR 的控制文件存放目录
15	mapreduce.jobtracker.staging.root.dir	${hadoop.tmp.dir}/mapred/staging	每个正在运行作业文件的存放区
16	mapred.temp.dir	${hadoop.tmp.dir}/mapred/temp	MR 临时共享文件存放区
17	mapred.local.dir.minspacestart	0	MR 本地中介文件删除时，不允许有任务执行的数量值
18	mapred.local.dir.minspacekill	0	MR 本地中介文件删除时，除非所有任务都已完成的数量值
19	mapred.tasktracker.expiry.interval	600000	任务管理跟踪器不发送心跳的累计时间间隔超过 600 秒，则任务管理跟踪器失效
20	mapred.tasktracker.resourcecalculatorplugin		指定的一个用户访问资源信息的类实例
21	mapred.tasktracker.taskmemorymanager.monitoring-interval	5000	监控任务管理跟踪器任务内存使用率的时间间隔
22	mapred.tasktracker.tasks.sleeptime-before-sigkill	5000	发出进程终止后，间隔 5 秒后发出进程消亡信号

（续）

序号	参数名称	参数值	功能描述
23	mapred.map.tasks	2	每个作业默认的 Map 任务数为 2
24	mapred.reduce.tasks	1	每个作业默认的 Reduce 任务数为 1
25	mapreduce.tasktracker.outofband.heartbeat	false	任务结束后发出一个额外的心跳信号
26	mapreduce.tasktracker.outofband.heartbeat.damper	1000000	当额外心跳信号发出量太多时，则适当阻止
27	mapred.jobtracker.restart.recover	false	允许任务管理器恢复时采用的方式
28	mapred.jobtracker.job.history.block.size	3145728	作业历史文件块的大小为 3MB
29	mapreduce.job.split.metainfo.maxsize	10000000	分隔元信息文件的最大值是 10MB
30	mapred.jobtracker.taskScheduler	org.apache.hadoop.mapred.JobQueueTaskScheduler	设定任务的执行计划实现类
31	mapred.jobtracker.taskScheduler.maxRunningTasksPerJob		作业同时运行的任务数的最大值
32	mapred.map.max.attempts	4	Map 任务的重试次数
33	mapred.reduce.max.attempts	4	Reduce 任务的重试次数
34	mapred.reduce.parallel.copies	5	在复制阶段时 Reduce 并行传送的值
35	mapreduce.reduce.shuffle.maxfetchfailures	10	取 Map 输出的最大重试次数
36	mapreduce.reduce.shuffle.connect.timeout	180000	Reduce 任务连接任务管理器获得 Map 输出时的总耗时是 3 分钟
37	mapreduce.reduce.shuffle.read.timeout	180000	Reduce 任务等待 Map 输出数据的总耗时是 3 分钟
38	mapred.task.timeout	600000	如果任务无读无写时的时间耗时为 10 分钟，将被终止
39	mapred.tasktracker.map.tasks.maximum	2	任管管理器可同时运行 Map 任务数为 2
40	mapred.tasktracker.reduce.tasks.maximum	2	任管管理器可同时运行 Reduce 任务数为 2
41	mapred.jobtracker.completeuserjobs.maximum	100	当用户的完成作业数达 100 个后，将其放入作业历史文件中
42	mapreduce.reduce.input.limit	−1	Reduce 输入量的限制
43	mapred.job.tracker.retiredjobs.cache.size	1000	作业状态为已不在执行的保留在内存中的量为 1000
44	mapred.job.tracker.jobhistory.lru.cache.size	5	作业历史文件装载到内存的数量
45	mapred.child.java.opts	−Xmx200m	启动 task 管理的子进程时的内存设置
46	mapred.child.env		子进程的参数设置
47	mapred.child.ulimit		虚拟机所需内存的设定
48	mapred.cluster.map.memory.mb	−1	
49	mapred.cluster.reduce.memory.mb	−1	
50	mapred.cluster.max.map.memory.mb	−1	
51	mapred.cluster.max.reduce.memory.mb	−1	

（续）

序号	参数名称	参数值	功能描述
52	mapred.job.map.memory.mb	−1	
53	mapred.job.reduce.memory.mb	−1	
54	mapred.child.tmp	/tmp	MapReduce 任务信息的存放目录
55	mapred.inmem.merge.threshold	1000	内存中的合并文件数设置
56	mapred.job.shuffle.merge.percent	0.66	
57	mapred.job.shuffle.input.buffer.percent	0.70	
58	mapred.job.reduce.input.buffer.percent	0.0	
59	mapred.map.tasks.speculative.execution	true	Map 任务的多实例并行运行标识
60	mapred.reduce.tasks.speculative.execution	true	Reduce 任务的多实例并行运行标识
61	mapred.job.reuse.jvm.num.tasks	1	每虚拟机运行的任务数
62	mapred.min.split.size	0	Map 的输入数据被分解的块数设置
63	mapred.jobtracker.maxtasks.per.job	−1	一个单独作业的任务数设置
64	mapred.submit.replication	10	提交作业文件的复制级别
65	mapred.tasktracker.dns.interface	default	任务管理跟踪器是否报告 IP 地址名的开关
66	mapred.tasktracker.dns.nameserver	default	作业和任务管理跟踪器之间通信方式采用的 DNS 服务的主机名或 IP 地址
67	tasktracker.http.threads	40	HTTP 服务器的工作线程数量
68	mapred.task.tracker.http.address	0.0.0.0:50060	任务管理跟踪器的 HTTP 服务器的地址和端口
69	keep.failed.task.files	false	失败任务是否保存到文件中
70	mapred.output.compress	false	作业的输出是否压缩
71	mapred.output.compression.type	RECORD	作业输出采用 NONE、RECORD、BLOCK 三种方式中的一种压缩的写入到流式文件
72	mapred.output.compression.codec	org.apache.hadoop.io.compress.DefaultCodec	压缩类的设置
73	mapred.compress.map.output	false	Map 的输出是否压缩
74	mapred.map.output.compression.codec	org.apache.hadoop.io.compress.DefaultCodec	Map 的输出压缩的实现类指定
75	map.sort.class	org.apache.hadoop.util.QuickSort	排序键的排序类指定
76	mapred.userlog.limit.kb	0	每个任务的用户日志文件大小
77	mapred.userlog.retain.hours	24	作业完成后的用户日志留存时间为 24 小时
78	mapred.user.jobconf.limit	5242880	Jobconf 的大小为 5MB
79	mapred.hosts		可与作业管理跟踪器连接的主机名
80	mapred.hosts.exclude		不可与作业管理跟踪器连接的主机名
81	mapred.heartbeats.in.second	100	作业管理跟踪器的每秒中到达的心跳数量为 100

（续）

序号	参数名称	参数值	功能描述
82	mapred.max.tracker.blacklists	4	任务管理跟踪器的黑名单列表的数量
83	mapred.jobtracker.blacklist.fault-timeout-window	180	任务管理跟踪器超时 180 分钟则任务将被重启
84	mapred.jobtracker.blacklist.fault-bucket-width	15	
85	mapred.max.tracker.failures	4	任务管理跟踪器的失败任务数设定
86	jobclient.output.filter	FAILED	控制任务的用户日志输出到作业端时的过滤方式
87	mapred.job.tracker.persist.jobstatus.active	false	是否持久化作业管理跟踪器的信息
88	mapred.job.tracker.persist.jobstatus.hours	0	持久化作业管理跟踪器的信息的保存时间
89	mapred.job.tracker.persist.jobstatus.dir	/jobtracker/	
	作业管理跟踪器的信息存放目录		
90	mapreduce.job.complete.cancel.delegation.tokens	true	恢复时是否变更令牌
91	mapred.task.profile	false	任务分析信息是否建设标志
92	mapred.task.profile.maps	0 ～ 2	设置 Map 任务的分析范围
93	mapred.task.profile.reduces	0 ～ 2	设置 Reduce 任务的分析范围
94	mapred.line.input.format.linespermap	1	每次切分的行数设置
95	mapred.skip.attempts.to.start.skipping	2	在跳转模式未被设定的情况下任务的重试次数
96	mapred.skip.map.auto.incr.proc.count	true	MapRunner 在调用 Map 功能后的增量处理方式设置
97	mapred.skip.reduce.auto.incr.proc.count	true	在调用 Reduce 功能后的增量处理方式设置
98	mapred.skip.out.dir		跳过记录的输出目录
99	mapred.skip.map.max.skip.records	0	
100	mapred.skip.reduce.max.skip.groups	0	
101	job.end.retry.attempts	0	Hadoop 尝试连接通知器的次数
102	job.end.retry.interval	30000	通知尝试回应的间隔操作为 30 秒
103	hadoop.rpc.socket.factory.class.JobSubmissionProtocol		指定与作业跟踪管理器的通信方式，默认采用 RPC 方式
104	mapred.task.cache.levels	2	任务缓存级别设置
105	mapred.queue.names	default	分隔作业队列的分隔符设定
106	mapred.acls.enabled	false	指定 ACL 访问控制列表
107	mapred.queue.default.state	RUNNING	定义队列的状态
108	mapred.job.queue.name	default	已提交作业的队列设定
109	mapreduce.job.acl-modify-job		指定可修改作业的 ACL 列表
110	mapreduce.job.acl-view-job		指定可浏览作业的 ACL 列表

（续）

序号	参数名称	参数值	功能描述
111	mapred.tasktracker.indexcache.mb	10	任务管理跟踪器的索引内存的最大容器
112	mapred.combine.recordsBeforeProgress	10000	在聚合处理时的记录块数
113	mapred.merge.recordsBeforeProgress	10000	在汇总处理时的记录块数
114	mapred.reduce.slowstart.completed.maps	0.05	
115	mapred.task.tracker.task-controller	org.apache.hadoop.mapred.DefaultTaskController	任务管理器的设定
116	mapreduce.tasktracker.group		任务管理器的组成员设定
117	mapred.healthChecker.script.path		脚本的绝对路径指定，这些脚本是为心跳服务的
118	mapred.healthChecker.interval	60000	节点心跳信息的间隔
119	mapred.healthChecker.script.timeout	600000	
120	mapred.healthChecker.script.args		参数列表
121	mapreduce.job.counters.limit	120	作业计数器的最小值

51CTO.com
技术成就梦想

联|袂|推|荐

随着大数据时代的到来，企业的数据量呈指数级增长，对海量数据的存储和分析能力成为一个企业，尤其是互联网企业能否适应这个时代的重要标志，也是企业面临的一个挑战。Hadoop技术很好地解决了海量数据存储和分析问题，短短几年，80%左右的大中型互联网公司开始使用Hadoop技术，银行、电信等传统行业也开始做相关的分析和调研。

Hadoop技术涉及的知识点较多，专业性较强，所以学习的曲线比较陡峭。本书作者在Hadoop开发和运维领域有近4年经验，对Hadoop技术人员学习过程中可能会遇到的问题有一定了解，在此基础上写了这本书，旨在帮助广大读者更快、更好地学习Hadoop技术。

本书主要内容

- Hadoop生态系统、常用项目及应用；
- Hadoop的安装和配置；
- MapReduce的开发流程、方法和技巧；
- MapReduce的工作原理和高级应用；
- HDFS和Hadoop文件I/O的原理及应用；
- MapReduce应用的测试与调试，以及源代码的编译；
- 数据仓库工具Hive和数据库HBase的原理和应用；
- Mahout算法框架的使用；

……

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

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

上架指导：计算机/程序设计/大数据
ISBN 978-7-111-45244-7
9 787111 452447 >
定价：79.00元