
HBase 权威指南

[美] *Lars George* 著

代志远 刘 佳 蒋 杰 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc. 授权人民邮电出版社出版

人 民 邮 电 出 版 社
北 京

图书在版编目 (C I P) 数据

HBase权威指南 / (美) 乔治 (George, L.) 著 ; 代志远, 刘佳, 蒋杰译. — 北京 : 人民邮电出版社, 2013.10

ISBN 978-7-115-31889-3

I. ①H… II. ①乔… ②代… ③刘… ④蒋… III. ①计算机网络—信息存储—指南 IV. ①TP393-62

中国版本图书馆CIP数据核字 (2013) 第105049号

版权声明

Copyright ©2011 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2011. Authorized translation of the English edition, 2011 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书中文简体版由 O'Reilly Media, Inc. 授权人民邮电出版社出版。未经出版者书面许可, 对本书的任何部分不得以任何方式复制或抄袭。

版权所有, 侵权必究。

-
- ◆ 著 [美] Lars George
译 代志远 刘佳 蒋杰
责任编辑 杨海玲
执行编辑 赵越
责任印制 程彦红 杨林杰
- ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
三河市海波印务有限公司印刷
- ◆ 开本: 787×1000 1/16
印张: 31.5
字数: 646千字
印数: 1-4 000册
- 2013年10月第1版
2013年10月河北第1次印刷

著作权合同登记号 图字: 01-2011-7480 号

定价: 89.00 元

读者服务热线: (010)67132692 印装质量热线: (010)67129223
反盗版热线: (010)67171154

内容提要

本书探讨了如何通过使用与 HBase 高度集成的 Hadoop 将 HBase 的可伸缩性变得简单；把大型数据集分布到相对廉价的商业服务器集群中；使用本地 Java 客户端，或者通过提供了 REST、Avro 和 Thrift 应用编程接口的网关服务器来访问 HBase；了解 HBase 架构的细节，包括存储格式、预写日志、后台进程等；在 HBase 中集成 MapReduce 框架；了解如何调节集群、设计模式、拷贝表、导入批量数据和删除节点等。

本书适合使用 HBase 进行数据库开发的高级数据库研发人员阅读。

O'Reilly Media, Inc. 介绍

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

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

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

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

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

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

——Linux Journal

感谢我的妻子 Katja, 感谢我的女儿 Laura,
以及我的儿子 Leon。我爱你们!

推荐序

近年来，新兴的互联网服务领域，以及电信、金融和交通等各传统行业出现了数据资产的爆炸性增长，这些数据资产的类型以非结构化和半结构化为主，如何低成本且高效率地存储和处理 PB 甚至 EB 量级的数据成为了极大的挑战。

Google 公司提出的 MapReduce 编程框架、GFS 文件系统和 BigTable 存储系统成为了大数据处理技术的开拓者和领导者，而源于这三项技术的 Apache Hadoop 等开源项目则成为了大数据处理技术的事实标准，迅速推广至国内外各大互联网企业，成为了 PB 量级大数据处理的成熟技术和系统。面对不同的应用需求，基于 Hadoop 的数据处理工具也应运而生，例如，Hive、Pig 等已能够很好地解决大规模数据的离线式批量处理问题。但是，Hadoop HDFS 适合于存储非结构化数据，且受限于 Hadoop MapReduce 编程框架的高延迟数据处理机制，使得 Hadoop 无法满足大规模数据实时处理应用的需求。

传统的信息系统和 Web 应用大多采用 LAMP 架构构建，并使用关系型数据库存储、组织和管理结构化或半结构化数据。通用的关系型数据库无法很好地应对在数据规模剧增时导致的系统扩展性和性能问题。因此，业界出现了一类面向半结构化数据存储和处理的高可扩展、低写入/查询延迟的系统，例如，键值存储系统、文档存储系统和类 BigTable 存储系统等，这些特性各异的系统也可统称为 NoSQL 系统。Apache HBase 就是其中已迈向实用的成熟系统之一。HBase 之所以能成为迈向实用的成熟系统，一是核心思想来源于 Google 的 BigTable，二是有 Apache 及 Hadoop 开源社区的支撑，三是有诸如 Facebook、淘宝和支付宝等互联网公司的应用实践，保证了 HBase 系统的稳定性和可用性。目前，作为关系型数据库的有益补充，HBase 已成功应用于互联网服务领域和传统行业的众多在线式数据分析处理系统中。

本书涉及 HBase 使用和开发过程中的各方面内容，章节组织由浅入深，内容阐述细致入微并且贴近实际，可以作为参考书以方便读者在开发过程中随时查阅。本书译者之一刘佳向 HBase 开源社区提交过多项错误修复和新功能，参与过多项 HBase 有关的大数据分析系统研发项目，积累了丰富的 HBase 系统开发经验。我相信本书对于 HBase 使用者和开发者来说，都是及时和不可或缺的。

查礼

于中科院计算所

2013 年 7 月

译者序

随着历史数据的积累和数据量的高速增长，海量数据领域越来越被重视，且该领域涌现出了非常多的新技术。技术的发展和时间的沉淀使得 HBase 开始被大家广泛认可，成为海量数据在线存储领域的首选。

本书是讲述 HBase 相关技术的第一本图书，也是著名图书出版商 O'Reilly 出版发行的 HBase 权威书籍。

本书从架构、开发、应用和运维等多个角度描述了 HBase，深入介绍了 HBase 内核的原理和机制以及社区的发展方向，并提供了应用层面的多种示例和源代码。本书为每个用例和知识点提供了丰富的解释和注意要点，使用户可以由浅入深地了解原理并深度使用其功能，并且体现了在 HBase 教学方面的最新进展和最高水平。

本书的成功离不开 Lars George 的努力。在 HBase 还处于萌芽时期时，Lars George 就开始投入了大量的精力，从修复 HBase 中的问题到优化性能，推广 HBase 并编写 HBase 可用性文档，他是 HBase 领域里大师级的人物。而这本《HBase 权威指南》花费了 Lars George 许多的时间和精力。

阅读本书后，我们不得不承认这本大师级的著作很好地应对了社区中 HBase 发展所面临的挑战。不得不说的是，本书著作和翻译经历的时间较长，而社区中 HBase 发展速度较快，许多版本已经发行，许多问题也得以修复，因此，本书最终落地后会与最新 HBase 版本的功能特性有少许描述性出入，还望广大读者见谅。

在翻译过程中，我们深刻地发现国外技术领域的专业性，深深地被世界级的高水平技术所震撼。我们由衷地希望本书中文版的出版能够推动国内 HBase 教学、使用和发展。本书译者代志远在翻译期间就职于阿里巴巴，译者刘佳是中科院计算所研究生，现为普泽天玑技术总监，译者蒋杰在腾讯担任数据与运营支撑平台副总经理。

感谢人民邮电出版社的编辑，他们为保证本书质量付出了大量的努力。

本书中概念和术语较多，许多概念和术语尚无公认的中文译法，加之译者水平有限，译文中若有不妥之处，恳请读者批评指正。

代志远

2013 年 7 月

序

HBase 的故事开始于 2006 年，当时旧金山的 Powerset 创业公司试图建立一个网页的自然语言搜索引擎，但他们构建索引时涉及一个复杂的过程，比标准的分词索引结果集大了两个数量级。他们曾经使用 Amazon Web Service 存储索引，但是爬虫抓取过程中的负荷主要集中在这一端。（叮铃铃，叮铃铃“您好！这里是 AWS，无论您正在运行什么，请停止运行！”）他们恰好在寻求解决方案，而此时 Google 的 BigTable 论文发表了。

Powerset 公司的工程负责人 Chad Walters 此时发表了如下的言论：

与 Google 基于 GFS (Google File System) 构建的 BigTable 一样，在 Hadoop 的分布式文件系统 (HDFS) 基础上构建一个开源系统是一个非常不错的主意：（1）这套架构是成熟的并且可拓展；（2）我们可以直接利用 Hadoop 的 HDFS；（3）我们可以扩大 Hadoop 生态系统的影响力。

BigTable 论文发表后，在社区中，人们一次又一次地讨论基于 Hadoop 构建类 BigTable 系统的可行性。在 2007 年年初，Mike Cafarella 出乎意料地在 Hadoop 的问题跟踪系统中上传了一个包含 30 多个 Java 文件的 tar 包：“我实现了一个类 BigTable 架构的存储系统 demo，叫做 HBase，虽然它还不完善，但是它已经做好准备让用户进行实验和检查了。”Mike 与 Doug Cutting 在 Nutch（一个开源搜索引擎）项目中长期共事，Doug Cutting 在 Nutch 中实现了一个类似于 Google 分布式文件系统的项目来管理磁盘，因此 Nutch 中构建的索引存储可以不仅仅存储在一台机器中（Nutch 分布式文件系统最后发展成为了 HDFS）。

Powerset 公司的 Jim Kellerman 增加了测试用例并填补了其他空白，使得 HBase 可以作为 Hadoop 的一部分代码进行提交。Doug Cutting 在 2007 年 4 月 3 日完成了 HBase 的第一次代码提交，代码提交到了 Hadoop 工程根目录的 *contrib* 子目录中。HBase 的第一个版本在 2007 年 10 月作为 Hadoop 0.15.0 的一部分发布了。

没过多久，本书作者 Lars 开始在 #hbase IRC 交流频道出现。当时 Lars 面临大数据的问题，并且尝试用 HBase 来解决这个问题。经过一番辛苦的摸索，Lars 成为了 Powerset 之外的 HBase 的第一个用户。我清楚地记得，Lars 当时记录了他在 WorldLingo 公司的生产集群的问题反馈清单，Lars 当时在这家公司担任 CTO。清单展示了他们的生产集群中 HBase 的 10 个版本（从 Hadoop 0.15.1 到 HBase 0.20），每个版本的集群都有将近 40 台机器。

在这些年来所有为 HBase 做出贡献的人中，具有史诗般意义的就是 Lars，因为他写了这

本书。Lars 一直在为 HBase 贡献文档，HBase 想要被更好地使用和推广，就需要有良好的文档。每个人都同意 Lars 的想法，并且能够专注地投入编程工作中，因此 Lars 在工作在欧洲旅行期间开始编写如何使用 HBase 的文档和架构描述，并承担起了 HBase 非官方的欧洲大使职责。Lars 在其关于 HBase 的博客 (<http://www.larsgeorage.com>) 中记录了 HBase 的工作原理，并在关键阶段推动了 HBase 社区的发展（一篇重要的博客文章解释了 HBase 依赖 Ivy 进行编译是个非常棒的主意）。

在微软公司赞助 HBase 的时期，HBase 也发生了非常有趣的事情。Powerset 在 2008 年 7 月被微软收购，在此期间其员工不允许贡献代码，因为微软法务部门需要审核 HBase 代码库并查看 HBase 与 SQLServer 的关系，直到一个月后才宣布重新贡献代码给社区（我是微软的一名员工，全职为 Apache 开源项目工作）。之后 Facebook 也开始使用 HBase，用于存储海量的邮件信息或点击信息，后来 Yahoo 部署了 1000 台 HBase 集群用于定位微软 Bing 的爬虫快照。同期非运行在 HDFS 上的 MapR 系统也仍处在开发中。

我很清楚，社区和 HBase 的发展得利于一群 HBase 的核心 committer 的辛苦努力。一些核心开发成员，如 Todd Lipcon、Gary Helmling 和 Nicolas Spiegelberg，已经付出了多年的努力，没有他们我们无法走到今天这一步，HBase 目前已经从一个分支代码成长为了一个独立存储项目。Jonathan Gray 冒险将其初创的 streamy.com 网站基于 HBase 进行建设，Andrew Purtell 在趋势科技组建了一只 HBase 团队，Ryan Rawson 得到了 StumbleUpon 的赞助，这是 HBase 在 Powerset、微软之后获得的最主要的赞助，并且还发掘了一个非常厉害的 commiter——John-Daniel Cryans，而当时 Cryans 还只是一个繁忙的学生。之后 Lars 不断地修复缺陷，并撰写文档。因此，Lars 是撰写第一本关键的 HBase 书籍的最佳人选，也让所有人都可以了解 HBase。

——Michael Stack，HBase 项目管理人

前言

你阅读本书的理由可能有很多。可能是因为听说了 Hadoop，并了解到它能够在合理的时间范围内处理 PB 级的数据，在研读 Hadoop 的过程中发现了一个处理随机读写的系统，它叫做 HBase。或者将其称为目前流行的一种新的数据存储架构，传统数据库解决大数据问题时成本更高，更适合的技术范围是 NoSQL。

无论你是如何来到这里的，我都希望你能够了解并学习如何在企业或组织中使用 HBase 解决海量数据问题。你可能有关系型数据库的背景，但更希望去研究这个“列式存储”系统；也许你听说 HBase 能够不费力地进行线性拓展，并且有足够的理由成为下一代网络系统。

在 2007 年年底，我曾面临百万级的文档存储需求，并且需要满足容错和可扩展等要求。我拥有丰富的 MySQL 数据库经验，并使用这种数据库来存储数据，最终服务于我的网站的用户。MySQL 可以在运行于一台服务器的同时，拥有另一台备份服务器，其无法应对如此海量数据的处理，于是我只好寻找其他可用的存储数据库。

我的口头禅是：“Google 是如何解决这类问题的？”后来我接触了 Hadoop，在短暂使用 Hadoop 之后，我面临随机读写的问题——但是这个问题已经得以解决：2006 年 Google 发表了 BigTable^① 论文，Hadoop 开发者拥有了 BigTable 的开源实现，并称其为 HBase。这就是解决我的问题的答案，所以这一切看起来顺理成章……

如今，我已经不再回忆自己刚开始接触 Hadoop 和 HBase 的日子有多艰难了。我希望可以从今天开始使用 HBase，HBase 目前已经成熟，接近 1.0 版本，并且目前已经有大量知名企业在使用，如 Facebook、Adobe、Twitter、Yahoo!、趋势科技和 StumbleUpon (<http://wiki.apache.org/hadoop/HBase/PoweredBy>)。我的集群是第一个生产集群（迄今为止），到目前也遇到了许多有趣的问题。

如预期所料，HBase 从 0.1x 版本开始成为社区项目，我有幸为这个项目贡献代码，并最终被要求成为全职的 committer。

过去几年我从其他开发者身上学到了许多知识，并且一直在努力地学习。我的信念是，我们还远没有达到这个技术的顶峰，而这个技术也会随着时间的推移不断地成长和演变。让我们用这本书对整个 HBase 开发者社区致以敬意，我的写作目标不仅仅是覆盖 HBase 的工作机制，而且还要为用户提供如何将这一技术用到自己的使用场景中。

① 见链接 <http://labs.google.com/papers/bigtable-osdi06.pdf>。

我强烈地感觉到你来到这里的原因是打算使用 HBase 解决你遇到的问题。现在让我们解开谜底。

基本信息

在我们开始之前，以下是一些基本介绍。

HBase 版本

在写这本书时，HBase 社区已经决定发布 0.92.0 版本，社区主干的代码已经开发完成 (<http://svn.apache.org/viewvc/hbase/trunk/>)，之前刚刚发布了 0.91.0- SNAPSHOT 版本。

我们很难跟上开发的步伐，因为这本书有一个截止日期——在 0.92.0 版本发布之前。因此，它只能记录到一个特定的修订版 1130916 (<http://svn.apache.org/viewvc/hbase/trunk/?pathrev=1130916>) 为止。如果你发现书中描述的和 HBase 提供的之间似乎并不匹配，你可以使用以上的版本与最新版本比对近期的修改。

我们正尽一切努力更新本书网站 (<http://www.hbasebook.com>) 上的 JDiff (比较不同软件版本的工具) 文档。使用它可以快速看到不同版本间有什么变化。

编译示例程序

有关这本书的示例代码可以在 GitHub 中 (<http://github.com/larsgeorge/hbase-book>) 找到更详细的信息。为了简洁，本书只提供了部分代码片段，即重要代码，尽量避免出现重复的样板代码。

每个例子的名字都能与链接库中的文件名匹配上，因此很容易就能找到这些示例。每章都有独立目录，呈树形结构，更利于用户查找。例如，正在读第 3 章，你可以到对应目录下查找第 3 章完整的示例源代码。

许多示例中所示特性都使用了内部辅助类来协助运行。例如，使用了 HBaseHelper 类来建立测试环境和收集测试结果。你可以根据具体情况来修改测试代码，或植入错误数据以观察示例中所示特性的行为。

编译示例代码需要借助以下辅助的命令行工具。

Java

HBase 是使用 Java 语言实现的系统，因此必然需要 Java 环境。2.2.3 节的“Java”描述了应该如何安装 Java 环境。

Git

示例源代码是通过 GitHub 来提供托管服务，因此我们还需要支持 Git——一个分布

式版本协作控制系统，Linux 内核开发最初就依托此系统做版本控制。^①开源社区提供了非常多的 Git 客户端二进制安装包。

此外，用户还可以通过 GitHub 提供的下载链接（<https://github.com/larsgeorge/hbase-book/archives/master>）下载文件的静态快照。

Maven

书中提供的代码编译过程是通过 Apache Maven^② 完成的。它使用 Project Object Model (POM) 来描述编译的过程，其中包括有哪些代码可以编译以及哪些不需要编译。因此读者需要首先下载 Maven 安装库并在本机上进行安装。

一旦读者按照上述信息安装好所需要的工具，就可以按照以下命令开始编译项目。

```
~$ cd /tmp
/tmp$ git clone git://github.com/larsgeorge/hbase-book.git
Initialized empty Git repository in /tmp/hbase-book/.git/
remote: Counting objects: 420, done.
remote: Compressing objects: 100% (252/252), done.
remote: Total 420 (delta 159), reused 144 (delta 58)
Receiving objects: 100% (420/420), 70.87 KiB, done.
Resolving deltas: 100% (159/159), done.
/tmp$ cd hbase-book/
/tmp/hbase-book$ mvn package
[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   HBase Book
[INFO]   HBase Book Chapter 3
[INFO]   HBase Book Chapter 4
[INFO]   HBase Book Chapter 5
[INFO]   HBase Book Chapter 6
[INFO]   HBase Book Chapter 11
[INFO]   HBase URL Shortener
[INFO] -----
[INFO] Building HBase Book
[INFO]   task-segment: [package]
[INFO] -----
[INFO] [site:attach-descriptor {execution: default-attach-descriptor}]
[INFO] -----
[INFO] Building HBase Book Chapter 3
[INFO]   task-segment: [package]
[INFO] -----
[INFO] [resources:resources {execution: default-resources}]
...
[INFO] -----
[INFO] Reactor Summary:
```

① 见网址 <http://git-scm.com/>。

② 见网址 <http://maven.apache.org/>。

```

[INFO] -----
[INFO] HBase Book ..... SUCCESS [1.601s]
[INFO] HBase Book Chapter 3 ..... SUCCESS [3.233s]
[INFO] HBase Book Chapter 4 ..... SUCCESS [0.589s]
[INFO] HBase Book Chapter 5 ..... SUCCESS [0.162s]
[INFO] HBase Book Chapter 6 ..... SUCCESS [1.354s]
[INFO] HBase Book Chapter 11 ..... SUCCESS [0.271s]
[INFO] HBase URL Shortener ..... SUCCESS [4.910s]
[INFO] -----
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 12 seconds
[INFO] Finished at: Mon Jun 20 17:08:30 CEST 2011
[INFO] Final Memory: 35M/81M
[INFO] -----

```

这段信息意味着有依赖关系的库已经下载到了本地目录，源代码也随后编译成功了。在每个子目录的 *target* 文件夹下都留下了编译后的 JAR 文件，即每章的源代码和 *.class* 文件归档：

```

/tmp/hbase-book$ ls -l ch04/target/
total 152
drwxr-xr-x 48 larsgeorge wheel 1632 Apr 15 10:31 classes
drwxr-xr-x 3 larsgeorge wheel 102 Apr 15 10:31 generated-sources
-rw-r--r-- 1 larsgeorge wheel 75754 Apr 15 10:31 hbase-book-ch04-1.0.jar
drwxr-xr-x 3 larsgeorge wheel 102 Apr 15 10:31 maven-archiver

```

上述场景中，*hbase-book-ch04-1.0.jar* 文件包含了第 4 章的示例代码。我们可以使用如下命令进行测试：

```

/tmp/hbase-book$ cd ch04/
/tmp/hbase-book/ch04$ bin/run.sh client.PutExample
/tmp/hbase-book/ch04$ bin/run.sh client.GetExample
Value: vall

```

bin/run.sh 已经配置了所需的 Java classpath，并添加了依赖的 JAR 文件。

Hush: HBase URL 简写

通过了解 HBase 提供的功能来了解 HBase 是一个好办法。本书使用了具有代表性的表的集合作为例子，并包含具体的数据，这样可以很容易地理解经过操作后数据状态前后改变的过程。读者可以执行每一个示例并查阅结果，该结果应该与书中提供的结果匹配。修改示例是进一步探讨和研究 HBase 的好方法，借助辅助类可以进行更有效的验证。

在前期的学习过程中，了解系统所有的功能是非常重要的一个环节，而这本书恰好提供了一个现实中的例子来展示 HBase 的大部分功能。同时这个例子也被用于与其他存储数据库进行对比，例如，与传统的基于 RDBMS 的系统进行对比。

这个应用的名称叫做 Hush——HBase URL Shortener (HBase URL 简写)。互联网提供了非常多的服务, 每个服务都可以通过 URL 来访问。例如, 你提交了一个网页, 但你得到了一个返回的短网址。例如, Twitter 每次只运行并发送最多 140 个字符, 但 URL 最长可达到 4096 字节, 因此不得不将 URL 简化到 20 字节以下, 以节省更多的空间给实际内容。

例如, 这是由 Google 地图引用 California 的 Sebastopol 的 URL:

```
http://maps.google.com/maps?f=q&source=s_q&hl=en&geocode=&q=Sebastopol, \
+CA,+United+States&aq=0&sll=47.85931,10.85165&sspn=0.93616,1.345825&ie=UTF8& \
hq=&hnear=Sebastopol,+Sonoma,+California&z=14
```

经过 Hush 模式, 其可以转化成如下 URL:

```
http://hush.li/1337
```

显然, 这个 URL 更短, 更有利于复制到电子邮件, 或通过限制字数的媒体 (如 Twitter 和 SMS) 发送。

但是, 这种服务并不仅仅是一张大的查询表。诚然, 非常多的用户希望能够映射短网址到完整网址, 但需求并不仅仅如此。用户更多的是想了解短网址究竟被使用了多少次? 因此短网址就需要保留一个计数器, 用于在用户点击网址的时候进行统计。

更高级的功能是, 用户可以使用固定的域名或定制的短网址 ID 而不是自动生成的地址, 就像上面例子描述的那样。用户必须能够登录短网址跟踪, 并查看日、周、月报表。

所有的这些都在 Hush 模式中实现了, 用户可以很容易地编译和运行。它使用了 HBase 的大多数功能, 在适当的时机我们会讨论这些问题。

用户可以自己创建账号并使用 Hush, 这也是一个了解如何从已有系统导入数据的好例子。本书使用了网络上提供的数据集: Delicious RSS feed。这里面有少量集合是可以随意下载的。

使用场景: Hush

留意本书中从 Hush 角度解释的功能。很多地方使用了 Hush 的示例代码, 但是它仅仅保持在非常简单的展示这一层面。Hush 作为一个使用场景更多的是应用在生产系统中。

Hush 没有华丽的 UI, 只有朴实的功能, Hush 在经过负载均衡后达到数千的 TPS 请求完全没有困难。

有关 Hush 如何运行的代码片段已经在本书中进行了描述, 完整的代码可以从 Git 库中下载并独立运行、调整和学习有关它的一切!

运行 Hush

使用示例代码编译和运行 Hush 非常简单, 一旦用户可以克隆或下载, 就执行以下命令:

```
$ mvn package
```

编译整个工程后，读者就可以使用以下命令来运行 Hush:

```
$ hush/bin/start-hush.sh
```

```
=====
Starting Hush...
=====
INFO [main] (HushMain.java:57) - Initializing HBase
INFO [main] (HushMain.java:60) - Creating/updating HBase schema
...
INFO [main] (HushMain.java:90) - Web server setup.
INFO [main] (HushMain.java:111)- Configuring security.
INFO [main] (Slf4jLog.java:55) - jetty-7.3.1.v20110307
INFO [main] (Slf4jLog.java:55) - started ...
INFO [main] (Slf4jLog.java:55) - Started SelectChannelConnector@0.0.0. 0:8080
```

看到控制台中的最后一行输出后，读者可以直接通过浏览器访问 <http://localhost:8080> 进入 Hush 服务器进行处理。

由于数据已经存储在了 HBase 中，因此用户可以放心地执行 Ctrl+C 来停止 Hush 的启动脚本。

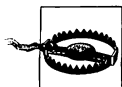
排版约定

本书使用如下排版约定。

- 斜体：URL、文件名、文件后缀和 Unix 命令。
- 等宽代码字体：用于程序清单、功能变量、环境变量、数据类型、描述和关键字等。
- 加粗的等宽代码字体：展示命令或其他应该由用户输入的文本，也用于代码清单中强调的部分。
- 斜的等宽代码字体：展示应该使用用户所提供的值或根据上下文确定的值来替换的文本。



这个图表示一个提示、建议或一般的注意。



这个图表示警告或需要谨慎处理。

代码示例

本书的目标是帮助你完成工作。一般而言，你可以在自己的程序和文档中使用本书中的代码，如果你要复制的不是核心代码，则无须取得我们的许可。例如，你可以在程序中使用本书中的多个代码块，无须获取我们许可。但是，要销售或分发来源于 O'Reilly 图书中的示例的光盘则需要取得我们的许可。通过引用本书中的示例代码来回答问题时，不需要事先获得我们的许可。但是，如果你的产品文档中融合了本书中的大量示例代码，则需要取得我们的许可。

在引用本书中的代码示例时，如果能列出本书的属性信息是最好不过了。属性信息通常包括书名、作者、出版社和 ISBN。例如：“*HBase: The Definitive Guide by Lars George* (O'Reilly). Copyright 2011 Lars George, 978-1-449-39610-7”。

在使用书中的代码时，如果不确定是否属于合理使用，或是否超出了我们的许可，请通过 permissions@oreilly.com 与我们联系。

我们的联系方式

如果你想就本书发表评论或有任何疑问，敬请联系出版社。

美国：

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

中国：

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

我们还为本书建立了一个网页，其中包含了勘误表、示例和其他额外的信息。你可以通过如下地址访问该网页：

<http://www.oreilly.com/catalog/9781449396107>

作者还为本书创建了一个网站：

<http://hbasebook.com/>

关于本书的技术性问题或建议，请发邮件到：

bookquestions@oreilly.com

欢迎登录我们的网站（<http://www.oreilly.com>），查看更多我们的书籍、课程、会议和最

新动态等信息。

我们的其他联系方式如下。

Facebook: <http://facebook.com/oreilly>

Twitter: <http://twitter.com/oreillymedia>

YouTube: <http://www.youtube.com/oreillymedia>

致谢

首先,我要感谢我已故的父亲 Reiner 和我的母亲 Ingrid,他们一直支持我的理想,使我成为一个更优秀的人。

在写这本书时我获得了 HBase 社区的很多支持,没有这些支持, HBase 不可能有今天的成功。现在遍布世界各地的核心公司向社区提交了非常多的代码,还有邮件列表中对许多问题的支持、博客文章,这些都促进了开源的发展。我是站在你们的肩上前进的!

感谢本书贡献者 Jean-Daniel Cryans、Jonathan Gray、Gary Helmling、Todd Lipcon、Andrew Purtell、Ryan Rawson、Nicolas Spiegelberg、Michael Stack 和 Ted Yu,以及名誉贡献者: Mike Cafarella、Bryan Duxbury 和 Jim Kellerman。

还要感谢本书的审校者 Patrick Angeles、Doug Balog、Jeff Bean、Po Cheung、Jean-Daniel Cryans、Lars Francke、Gary Helmling、Michael Katzenellenbogen、Mingjie Lai、Todd Lipcon、Ming Ma、Doris Maassen、Cameron Martin、Matt Massie、Doug Meil、Manuel Meßner、Claudia Nielsen、Joseph Pallas、Josh Patterson、Andrew Purtell、Tim Robertson、Paul Rogalinski、Stefan Rudnitzki、Eric Sammer、Michael Stack 和 Suraj Varma。

所有的 HBase 贡献者,请继续努力!

最后,还要感谢我的雇主——Cloudera 公司,它为我提供了写这本书的时间。

目录

第 1 章 简介	1
1.1 海量数据的黎明	1
1.2 关系数据库系统的问题	5
1.3 非关系型数据库系统 Not-Only-SQL (简称 NoSQL)	7
1.3.1 维度	9
1.3.2 可扩展性	12
1.3.3 数据库的范式化和反范式化	12
1.4 结构	15
1.4.1 背景	15
1.4.2 表、行、列和单元格	16
1.4.3 自动分区	20
1.4.4 存储 API	21
1.4.5 实现	22
1.4.6 小结	25
1.5 HBase: Hadoop 数据库	25
1.5.1 历史	26
1.5.2 命名	27
1.5.3 小结	27
第 2 章 安装	28
2.1 快速启动指南	28
2.2 必备条件	31
2.2.1 硬件	31
2.2.2 软件	37
2.3 HBase 使用的文件系统	47
2.3.1 本地模式	48
2.3.2 HDFS	49
2.3.3 S3	49
2.3.4 其他文件系统	50
2.4 安装选项	50
2.4.1 Apache 二进制发布包	50
2.4.2 编译源码	52
2.5 运行模式	53

2.5.1	单机模式	53
2.5.2	分布式模式	53
2.6	配置	57
2.6.1	hbase-site.xml 与 hbase-default.xml	58
2.6.2	hbase-env.sh	59
2.6.3	regionserver	59
2.6.4	log4j.properties	59
2.6.5	配置示例	59
2.6.6	客户端配置	61
2.7	部署	61
2.7.1	基于脚本	62
2.7.2	Apache Whirr	63
2.7.3	Puppet 与 Chef	63
2.8	操作集群	64
2.8.1	确定安装运行	64
2.8.2	Web UI 介绍	65
2.8.3	Shell 介绍	66
2.8.4	关闭集群	66
第 3 章	客户端 API: 基础知识	68
3.1	概述	68
3.2	CRUD 操作	69
3.2.1	put 方法	69
3.2.2	get 方法	87
3.2.3	删除方法	97
3.3	批量处理操作	107
3.4	行锁	110
3.5	扫描	114
3.5.1	介绍	114
3.5.2	ResultScanner 类	117
3.5.3	缓存与批量处理	119
3.6	各种特性	125
3.6.1	HTable 的实用方法	125
3.6.2	Bytes 类	127
第 4 章	客户端 API: 高级特性	129
4.1	过滤器	129
4.1.1	过滤器简介	129
4.1.2	比较过滤器	132

4.1.3	专用过滤器	139
4.1.4	附加过滤器	147
4.1.5	FilterList	151
4.1.6	自定义过滤器	153
4.1.7	过滤器总结	159
4.2	计数器	160
4.2.1	计数器简介	160
4.2.2	单计数器	163
4.2.3	多计数器	164
4.3	协处理器	166
4.3.1	协处理器简介	167
4.3.2	Coprocessor 类	168
4.3.3	协处理器加载	171
4.3.4	RegionObserver 类	174
4.3.5	MasterObserver 类	180
4.3.6	endpoint	184
4.4	HTablePool	190
4.5	连接管理	194
第 5 章	客户端 API: 管理功能	197
5.1	模式定义	197
5.1.1	表	197
5.1.2	表属性	199
5.1.3	列族	202
5.2	HBaseAdmin	207
5.2.1	基本操作	208
5.2.2	表操作	209
5.2.3	模式操作	217
5.2.4	集群管理	219
5.2.5	集群状态信息	222
第 6 章	可用客户端	230
6.1	REST、Thrift 和 Avro 的介绍	230
6.2	交互客户端	233
6.2.1	原生 Java	233
6.2.2	REST	233
6.2.3	Thrift	240
6.2.4	Avro	244
6.2.5	其他客户端	245

6.3	批处理客户端	246
6.3.1	MapReduce	246
6.3.2	Hive	246
6.3.3	Pig	252
6.3.4	Cascading	256
6.4	Shell	257
6.4.1	基础	257
6.4.2	命令	259
6.4.3	脚本	263
6.5	基于Web的UI	265
6.5.1	master的UI	265
6.5.2	region服务器的UI	270
6.5.3	共享页面	272
第7章	与MapReduce集成	275
7.1	框架	275
7.1.1	MapReduce介绍	275
7.1.2	类	276
7.1.3	支撑类	279
7.1.4	MapReduce的执行地点	279
7.1.5	表拆分	280
7.2	在HBase之上的MapReduce	281
7.2.1	准备	281
7.2.2	数据流向	286
7.2.3	数据源	291
7.2.4	数据源与数据流向	293
7.2.5	自定义处理	296
第8章	架构	299
8.1	数据查找和传输	299
8.1.1	B+树	299
8.1.2	LSM树	300
8.2	存储	302
8.2.1	概览	303
8.2.2	写路径	304
8.2.3	文件	305
8.2.4	HFile格式	313
8.2.5	KeyValue格式	316
8.3	WAL	316

8.3.1	概述	317
8.3.2	HLog 类	318
8.3.3	HLogKey 类	319
8.3.4	WALEdit 类	319
8.3.5	LogSyncer 类	319
8.3.6	LogRoller 类	320
8.3.7	回放	321
8.3.8	持久性	324
8.4	读路径	325
8.5	region 查找	328
8.6	region 生命周期	330
8.7	ZooKeeper	330
8.8	复制	333
8.8.1	Log Edit 的生命周期	334
8.8.2	内部机制	335
第 9 章	高级用法	339
9.1	行键设计	339
9.1.1	概念	339
9.1.2	高表与宽表	341
9.1.3	部分键扫描	342
9.1.4	分页	343
9.1.5	时间序列	344
9.1.6	时间顺序关系	348
9.2	高级模式	350
9.3	辅助索引	350
9.4	搜索集成	354
9.5	事务	357
9.6	布隆过滤器	358
9.7	版本管理	361
9.7.1	隐式版本控制	361
9.7.2	自定义版本控制	364
第 10 章	集群监控	366
10.1	介绍	366
10.2	监控框架	367
10.2.1	上下文、记录和监控指标	367
10.2.2	master 监控指标	372
10.2.3	region 服务器监控指标	373

10.2.4	RPC 监控指标	375
10.2.5	JVM 监控指标	376
10.2.6	info 监控指标	377
10.3	Ganglia	378
10.3.1	安装	379
10.3.2	用法	383
10.4	JMX	386
10.4.1	JConsole	388
10.4.2	JMX 远程 API	390
10.5	Nagios	394
第 11 章	性能优化	395
11.1	垃圾回收优化	395
11.2	本地 memstore 分配缓冲区	398
11.3	压缩	399
11.3.1	可用的编解码器	400
11.3.2	验证安装	401
11.3.3	启用压缩	403
11.4	优化拆分和合并	404
11.4.1	管理拆分	404
11.4.2	region 热点	405
11.4.3	预拆分 region	406
11.5	负载均衡	407
11.6	合并 region	408
11.7	客户端 API: 最佳实践	409
11.8	配置	411
11.9	负载测试	414
11.9.1	性能评价	414
11.9.2	YCSB	416
第 12 章	集群管理	421
12.1	运维任务	421
12.1.1	减少节点	421
12.1.2	滚动重启	423
12.1.3	新增服务器	424
12.2	数据任务	428
12.2.1	导入/导出	428
12.2.2	CopyTable 工具	433
12.2.3	批量导入	435

12.2.4	复制	438
12.3	额外的任务	440
12.3.1	集群共存	440
12.3.2	端口要求	442
12.4	改变日志级别	442
12.5	故障处理	443
12.5.1	HBase Fsck	443
12.5.2	日志分析	445
12.5.3	常见问题	447
附录 A	HBase 配置属性	451
附录 B	计划	467
附录 C	版本升级	469
附录 D	分支	471
附录 E	Hush SQL Schema	473
附录 F	对比 HBase 和 BigTable	475

在探究 HBase 的功能之前，我认为有必要先来思考一下为什么要设计出这样一套新的存储架构。传统的关系型数据库管理系统（Relational Database Management System, RDBMS）早在 20 世纪 70 年代已经出现，并且帮助无数的公司和机构实现了给定问题的解决方案，时至今日，RDBMS 仍旧非常有用。很多情况下关系模型都能够非常完美地阐述问题，但是在面对一些特殊的场景时关系模型并不是最佳的解决方案。^①

1.1 海量数据的黎明

我们生活在一个互联网时代，无论是想搜索最佳的火鸡菜谱，还是送妈妈什么样的生日礼物，都希望能够通过互联网迅速地检索到问题的答案，同时希望查询到的结果有用，而且非常切合我们的需要。

因此，很多公司开始致力于提供更有针对性的信息，例如推荐或在线广告，这种能力会直接影响公司在商业上的成败。现在类似 Hadoop[®] 这样的系统能够为公司提供存储和处理 PB 级数据的能力，随着新机器学习算法的不断发展，收集更多数据的需求也在与日俱增。

以前，因为缺乏划算的方式来存储所有信息，很多公司会忽略某些数据源，但是现在这样的处理方式会让公司丧失竞争力。存储和分析每一个数据点的需求在不断增长，这种需求的增长直接导致各公司电子商务平台产生了更多的数据。

过去，唯一的选择就是将收集到的数据删减后保存起来，例如只保留最近 N 天的数据。然而，这种方法只在短期内可行，它无法存储几个月或几年收集到的所有数据，因此

① 例如，参见 Michael Stonebraker 和 Uğur Çetintemel 撰写的文章“‘One Size Fits All’: An Idea Whose Time Has Come and Gone” (http://www.cs.brown.edu/~ugur/fits_all.pdf)。

② 相关信息可以在 Hadoop 的官方网站 <http://hadoop.apache.org/> 中找到。也可以到 Tom White 编写的《Hadoop 权威指南（第 2 版）》（原出版社为 O'Reilly）一书中查阅你想了解的 Hadoop 知识。

建议：构建一种数学模型覆盖整个时间段或者改进一个算法，重跑以前所有的数据，以达到更好的效果。

对于海量数据的重要性，Ralph Kimball 博士指出^①：

“数据资产会取代 20 世纪传统有形资产的地位，成为资产负债表的重要组成部分。”

还指出：

“数据的价值已经超越了传统企业广泛认同的价值边界。”

Google 和 Amazon 是认识到数据价值的典范，它们已经开始开发满足自己业务需求的解决方案。例如，Google 在一系列的技术出版物中描述了基于商业硬件的可扩展的存储和处理系统。开源社区利用 Google 的这些思想实现了开源 Hadoop 项目的两个模块：HDFS 和 MapReduce。

Hadoop 擅长存储任意的、半结构化的数据，甚至是非结构化的数据，可以帮助用户在分析数据的时候决定如何解释这些数据，同样允许用户随时更改数据分类的方式：一旦用户更新了算法，只需要重新分析数据。

目前 Hadoop 几乎是所有现有数据库系统的一种补充，它给用户提供了数据存储的无限空间，支持用户在恰当的时候存储和获取数据，并且针对大文件的存储、批量访问和流式访问做了优化。这使得用户对数据的分析变得简单快捷，但是用户同样需要访问分析后的最终数据，这种需求需要的不是批量模式而是随机访问模式，这种模式相对于在数据库系统来说，相当于一种全表扫描和使用索引。

通常用户在随机访问结构化数据的时候都会查询数据库。RDBMS 在这方面最为突出，但是也有一些少量的有差异的实现方式，比如面向对象的数据库。大多数 RDBMS 一直遵守科德十二定律（Codd's 12 rules）^②，这个定律对于 RDBMS 来说是刚性标准，并且由于 RDBMS 的底层架构是经过仔细研究的，所以在相当长的一段时间里这种架构都不会有明显的改变。近年来出现的各种处理方法，如列式存储的（column-oriented）数据库和大规模并行处理（Massively Parallel Processing, MPP）数据库，表明业界重新思考了技术方案以满足新的工作负载，但是大多数解决方案仍旧是基于科德十二定律来实现的，并没有打破传统的法则。

① 此处引用的是 Kimball 集团的 Ralph Kimball 博士的一篇题为“Rethinking EDW in the Era of Expansive Information Management”的演讲（http://www.informatica.com/campaigns/rethink_edw_kimball.pdf），这个演讲讨论了一个不断发展的企业数据仓库市场的需求。

② Edgar F. Codd 定义了 13 个规则（编号为 0~12），这些规则促使数据库管理系统（Database Management System, DBMS）被考虑为 RDBMS。HBase 需要满足更多的通用规则，但也有一些规则没有满足，最重要的是规则 5：全面的数据子语言规则，这个规则定义了至少需要支持一种关系型语言。详情见维基百科关于科德十二定律的链接 http://en.wikipedia.org/wiki/Codd's_12_rules。

列式存储数据库

列式存储数据库以列为单位聚合数据，然后将列值顺序地存入磁盘，这种存储方法不同于行式存储的传统数据库，行式存储数据库连续地存储整行。图 1-1 形象地展示了列式存储和行式存储的不同物理结构。

列式存储的出现主要基于这样一种假设：对于特定的查询，不是所有的值都是必需的。尤其是在分析型数据库里，这种情形很常见，因此需要选择一种更为合适的存储模式。

在这种新型的设计中，减少 I/O 只是众多主要因素之一，它还有其他的优点：因为列的数据类型天生是相似的，即便逻辑上每一行之间有轻微的不同，但仍旧比按行存储的结构聚集在一起的数据更利于压缩，因为大多数的压缩算法只关注有限的压缩窗口。

像增量压缩或前缀压缩这类的专业算法，是基于列存储的类型定制的，因而大幅度提高了压缩比。更好的压缩比有利于在返回结果时降低带宽的消耗。

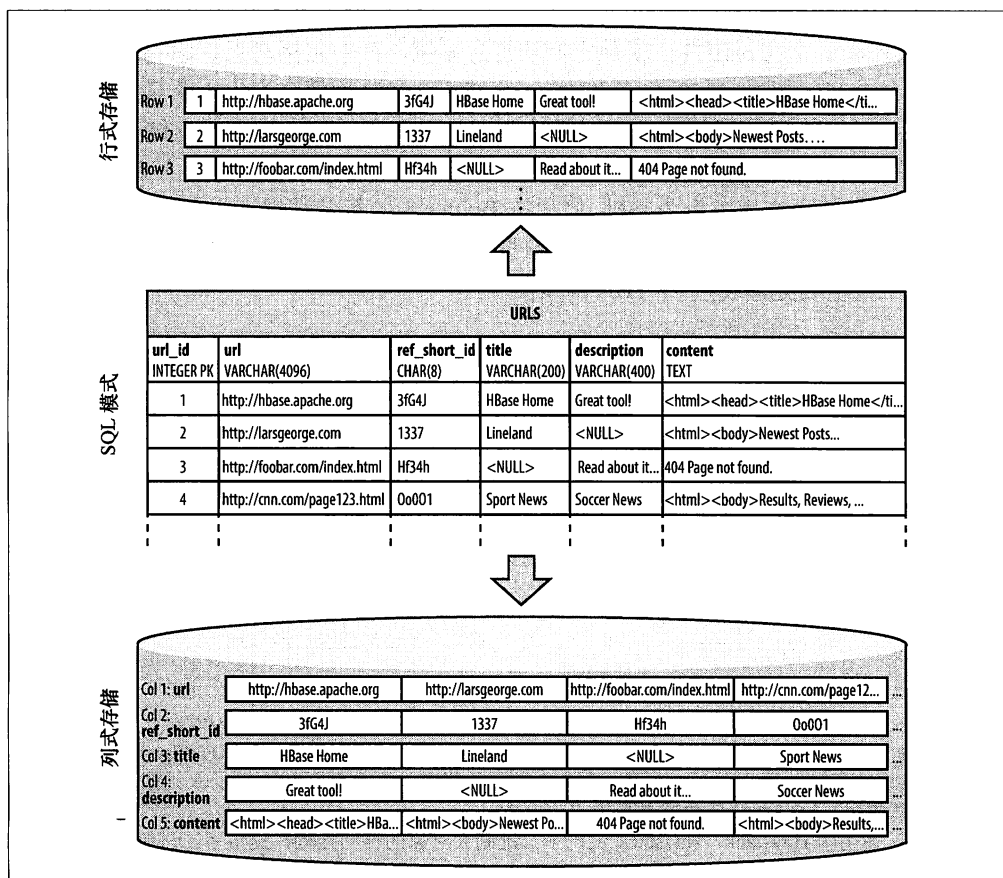


图 1-1 列式存储结构与行式存储结构

值得注意的是,从典型 RDBMS 的角度来看, HBase 并不是一个列式存储的数据库,但是它利用了磁盘上的列存储格式,这也是 RDBMS 与 HBase 最大的相似之处,因为 HBase 以列式存储的格式在磁盘上存储数据。但它与传统的列式数据库有很大的不同:传统的列式数据库比较适合实时存取数据的场景, HBase 比较适合键值对的数据存取,或者有序的数据存取。

如今数据的产生速度比几年以前已经有了迅猛的增长,随着全球化步伐加快,这个增长速度只会越来越迅猛,由此所产生的数据处理问题也会越来越严峻。像 Google、Amazon、eBay 和 Facebook 这样网站的用户已经覆盖到了地球上的绝大多数人。全球化网络应用 (planet-size web application) 的概念已经形成,在这种背景下,企业使用 HBase 更合适。

举例来说, Facebook 每天增量存储到它们 Hadoop 集群的数据量超过 15 TB^①,并且随后会对所有这些数据进行处理。这些数据一部分是点击流日志,用户点击了它们的网站或点击了使用 Facebook 提供的社交插件的网站,每一步点击操作都会被记录并保存,这非常适合以批处理的模式,为预测和推荐系统构建机器学习模型。

Facebook 还有一个实时组件,就是它们的消息系统,其中包括聊天、涂鸦墙和电子邮件,每个月会产生超过 1350 亿条数据^②,存储几个月之后便会产生一个量级庞大的尾部数据,并且这些尾部数据需要被有效地处理。尽管电子邮件中占用存储量较大的部分(如附件)通常存储在二级系统中^③,但这些消息产生的数据量还是令人难以置信的。以 Facebook 的数据产生条目为基础,假如按 Twitter 中的每条数据占用 140 字节来计算, Facebook 每个月将会产生超过 17 TB 的数据,在将这些数据导入到 HBase 之前,现存的系统每个月也要处理超过 25 TB 的数据。^④

目前在少数的重点行业中,面向 Web 业务的公司收集的数据量也在不断增长。

金融

如股票涨跌产生的数据。

生物信息学

如全球生物多样性信息机构 (Global Biodiversity Information Facility, <http://www.gbif.org/>)。

① 见 Facebook 提供的信息 http://www.facebook.com/note.php?note_id=89508453919。

② 请看博文 http://www.facebook.com/note.php?note_id=454991608919,这篇博文来自 Facebook 的工程团队。150 亿条墙消息和 1200 亿条聊天消息,共计 1350 亿条消息一个月。此外, Facebook 还添加了 SMS 和其他一些应用,这些都会使数据量变得更为庞大。

③ Facebook 使用了 Haystack, Haystack 优化了二进制大对象的存储结构,提供了二进制小对象存储,例如图片。

④ 见 <http://www.slideshare.net/brizzzdotcom/facebook-messages-hbase>,这是 Facebook 的员工 Nicolas Spiegelberg 写的,他也是 HBase 的 committer。

智能电网

如 OpenPDC (<http://openpdc.codeplex.com/>) 项目。

销售

如销售终端 (POS 机) 产生的数据, 或者是股票系统、库存系统。

基因组学

如 Crossbow (<http://bowtie-bio.sourceforge.net/crossbow/index.shtml>) 项目。

移动电话服务、军事、环境工程

也产生了海量的数据。

有效存储 PB 级的数据并能高效地检索和更新并不是一件容易的事情。下面深入探讨一下我们将面临的挑战。

1.2 关系数据库系统的问题

RDBMS 在设计和实现商业应用方面扮演了一个不可或缺的角色 (至少在可预见的未来仍旧如此)。只要用户需要保留用户、产品、会话、订单等信息, 就会采用一些存储后端为前端应用服务器提供持久化数据的服务。这种结构非常适合有限的数据量, 但对于数据急剧增长的情况, 这种结构就显得力不从心了。

以我们之前提到的 HBase 短网址 (HBase URL Shortener) 服务——Hush 为例。假设要构建一个初始就能支持几千个用户的系统, 并且要节省成本, 换句话说就是使用免费软件。这种经典案例就可以使用开源的 LAMP^① 快速地搭建一个原型。

关系数据库模型通过用外键关联 url 表、shorturl 表和 click 表, 规范了存入 user 表的数据。其中这些数据表都有索引, 可以保证你既能通过 short ID 检索到 URL, 又能通过 username 检索到用户。如果需要找出一个特定用户列表的所有短址 URL, 可以运行一个 SQL 的 JOIN 语句关联这两张表, 得到一个全面的 URL 表, 其中不仅包括短址 URL, 还包括所需的用户明细。

此外, 还可以利用数据库内置的功能, 如存储过程 (stored procedure)。当数据系统需要始终保证多张表的数据一致性时, 可以利用存储过程来解决多个客户端同时更新数据的一致性问题。

事务 (transaction) 提供了原子性跨表更新数据的特性, 可以让修改同时可见或同时不可见。RDBMS 提供了所谓的 ACID^② 特性, 这意味着用户数据是强一致性的 (在稍后

① Linux、Apache、MySQL 和 PHP (或者 Perl 和 Python) 的缩写。

② Atomicity、Consistency、Isolation 和 Durability 的缩写。

的“一致性模型”中有详细介绍)。参照完整性(referential integrity)负责约束不同的表结构之间的关系,利用特定域语言,即 SQL,能够写出任意复杂的查询语句。最终,用户不需要关心实际上数据是怎样存储的,只需要关心更高层次的概念,例如,表结构,表结构在应用程序中提供了非常固定的访问模型。

通常这种模式的设计能在较长的一段时间里满足需求。如果足够幸运,你可能会成为下一个互联网上的热点,每天有越来越多的用户加入你的网站。但随着你网站用户数量的增加,共享数据库服务器的压力也会越来越大。增加应用服务器的数量相对来说比较容易,因为应用服务器之间是共享中央数据库的,但随着共享中央数据库的 CPU 和 I/O 负载的上升,你将很难预测你能承受这种增长速度多久。

减少压力的第一步是增加用于并行读取的从服务器,将读写分离。这种方案保留了一个主数据库服务器,但是这个主数据库服务器现在只服务于写请求,这样做主要是考虑到网站的请求主要由用户浏览产生,因此写请求远少于读请求。如果这个方案也因用户数的持续增加而失败了,或者降低了系统的性能,又该怎么办呢?

下一步常见的做法是增加缓存,如 Memcached^①。现在可以将读操作接入到高速的在内存中缓存数据的系统中,但是这种方案无法保证数据一致性,因为用户更新数据到数据库,而数据库并不会主动更新缓存系统中的数据,所以需要尽可能快地同步缓存和数据库视图,把更新缓存数据与更新数据库数据的时间差最小化。

虽然这种方案能够缓解读请求的压力,但是写请求压力的增加问题还是没有得到解决。一旦主数据库服务器写性能下降,可以把主服务器换成加强服务器,即垂直扩容,让加强服务器使用更多的内核、更多的内存、更快的磁盘……总之,与初始的主服务器相比要花费更多的金钱。同时值得注意的是,用户如果已经选择了主从的配置方案,就必须要让从服务器的性能与主服务器同样强,否则从服务器的更新速度就会跟不上主服务器的更新速度,这样增加一倍到两倍的成本,甚至更多。

伴随着网站受欢迎程度增加,网站会需要增加更多新功能,而这些新功能无疑都会转化为后台数据库的查询语句。以前顺利执行的 SQL JOIN 语句执行突然变慢了,或是干脆无法执行,这时候就不得不采用逆范式化存储结构。如果情况越来越糟,就不得不停止使用存储过程,因为存储过程最后会慢得无法执行。本质上讲,减少数据库中的存储数据才能优化访问。

所以随着用户越来越多,负载会不断提高,合乎逻辑的方式就是不时地预先实现最昂贵的查询方案,从而给用户提供更快的数据服务。最终,不得不放弃辅助索引的使用,原因就是数据量增大的同时,索引量也大到了足以让数据库的性能直线下降。最后所

① Memcached 是基于内存的、非持久化的、非分布式的键值存储系统。参见 Memcached 项目的主页 <http://memcached.org/>。

能提供的查询模式只剩下了按照主键查询。

这时该怎么办？如果负载在未来的几个月里预期会增加一个数量级或更多又该怎么办？此时用户可以考虑将数据分区（sharding）到多个数据库中，但是采用此方案会使运维操作变成噩梦，而且代价非常昂贵，因此也不是最合理的解决方案。但从本质来讲，采用 RDBMS 也是因为没有其他可以选择的方案。

分区

分区（sharding）主要描述了逻辑上水平划分数据的方案。这个方案的特点是将数据分文件或分服务器存储，而不是连续存储。

数据的分区是在固定范围内实施的：在传入数据之前，必须提前划分好数据的存储范围，如果一个水平划分的压力超过其所能提供的容量，就需要将数据重分区（reshard）并迁移数据。

重分区并迁移数据是非常消耗资源的操作，等同于数据重做。需要重新划分边界然后横向拆分（split）。大规模的复制操作会消耗大量的 I/O 资源，同时还会临时性地增加存储需求。在对数据重分区的过程中，客户端应用仍然会有更新操作要执行，不过此时的更新操作受重分区的影响会执行得非常慢。

可以采用虚拟分区（virtual shard）的方式来减少这种资源消耗，虚拟分区按照关键词定义范围较大的数据分区，每个服务器加载同等数量的数据分区。但是在新增服务器的时候，需要重新加载数据分区到新服务器，并且这个过程仍旧需要将数据迁移到新服务器。

分区是简单的完全脱离用户操作的事后操作，如果没有数据库的支持，可能会对生产系统造成严重破坏。

对于关系数据库系统的问题就讲到这里，客观地讲，RDBMS 已经在大量的公司里使用，并形成了较大规模的技术体系。例如，Facebook、Google 都已经大规模地使用了 MySQL，因为 MySQL 的安装和使用都已经非常简洁，相关的参考资料和范例也非常充分。这个数据库适合特定场景的业务，并且短期内不会被取代。这里有个问题是：如果你想开发一个新产品，并且在设计阶段就已经预料到系统拓展速度非常快，那么，你是希望所有的功能都可用，还是使用某些有一定制约的功能？

1.3 非关系型数据库系统 Not-Only-SQL（简称 NoSQL）

过去的四五年时间里，为了解决问题，创新的前进步伐由缓慢变得出奇得快，好像每

周都会发布新的框架和项目来满足需求。我们看到了所谓的 NoSQL 解决方案问世了，NoSQL 是 Eric Evans 针对 Johan Oskarsson 提出的“为新兴的新数据存储空间^①命名”问题而创造的一个名词。

正是因为这类新产品还没有合适的名称，NoSQL 一举成名。在激烈的讨论中，它被认为是“SQL”的克星，或者说，它给仍旧考虑使用传统 RDBMS 的人带来了瘟疫……只是开个玩笑！



实际上，为特定的问题制定差异化的专用解决方案的想法并不新鲜。像 Berkeley DB、Coherence、GT.M 这样的系统，以及面向对象的数据库系统都已经出现了好多年，有些甚至都可以追溯到 20 世纪 80 年代初，从定义上来看，它们都属于 NoSQL。

标示符号化（tagword）实际上是一个不错的选择：最新的存储系统不提供通过 SQL 查询数据的手段，只提供一些比较简单的、类似于 API 接口的方式来存取数据。

但是，也有一些工具为 NoSQL 数据存储提供了 SQL 语言的入口，用于执行一些关系数据库中常用的复杂条件查询。因此，从查询方式上的限制来说，关系型数据库和非关系型数据库并没有严格的区分。

实际上两者在底层上是有区别的，尤其涉及到模式或者 ACID 事务特性时，因为这与实际的存储架构是相关的。很多这一类的新系统首先做的事情是：抛弃一些限制因素以提升扩展性（这一点会在 1.3.1 节讨论）。例如，它们通常不支持事务或辅助索引。更重要的是，这一类系统是没有固定模式的，可以随着应用的改变而灵活变化。

一致性模型

在这本书里，我们经常会提到一致性问题，所以有必要在这里对它稍加介绍。一开始的一致性是指数据库客户端操作的正确性，数据库必须保证每一步操作都是从一个一致的状态到下一个一致的状态。系统没有明确地指定如何实现这个功能，以便系统可以有多种选择。最终，系统要选择是进入下一个一致的状态，还是回退到上一个一致的状态，从而保证一致性。

一致性可以按照严格程度由强到弱分类，或者是按照对客户端的保证程度分类，下面是一个非正式的分类列表。

严格一致性：数据的变化是原子的，一经改变即时生效，这是一致性的最高形式。

顺序一致性：每个客户端看到的数据依照它们操作执行的顺序而变化。

因果一致性：客户端以因果关系顺序观察到数据的改变。

^① 见维基百科中的“NoSQL”（<http://en.wikipedia.org/wiki/NoSQL>）。

最终一致性：在没有更新数据的一段时间里，系统将通过广播保证副本之间的数据一致性。

弱一致性：没有做出保证的情况下，所有的更新会通过广播的形式传递，展现给不同客户端的数据顺序可能不一样。

采用最终一致性策略的系统还可以细分为几个子类，并且这些子策略还可以共存。亚马逊的首席技术官 Werner Vogels 在一篇名为“Eventually Consistent”的文章中列举了这几个子类。这篇文章还谈到了 CAP 定理（CAP theorem）^①，其中指出，一个分布式系统只能同时实现一致性、可用性和分区容忍性（或分区容错性）中的两个。CAP 定理是热点话题，不过它不是区分分布式系统的唯一方法，但 CAP 定理指出了，开发一套同时满足以上需求的分布式系统是比较困难的。例如，Vogels 提到：

“在一系列的研究结果里发现，在较大型的分布式系统中，由于网络分隔，一致性与可用性不能同时满足。这意味着三个要素最多只能同时实现两个，不可能三者兼顾；放宽一致性的要求会提升系统的可用性……提升一致性意味着系统需要牺牲一定的可用性。”

放宽一致性来提高系统可用性是一个非常有效的提议。不过这种方案会强制让应用层去解决一致性的问题，因此也会增加系统的复杂度。

各种非关系型数据库有许多共同的特性，同时这其中的许多特性与传统的存储方案也有很多共同点。因此新系统并不是革命性的产品，从工程的角度来看更像是产品的进化。

假使连 Memcached 这样的项目都划入到 NoSQL 范畴的话，那就成了只要不是 RDBMS 就可以认为是 NoSQL。这个说法导致了错误的二分法，二分法掩盖了这些系统提供的令人振奋的技术可行性。在 NoSQL 范畴内，还有很多的维度可以区分系统的特定优势所在。

1.3.1 维度

让我们来挑几种维度简单介绍一下。需要注意的是，列举的这些维度并不全面，并且这也不是唯一的区分方式。

数据模型

数据有多种存储的方式，包括键/值对（类似于 HashMap）、半结构化的列式存储和文档结构存储。用户的应用如何存取数据？同时数据模式是否随着时间而变化？

^① 见 Eric Brewer 的论文 <http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf>，后期 Gilbert 与 Lynch 发表了 PDF 版，详情见 <http://lpd.epfl.ch/sgilbert/pubs/BrewersConjecture-SigAct.pdf>。

存储模型

内存还是持久化？坦率来说做出这个决定并不难，其主要原因是，我们可以将其与 RDBMS 进行对比，它们通常持久化存储数据到磁盘中。即使需要的是纯粹的内存模式，也仍旧有其他方案。一旦考虑持久化存储，就需要考虑选择的方案是否会影响到访问模式？

一致性模型

严格一致性还是最终一致性？问题是存储系统如何实现它的目标：必须降低一致性要求吗？虽然这种问题很粗浅，但是在特定的场景中会产生巨大影响。因为一致性可能会影响操作延时，即系统响应读写请求的速度。这需要权衡投入和产出后得到一个折中结果。^①

物理模型

分布式模式还是单机模式？这种架构看起来像什么？是仅仅运行在单个机器上，还是分布在多台机器上，但分布及扩展规则由客户端管理，换句话说，由用户自己的代码管理？也许分布式模式仅仅是个事后的工作，并且只会在用户需要扩展系统时产生问题。如果系统提供了一定的扩展性，那么需要用户采取特定的操作吗？最简单的解决方案就是一次增加一台机器，并且设置好分区（这点对于不支持虚拟分区的系统非常重要），设置时需要考虑同时提高每个分区的处理能力，因为系统的每个部分都需要提供均衡的性能。

读/写性能

用户必须了解自己的应用程序的访问模式。是读多写少？还是读写相当？或者是写多读少？是用范围扫描数据好，还是用随机读请求数据更好？有些系统仅仅对这些情况中的一种支持得非常好，有些系统则对各种情况都提供了很好的支持。

辅助索引

辅助索引支持用户按不同的字段和排序方式来访问表。这个维度覆盖了某些完全没有辅助索引支持且不保证数据排序的系统（类似于 HashMap，即用户需要知道数据对应键的值），到某些可能通过外部手段简单支持这些功能的系统。如果存储系统不提供这项功能，用户的应用可以应对或自己模拟辅助索引吗？

故障处理

机器会崩溃是一个客观存在的问题，需要有一套数据迁移方案来应对这种情况（关于这一点可以参考在“一致性模型”中讨论的 CAP 定理）。每个数据存储如

① 见 Brewer 的论文“Lessons from giant-scale services. Internet Computing”，IEEE，2001，5 (4)：46~55 (http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=939450)。

何进行服务器故障处理？故障处理完毕之后是否可以正常工作？这与之前讨论的“一致性模型”维度有关系，因为失去一台服务器可能会造成数据存储的空洞(hole)，甚至使整个数据存储不可用。如果替换掉故障服务器，那么恢复 100% 服务的难度有多大？从一个正在提供服务的集群中卸载一台服务器时，也会遇到类似的问题。

压缩

当用户需要存储 TB 级的数据时，尤其当这些数据差异性很小或由可读性文本组成时，压缩会带来非常好的效果，即能节省大量的原始数据存储。有些压缩算法可以将此类的数据压缩到原始文件大小的十分之一。有可选择的压缩组件吗？又有哪些压缩算法可用？

负载均衡

假如用户有高读写吞吐率的需求，就要考虑配置一套能够随着负载变化自动均衡处理能力的系统。虽然这样不能完全解决该问题，但是也可以帮助用户设计高读写吞吐量的程序。

原子操作的读-修改-写

RDBMS 提供了很多这类的操作（因为它是一个集中式的面向单服务器的系统），但这些操作在分布式系统中较难实现，这些操作可以帮助用户避免多线程造成的资源竞争，也可以帮助用户完成无共享应用服务器的设计。有了这些比较并交换(compare and swap, CAS)操作，或者说检查并设置(check and set)操作，在设计系统的时候可以有效地降低客户端的复杂度。

加锁、等待和死锁

众所周知，复杂的事务处理，如两阶段提交，会增加多个客户端竞争同一个资源的可能性。最糟糕的情况就是死锁，这种情况也很难解决。用户需要支持的系统采用哪种锁模型？这种锁模型能否避免等待和死锁？



稍后我们会回顾这些维度，看看 HBase 适合用在哪里，其优势何在。现在需要指出的是，一定要根据实际的需求来仔细选择最适合的维度。按照实际情况来设计解决方案，要知道没有硬性规定说：RDBMS 不能很好地解决的问题，NoSQL 就能完美解决。重要的是正确地评估需求，然后再做出明智的选择，有需要的话甚至可以采用混合使用的方案。

可以用一个有趣的词来形容这个情况——阻抗匹配(impedance match)，意思就是要为一个给定问题找到一个理想的解决方案，除了使用通用的解决方案，还应该知道有什么可用的解决方案，从而找到最适合于解决该问题的系统。

1.3.2 可扩展性

RDBMS 非常适合事务性操作，但不见长于超大规模的数据分析处理，因为超大规模的查询需要进行大范围的数据记录扫描或全表扫描。分析型数据库可以存储数百或数千 TB 的数据，在一台服务器上做查询工作的响应时间，会远远超过用户可接受的合理响应时间。垂直扩展服务器性能，即增加 CPU 核数和磁盘数目，也并不能很好地解决该问题。

更糟糕的是，RDBMS 的等待和死锁的出现频率，与事务和并发的增加并不是线性关系，准确地说，与并发数目的平方以及事务规模的 3 次方甚至 5 次方相关^①。分区通常是一个不切合实际的解决方案，因为它需要客户端采用非常复杂的方式和较高的代价来维护分区信息。

一些商业 RDBMS 也解决过类似的问题，但它们往往只是特定地解决了问题的某几个方面，更重要的是，它们非常非常的昂贵。而一些开源的 RDBMS 解决方案中，往往放弃了其中的一些甚至全部的关系型特性，如辅助索引，来换取更高的性能拓展能力。

问题是，为了性能而一直放弃以上关系型特性是否值得？用户可以反范式化（见 1.3.3 节）数据模型来避免等待，并且可以通过降低锁粒度的方式来尽量避免死锁。数据增长时，无需重新分区迁移数据并内嵌水平扩展性的方法。最后，用户还要面对容错和数据可用性问题，采用提高扩展性的机制，用户最终会得到一个 NoSQL 的解决方案，更确切地说，HBase 可以满足以上多种需求。

1.3.3 数据库的范式化和反范式化

不同的规模，经常需要设计不同的系统结构，对这种原则的最佳描述是：反范式化、复制和智能的主键（Denormalization, Duplication, and Intelligent Keys，简称 DDI^②）。这就需要重新思考在类似 BigTable 的存储系统中如何才能高效合理地存储数据。

部分原则是采用反范式化模式，例如将数据复制到多张表中，这样在读取的时候就不需从多张表中聚合数据了。或者预先物化所需的视图，一次优化从而避免进一步的处理来提高读取性能。

这一主题在第 9 章里有更详细的介绍，主要阐述了如何充分利用 HBase 的特性去解决实际问题。让我们来看一个例子，理解传统的关系数据库模型转到列式存储的 HBase 的几点基本原则。

再来看看 HBase 短网址，即 Hush，Hush 允许用户将长网址映射为短网址（short URL），见图 1-2 表示的实体关系图（entity relationship diagram，ERD，简称 ER 图）。在附录 E^③中可以查看完整的 SQL 模式。

① 见 Jim Gray 等人的“FT 101”（http://research.microsoft.com/en-us/um/people/gray/talks/UCBerkeley_Gray_FT_Availability_talk.ppt）。

② DDI 这个词是 Salmen 博士等人于 2009 年在“Cloud Data Structure Diagramming Techniques and Design Patterns”一文中提出的。

③ 请注意，这仅仅是一个演示用例，所以故意将模式设计得很简单。

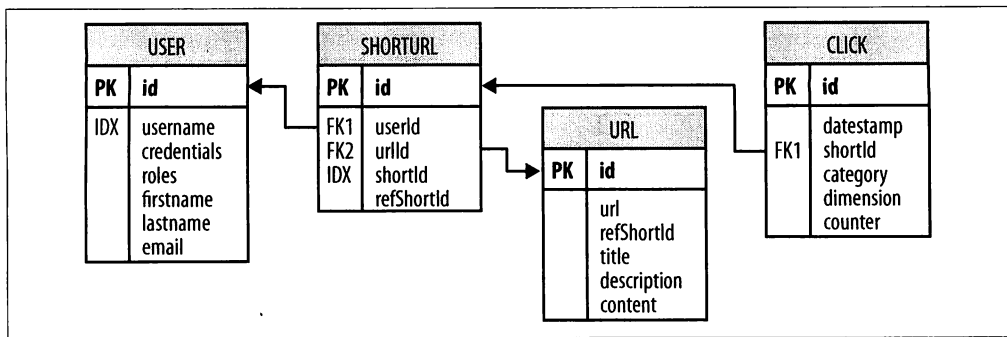


图 1-2 Hush 模式的实体关系图

短网址存储在 `shorturl` 表中，用户可以点击短网址来链接到完整网址。系统会跟踪每次点击，记录该网址的使用次数，还会记录一些其他信息，例如，点击该链接的用户所在的国家。这些信息会记录在 `click` 表中，这个表的功能类似于一个计数器，以天为周期统计每天的访问量。

用户信息存储在 `user` 表中，用户可以在 Hush 网站上注册并创建个人短网址列表，同时也可以在此网站上增加描述。`user` 表与 `shorturl` 表之间维护了一个外键关系。

系统还会在后台下载链接到的页面，并提取一些 TITLE 之类的 HTML 标签。将整个页面保存下来的目的是，供后续的异步任务进行处理和分析。这些内容都由 `url` 表存储。每个链接页面只存储一份，不过，由于许多用户可能会链接同一个长网址，并且还想保存他们自己的详细信息，例如，使用统计信息，因此会在短链接表中创建多个项来加以区分。通过这段逻辑，将 `url` 表、`shorturl` 表和 `click` 表关联在了一起。

这使得通过统计原始的短网址标识 `refShortId`，就可以统计任意一个短网址映射到同一个长网址的使用率。`shortId` 和 `refShortId` 利用散列 ID 的方式被唯一地分配给了短网址。例如：

```
http://hush.li/a23eg
```

在上述地址中，散列 ID 是 `a23eg`。

图 1-3 展现了 Hush 应用在 HBase 中的对应模式。每个短网址都存储在独立的表 `shorturl` 中，表中还包含了使用统计信息，按统计时间范围不同，存放于不同的列族中，同时每个值都有其生存期（time-to-live）。列名是日期和一个可选维度后缀的组合，例如国家代码，列值则是对应计数器的值。

下载下来的页面和提取的详细信息存储在 `url` 表中，并且要通过压缩最大限度地减少存储量，因为存储的页面主要是 HTML，这种格式本身冗余量大，并且包含了大量文本。

Table: shorturl		
Row Key:	shortId	
Family:	data:	Columns: url, refShortId, userId, clicks
	stats-daily: [ttl: 7days]	Columns: YYYYMMDD, YYYYMMDD\x00<country-code>
	stats-weekly: [ttl: 4weeks]	Columns: YYYYWW, YYYYWW\x00<country-code>
	stats-monthly: [ttl: 12months]	Columns: YYYYMM, YYYYMM\x00<country-code>

Table: url		
Row Key:	MD5(url)	
Family:	data: [compressed]	Columns: refShortId, title, description
	content: [compressed]	Columns: raw

Table: user-shorturl		
Row Key:	username\x00shortId	
Family:	data:	Columns: timestamp

Table: user		
Row Key:	username	
Family:	data:	Columns: credentials, roles, firstname, lastname, email

图 1-3 HBase 中的 Hush 模式

系统通过 user-shorturl 表可以快速查到指定用户的所有短网址标识。这个功能被用在用户主页中，只要用户一登录就会被记录下来。user 表存储着实际用户的详细信息。

虽然表的数量相同，都是 4 个，但表的含义发生了变化：clicks 表被合并到了 shorturl 表中，统计列使用日期为列键，格式为 YYYYMMDD，例如，20110502，这样用户可以顺序访问数据。新增的 user-shorturl 表代替了外键，使查询用户相关信息变得更为快捷。

有非常多的方法来转换一对一、一对多、多对多的关系，以适应 HBase 的底层架构。这种简单的例子有多种实现方式，用户需要充分理解 HBase 存储设计的潜在能力，然后深思熟虑地决定用哪一种实现方式。

对稀疏矩阵、宽表、列式存储的支持使得数据在存储的时候无需范式化，同时也可以避免查询时采用开销很大的 JOIN 操作聚合数据。使用智能的主键可以控制数据怎样去存储以及存储在什么位置。由于可以使用行键的部分内容进行范围检索，行键作为组合键设计时，与字典序左部分为头的索引效果相似。因此，正确的设计能够使性能不会因为数据增长而下降，例如当数据条目从 10 条增加到 1000 万条时，系统仍旧可以保持相同的读写性能。

1.4 结构

本节首先介绍 HBase 的架构，然后介绍一些关于 HBase 起源的背景资料，之后将介绍其数据模型的一般概念和可用的存储 API，最后在一个更高的层次上对其实现细节进行分析。

1.4.1 背景

2003 年，Google 发表了一篇论文，叫“The Google File System”(<http://labs.google.com/papers/gfs.html>)。这个分布式文件系统简称 GFS，它使用商用硬件集群存储海量数据。文件系统将数据在节点之间冗余复制，这样的话，即使一台存储服务器发生故障，也不会影响数据的可用性。它对数据的流式读取也做了优化，可以边处理边读取。

不久，Google 又发表了另外一篇论文，叫“MapReduce: Simplified Data Processing on Large Clusters”，参见 <http://labs.google.com/paper/mapreduce.html>。MapReduce 是 GFS 架构的一个补充，因为它能够充分利用 GFS 集群中的每个商用服务器提供的大量 CPU。MapReduce 加上 GFS 形成了处理海量数据的核心力量，包括构建 Google 的搜索索引。

不过以上描述的两个系统都缺乏实时随机存取数据的能力（意味着尚不足以处理 Web 服务）。GFS 的另一个缺陷就是，它适合存储少许非常非常大的文件，而不适合存储成千上万的小文件，因为文件的元数据信息最终要存储在主节点的内存中，文件越多主节点的压力越大。

因此，Google 尝试去找到一个能够驱动交互应用的解决方案，例如，Google 邮件或者 Google 分析，能够同时利用这种基础结构、依靠 GFS 存储的数据冗余和数据可用性较强的特点。存储的数据应该拆分成特别小的条目，然后由系统将这些小记录聚合到非常大的存储文件中，并提供一些索引排序，让用户可以查找最少的磁盘就能够获取数据。最终，它应该能够及时存储爬虫的结果，并跟 MapReduce 协作构建搜索索引。

意识到 RDBMS 在大规模处理中的缺点（8.1 节会针对这一点进行深入讨论），工程师们开始考虑问题的其他切入点：摒弃关系型的特点，采用简单的 API 来进行增、查、改、删（Create, Read, Update, and Delete，简称 CRUD）操作，再加一个扫描函数，在

较大的键范围或全表范围上迭代扫描。这些努力的成果最终在 2006 年的论文“BigTable: A Distributed Storage System for Structured Data”中发表了。

“BigTable 是一个管理结构化数据的分布式存储系统，它可以扩展到非常大：如在成千上万的商用服务器上存储 PB 级的数据。……一个稀疏的、分布式的、持久的多维排序映射。”

强烈建议对 HBase 感兴趣的人去阅读这篇论文，它介绍了很多 BigTable 的设计原理，用户最终都能在 HBase 中找到 BigTable 的影子。我们会借用这篇论文的基本概念来贯穿我们的这本书。

HBase 实现了 BigTable 存储架构，因此我们也可以用 HBase 来解释每样东西。附录 F 介绍了两者之间的不同。

1.4.2 表、行、列和单元格

首先，做一个简要总结：最基本的单位是列（column）。一列或多列形成一行（row），并由唯一的行键（row key）来确定存储。反过来，一个表（table）中有若干行，其中每列可能有多个版本，在每一个单元格（cell）中存储了不同的值。

除了每个单元格可以保留若干个版本的数据这一点，整个结构看起来像典型的数据库的描述，但很明显有比这更重要的因素。

所有的行按照行键字典序进行排序存储。例 1.1 展现了如何通过不同的行键增加多行数据。

例 1.1 行序是按照行键的字典序进行排序的

```
hbase(main):001:0> scan 'table1'
ROW                                COLUMN+CELL
row-1                             column=cf1:,timestamp=1297073325971 ...
row-10                             column=cf1:,timestamp=1297073337383 ...
row-11                             column=cf1:,timestamp=1297073340493 ...
row-2                              column=cf1:,timestamp=1297073329851 ...
row-22                             column=cf1:,timestamp=1297073344482 ...
row-3                              column=cf1:,timestamp=1297073333504 ...
row-abc                            column=cf1:,timestamp=1297073349875 ...
7 row(s) in 0.1100 seconds
```

注意，排列的顺序可能和你预期的不一样，可能需要通过补键来获得正确排序。在字典序中，是按照二进制逐字节从左到右依次对比每一个行键，例如，row-1...小于 row-2...，因此，无论后面是什么，将始终按照这个顺序排列。

按照行键排序可以获得像 RDBMS 的主键索引一样的特性，也就是说，行键总是唯一的，并且只出现一次，否则你就是在更新同一行。虽然 BigTable 的论文里只考虑了行键单一索引，但是 HBase 增加了对辅助索引（见 9.3 节）的支持。行键可以是任意的字节数组，但它并不一定是人直接可读的。

一行由若干列组成，若干列又构成一个列族（column family），这不仅有助于构建数据的语义边界或者局部边界，还有助于给它们设置某些特性（如压缩），或者指示它们存储在内存中。一个列族的所有列存储在同一个底层的存储文件里，这个存储文件叫做 *HFile*。

列族需要在表创建时就定义好，并且不能修改得太频繁，数量也不能太多。在当前的实现中有少量已知的缺陷，这些缺陷使得列族数量只限于几十，实际情况可能还小得多（详情见第 9 章）。列族名必须由可打印字符组成，这与其他名字或值的命名规范有显著不同。

常见的引用列的格式为 *family:qualifier*，*qualifier* 是任意的字节数组。^①与列族的数量有限制相反，列的数量没有限制：一个列族里可以有数百万个列。列值也没有类型和长度的限定。

图 1-4 用可视化的方式展现了普通数据库与列式 HBase 在行设计上的不同，行和列没有像经典的电子表格模型那样排列，而是采用了标签描述（tag metaphor），也就是说，信息保存在一个特定的标签下。

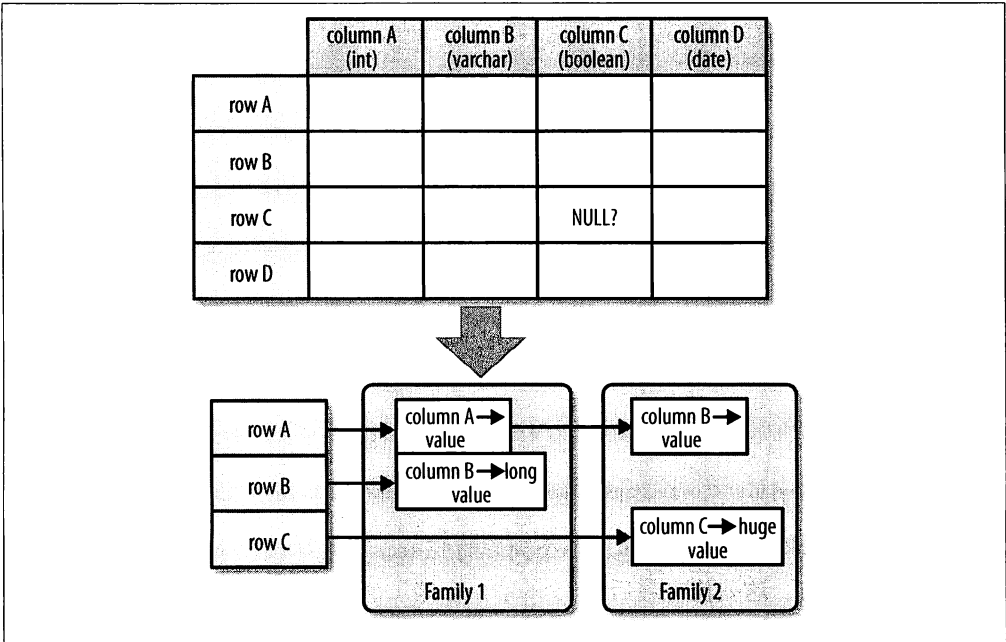


图 1-4 HBase 中的行与列

^① 在 5.1.3 节中会看到，还可以不设置 *qualifier*。



图 1-4 中的“NULL?”表明了固定模式的数据库在没有值的地方必须存储 NULL 值，但是在 HBase 的存储架构中，可以干脆省略整个列，换句话说，空值是没有任何消耗的：它们不占用任何存储空间。

所有列和行的信息都会通过列族在表中定义，关于这一点我们在下文进行讨论。

每一列的值或单元格的值都具有时间戳，默认由系统指定，也可以由用户显式设置。时间戳可以被使用，例如通过不同的时间戳来区分不同版本的值。一个单元格的不同版本的值按照降序排列在一起，访问的时候优先读取最新的值。这种优化的目的在于让新值比老值更容易被读取。

用户可以指定每个值所能保存的最大版本数。此外，还支持谓词删除 (predicate deletion, 见 8.1.2 节关于 LSM 树的内容)，例如，允许用户只保存过去一周内写入的值。这些值 (或单元格) 也只是未解释的字节数组，客户端需要知道怎样去处理这些值。

前面提到过，HBase 是按照 BigTable 模型实现的，是一个稀疏的、分布式的、持久化的、多维的映射，由行键、列键和时间戳索引。将以上特点联系在一起，我们就有了如下的数据存取模式：

$(Table, RowKey, Family, Column, Timestamp) \rightarrow Value$

可以用一种更像编程语言的风格表示如下：

```
SortedMap<
  RowKey, List<
    SortedMap<
      Column, List<
        Value, Timestamp
      >
    >
  >
>
```

或者用一行来表示：

```
SortedMap<RowKey, List<SortedMap<Column, List<Value, Timestamp>>>>>
```

第一个 SortedMap 代表那个表，包含一个列族的 List。列族中包含了另一个 SortedMap 存储列和相应的值。这些值在最后的 List 中，存储了值和该值被设置的时间戳。

这个数据存取模型的一个有趣的特性是单元格可以存在多个版本，不同的列被写入的次数不同。API 默认提供了一个所有列的统一视图，API 会自动选择单元格的当前值。图 1-5 展示了示例表中的某一行。

图 1-5 用表示单元格被写入的时间戳 t_n 可视化了时间组件，升序显示了这些值被插入的不同时间。图 1-6 是另一种查看数据的方式，在这种更类似电子表格的形式中，将时间

戳添加到了它自己的那一列中。

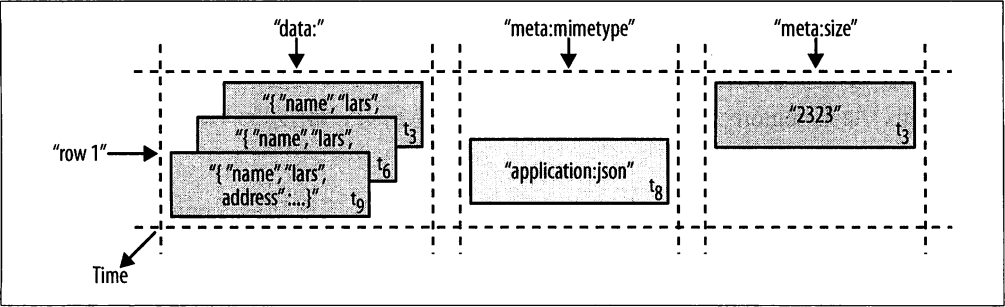


图 1-5 基于时间的行的组成部分

Row Key	Time Stamp	Column "data:"	Column "meta:"		Column "counters:" "updates"
			"mimetype"	"size"	
"row1"	t ₃	"{"name": "lars", "address": ...}"		"2323"	"1"
	t ₆	"{"name": "lars", "address": ...}"			"2"
	t ₈		"application/json"		
	t ₉	"{"name": "lars", "address": ...}"			"3"

图 1-6 用电子表格展示行中相同的部分

尽管这些值插入的次数不同，并且存在多个版本，但是仍然能将行看作是所有列以及这些列的最新版本（即每一列的最大 t_n ）的组合。这里提供了查询一个特定时间戳或者是特定时间戳之前的值的方式，也可以一次查询多个版本的值，关于这一点请参见第 3 章。

webtable

BigTable 和 HBase 的典型使用场景是 webtable，存储从互联网中抓取的网页。

行键是反转的网页 URL，如 org.hbase.www。有一个用于存储网页 HTML 代码的列族 contents，还有其他的列族，如 anchor，用于存储外向链接和入站链接，还有一个用于存储元数据的列族 language。


contents 列族使用多版本，允许用户存储一些旧的 HTML 副本，使用多版本是有益的，例如帮助分析一个页面的变化频率。使用的时间戳是抓取该页面的实际次数。

行数据的存取操作是原子的（atomic），可以读写任意数目的列。目前还不支持跨行事务和跨表事务。原子存取也是促成系统架构具有强一致性（strictly consistent）的一个因素，因为并发的读写者可以对行的状态作出安全的假设。

使用多版本和时间戳同样能够帮助应用层解决一致性问题。

1.4.3 自动分区

HBase 中扩展和负载均衡的基本单元称为 **region**, **region** 本质上是以行键排序的连续存储的区间。如果 **region** 太大, 系统就会把它们动态拆分, 相反地, 就把多个 **region** 合并, 以减少存储文件数量。^①

 HBase 中的 **region** 等同于数据库分区中用的范围划分 (range partition)。它们可以被分配到若干台物理服务器上以均摊负载, 因此提供了较强的扩展性。

一张表初始的时候只有一个 **region**, 用户开始向表中插入数据时, 系统会检查这个 **region** 的大小, 确保其不超过配置的最大值。如果超过了限制, 系统会在中间键 (middle key, **region** 中间的那个行键) 处将这个 **region** 拆分成两个大致相等的子 **region** (详情见第 8 章)。

每一个 **region** 只能由一台 **region** 服务器 (**region server**) 加载, 每一台 **region** 服务器可以同时加载多个 **region**。图 1-7 展示了一个表, 这个表实际上是一个由很多 **region** 服务器加载的 **region** 集合的逻辑视图。

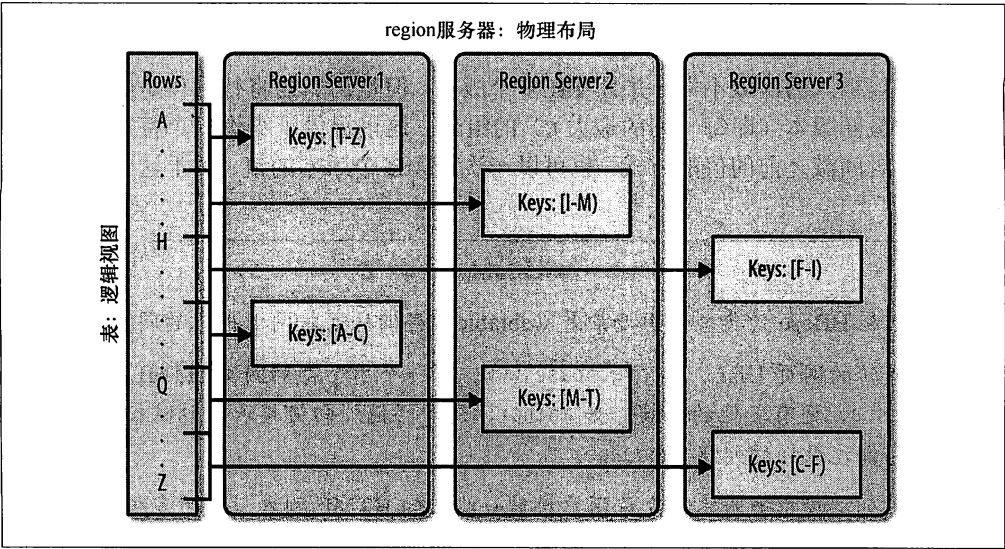


图 1-7 **region** 中的行分组加载到不同的服务器中

① 虽然 HBase 不支持在线的 **region** 合并, 但是有离线处理合并的工具, 详情见 11.6 节。



BigTable 的论文中指出, 每台服务器中 region 的最佳加载数量是 10~1000, 每个 region 的最佳大小是 100 MB~200 MB。这个标准是以 2006 年 (以及更早以前) 的硬件配置为基准参数建议的。按照 HBase 和现在的硬件能力, 每台服务器的最佳加载数量差不多还是 10~1000, 但每个 region 的最佳大小是 1 GB~2 GB 了。

虽然数量增加了, 但是基本原理还是一样的: 每台服务器能加载的 region 数量和每个 region 的最佳存储大小取决于单台服务器的有效处理能力。

region 拆分和服务相当于其他系统提供的自动分区 (autosharding)。当一个服务器出现故障后, 该服务器上的 region 可以快速恢复, 并获得细粒度的负载均衡, 因为当服务于某个 region 的服务器当前负载过大、发生错误或者被停止使用导致不可用时, 系统会将该 region 移到其他服务器上。

region 拆分的操作也非常快——接近瞬间, 因为拆分之后的 region 读取的仍然是原存储文件, 直到合并把存储文件异步地写成独立的文件, 详情见第 8 章。

1.4.4 存储 API

“BigTable 并不支持完整的关系数据模型; 相反, 它提供了具有简单数据模型的客户端, 这个简单的数据模型支持动态控制数据的布局格式……”

API 提供了建表、删表、增加列族和删除列族操作, 同时还提供了修改表和列族元数据的功能, 如压缩和设置块大小。此外, 它还提供了客户端对给定的行键值进行增加、删除和查找操作的功能。

scan API 提供了高效遍历某个范围的行的功能, 同时可以限定返回哪些列或者返回的版本数。通过设置过滤器可以匹配返回的列, 通过设置起始和终止的时间范围可以选择查询的版本。

在这些基本功能的基础上, 还有一些更高级的特性。系统支持单行事务, 基于这个特性, 系统实现了对单个行键下存储的数据的原子读-修改-写 (read-modify-write) 序列。虽然还不支持跨行和跨表的事务, 但客户端已经能够支持批量操作以获得更好的性能。

单元格的值可以当作计数器使用, 并且能够支持原子更新。这个计数器能够在一个操作中完成读和修改, 因此尽管是分布式的系统架构, 客户端依然可以利用此特性实现全局的、强一致的、连续的计数器。

还可以在服务器的地址空间中执行来自客户端的代码, 支持这种功能的服务端框架叫做协处理器 (coprocessor)。这个代码能直接访问服务器本地的数据, 可以用于实现轻量级批处理作业, 或者使用表达式并基于各种操作来分析或汇总数据。



HBase 在 0.91.0 版本中加入了协处理器。

最后，系统通过提供包装器集成了 MapReduce 框架，该包装器能够将表转换成 MapReduce 作业的输入源和输出目标。

与 RDBMS 不同，HBase 系统没有提供查询数据的特定域语言，例如 SQL。数据存取不是以声明的方式完成的，而是通过客户端 API 以纯粹的命令完成的。HBase 的 API 主要是 Java 代码，但是也可以用其他编程语言来存取数据。

1.4.5 实现

“BigTable……允许客户端推断在底层存储中表示的数据的位置属性。”

数据存储存储在存储文件（store file）中，称为 HFile，HFile 中存储的是经过排序的键值映射结构。文件内部由连续的块组成，块的索引信息存储在文件的尾部。当把 HFile 打开并加载到内存中时，索引信息会优先加载到内存中，每个块的默认大小是 64KB，可以根据需要配置不同的块大小。存储文件提供了一个设定起始和终止行键范围的 API 用于扫描特定的值。



关于实现的细节会在第 8 章中讨论，接下来只是对系统的实现进行简单介绍。

每一个 HFile 都有一个块索引，通过一个磁盘查找就可以实现查询。首先，在内存的块索引中进行二分查找，确定可能包含给定键的块，然后读取磁盘块找到实际要找的键。

存储文件通常保存在 Hadoop 分布式文件系统（Hadoop Distributed File System，HDFS）中，HDFS 提供了一个可扩展的、持久的、冗余的 HBase 存储层。存储文件通过将更改写入到可配置数目的物理服务器中，以保证不丢失数据。

每次更新数据时，都会先将数据记录在提交日志（commit log）中，在 HBase 中这叫做预写日志（write-ahead log，WAL），然后才会将这些数据写入内存中的 memstore 中。一旦内存保存的写入数据的累计大小超过了一个给定的最大值，系统就会将这些数据移出内存作为 HFile 文件刷写到磁盘中。数据移出内存之后，系统会丢弃对应的提交日志，只保留未持久化到磁盘中的提交日志。在系统将数据移出 memstore 写入磁盘中过程中，可以不必阻塞系统的读写，通过滚动内存中的 memstore 就能达到这个目的，即用空的新 memstore 获取更新数据，将满的旧 memstore 转换成一个文件。请注意，memstore 中的数据已经按照行键排序，持久化到磁盘中的 HFile 也是按照这个顺序排列的，所以不必执行排序或其他特殊处理。



现在我们开始讨论本节开头提到的 BigTable 引文，并搞清位置属性（locality property）是什么。所有文件包含的键/值对都是按行键顺序归类的，并且对块级别的操作做了优化，比如顺序地读取这些键/值对的操作，因此，行键需要特殊指定。在前面介绍 webtable 的例子中，你可能已经注意到了，使用的行键是反转的 FQDN（网址的域名部分），如 org.hbase.www。主要原因是通过网址反转，把网址中最重要的部分——顶级域名（TLD）放在最前面，可以让来自 hbase.org 的网页在 HBase 中顺序地排列在一起。例如，blog.hbase.org 下面的页面和那些来自 www.hbase.org 的页面会归为一类，或者说 org.hbase.www 会排列在 org.hbase.blog 的后面。

因为存储文件是不可被改变的，所以无法通过移除某个键/值对来简单地删除值。可行的解决办法是，做个删除标记（delete marker，又称墓碑标记），表明给定行已被删除的事实。在检索过程中，这些删除标记掩盖了实际值，客户端读不到实际值。

读回的数据是两部分数据合并的结果，一部分是 memstore 中还没有写入磁盘的数据，另一部分是磁盘上的存储文件。值得注意的是，数据检索时用不着 WAL，只有服务器内存中的数据在服务器崩溃前没有写入到磁盘，而后进行恢复数据时才会用到 WAL。

随着 memstore 中的数据不断刷写到磁盘中，会产生越来越多的 HFile 文件，HBase 内部有一个解决这个问题的管家机制，即用合并将多个文件合并成一个较大的文件。合并有两种类型：minor 合并（minor compaction）和 major 压缩合并（major compaction）。minor 合并将多个小文件重写为数量较少的大文件，减少存储文件的数量，这个过程实际上是个多路归并的过程。因为 HFile 的每个文件都是经过归类的，所以合并速度很快，只受到磁盘 I/O 性能的影响。

major 合并将一个 region 中一个列族的若干个 HFile 重写为一个新 HFile，与 minor 合并相比，还有更独特的功能：major 合并能扫描所有的键/值对，顺序重写全部的数据，重写数据的过程中会略过做了删除标记的数据。断言删除此时生效，例如，对于那些超过版本号限制的数据以及生存时间到期的数据，在重写数据时就不再写入磁盘了。



这种架构来源于 LSM 树（见 8.1.2 节）。唯一的区别是，LSM 树将多页块（multipage block）中的数据存储在磁盘中，其存储结构布局类似于 B 树。在 HBase 中，数据的更新与合并是轮流进行的，而在 BigTable 中，更新是更粗粒度的操作，整个 memstore 会存储为一个新的存储文件，不会马上合并。可以把 HBase 的这种架构称为“LSM 映射”（Log-Structured Sort-and-Merge-Map）。后台合并过程与 LSM 树的结构合并过程相对应，只不过 HBase 合并重写整个文件，而不会像 LSM 树一样只操作树结构的部分数据，LSM 树结构也正是因为这种操作而得名。

HBase 中有 3 个主要组件：客户端库、一台主服务器、多台 region 服务器。可以动态地增加和移除 region 服务器，以适应不断变化的负载。主服务器主要负责利用 Apache ZooKeeper 为 region 服务器分配 region，Apache ZooKeeper 是一个可靠的、高可用的、持久化的分布式协调系统。

Apache ZooKeeper

ZooKeeper^① 是 Apache 软件基金会旗下的一个独立开源系统，它是 Google 公司为解决 BigTable 中问题而提出的 Chubby 算法的一种开源实现。它提供了类似文件系统一样的访问目录和文件（称为 znode）的功能，通常分布式系统利用它协调所有权、注册服务、监听更新。

每台 region 服务器在 ZooKeeper 中注册一个自己的临时节点，主服务器会利用这些临时节点来发现可用服务器，还可以利用临时节点来跟踪机器故障和网络分区。

在 ZooKeeper 服务器中，每个临时节点都属于某一个会话，这个会话是客户端连接上 ZooKeeper 服务器之后自动生成的。每个会话在服务器中有一个唯一的 id，并且客户端会以此 id 不断地向 ZooKeeper 服务器发送“心跳”，一旦发生故障 ZooKeeper 客户端进程死掉，ZooKeeper 服务器会判定该会话超时，并自动删除属于它的临时节点。

HBase 还可以利用 ZooKeeper 确保只有一个主服务器在运行，存储用于发现 region 的引导位置，作为一个 region 服务器的注册表，以及实现其他目的。ZooKeeper 是一个关键组成部分，没有它 HBase 就无法运作。ZooKeeper 使用分布式的一系列服务器和 Zab 协议（确保其状态保持一致）减轻了应用上的负担。

图 1-8 展示了 HBase 的各个组件是如何利用像 HDFS 和 ZooKeeper 这样的现有系统协调地组织起来的，而且还增加了自己的层以形成一个完整的平台。

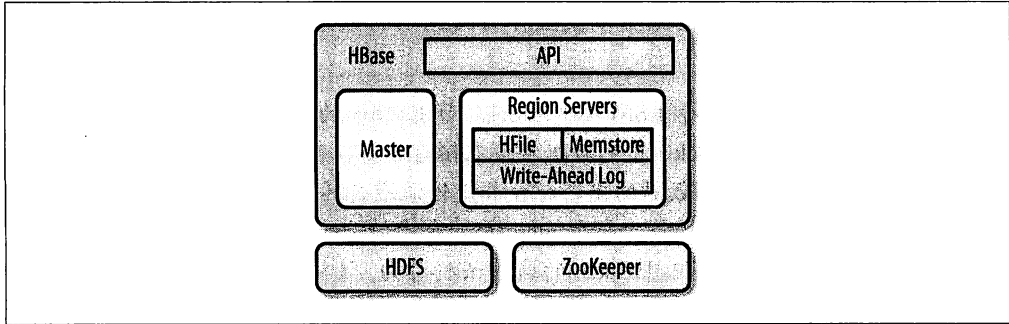


图 1-8 HBase 利用自身组件的同时还平衡地利用了已有的系统

① 有关 Apache ZooKeeper 的更多信息请参见 Apache ZooKeeper 官方网站 (<http://hadoop.apache.org/zookeeper/>)。

master 服务器负责跨 region 服务器的全局 region 的负载均衡，将繁忙的服务器中的 region 移到负载较轻的服务器中。主服务器不是实际数据存储或者检索路径的组成部分，它仅提供了负载均衡和集群管理，不为 region 服务器或者客户端提供任何的数据服务，因此是轻量级服务器。此外，主服务器还提供了元数据的管理操作，例如，建表和创建列族。

region 服务器负责为它们服务的 region 提供读写请求，也提供了拆分超过配置大小的 region 的接口。客户端则直接与 region 服务器通信，处理所有数据相关的操作。

8.5 节详细介绍了客户端如何执行 region 查找。

1.4.6 小结

“数十亿行 × 数百万列 × 数千个版本 = TB 级或 PB 级的存储”

我们已经见识到，BigTable 的存储架构是怎样使用多台服务器将按键归类的行拆分成多个范围来负载均衡的，还看到了它是怎样使用上千台机器存储 PB 级数据的。使用的存储格式对于顺序读相邻的键/值来说是很理想的，这种格式针对块 I/O 操作做了优化，能最大限度地利用磁盘传输通道。

表的扫描与时间呈线性关系，行键的查找以及修改操作与时间呈对数关系——极端情况下是常数关系（使用了布隆过滤器）。HBase 在设计上完全避免了显式的锁，提供了行原子性操作，这使得系统不会因为读写操作性能而影响系统扩展能力。

当前的列式存储结构允许表在实际存储时不存储 NULL 值，因此表可以看作是个无限的、稀疏的表。表中每行数据只由一台服务器所服务，因此 HBase 具有强一致性，使用多版本可以避免因并发解耦过程引起的编辑冲突，而且可以保留这一行的历史变化。

事实上，至少从 2005 年开始，BigTable 就已经应用于 Google 生产，拥有非常多的应用场景，从批处理到实时数据服务都有它的身影。BigTable 存储的数据既可以非常小（例如 URL），也可以特别大（如网页和卫星图片），还成功地为许多知名的 Google 产品提供了灵活的、高性能的解决方案，这些产品包括 Google Earth、Google Reader、Google Finance 和 Google Analytics。

1.5 HBase: Hadoop 数据库

看过 BigTable 的架构之后，我们可能会简单地认为 HBase 完全是 Google 的 BigTable 的开源实现。但是这个说法可能过于简单，因为两者之间还有些差异（大多是细微的）值得一提。

1.5.1 历史

HBase 是 Powerset^① 在 2007 年创建的，最初是 Hadoop 的一部分。之后，它逐步成为 Apache 软件基金会旗下的顶级项目，具备 Apache 软件许可证，版本为 2.0。

HBase 项目的主页是 <http://hbase.apache.org/>，通过这个主页可以链接到文档 (documentation)、wiki、源代码库 (source repository)，以及已经发布的库和源代码的下载站点。

下面是一个 HBase 随时间发展的简短概述。

- 2006 年 11 月：Google 发布 BigTable 论文。
- 2007 年 2 月：HBase 宣布在 Hadoop 项目中成立。^②
- 2007 年 10 月：HBase 第一个“可用”版本 (Hadoop 0.15.0)。
- 2008 年 1 月：Hadoop 成为 Apache 的顶级项目，HBase 成为 Hadoop 的子项目。
- 2008 年 10 月：HBase 0.18.1 发布。
- 2009 年 1 月：HBase 0.19.0 发布。
- 2009 年 9 月：HBase 0.20.0 发布，性能有明显提升。
- 2010 年 5 月：HBase 成为 Apache 的顶级项目。
- 2010 年 6 月：HBase 0.89.20100621，第一个开发版本。
- 2011 年 1 月：HBase 0.90.0 发布，稳定性和持久性有所提升。
- 2011 年年中：HBase 0.92.0 发布，支持协处理器和安全控制。



2010 年 5 月前后，HBase 的开发者决定打破一直依赖的、步调一致的 Hadoop 的版本编号。原因是 HBase 有一个更快的发布周期，同时更接近 1.0 版本的水平，比 Hadoop 的预期更快。

为此，版本号从 0.20.x 跳到了 0.89.x，跳跃相当明显。此外，还做了一个决定，将 0.89.x 定为早期的开发版本。在 0.89 的基础上最终发布了 0.90，即面向所有用户的稳定版。

① Powerset 公司位于旧金山，开发了一套用于互联网的自然语言搜索引擎。在 2008 年 7 月 1 日，微软公司收购了 Powerset，之后 Powerset 放弃了对 HBase 开发的后续支持。

② 在 Apache JIRA（网站上的问题追踪系统）中找到 HBASE-287，里面可以找到当时的记录，读者可以看到 Mike Cafarella 提交的最初代码，这个代码很快就被 Jim Kellerman 采纳了，Jim Kellerman 当时就职于 Powerset。

1.5.2 命名

HBase 与 BigTable 最大的不同就是命名。表 1-1 罗列了两个系统之间相同组件的命名有哪些不同。

表 1-1 命名差异

HBase	BigTable
Region	Tablet
RegionServer	Tablet server
Flush	Minor compaction
Minor compaction	Merging compaction
Major compaction	Major compaction
Write-ahead log	Commit log
HDFS	GFS
Hadoop MapReduce	MapReduce
MemStore	memtable
HFile	SSTable
ZooKeeper	Chubby

更多的差异参见附录 F。

1.5.3 小结

让我们回到 1.3.1 节来看看怎样用维度来描述 HBase 系统。HBase 是一个分布式的、持久的、强一致性的存储系统，具有近似最优的写性能（能使 I/O 利用率达到饱和）和出色的读性能，它充分利用了磁盘空间，支持特定列族切换可选压缩算法。

HBase 继承自 BigTable 模型，只考虑单一的索引，类似于 RDBMS 中的主键，提供了服务器端钩子，可以实施灵活的辅助索引解决方案。此外，它还提供了过滤器功能，减少了网络传输的数据量。

HBase 并未将说明性查询语言作为核心实现的一部分，对事务的支持也有限。但行原子性和“读-修改-写”操作在实践中弥补了这个缺陷，它们覆盖了大部分的使用场景并消除了在其他系统中经历过的死锁、等待问题。

HBase 在进行负载均衡和故障恢复时对客户端是透明的。在生产系统中，系统的可扩展性体现在系统自动伸缩的过程中。更改集群并不涉及重新全量负载均衡和数据重分区，但整个处理过程完全是自动化的。

第 2 章

安装

本章将讲述如何安装 HBase 以及如何对 HBase 进行初始化配置。我们可以在命令行中看到如何使用 HBase 的基本操作，例如，添加、检索和删除数据。



以下讲述的内容均假设用户已经安装了 Java 运行时环境 (Java Runtime Environment, JRE), Hadoop 和 HBase 要求都至少是 1.6 版本(也称为 Java 6)以上的 JRE, 推荐使用 Oracle 公司(原来是 Sun 公司, 后来 Oracle 收购了 Sun 公司)提供的版本, 可以在 <http://www.java.com/download/> 下载。如果用户还未安装 Java 运行时环境, 或者在使用过程中存在问题, 请参见 2.2.3 节的“Java”。

2.1 快速启动指南

想知道如何运行 HBase 吗? 想即刻知道 HBase 是如何工作的吗? 从“tl;dr”^①部分开始了解吧! 这是最容易理解的部分, 因为用户只需要从 Apache HBase 发布版本页面下载最新的 HBase 版本 (<http://www.apache.org/dyn/closer.cgi/hbase/>), 并将内容解压到合适的目录, 如 `/usr/local` 或者 `/opt` 即可, 就像这样:

```
$ cd /usr/local
$ tar -zxvf hbase-x.y.z.tar.gz
```

设置数据路径

此刻, 就可以准备启动 HBase 了。在启动 HBase 之前, 建议先把数据目录设置到合适的位置。需要编辑配置文件 `conf/hbase-site.xml`, 并设置合适的数据路径: 通过对

^① tl;dr 是英文 “too long; didn’t read” 的缩写, 意思是 “太长, 别读”。——译者注

属性键 `hbase.rootdir` 赋值，来配置用户想要的 HBase 进行写操作的路径写路径。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>hbase.rootdir</name>
    <value>file:///<PATH>/hbase</value>
  </property>
</configuration>
```

用户可以自定义上述示例配置文件中的 `<PATH>`，将其替换为希望存储 HBase 数据的路径。默认情况下，`hbase.rootdir` 设置为 `/tmp/hbase-${user.name}`，当服务器重启的时候，数据可能会丢失，因为很多操作系统会在重启的时候清空 `/tmp` 目录。

替换之后，就可以启动 HBase 并开始第一次交互了。在交互窗口中的提示符后面输入 `status` 命令（按回车键完成命令操作）：

```
$ cd /usr/local/hbase-0.91.0-SNAPSHOT
$ bin/start-hbase.sh
starting master, logging to \
/usr/local/hbase-0.91.0-SNAPSHOT/bin/../logs/hbase-<username>-master-localhost.out
$ bin/hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version: 0.91.0-SNAPSHOT, r1130916, Sat Jul 23 12:44:34 CEST 2011

hbase(main):001:0> status
1 servers, 0 dead, 2.0000 average load
```

以上显示信息表明 HBase 已启动并且正在运行。下面通过使用一些命令证明 HBase 能够将数据存放进去，接着对其进行检索。



现在看来读者也许还不是很清楚我们正在做的事情的作用，但这都是必要的准备工作，正如坐在一辆汽车中启动发动机的时候，也需要时刻掌控好刹车装置一样。其实，在类似生产的环境中使用 HBase，还需要很多配置和要了解的知识，不过我们可以从基础的 HBase 命令开始学习，然后逐渐熟悉高层次的概念。

现在我们运行的是所谓的单机模式 (Standalone Mode)，查看后面的章节（见 2.5 节）将会看到多种可用的模式。现在了解下面这一点很重要：在单机模式中，一切事物都运行在单个 Java 进程中，并且所有的文件默认情况下都将存储在 `/tmp` 路径下——除非用户按照前文中给出的提示，改变了存储路径。如果数据存储在默认路径下，服务器一旦重启，测试数据就会丢失。数据一旦被操作系统删除，将无法恢复。

现在来创建一个简单的表并新增几行数据：

```
hbase(main):002:0> create 'testtable','colfam1'
0 row(s) in 0.2930 seconds

hbase(main):003:0> list 'testtable'
TABLE
testtable
1 row(s) in 0.0520 seconds

hbase(main):004:0> put 'testtable','myrow-1','colfam1:q1','value-1'
0 row(s) in 0.1020 seconds

hbase(main):005:0> put 'testtable','myrow-2','colfam1:q2','value-2'
0 row(s) in 0.0410 seconds

hbase(main):006:0> put 'testtable','myrow-2','colfam1:q3','value-3'
0 row(s) in 0.0380 seconds
```

通过一条命令，我们创建了一张带有一个列族的表，读者可以通过 list 命令来检查这张表是否已经存在了。目前只有一张表的情况下，读者可以看到它是如何输出表名 testtable 的。然后，我们存放几行数据。读者仔细阅读示例的话可以发现：我们通过两个不同的行键 myrow-1 和 myrow-2 把新增数据添加到两个不同的行中。由第 1 章中描述的表结构可知，在有了一个名为 colfam1 的列族之后，还要添加一个任意的限定符才能形成实际的列，如 colfam1:q1、colfam1:q2 和 colfam1:q3。

接下来要做的是，确认新增的数据是否能被检索，使用 scan 操作就能实现：

```
hbase(main):007:0> scan 'testtable'
ROW                                COLUMN+CELL
myrow-1                            column=colfam1:q1, timestamp=1297345476469, value= value-1
myrow-2                            column=colfam1:q2, timestamp=1297345495663, value= value-2
myrow-2                            column=colfam1:q3, timestamp=1297345508999, value= value-3

2 row(s) in 0.1100 seconds
```

你已经看到 HBase 是如何打印数据的，它通过面向单元格的方式分别输出每一列数据。可以看出它确实打印了两次 myrow-2，这与预期的一样，后面还显示了每一列的实际数值。

如果想要获得单行数据，可以使用 get 命令，这个命令有很多选项，后面会详细讨论，现在只简单地尝试一下这个命令：

```
hbase(main):008:0> get 'testtable','myrow-1'
COLUMN                                CELL
colfam1:q1                            timestamp=1297345476469,value=value-1

1 row(s) in 0.0480 seconds
```

删除数据也是基本操作之一，同样，delete 命令也有很多选项，但现在我们只简单地删除一个具体的单元格，并检查数据是否真的删除了：

```
hbase(main):009:0> delete 'testtable','myrow-2','colfam1:q2'
0 row(s) in 0.0390 seconds

hbase(main):010:0> scan 'testtable'
ROW                COLUMN+CELL
myrow-1            column=colfam1:q1,timestamp=1297345476469,value=value-1
myrow-2            column=colfam1:q3,timestamp=1297345508999,value=value-3

2 row(s) in 0.0620 seconds
```

在对这个简单的练习进行总结之前，需要先禁用并删除这张练习表：

```
hbase(main):011:0> disable 'testtable'
0 row(s) in 2.1250 seconds

hbase(main):012:0> drop 'testtable'
0 row(s) in 1.2780 seconds
```

然后，通过输入 exit 命令关闭 Shell 并返回到命令行窗口：

```
hbase(main):013:0> exit
$ _
```

最后，运行 stop-hbase.sh 脚本关闭 HBase 系统：

```
$ bin/stop-hbase.sh
stopping hbase.....
```

这样就完成了整个流程。我们成功地创建了表，进行了数据的添加、检索以及删除，最终通过 HBase Shell 删除了表。

2.2 必备条件

下面所描述的条件并不是所有 HBase 支持的运行模式都需要的。如果读者纯粹是出于本地测试的目的，就只需要安装 Java，如 2.1 节所述。

2.2.1 硬件

很难为 HBase 推荐一款具体类型的服务器。事实上，HBase 能在多种不同配置的硬件上运行。通常的描述是商用（commodity）硬件，但这代表什么意思呢？

对于初学者，我们在此不讨论桌面 PC 系统，只讨论服务器级别的机器。因为 HBase 是 Java 编写的，所以至少需要支持当前的 Java 运行时环境。region 服务器的内存主要服务于内部数据结构，例如，memstore 和块缓存，因此你需要安装 64 位操作系统才能

分配和使用大于 4 GB 的内存空间。

在实践中,为了能够像 MapReduce 一样有效地利用 HDFS, HBase 大多是与 Hadoop 安装在一起的。这样能很大程度地减少对网络 I/O 的需求,同时能加快处理速度。当在同一服务器上运行 Hadoop 和 HBase 时,最少会有 3 个 Java 进程(DataNode、TaskTracker 和 RegionServer)在运行,而且在执行 MapReduce 作业时,进程数还会激增。要有效地运行所有这些进程,需要保证拥有一定数量的内存、磁盘和 CPU 资源。



因为在所有已知的产品系统中, Hadoop 都是 HBase 的后备存储,所以在此假设读者已较好地掌握了 Hadoop。如果确实未接触过 HBase 和 Hadoop,建议先从最基础的 Hadoop 开始学习,哪怕先了解点皮毛,这里推荐一本 Tom White 撰写的《Hadoop 权威指南(第 2 版)》供大家参考。除此之外,需要建立正常工作的 HDFS 和 MapReduce 集群。

鉴于大多数操作系统都需要一定的空闲资源来保证工作更有效,因此将可用的内存都给 Java 进程是不明智的,例如,磁盘 I/O 缓冲区是由 Linux 内核维护的。HBase 间接地利用了已有的本地磁盘 I/O,并将进程归属于同一服务器,使得操作系统在维持自己的块缓存时性能更好。

我们将需求分为两类:服务器和网络化。下面首先探究的是服务器硬件方面的需求,然后是网络建立所需要的条件。

1. 服务器

在 HBase 和 Hadoop 中有两种类型的机器: master (HDFS 的 NameNode、MapReduce 的 JobTracker, 以及 HBase 的 Master) 和 slave (HDFS 的 DataNode、MapReduce 的 TaskTracker, 以及 HBase 的 RegionServer)。有可能的话,两类机器的硬件配置有所区别会更好,然而,为了省事,配置相同的情况也很普遍。实际上 master 并不需要大存储空间,因此不需要挂载过多的磁盘。由于 master 的重要性大于 slave,因此 master 可以通过冗余来提升硬件可用率。必要时我们会解释这两类机器的不同之处。

由于 Java 是在用户空间 (user land) 运行的,所以可以在每一个支持 Java 运行时的操作系统上运行它,虽然有一些备受推荐的运行 Java 的操作系统,但是没有用户干预时,Java 将无法运行 (详情见 2.2.3 节)。因此,用户可以选择的硬件供应商有很多,甚至可以自己组装硬件。以下是通用硬件需求的要求。

CPU

使用单核 CPU 机器,同时运行 3 个或者更多的 Java 进程和操作系统的服务进程是不合理的。在生产系统中,通常采用的是多核处理器^①。四核的处理器能够满足需求,

^① 见维基百科中的“Multi-core processor” (<http://en.wikipedia.org/wiki/Multi-core>)。

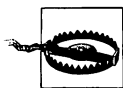
但六核的处理器更受欢迎。大多数硬件支持一个以上的 CPU，所以系统可以使用两个四核 CPU，达到了八核。这样每一个基本的 Java 进程都可以独立占有一个核，而像 Java 垃圾回收（Garbage Collection，GC）这样的后台任务则可以并行执行。此外还有超线程（hyperthreading），它更加大了这种处理优势。

就 CPU 而言，master 机器与 slave 机器的规格应该是一样的。

节点类型	推荐
master	双四核 CPU，2.0 GHz~2.5 GHz
slave	双四核 CPU，2.0 GHz ~2.5 GHz

内存

真正的问题是：给单个进程分配过多的内存会产生问题么？在理论上不会，但实践证明，使用 Java 时，不应该为一个进程设置过多的内存。内存在 Java 术语中称为堆（heap），会在使用过程中产生许多碎片，在最坏的情况下，整个堆需要重写一次，这与众所周知的磁盘碎片整理相似，但重写堆不能在后台运行。Java 运行时环境会暂停所有进程内的逻辑并进行清理，这可能会导致不少问题（稍后详细讨论）。设置的堆越大，这个过程花的时间就越长。进程并不需要大量的内存，合适的内存大小可以避免上述问题，但是 region 服务器和块缓存在理论上没有上限。因此为了避免上述问题的出现，需要根据用户访问模式找到一个平衡点。



在写这本书时候，一般认为给 region 服务器设置超过 16 GB 的堆是很危险的。一旦发生 Full 垃圾回收会造成很长时间的重写内存堆操作。这个时候 master 可能会判定进程已经死掉，并将其从工作列表中移除。

这种情况有时候依赖于使用的 JRE，有的 JRE 实现可以在垃圾回收时不阻塞进程内的工作线程。

表 2-1 展现了一个非常基本的内存配置标准。值得注意的是，这仅仅是个例子，配置不仅仅高度依赖集群的大小和数据写入量，还依赖于用户的访问模式，比如是只有交互式访问，还是交互式和批量处理混合使用（如 MapReduce）。

表 2-1 拥有 800 TB 存储空间的集群中每个 Java 进程的典型内存配置

进程	堆	描述
NameNode	8 GB	每 100 TB 的数据或者是每 100 万个文件大约会占用 NameNode 堆 1 GB 的内存
SecondaryNameNode	8 GB	在内存中重做主 NameNode 的 EditLog，因此配置需要与 NameNode 一样
JobTracker	2 GB	适度

进程	堆	描述
HBase Master	4 GB	轻量级负载, 适度
DataNode	1 GB	适度
TaskTracker	1 GB	适度
HBase RegionServer	12 GB	大部分可用内存, 同时为操作系统(缓冲区缓存)和任务进程留下足够的空间
Task Attempts	1 GB (ea.)	剩余内存除以允许的任务最大的单机进程数
ZooKeeper	1 GB	适度

推荐配置: master 机器要运行 NameNode、SecondaryNameNode、JobTracker 和 HBase Master, 推荐 24 GB 内存; slave 机器要运行 DataNode、TaskTracker 和 HBase RegionServer, 推荐 24 GB 内存及以上的配置。

节点类型	推荐
master	24 GB
slave	24 GB (及以上)



这里主要是建议优化内存通道。例如, 使用双通道内存时, 每台机器应该配备成对的插槽 (DIMM); 使用三通道内存时, 每台机器配备的插槽个数应该是 3 的倍数。这可能意味着机器可以用 18 GB (9 × 2 GB) 内存替换 16 GB (4 × 4 GB) 内存。

这不仅仅需要服务器的主板支持, 同时也需要 CPU 兼容这种模式。例如, 有的 CPU 只兼容双通道内存, 即使主板支持三通道模式, CPU 也只能在双通道模式下工作。

磁盘

数据存储在 slave 机器上, 因此 slave 服务器需要大量的存储空间。用户需要根据主要是面向读/写, 还是面向数据加工, 来平衡可用的 CPU 内核数量与磁盘数量的使用。通常情况下, 用户应该保证每个磁盘上至少有一个核, 所以在 8 核心服务器增加 6 块磁盘是较优的, 加入更多磁盘可能并不会带来显著的性能提升。

RAID 还是简单 JBOD?

一个常见的问题是磁盘如何挂载到服务器上。这里要区别对待 master 和 slave。对于 slave 来说, 并不建议使用 RAID^① 模式, 而是要使用所谓的 JBOD^② 模式。RAID 比单个磁盘慢, 因为 RAID 受管理开销和流水线写能力的限制, 并取决于

① 见维基百科中的“RAID” (<http://en.wikipedia.org/wiki/RAID>)。

② 见维基百科中的“JBOD” (<http://en.wikipedia.org/wiki/JBOD#JBOD>)。

RAID 的等级（通常采用的 RAID 模式是 RAID 0，这种数据上的并行操作可以充分利用总线的带宽，显著提高磁盘整体存取性能），但一块磁盘出现故障会使所有数据节点不可用。

对 master 节点来说，使用 RAID 主要是为了保护关键性的文件系统数据，通常的配置是 RAID 1+0 或 RAID 0+1。

master 和 slave 一定要使用带 RAID 固件（RAID firmware）的磁盘。这类磁盘与消费级磁盘的主要区别是，一旦硬件出错，RAID 固件马上失效，因此 DataNode 进程可以快速知道发生了故障。

还要考虑磁盘驱动器的类型，例如，是 2.5 英寸还是 3.5 英寸？是 SATA 还是 SAS？一般更推荐使用 SATA 驱动器，因为 SATA 比 SAS 更节省成本，虽然 SAS 盘安全性高于 SATA，但一般的软件策略中是跨机架数据冗余，因此可以放心地使用 SATA 盘。实际使用哪种类型的磁盘驱动器，最终取决于负担得起哪一种的成本。虽然 3.5 英寸的磁盘比 2.5 英寸的磁盘可靠，但考虑到服务器机架的因素，你可能会选择 2.5 英寸的磁盘。

通常使用的磁盘大小是 1 TB，如果有需要也可以使用 2TB 的磁盘。使用 6~12 个配有 1 TB 或 2 TB 磁盘的高密度服务器比较好，既可以得到较高的存储容量，又拥有足够 CPU 内核的 JBOD 模式，以获得较高的磁盘带宽。

节点类型	推荐
master	4×1 TB SATA，RAID 0+1（也可以用 2 TB 的）
slave	6×1 TB SATA，JBOD

IOPS

磁盘的数量是影响每秒进行读写操作次数（I/O Operation Per Second，IOPS）的重要指标。例如，一般使用 4×1 TB 的磁盘比较好，这种方式可以使节点的 IOPS 达到 400 次，即 400 MB/s 的传输吞吐量，非常适合冷数据的访问需求。^①

如果有更高的需求还可以使用 8×500 GB 的磁盘，这样每个节点的磁盘吞吐量能够达到 800IO PS/s，接近千兆以太网的线路速率。总之，需要结合具体需求并采用适量的磁盘来实现目标。

机架

实际上，服务器机架（chassis）不是至关重要的因素，机架的价格虽然不同，但提

① 这里假定每个磁盘驱动器支持的 IOPS 为 100 次，单个磁盘驱动器的传输吞吐量就是 100 MB/s。

供的功能大体类似。对于通用服务器来说，一般情况下最好是回避具有专有功能和选项的特殊硬件，这样这些服务器就可以根据需求通过简单的组合，达到扩容集群容量的目的。

就网络而言，推荐使用双端口千兆以太网卡，即双通道绑定网卡。如果已经能够支持万兆网或者无限带宽（InfiniBand）技术，应该毫不犹豫地使用它。

slave 采用一个电源供应器（Power Supply Unit, PSU）就够了，但 master 节点应该使用冗余的 PSU，如双 PSU。

从硬件部署密度来看，机架单元（简称 U）越少越好。通常 1U 和 2U 的服务器部署在 19 英寸的机架中。在选择机架大小时要考虑的是，能容纳多少磁盘以及它们的能源消耗。通常 1U 的服务器适合挂载较少的磁盘，或者为了达到容量要求只能用 2.5 英寸的磁盘。

节点类型	推荐
master	万兆以太网，双 PSU，1U 或者 2U
slave	万兆以太网，单 PSU，1U 或者 2U

2. 网络

数据中心里，服务器通常是挂载在 19 英寸的机架上，这种机架能容纳 40U 甚至更多，用户可以使用置顶式（Top-of-Rack, ToR）交换机将 40 台机器（放在半边机架上）连接在一起（有些公司使用一个整体机架挂载 80 台机器，每边 40 台）。如果每台服务器用的都是千兆网卡，那么要确保 ToR 交换机有足够快的速度处理服务器创建的吞吐。通常交换机的背板不能以线路速率处理所有的端口，换句话说，理论上承诺的实际达不到。

交换机通常有 24 或 48 个端口，配有前面提到的通道绑定技术及双端口网卡，用户需要保证交换机连网的规模足够大，从而能够提供足够的带宽。安装 40 个 1U 服务器需要 80 个网络端口，所以在实践中需要设置多个机架交换机交错使用，然后汇总到一个更大的核心汇聚交换机（Core Aggregation Switch, CaS）。最终得到一个两层（two-tier）架构，由 ToR 交换机分配，CaS 汇聚。

尽管不能满足所有的大规模安装需求，但我们要知道这是一种常用的设计模式。鉴于运维是规划的一部分，因此用户需要知道有多少数据要存储，有多少客户端并发读写，然后计算出需要多少台服务器，最后还要考虑每台服务器的联网能力。

用户通过公开的邮件列表或其他渠道报告了 HBase 的一些问题，反映得比较多的是，I/O 性能在批量插入大量数据的时候比预期的要差，很显然，这是连网问题导致的。而导致连网问题的原因可能是连网配置不当或者使用了错误的网络接口卡（Network Interface Card, NIC），也可能是服务器的数据吞吐量完全超出了交换机的载荷。一定

要仔细验证集群的每个硬件组件，从而避免突然出现的故障问题，这些问题本来就可以通过合适的硬件配置避免。

最后，目前 Hadoop 和 HBase 的安全功能是内置的，通常整个集群都位于自己的网络中，因此，可以通过防火墙保护集群，控制客户端对集群的请求。

2.2.2 软件

讨论了所需要的硬件，也购买了服务器，现在要讨论一下软件了。除了要考虑底层的操作系统，还要考虑操作系统挂载的文件系统以及各种其他服务的配置。



大多数列出的需求都不依赖于 HBase，都是在底层、操作层中用到的。管理员可以在应用过程中验证该问题。

1. 操作系统

推荐操作系统（OS）是一件非常棘手的事情，尤其是在开源领域中。在过去的两年里，好像 HBase 偏好使用 Linux 系统工作，事实上，Hadoop 与 HBase 本来就是基于 Linux 系统或者 Unix 系统开发的，还可以在其他类 Unix 系统上运行。不过，用户可以在支持 Java 的任何一个 OS 中运行 Hadoop 和 HBase，例如，Windows，不过 Hadoop 和 HBase 的测试只能在类 Unix 系统上进行，原因是启动与关闭等管理脚本都是由 Linux 或 Unix 命令行 Shell 提供的。

Unix 与类 Unix 系统的区别是开源免费与闭源收费。此外，这两类系统都能用，具体用哪个就看用户所在公司的具体规定了。以下是可以支持 HBase 集群的操作系统列表。

CentOS

CentOS 是一个社区支持的免费软件操作系统，基于红帽的企业版 Linux 操作系统（Red Hat Enterprise Linux, RHEL）改造而来。CentOS 利用红帽为其自身企业提供的源代码包，创建了相应 CentOS 的各个部件，从而镜像了 RHEL 的功能、特性并按等级发布包。类似于 RHEL，它提供了 RPM 格式的软件包。

由于 CentOS 也侧重于企业应用，所以版本更新不会太快。它的目标是应用于大型基础设施，因此不必考虑短期的小增量包更新。

Fedora

Fedora 也是由一个社区支持的，红帽公司赞助的免费开源操作系统。但与 RHEL 和 CentOS 相比，Fedora 是一个全新的技术平台，致力于推进新的思路 and 特性。因此，与面向企业的产品相比，Fedora 的发布周期更短，平均每 13 个月就会更新一次版本。

事实上，它是针对工作站设计的，经常会利用到 Fedora 的一些新特性，不过流行程度与面向桌面的操作系统^①相比略逊一筹。在生产中使用 Fedora 要考虑到单一版本生命周期短的因素，还需要考虑使用一个稳定版本，而不使用最新发布的 Fedora 版本，同时依靠社区的力量修复问题。

Debian

Debian 是另一个基于 Linux 内核的操作系统，拥有免费开源的软件升级包。Debian 可用于桌面系统以及服务器系统，但是 Debian 社区的升级更新策略相对保守，所有包文件都需要经过充分的测试，且被视为稳定后，才会发布 Debian 的新版本。

与其他操作系统不同，Debian 虽然没有商业实体支持但是却有独立的项目规范。它有一套独立的安装包规则，并且只支持后缀名为 DEB 的包。Debian 能够在多种硬件平台上运行，并且拥有一个非常庞大的类库。

Ubuntu

Ubuntu 是一个基于 Debian 的 Linux 分支系统。Ubuntu 是一款由 Canonical 公司提供支持的免费并开源的软件，Canonical 公司的目的并不是销售软件而是销售 Ubuntu 服务，并提供技术支持。

Ubuntu 的发布周期长短结合，桌面版本采用 3 年的长周期(Long-Term Support, LTS)更新策略，服务器版本采用 5 年的长周期更新策略。Ubuntu 的安装包虽然是 DEB 格式的，但却是基于 Debian 的不稳定分支：在某种意义上看，Ubuntu 与 Debian 的关系和 Fedora 与红帽的关系类似。事实上，由于 Ubuntu 关键模块的更新过于频繁，所以 Ubuntu 很难被用作服务器操作系统。

Solaris

Solaris 是由 Oracle 公司提供的有限可用平台。它的前身是 Unix V4 版本，因此它与其他操作系统有很大的不同。Solaris 中部分源代码是开源的，其余则是闭源的。Solaris 是一个商业化产品，需要经过购买才能使用，每个购买到的 Solaris 版本都能够得到 10 年至 12 年的商业支持。

Red Hat Enterprise Linux

简称 RHEL，Red HatLinux 发行版的目的在于支持商业和企业级用户，分为服务器和桌面版本，官方还提供了产品培训和产品认证计划许可证。

RHEL 的安装包被称为 RPM(Red Hat Package Manager, 红帽软件包管理器)，由 *rpm* 格式的文件和自身的包管理器组成。

① DistroWatch (<http://distrowatch.com/>) 有一个 Linux 或类 Linux 操作系统的流行列表并按照流行程度进行了排名。

RHEL 提供了非常长的商业支持周期，大概有 7~10 年。



当用户需要选择服务器操作系统时需要考虑现有的基础设施，并选择一个与现有设施相适应的操作系统。

我们推荐大部分运行 HBase 的生产系统使用 CentOS 或 RHEL。

2. 文件系统

与选择操作系统一样，对于用户来说，磁盘文件系统也有多种选择。但是，实际上大量缺乏公开可用的经验数据用于比较 HBase 采用不同文件系统的效果。比较常见的文件系统有 ext3、ext4 以及 XFS，用户也可以使用其他的文件系统。一些 HBase 的用户则反馈了他们针对其中一些文件系统的研究结果，但是在生产环境中使用这些文件系统需要经过足够的测试。以下是较常用的文件系统上的一些注意事项。



需要注意这里提到的文件系统是 HDFS 的数据节点依赖的本地文件系统，而 HBase 是直接和 HDFS 打交道的。

ext3

Linux 操作系统中使用最普遍的文件系统是 ext3（详情见链接 <http://en.wikipedia.org/wiki/Ext3>）。它已经被证明是稳定可靠的文件系统，这意味着生产机器中采用 ext3 为本地文件系统是一个相对安全的选择。自从 2001 年 ext3 成为 Linux 一部分以来，ext3 一直在稳步提升，并且多年来一直是 Linux 系统的默认文件系统。

用户在使用 ext3 时需要切记以下几点优化。首先用户在挂载文件系统时应该设置 noatime 属性来禁止记录文件访问时间戳以减少内核的管理开销。HBase 不需要记录每个文件的访问时间，并且禁用这个选项可以大幅度提高磁盘的读取性能。



禁止记录最近一次文件访问时间戳能够提升性能，并且是推荐的配置。挂载选项通常配置在 `/etc/fstab` 文件中。下面一行是关于 Linux 设置 noatime 选项的例子：

```
/dev/sdd1 /data ext3 defaults,noatime 0 0
```

请注意上述描述页包含了 nodiratime 的配置选项。

另一个优化是为了更好地利用 ext3 提供的磁盘空间。默认情况下，磁盘每个块都为关键系统进程保留了一个固定的空间，以保证在磁盘存储已满的情况下不影响关键进程的使用。这个功能对关键磁盘比较有用，比如操作系统依赖的磁盘，但这个功能对于数据存储盘来说几乎无用，并且对一个大型集群中的可用存储空间

造成了极大的影响。



用户在 ext3 中可以使用 Linux 提供的命令行工具 tune2fs 减少保留块的数量以获取更多的可用磁盘空间。默认情况下每块磁盘的保留块数量是 5%，但可以安全地减少到 1%（甚至 0%）。以下是所需使用的命令：

```
tune2fs -m 1 <device-name>
```

用户需要使用被处理的磁盘替换 <device-name>，例如 /dev/sdd1。所有数据存储盘上设置 -m 1 定义保留块的比例，例如，使用 -m 0 就可以设置保留块数量为 0。

切记：这种设置只适用于数据存储磁盘，不适用于操作系统依赖的磁盘，更不适用于 master 节点上的任何磁盘！

Yahoo！曾公开表示 ext3 是它们的大型 Hadoop 集群的首选文件系统。这说明 ext3 虽然不是迄今为止最主流最新的文件系统，但它在大型集群中表现很好。事实上，用户使用 ext3 的过程中，其他级别的栈的极限会比 I/O 极限更先到达。

ext3 的最大缺点是服务器启动过程需要消耗很多的时间。ext3 的格式化磁盘操作会花费数分钟时间，这对机器中定期向上旋转磁盘可能会造成不必要的麻烦——尽管这不是常用的操作。

ext4

ext3 的下一代是 ext4（详见链接 <http://en.wikipedia.org/wiki/Ext4>），ext3 与 ext4 最初基于相同的代码，但 ext4 随后被移动到独立的项目。2008 年以后 ext4 正式成为官方 Linux 内核的一部分。ext4 只经过了几年光景就达到了这个程度，足以证明其稳定性和可靠性。不仅如此，Google 已经宣布^① 把其存储基础设施从 ext2 升级到 ext4，这可以认为是一个极其有震撼力的消息，并表明了 ext 系列文件系统（ext3、ext4 等）可以在不转存文件的情况下升级的优势。然而，选择其他文件系统如 XFS 则做不到这种升级。

从性能方面来说，ext4 打败了 ext3 并接近了高性能文件系统 XFS，同时还拥有很多高级特性，例如，能够允许单文件达到 16 TB 的大小并，支持 EB（ 10^{18} 字节）的存储空间。

ext4 中一个更重要的特性是延迟分配（delayed allocation），我们建议用户在 Hadoop 和 HBase 中将其关闭。采用延迟分配策略的数据会保留在内存中，并会在内存中保留若干数据块，直到数据最终被刷写到磁盘。这个特性会帮助块中的文件保持连续，并在某一时刻整体写入到磁盘的连续块中。这个特性减少了磁盘碎片并提高了文件读取性能。但另一方面，它增加了在服务器崩溃时数据丢失的概率。

① 在 Ars Technica 网站上看到的招聘信息，Google 聘请了 ext4 的主要开发者 Theodore Ts'o。他宣布继续专注于 ext4 和其他的 Linux 内核功能的研发。

XFS

Linux 在同一时间段开始支持 ext3 与 XFS（详情见链接 <http://en.wikipedia.org/wiki/Xfs>）。XFS 最初是在 1993 年由 Silicon Graphics 公司研发的，迄今为止大多数 Linux 已经支持 XFS。

它与 ext4 功能类似，例如，分别有 extents（分组存储块，减少每个文件需要保持的块数）以及上面提到的延迟分配。

XFS 一个很大的优势是，它引导服务器时格式化非常快，这样可以有效地减少使用磁盘组建新服务器的时间。

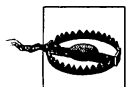
另一方面来说，XFS 也有缺点。在 XFS 的设计中有一个众所周知的缺点，它的一些操作牵涉到了元数据的变更，比如删除大量文件的操作。不过开发人员已经开始着手解决这个问题，并且提交了一些修复。使用 HBase 时一定要仔细核实可能会出问题的关键点，否则极有可能影响使用。不过 HBase 需要操作的文件一般少而大，因此用户在 XFS 上不会受到太多的限制。

ZFS

ZFS（详情见 <http://en.wikipedia.org/wiki/ZFS>）是由 Sun 公司研发并在 2005 年推出的文件系统。这个名字是 Zettabyte File System 的缩写，因为它有能力存储 2^{58} ZB（大概是 10^{21} 字节）。

ZFS 主要被 Solaris 操作系统支持，不过 ZFS 拥有非常适合 HBase 应用场景的高级功能。它支持内置的压缩，可以替换 HBase 中以插件形式提供的压缩功能。

选择文件系统与选择操作系统一样都需要适合用户现有的基础设施。不经过测试和比较，仅靠单纯的数字，是很难做出选择的。如果用户要选择的话，我们建议选择比较新的文件系统，如 ext4 或 XFS，它们迟早会取代 ext3，并且比其他旧版本的文件系统更容易扩展。



我们不建议在一台服务器上安装不同的文件系统。内核可能需要分割缓冲区来支持不同的文件系统，进而影响性能，据报道这样做可能会对某些操作系统有破坏性的性能影响。如果你已经挂载了混合文件系统，请务必仔细测试这个问题。

3. Java

不得不提的是 Java！需要 Java 才能运行 HBase，Java 1.6 以及更高的版本才能很好地支持 HBase。最佳的选择就是使用 Oracle 公司推荐的版本（原来是 Sun 公司，后来 Oracle 收购了 Sun 公司），详情见链接 <http://www.java.com/download/>。

用户应该确保 Java 二进制文件可执行，并且可执行的类库目录中能找到这些文件。在命令行中输入 `java -version` 可以获取已安装的 Java 版本信息，同时可以确认安装是

否正常。例如，执行 `java -version` 可以得到正常输出 "1.6.0_22"。通常，用户使用最新的 Java 版本时会碰到一些意想不到的问题（例如，1.6.0_18 版本存在 JVM 随机崩溃的问题），因此用户最好尝试一个稍旧的版本。

如果 HBase 警告无法找到可执行的 Java 安装路径（见例 2.1）就需要在 `conf/hbase-env.sh` 文件中编辑 `JAVA_HOME` 这一行，设置正确的 Java 安装路径。

例 2.1 HBase 启动时无法找到可执行的 Java 安装路径，从而打印的错误信息

```
+=====+
|      Error: JAVA_HOME is not set and Java could not be found      |
+-----+
| Please download the latest Sun JDK from the Sun Java web site    |
|       > http://java.sun.com/javase/downloads/ <                  |
|                                                                     |
| HBase requires Java 1.6 or later.                                  |
| NOTE: This script will find Sun Java whether you install using the |
|       binary or the RPM based installer.                           |
+=====+
```



提供的脚本会在系统的默认类库存放目录寻找 Java，所以大部分情况下它都能自动找到系统安装的 Java。如果没有找到已安装的 Java 运行时环境，则极有可能是没有安装 Java 运行时环境。此时，就要从上述的下载链接下载 Java 安装程序，并阅读操作系统手册学习如何安装 Java。

4. Hadoop

目前 HBase 只能依赖特定的 Hadoop 版本，其中的主要原因之一是 HBase 与 Hadoop 之间的远程过程调用（Remote Procedure Call, RPC）API，RPC 协议是版本化的，并且需要调用方与被调用方相互匹配，细微的差异就可能导致通信失败。

当前 HBase 仅仅依赖 Hadoop 0.20.x 系列（<http://hadoop.apache.org/common/release.html>），而不能运行在 Hadoop 0.21.x 和 0.22.x 系列上。运行在不支持 sync 功能的 HDFS 上，HBase 可能会在灾难性场景中丢失数据。Hadoop 0.20.2 与 Hadoop 0.20.203.0 并不支持这个功能，只有分支 `branch-0.20-append` 支持该功能。^① 目前为止官方所有发布版本都不是从这个分支发展而来的，因此最好直接使用 `branch-0.20-append` 分支。访问 Hadoop How To Release（<http://wiki.apache.org/hadoop/HowToRelease>）以获得如何编译 Hadoop 工程的信息。^②

另一种选择是使用已经打了对应补丁的 Hadoop 工程，例如用户可以使用 Cloudera 的

① 查阅分支 `branch-0.20-append` 的 `CHANGES.txt`（<http://svn.apache.org/viewvc/hadoop/common/branches/branch-0.20-append/CHANGES.txt>）文件可以看到所有新增的补丁。

② 这本书出版后，信息发生更改是很正常的，有关详情和最新细节都可以在配置向导页面 <http://hbase.apache.org/book/configuration.html> 中查询，尤其是 <http://hbase.apache.org/book/hadoop.html> 这一节。

CDH3 版本 (<http://archire.cloudera.com/docs/>)。CDH 已经打入了 0.20-append 中增加持久化 sync 功能的补丁。更多细节见附录 D 中关于 Cloudera 的内容。

由于 HBase 依赖于 Hadoop，它要求 Hadoop 的 JAR 必须部署在 HBase 的 *lib* 目录下。默认依赖的 Hadoop 的 JAR 是从 Apache branch-0.20-append 这个分支编译出来的。关键在于 HBase 使用的 Hadoop 版本必须与底层 Hadoop 集群上使用的 Hadoop 版本一致，使用 Hadoop 集群正在运行的 JAR (*hadoop-xyz.jar*) 替换 HBase 的 *lib* 目录中依赖的 Hadoop 的 JAR 可以避免版本不匹配问题，此外必须确认集群中所有的节点都需要更新为一样的 JAR，否则版本不匹配问题会造成集群无法启动或假死现象。



HBase 类库中默认 Hadoop 的 JAR 只能在单机模式中使用。

另一种选择就是 Hadoop 集群使用 HBase 中默认 Hadoop 的 JAR 启动，但是这种方式没有经过广泛的测试，用户的实际使用情况可能不同。



HBase 能够运行在任意追加了安全功能的 Hadoop 0.20.x 系列上——例如 CDH3——只要按照以上的步骤替换 Hadoop 的 JAR 为附带安全版本的 HBase 就可以了。

5. SSH

如果用户需要通过脚本来管理 Hadoop 与 HBase 进程必须要安装 ssh 并运行 *sshd*。常用的并可以提供这些命令的软件包是 OpenSSH，详情见 <http://www.openssh.com/>。但是，首先请查看你的操作系统，它很可能已经提供了二进制程序的安装功能，且不需要重新编译。在 Ubuntu 工作站可以这样使用：

```
$ sudo apt-get install openssh-client
```

在服务器上还应该安装与之匹配的服务器软件包：

```
$ sudo apt-get install openssh-server
```

用户必须能够通过密码登录，并 ssh 跳转到所有节点，包括本地节点。ssh 需要有一个公共密钥——要么是用户已经在使用的（见 *home* 目录中的 *.ssh* 目录）或者新生成一个——每个服务器上都需要添加用户公共密钥，这就可以让脚本可以没有阻碍地访问任意一台远程服务器。



HBase 提供的 Shell 脚本需要通过 SSH 将命令发送到集群中的每个服务器中并执行，因此强烈建议不要使用简单的密码验证，而应该使用公共密钥认证！

当用户创建密钥对时，同时应该创建一个口令以保护私钥。每一个命令

的口令发都需要发送到远程服务器中，为了避免麻烦建议使用 `ssh-agent`，它是一个 SSH 助手，能够帮助用户只输入一次密码就可以连续处理请求。

理想的情况下，用户还可以使用内置的代理转发 (`agent forwarding`) 命令，它可以帮助用户从集群中的节点登录到其他远程服务器。

6. 域名服务

HBase 使用本地域名汇报 IP 地址。正向与反向 DNS (Domain Name Service, 域名服务) 均可以工作，用户可以通过以下命令验证正向 DNS 的正确性：

```
$ ping -c 1 $(hostname)
```

用户需要确保服务器使用了公共 IP 地址，而不是环路地址 127.0.0.1。出现这种情况的典型原因是 `/etc/hosts` 文件配置不正确，其中包含了计算机域名到环路地址的映射。

如果用户的服务器有多个接口，HBase 将使用主要接口解析域名。如果这些不够用，用户可以通过设置 `hbase.regionserver.dns.interface` (见 2.6 节如何配置参数的内容) 指出主接口，但这样做的前提是集群内的配置必须是一致的，且所有主机要有相同的网络接口配置。

另一种方法是设置 `hbase.regionserver.dns.nameserver` 以选择与系统默认配置不同的域名服务器。

7. 同步时间

集群中节点的时间必须是一致的，稍微有一点时间偏差是可以容忍的，但是偏差较多会产生一些奇怪的行为，仅仅一分钟的偏差就有可能使集群产生莫名其妙的行为。所以，用户需要在集群中运行 NTP (http://en.wikipedia.org/wiki/Network_Time_Protocol) 或同等功能的应用来同步集群的时间。

如果在运行正常的集群中读取数据时发生了一些奇怪的行为，请检查下集群的系统时间！

8. 文件句柄和进程限制

HBase 是数据库，它会同时使用很多文件。在 Unix 或其他类 Unix 系统中，默认的 `ulimit-n` 是 1024，但这个值这不够。任何大量的加载操作都会导致显而易见的 I/O 异常：

java.io.IOException: Too many open files。还有如下类似的异常信息：

```
2010-04-06 03:04:37,542 INFO org.apache.hadoop.hdfs.DFSClient: Exception
in createBlockOutputStream java.io.EOFException
2010-04-06 03:04:37,542 INFO org.apache.hadoop.hdfs.DFSClient: Abandoning
block blk_-6935524980745310745_1391901
```



所有错误信息都会被记录在日志文件中。详情见 12.5.2 节，该节介绍了如何分析这些日志内容。

用户需要改变文件描述符的数量上限，将上限设置为超过 10000 的数字。要清楚这个参数是运行 HBase 的操作系统参数，而不是 HBase 的配置。此外，一个常见的错误是管理员为特定的用户增加了文件描述符，但 HBase 却是用其他账户运行的。



用户可以大致估算所需的文件句柄数，如下所示：每个列族至少有一个存储文件，一个已加载的 region 可能有多达 5 或 6 个文件，且平均每个列族有 2 或 3 个存储文件。为了确定所需的文件句柄数，用列族数乘以每个 region 服务器中 region 的数量，例如，每个 region 有 3 个列族，每个 region 服务器有 100 个 region，不算打开的 JAR 文件、配置文件、CRC32 文件等，JVM 将打开 $3 \times 3 \times 100 = 900$ 个文件。运行 `lsof -p REGIONSERVER_PID` 能够看到准确的数字。

HBase 在它的日志第一行打印了 ulimit 信息，因此要确保打印的信息是正确的。^①12.5.2 一节会介绍怎样在日志中查找此类关键信息，这些信息可以帮助用户发现和解决 HBase 的安装问题。

用户还需要编辑 `/etc/sysctl.conf`，并调整 `fs.file-max` 的值。有关的详细信息请参考 Server Fault 的这篇文章 <http://serverfault.com/questions/165316/how-to-configure-linux-file-descriptor-limit-with-fs-file-max-and-ulimit/>。

例子：在 Ubuntu 上设置文件句柄

如果在 Ubuntu 下，你需要做以下更改：在文件 `/etc/security/limits.conf` 中添加这样一行：

```
hadoop -      nofile 32768
```

用户应该使用 hadoop 账号运行 Hadoop 和 HBase，如果用户使用单独的账号运行还需要设置两个参数，其中一个参数需要在每个账号中都设置。在文件 `/etc/pam.d/common-session` 的最后一行添加：

```
session required pam_limits.so
```

否则，`/etc/security/limits.conf` 中的变化不会生效。

另外切记不要忘记重新登录以便更改及时生效！

用户可以在配置文件 `/etc/security/limits.conf` 中设置 `nproc` 值以调整文件可以被引用

^① <http://www.cloudera.com/blog/2009/03/configuration-parameters-what-can-you-just-ignore/> 是篇来自 Aaron Kimball 的博文，名为 “What can you just ignore?”，它可以帮助用户在 Hadoop 集群中设置配置参数。

的进程数上限。数值设置较小会引起 `OutOfMemoryError` 异常，这会导致 Java 进程直接退出。文件句柄是非常关键的参数，用户需要在运行进程时着重确认当前账号下的此参数。

9. DataNode 处理线程数

HDFS 的 `DataNode` 会设置服务时可处理的文件数上限，这个参数叫做 `xcievers`。在加载前，用户必须确保拥有 Hadoop 的配置文件 `conf/hdfs-site.xml`，且设置 `xcievers` 参数至少为以下数值：

```
<property>
  <name>dfs.datanode.max.xcievers</name>
  <value>4096</value>
</property>
```



修改了配置后需要重启 HDFS。

不配置这个选项有可能导致一些奇怪的问题。用户最终会在数据节点的日志中发现 `xcievers` 使用超过限额的异常，但是这个异常是以块丢失的异常抛出给客户端的，相关的异常信息如下：

```
10/12/08 20:10:31 INFO hdfs.DFSCliet: Could not obtain block
blk_XXXXXXXXXXXXXXXXXXXXX_YYYYYYYY from any node: java.io. IOException:
No live nodes contain current block. Will get new block locations from
namenode and retry...
```

10. 交换区

用户为了避免运行时发生内存溢出，比较好的方式是给操作系统的进程预留足够的内存，并且 JVM 堆大小设置不要太大。一旦使用内存接近最大可用物理内存，操作系统会开始使用交换区（swap），通常是机器磁盘中独立的分区，此时内存会重新分配。

交换区——在工作站上或许是好事——但是在服务器上已经逐渐地被禁止了。因为一旦服务器开始使用交换区，整体的性能就会显著降低，用户甚至可能无法登录系统，因为远程访问（如 SSHD）在这个过程中会被挂起。

HBase 需要保证 CPU 周期，并且需要遵守一定的租约。例如，HBase 需要刷新 ZooKeeper 会话，一旦服务器发生交换，HBase 服务器就无法与 ZooKeeper 服务器交换信息，ZooKeeper 服务会认为这个会话已经超时并失效，即租约失效。这种情况会导致这些服务器上部署的 region 被重新部署到其他服务器中，如果集群遇到额外的压力也会引发此类问题。

更糟糕的是，服务器在交换过程中被唤醒，而此时 master 节点认为当前节点已经死掉

了，于是 region 服务器仿佛什么事情也没有发生过，并再次向 master 汇报信息，这时 region 服务器会收到 master 返回的 `YouAreDeadException` 异常，并认定自己需要进入死亡状态，并终止自身的进程。此外还有不少隐含的问题，例如，挂起状态的更新，关于这一点后面有详细讨论。以上足以说明这并不是件好事。

用户可以通过编辑配置文件 `/etc/sysctl.conf` 加入下面命令行来对 Linux 和类 Unix 系统服务器交换区的相关参数进行调整：

```
vm.swappiness=5
```

用户可以尝试将这个值设置为 0 或 5，以减少使用交换空间的概率。

一些激进的系统管理员已经完全关闭了交换区（见 Linux 上的 `swappoff`），主要原因是他们希望系统能够“撞墙似”的处理，而不必处理交换引发的问题。用户可以选择自己习惯的方案，但要时刻关注这个问题。

最终，重启服务器就可以使配置生效：

```
sysctl -p
```

但是，这个配置也许不能完全满足需求。这显然是在类 Unix 系统中才有的问题，用户在使用时最好要调整操作系统。

11. Windows

HBase 很少在 Windows 上进行测试，因此不建议将 HBase 的生产环境建在 Windows 上。如果用户坚持在 Windows 上运行 HBase，必须安装 Cygwin 环境（<http://cygwin.com/>），因为 Cygwin 为 Shell 脚本提供了一个类 Unix 环境。在 HBase 的主页上有 Windows 安装指南（<http://hbase.apache.org/cygwin.html>），里面有详细的解释。

2.3 HBase 使用的文件系统

HBase 最常使用的文件系统是 HDFS，但并不仅仅是 HDFS，因为 HBase 使用的文件系统是一个可插拔的架构，用户可以使用其他任何支持 Hadoop 接口的文件系统代替 HDFS。事实上，用户可以实现自己的文件系统——甚至基于另一个数据库。一切皆有可能！



我们现在讨论的不是操作系统使用的底层文件系统（见 2.2.3 节），而是指 HBase 直接依赖的存储文件系统。这些是更高层次功能和 API 的抽象定义，然后由 Hadoop 调用并存储数据，最终 Hadoop 会将数据存储到操作系统使用的磁盘文件系统。

HDFS 是生产中使用最广泛的且经过检验的文件系统。几乎所有的生产集群都使用

HDFS 作为底层存储层，它被证明是稳定可靠的系统，然而不使用 HDFS 可能会产生不可控的风险和一些后续的问题。

HDFS 如此受欢迎的主要原因是，它的机制包涵了冗余、容错性和可扩展性。无论选择哪个文件系统都应该提供类似的保障，因为 HBase 需要假定文件系统的数据存储是可靠的，并且 HBase 本身没有办法复制数据并维护自身存储文件的副本。因此较低层次的文件系统必须提供此功能。

通过设置 URI^① 模式，就可以选择不同的文件系统，URI 标识符中的 scheme（即第一个冒号之前的部分）标识了使用的磁盘。图 2-1 显示了在 Hadoop 文件系统与实际磁盘的底层操作系统的文件系统有哪些不同。

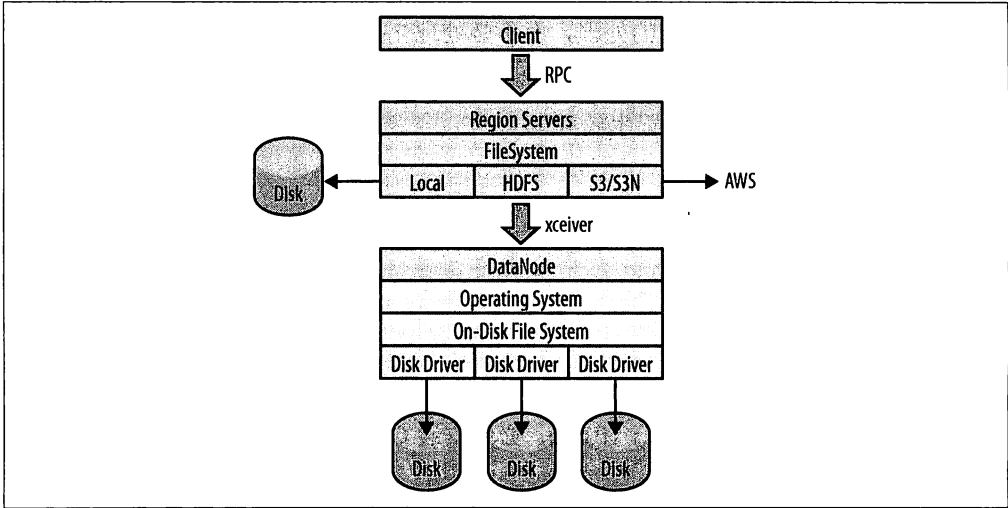


图 2-1 文件系统协议

用户可以使用 Hadoop 提供的文件系统：上图罗列了支持 Hadoop 接口的所有文件系统^②，用户可以对此先做一些尝试。对有经验的开发者来说，开发者也可以自行实现自己的文件系统。

2.3.1 本地模式

本地文件系统实际上完全绕过了 Hadoop，即不使用 HDFS 或任何其他集群。HBase 使用 FileSystem 类连接到文件系统实现，Hadoop 客户端加载并使用 Hadoop 提供的 ChecksumFileSystem 类直接操作本地磁盘路径来存储所有数据。

这种方法的特点是，HBase 直接使用本地文件系统，而不需要关注使用的是一个远程的分布式文件系统还是一个托管集群。HBase 单机模式会直接使用上述特性。用户设置以下参数就可以直接使用本地文件系统了：

① 见维基百科的“Uniform Resource Identifier”(http://en.wikipedia.org/wiki/Uniform_Resource_Identifier)。

② 完整列表已经在 Tom White 的文章“Get to Know Hadoop Filesystems”中介绍过了。


```
file:///<path>
```

形式上类似于 Web 浏览器的 URI，使用 file: 直接定位本地文件。

2.3.2 HDFS

Hadoop 分布式文件系统（Hadoop Distributed File System，HDFS）是默认的文件系统，它部署在一个完全分布式的集群中。HBase 选择 HDFS 作为文件系统，是因为 HDFS 具有所有必需的功能。如之前讨论的与 MapReduce 的耦合，能够充分利用其并行流式的处理能力。并且拥有较好的扩展性、系统可靠性和自动冗余功能，是理想可靠的文件存储系统。HBase 增加了随机存取层，是 HDFS 缺失的部分，是对 Hadoop 的理想补充。另外利用 MapReduce 的并行能力可以执行批量导入数据的功能，最大限度利用磁盘带宽。

使用以下 URI 可以访问 HDFS：

```
hdfs://<namenode>:<port>/<path>
```

2.3.3 S3

Amazon S3（Simple Storage Service）^①是一个存储系统，并主要与动态服务器结合使用，S3 一般运行在 Amazon 的另一个互补服务 EC2（Elastic Compute Cloud）^②上。

S3 可以不依赖 EC2 直接使用，S3 输入和导出数据消耗的带宽成本是非常高的。EC2 和 S3 之间的数据迁移是免费的，因此这是一个可行方案。关于如何启动基于 EC2 的集群在 2.7.2 节有介绍。

S3 文件系统实现了 Hadoop 支持的两种不同模式：原生（raw 或 native）模式和块（block）模式。原生模式使用 s3n:URI scheme，并直接将数据写入到 S3，与本地文件系统类似。与本地磁盘类似，用户可以在桶中看到所有的文件。

S3:是基于块的模式，可用来克服 S3 最大文件为 5 GB 的限制。情况已经发生变化，这让选择哪种模式变得更困难——或者更容易：如果用户文件不超过 5 GB 就可以选择 s3n:。

S3 的块模式模拟了 HDFS 文件系统，它使浏览桶的内容变得非常困难，因为只有内部的文件块是可见的，而 HBase 存储文件的文件块散落在集群的各个地方。可以使用如下 URI 配置选择文件系统：

```
s3://<bucket-name>  
s3n://<bucket-name>
```

① 点击链接 http://en.wikipedia.org/wiki/Amazon_S3，可以查看更多有关 Amazon S3 的背景信息。

② 见维基百科的 EC2。

2.3.4 其他文件系统

其他文件系统，如 CloudStore（过去称为 Kosmos filesystem，简称 KFS，同名的 URI scheme 将在下一段末进行描述）。KFS 是一个用 C++ 语言编写的、开源的、分布式的、高性能的文件系统，功能上与 HDFS 类似。在 CloudStore 网站可以查找到更多有关它的信息（<http://kosmosfs.sourceforge.net/>）。

KFS 可以在 Solaris 和 Linux 上使用，最初 KFS 是由 Kosmix 开发，并在 2007 年发布为开源应用的。选择 CloudStore 作为 HBase 的文件系统需要使用以下 URI 格式：

```
kfs:///<path>
```

2.4 安装选项

用户一旦决定了操作系统相关的基本参数，就必须马上安装 HBase 到服务器中。用户有多种选择，我们接下来将详细讨论。在附录 D 中可查看其他选项。

2.4.1 Apache 二进制发布包

大多数 Apache 项目安装过程中需要下载一个发布版本，通常是一个包含所有必需文件的归档文件。有些项目将归档文件分为二进制文件和源文件两种不同的类型——前者拥有所有运行项目所需的文件，后者则包含了编译项目所需的所有文件。HBase 使用一个包含了二进制文件和源文件的独立软件包。更多关于 HBase 版本的信息可以查看 Release Notes^① 页面。另一个有趣的页面是 Change Log^②，其中列出了增加、修改的功能和修复的缺陷。

用户可以从 Apache HBase 的发布网站（<http://www.apache.org/dyn/closer.cgi/hbase/>）下载 HBase 的最新版本，并将内容解压到一个合适的目录中，如 `/usr/local` 或 `/opt`，像这样：

```
$ cd /usr/local
$ tar -zxvf hbase-x.y.z.tar.gz
```

解压所有文件后，用户可以了解到项目目录中都包含哪些文件，解压归档文件后得到的目录结构如下：

```
$ ls -lr
-rw-r--r--  1 larsgeorge  staff  192809 Feb 15 01:54 CHANGES.txt
-rw-r--r--  1 larsgeorge  staff   11358 Feb  9 01:23 LICENSE.txt
-rw-r--r--  1 larsgeorge  staff    293 Feb  9 01:23 NOTICE.txt
```

① 见链接 <https://issues.apache.org/jira/browse/HBASE?report=com.atlassian.jira.plugin.system.project:changelog-panel>。

② 见链接 <https://issues.apache.org/jira/browse/HBASE?report=com.atlassian.jira.plugin.system.project:changelog-panel#selectedTab=com.atlassian.jira.plugin.system.project%3Achangelog-panel>。

```

-rw-r--r--  1 larsgeorge staff    1358 Feb  9 01:23 README.txt
drwxr-xr-x 23 larsgeorge staff     782 Feb  9 01:23 bin
drwxr-xr-x  7 larsgeorge staff     238 Feb  9 01:23 conf
drwxr-xr-x 64 larsgeorge staff    2176 Feb 15 01:56 docs
-rwxr-xr-x  1 larsgeorge staff 905762 Feb 15 01:56 hbase-0.90.1-tests.jar
-rwxr-xr-x  1 larsgeorge staff 2242043 Feb 15 01:56 hbase-0.90.1.jar
drwxr-xr-x  5 larsgeorge staff     170 Feb 15 01:55 hbase-webapps
drwxr-xr-x 32 larsgeorge staff    1088 Mar  3 12:07 lib
-rw-r--r--  1 larsgeorge staff 29669 Feb 15 01:28 pom.xml
drwxr-xr-x  9 larsgeorge staff     306 Feb  9 01:23 src

```

根目录包含一些文本文件，说明文档以及许可条款（*LICENSE.txt* 和 *NOTICE.txt*）和其他一些生成信息（*README.txt* 文件）。*CHANGES.txt* 是变更日志的静态快照页面，它包含了当前下载版本中所有的变更记录。

根目录中还能发现 Java 的归档文件，又名 JAR 文件，包含已编译的 Java 代码以及所有其他必要的资源。JAR 文件有两个，一个只包括名称和版本号，另一个还包含 *tests* 后缀，此文件包含 HBase 的测试用例，开发人员使用其中的单元测试验证一个版本是完全可用并且没有退化。

最后一个文件是 *pom.xml*，这是 Maven 编译工程时依赖的文件，详情见 2.4.2 节。

根目录下的其他目录如下所示。

bin

bin，即二进制文件，此目录包含了 HBase 提供的所有脚本，可以完成启动和停止，运行独立的守护进程^①或启动额外的 *master* 节点等功能，请在 2.8.1 节中查阅如何使用它们。

conf

配置目录中包含了定义 HBase 配置的文件，在 2.6 节中非常详细地解释了其中包含的文件。

docs

这个目录包含了 HBase 工程网页的副本，以及工具、API 和项目自身的文档信息。打开浏览器把 *docs/index.html* 文件拖入浏览器，双击该文件使用文件→打开（或类似）菜单就可以使用文档。

hbase-webapps

HBase 提供了 Java 实现的 Web 接口，其中所用到的文件都在这个目录下。用户在布置 HBase 到生产环境中或使用 HBase 时，很少有机会接触到这个目录中的文件。

① 进程以守护模式在后台运行，不会因为执行的任务终止而停止运行。

lib

Java 应用程序依赖很多类库，这些类库包含了实际的执行程序，并且这些文件都放置在 *lib* 目录里。

logs

HBase 进程通常以守护进程的形式运行，即在操作系统的后台运行，在生命周期内它会将一些状态、程度、异常等信息打印到日志文件中，12.5.2 节介绍了相关内容。



首次启动时，*logs* 目录可能不存在，但 HBase 会通过日志框架自动创建日志文件夹。

src

如果用户计划编译二进制文件（如何做请查阅 2.4.2 节），或加入 HBase 的开发团队，用户将需要源文件，源文件包含了所有的发布信息。

如果你已经解压了一个发布版，现在就可以跳到 2.5 节去研讨如何启动 HBase 了。

2.4.2 编译源码

HBase 依赖 Maven 编译工程，因此用户需要安装 Maven 以及完整的 Java 开发工具包（Java Development Kit, JDK）——不仅仅是在 Java 运行时才会被用到，关于这里的细节在 2.1 节中有介绍。



本节仅仅介绍了如何通过源码编译 HBase 工程，这对需要打补丁或增加功能来说才是需要掌握的技能。

一旦确认以上两者都已经安装，就可以通过以下命令来编译二进制包：

```
$ mvn assembly:assembly
```

需要注意的是，HBase 的运行测试会超过一个小时。如果用户相信代码是没问题的，可以节省这个时间，直接通过以下命令跳过测试阶段：

```
$ mvn -DskipTests assembly:assembly
```

这个编译过程大概需要几分钟，如果没有跳过测试阶段可能需要几十分钟。最后，HBase 的 *home* 目录里会创建 *target* 目录。一旦编译完成并打印出提示信息 Build Successful，在 *target* 目录里，你将能找到编译和打包好的 *tarball* 格式归档文件。此时，你可以跳转回 2.4.1 节，按照步骤将其安装在自己的私有服务器中。

2.5 运行模式

HBase 有两个运行模式：单机模式和分布式模式。2.1 节介绍过如何启动单机版 HBase，如果用户想启动分布式模式只需要编辑 `conf` 目录中的配置文件即可。

无论启动什么模式，都必须编辑 `conf/hbase-env.sh` 文件以指定运行 HBase 的 `java` 安装目录，只需要设置 `JAVA_HOME` 参数即可，让该参数指向 `Java` 安装的根目录。在这个文件中，用户还能够设置 HBase 的环境变量，如堆大小和其他一些 JVM 参数，以及日志文件的输出目录等。

2.5.1 单机模式

单机模式是默认模式，具体描述见 2.1 节。在单机模式中，HBase 并不使用 HDFS——仅使用本地文件系统——ZooKeeper 程序与 HBase 程序运行在同一个 JVM 进程中，ZooKeeper 绑定到客户端的常用端口上，以便客户端可以与 HBase 进行通信。

2.5.2 分布式模式

分布式模式可以进一步细分成伪分布式模式（pseudo distributed）——所有守护进程都运行在单个节点上，以及完全分布式模式（fully-distributed）——进程运行在物理服务器集群中。^①

分布式模式依赖 Hadoop 分布式文件系统实例（Hadoop Distributed File System, HDFS），详情见链接 http://hadoop.apache.org/common/docs/current/api/overview-summary.html#overview_description，从中可以查阅如何建立一个 HDFS 集群。进行布置前，一定要确保有一个合适的、正在工作的 HDFS 集群。

下面我们介绍如何启动不同的分布式模式，关于完全分布式模式与伪分布式模式的验证和安装信息在 2.8.1 节中有详细描述。经过验证的部署脚本适合两种部署类型（伪分布式模式与完全分布式模式）。

1. 伪分布式模式

伪分布式模式是在一台主机上运行所有进程的模式。此配置仅仅是协助 HBase 用于测试和原型，不要在生产环境中使用此配置，也不要用它配置做 HBase 的性能比较。

如果你确认 HDFS 已经启动，就请编辑 `conf/hbase-site.xml` 文件，在这个文件中可以添加自定义的本地参数，并覆盖 HBase 的默认配置（在附录 A 中可查阅完整属性列表，以及 2.6.1 节中的“HDFS 相关配置”）。设置 `hbase.rootdir` 属性可以指定 HDFS 实例。

^① 伪分布式与完全分布式的概念来源于 Hadoop。

例如，添加以下配置属性到 *hbase-site.xml* 文件是指：HBase 使用 HDFS 的 */hbase* 目录作为根目录，HDFS 的服务端口是本地端口 9000，并且数据只保留一个副本（建议伪分布式模式这样做）：

```
<configuration>
...
<property>
  <name>hbase.rootdir</name>
  <value>hdfs://localhost:9000/hbase</value>
</property>
<property>
  <name>dfs.replication</name>
  <value>1</value>
</property>
...
</configuration>
```



在这个例子中，服务绑定在本机中，这意味着远程客户端无法连接。如果用户想从远程位置连接，其需要做相应的修改。

如果用户想尝试伪分布式模式，那么现在就可以跳转到 2.8.1 节查看如何启动和验证。第 12 章中有关于如何在伪分布式模式中启动 master 和 region 服务器的相关信息。

2. 完全分布式模式

如果用户需要在多台主机中运行完全分布式操作，需要进行以下配置。在 *hbase-site.xml* 中添加 *hbase.cluster.distributed* 属性并设置为 *true*，添加 *hbase.rootdir* 属性并设置 HDFS NameNode 的访问地址，这将决定 HBase 的数据会写到哪里，例如，NameNode 运行在主机名为 *namenode.foo.com* 的服务器上，并以 9000 为服务端口，HBase 在 HDFS 上使用的根目录为 */hbase*，可以做如下配置：

```
<configuration>
...
<property>
  <name>hbase.rootdir</name>
  <value>hdfs://namenode.foo.com:9000/hbase</value>
</property>
<property>
  <name>hbase.cluster.distributed</name>
  <value>true</value>
</property>
...
</configuration>
```

配置 region 服务器。此外，完全分布式模式需要修改 *conf/regionservers* 文件，该文件列出

了所有运行 HRegionServer 守护进程的主机，每个主机独立占用一行（类似于 Hadoop 中的 *slaves* 文件）。HBase 集群启动和关闭时会按照该文件中罗列的主机逐一执行。

ZooKeeper 安装。分布式的 HBase 依赖于 ZooKeeper 集群。所有的节点和客户端都必须能够正常访问 ZooKeeper。HBase 默认管理一个单点的 ZooKeeper 集群（ZooKeeper 能够以一个单独的节点启动），用户通过启动和关闭脚本就可以把 ZooKeeper 当做 HBase 的一部分来启动和关闭进程。用户也可以不依赖于 HBase 管理 ZooKeeper 集群，只需为 HBase 指出需要使用的集群即可。在 *conf/hbase-env.sh* 中设置 `HBASE_MANAGES_ZK` 变量为 `true`，就可以将 ZooKeeper 作为 HBase 的一部分管理启动（这个参数默认为 `true`）。

当用户需要通过 HBase 管理 ZooKeeper 时，ZooKeeper 可以直接使用本地 *zoo.cfg* 文件作为依赖的配置文件，或者直接使用 *conf/hbase-site.xml* 中的 ZooKeeper 配置。ZooKeeper 的相关配置可以在 *hbase-site.xml* 中通过 XML 格式设置，属性以 `hbase.zookeeper.property` 为前缀，例如，`clientPort` 可以设置为 `hbase.zookeeper.property.clientPort`。在 HBase 中这些参数都有默认值，包括 ZooKeeper 的配置，详情见附件 A，查询前缀为 `hbase.zookeeper.property` 的属性。^①

zoo.cfg 与 hbase-site.xml 的对比

将 *hbase-site.xml* 与 *zoo.cfg* 结合起来配置 ZooKeeper 参数很容易引起混乱。对于初学者来说，如果在 classpath 中配置 *zoo.cfg*（这意味着 Java 进程可以找到这个配置），这样就可以覆盖 *hbase-site.xml* 中除了以 `hbase.zookeeper.property` 为前缀的配置。

有一些 ZooKeeper 的客户端配置无法在 *zoo.cfg* 中读取，因此必须要在 *hbase-site.xml* 中设置，例如，客户端会话超时配置——`zookeeper.session.timeout`。其他更详细的描述如下表所示。

属性	zoo.cfg + hbase-site.xml	仅 hbase-site.xml
<code>hbase.zookeeper.quorum</code>	<i>zoo.cfg</i> 文件中格式为 <i>server.n</i> 的若干行配置会直接覆盖 <i>hbase-site.xml</i> 中的配置	仅使用
<code>hbase.zookeeper.property.*</code>	<i>zoo.cfg</i> 中的配置会覆盖 <i>hbase-site.xml</i> 中的配置	仅使用
<code>zookeeper.*</code>	仅在 <i>hbase-site.xml</i> 中才有	仅在 <i>hbase-site.xml</i> 中才有

为了避免部署中出现混乱，不推荐使用 *zoo.cfg*，文件推荐只使用 *hbase-site.xml* 文件。尤其在完全分布式模式中，用户完全没有必要将 ZooKeeper 的特殊配置文件复制到其他 HBase 服务器中。

① ZooKeeper 的完整配置列表可在 *zoo.cfg* 中见到。HBase 默认不带这个文件，所以需要用户自己去 ZooKeeper 的网站浏览 *conf* 目录并下载。

如果用户使用 *hbase-site.xml* 配置 ZooKeeper, 首先要设置的是 `hbase.zookeeper.quorum` 属性, 通过这个属性可以设置可用服务器列表。这个属性默认是本地的单一成员, 这个默认属性不适合完全分布式模式的 HBase(如果不设置该属性客户端就无法进行远程访问)。

应该启动几台 ZooKeeper?

用户可以运行一台只包括一个节点的 ZooKeeper, 但是在生产环境中, 推荐使用 3 台、5 台或 7 台; 数量越多集群容灾能力越强。但是最好运行奇数个节点, 因为运行偶数个节点不容易让服务器之间达成共识——ZooKeeper 集群需要一个绝大多数的投票, 无论 3 台或 4 台服务器都需要 3 个以上节点的投票, 使用奇数的目的是只有在两台服务器都失败的情况下才不可用而使用偶数在一台服务器失败的情况下就会不可用。

给每个 ZooKeeper 服务器大约 1 GB 的内存和专用磁盘(如果可能的话, 专用磁盘有利于确保 ZooKeeper 的性能)。在负载非常高的集群上, ZooKeeper 服务器应该独立于 RegionServer、DataNode 和 TaskTracker。

例如, 如果需要通过 HBase 在节点 `rs{1, 2, 3, 4, 5}.foo.com` 上管理 ZooKeeper, 并绑定客户端服务端端口 2222(默认是 2181), 用户就需要在 `conf/hbase-env.sh` 中将 `HBASE_MANAGE_ZK` 设置为 `true`, 并且需要编辑 `conf/hbase-site.xml` 文件和设置 `hbase.zookeeper.property.clientPort` 和 `hbase.zookeeper.quorum`。此外用户还应该设置 `hbase.zookeeper.property.dataDir` 以存储 ZooKeeper 的元数据, 如果不设置这个属性, 默认将是 `/tmp`, 系统重启后会自动清理这个目录。下面的例子描述了 ZooKeeper 设置该属性为 `/var/zookeeper`。

```
<configuration>
...
<property>
  <name>hbase.zookeeper.property.clientPort</name>
  <value>2222</value>
</property>
<property>
  <name>hbase.zookeeper.quorum</name>
  <value>rs1.foo.com,rs2.foo.com,rs3.foo.com,rs4.foo.com,rs5.foo.com</value>
</property>
<property>
  <name>hbase.zookeeper.property.dataDir</name>
  <value>/var/zookeeper</value>
</property>
...
</configuration>
```


使用已有 ZooKeeper 集群。HBase 采用已有的 ZooKeeper 集群，因此不能依赖 HBase 来管理 ZooKeeper，用户需要在 `conf/hbase-env.sh` 中将 `HBASE_MANAGES_ZK` 属性设置为 `false`。

```
...
# Tell HBase whether it should manage it's own instance of Zookeeper or not.
export HBASE_MANAGES_ZK=false
```

下一步需要在 `hbase-site.xml` 中设置 ZooKeeper 的连接地址与客户端端口号，或将在 `zoo.cfg` 中配置相关信息，并在 HBase 的 `CLASSPATH` 中添加 `zoo.cfg` 路径。HBase 启动时会读取 `zoo.cfg` 中的配置，并覆盖 `hbase-site.xml` 中的配置。

使用 HBase 管理 ZooKeeper，ZooKeeper 会作为 HBase 的一部分随着启动/关闭脚本进行启动/关闭。如果需要独立运行 ZooKeeper，不依赖 HBase 的启动与关闭，需要执行以下命令：

```
${HBASE_HOME}/bin/hbase-daemons.sh {start,stop} zookeeper
```

采取这样的方式可以让 ZooKeeper 与 HBase 脱离关系，只有在将 `HBASE_MANAGES_ZK` 设置为 `false` 后，ZooKeeper 才不会因 HBase 的关闭而关闭。

关于独立运行的 ZooKeeper 集群的相关信息，请参考 ZooKeeper 入门指导的相关内容（<http://hadoop.apache.org/zookeeper/docs/current/zookeeperStarted.html>），此外还可以参考 ZooKeeper 的维基百科（<http://wiki.apache.org/hadoop/ZooKeeper/FAQ#A7>）或 ZooKeeper 的相关文档（http://zookeeper.apache.org/doc/r3.3.3/zookeeperAdmin.html#sc_zkMultiserverSetup）。

2.6 配置

通过学习如何选择文件系统、运行模式、调整操作系统，我们已经完全掌握了这种基本操作，现在让我们看看如何配置 HBase。HBase 的配置类似于 Hadoop 的配置参数，配置文件放在 `conf` 目录下。这些文件都是简单的文本文件，有些是包括一系列属性的 XML 文件，另外一些是纯文本文件，且其中每行都是一个配置项。



关于更多如何修改配置文件的信息参见 2.6 节。

一个名为 `conf/hbase-env.sh` 的文件中包括 HBase 启动时要使用到的环境变量，这些配置经常被脚本用于启动和关闭集群（详情见 5.2.4 节）。用户可能还需要在 XML 文件^① `conf/hbase-site.xml` 中添加一些配置项，这个文件中的配置会覆盖 HBase 的默认配置，例如，用户可以在其中设

① 编辑 XML 时要谨慎，确保所有的元素都已经关闭，用户可以通过 `xmllint` 或类似的工具进行检查，以确保编辑会话后 XML 仍保持一个良好的格式。

置可用文件系统和 ZooKeeper 可用地址。

用户以分布式模式运行 HBase 时, 首先需要编辑 HBase 的配置文件, 然后复制 *conf* 目录到集群的其他节点中, 但 HBase 不会自动完成这些工作。



同步集群的配置有很多种方式, 最简单的方法是使用 *rsync* 工具。同时还有更多非常精巧的同步配置方式, 用户可以参考 2.7 节。

2.6.1 hbase-site.xml 与 hbase-default.xml

在 Hadoop 中, 如果用户需要增加 HDFS 的特定配置就要添加到 *hdfs-site.xml* 文件中。与此类似, 在 HBase 中, 用户需要增加配置信息就需要将配置添加到 *conf/hbase-site.xml* 文件中。配置参数的全部列表参考附件 A, 或直接查看 HBase 目录 *src/main/resources* 中的源文件 *hbase-default.xml*, *doc* 目录中也有配置参数信息的 HTML 文档。



并非所有的配置信息都罗列在了 *hbase-default.xml* 中。配置中有些参数并不常用并且只在源码中存在; 因此, 唯一的办法是通过阅读源码来查找这些配置参数的作用。

进程启动后, 服务器会先读取 *hbase-default.xml* 文件, 然后读取 *hbase-site.xml* 文件, *hbase-site.xml* 的内容会覆盖 *hbase-default.xml* 中的内容。

修改 *site* 文件后必须要重启进程才能得到最新的配置。

HDFS 相关配置

如果用户需要改动 Hadoop 集群中 HDFS 的相关配置, 也就是说用户修改了 HDFS 客户端相关配置而不是服务器端相关配置, HBase 中需要经过以下操作才能使配置生效。

- 在 *hbase-env.sh* 中, 将 *HADOOP_CONF_DIR* 配置到 *HBASE_CLASSPATH* 中。
- 复制一份 *hdfs-site.xml* (或 *hadoop-site.xml*) 文件到 */\${HBASE_HOME}/conf* 中, 或在 */\${HBASE_HOME}/conf* 中设置指向该文件的符号连接。
- 直接把这些配置添加到 *hbase-site.xml*。

例如, HDFS 客户端需要使用 *dfs.replication* 这个参数, 希望能够设置复制因子为 5, HBase 默认使用的是 3, 只有按照以上的步骤才能让 HDFS 的配置在 HBase 中可用。

Hadoop 配置文件在 HBase 中的使用优先级最低, 换句话说, 在 HBase 中的配置与 Hadoop 配置属性有重复的情况下, 无论是 *default* 还是 *site* 文件, HBase 配置的优先级都会高于 Hadoop 配置的优先级。这意味着用户可以使用 HBase 配置文件的参数覆盖 Hadoop 的参数。

2.6.2 hbase-env.sh

HBase 的环境变量等信息需要在这个文件中设置，例如，HBase 守护进程的 JVM 启动参数：Java 堆大小和垃圾回收策略等。在这个文件中还可以设置 HBase 配置文件的目录、日志目录、SSH 选项、进程 *pid* 文件的目录等。用户最好打开 *conf/hbase-env.sh* 文件，并仔细阅读其中内容，每个配置在文件中均有注释。在 HBase 守护进程启动前可以在这里设置自己需要的环境变量以便 HBase 读取。

改变配置后需要重启 HBase 才能生效。^①

2.6.3 regionserver

这个文件罗列了所有 region 服务器的主机名，它是纯文本文件，文件中的每一行都是主机名。HBase 的运维脚本会依次迭代访问每一行来启动所有 region 服务器进程。



如果用户使用了较早的 0.20.x 系列版本，里面有 *masters* 文件，但是之后的版本中已经将这个文件移除了，并且不再需要这个文件了。现在每个 master 在启动的时候都会注册到 ZooKeeper 中，因此 master 的地址可以在 ZooKeeper 中动态获取。

2.6.4 log4j.properties

修改这个文件中的参数可以改变 HBase 的日志级别，改变后重启 HBase，新配置才能生效，此外还可以通过 HBase 的 UI 界面来更改特定守护进程的日志级别，详情见 12.4 节。12.5.2 节可以帮助用户学习如何通过分析日志来查找和解决问题。

2.6.5 配置示例

下面示例是一个 10 个节点的集群。节点的名字分别是 *master.foo.com*，*host1.foo.com* 到 *host9.foo.com*。HBase master 与 HDFS NameNode 都运行在 *master.foo.com* 上，region 服务器运行在 *host1.foo.com* 到 *host9.foo.com* 上。3 个节点的 ZooKeeper 分别运行在 *zk1.foo.com*、*zk2.foo.com* 和 *zk3.foo.com* 中。ZooKeeper 的元数据目录设置为 */var/zookeeper*。主要的配置文件——*hbase-site.xml*、*regionservers*、*hbase-env.sh*——都可以在 HBase 的 *conf* 目录下找到，配置如下所示。

1. hbase-site.xml

该文件包括定义了启动 HBase 集群的基本配置信息。

^① 编辑完配置后，一定要重启服务器，但这种工作模式随着时间的推移会改变。

```

<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>hbase.zookeeper.quorum</name>
    <value>zk1.foo.com,zk2.foo.com,zk3.foo.com</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>/var/zookeeper</value>
  </property>
  <property>
    <name>hbase.rootdir</name>
    <value>hdfs://master.foo.com:9000/hbase</value>
  </property>
  <property>
    <name>hbase.cluster.distributed</name>
    <value>true</value>
  </property>
</configuration>

```

2. regionservers

这个文件中记录了所有 **region** 服务器的主机列表。在我们的例子中，除了运行 **HBase Master** 和 **HDFS NameNode** 的节点 `master.foo.com` 之外，所有节点都启动了 **region** 服务器进程。

```

host1.foo.com
host2.foo.com
host3.foo.com
host4.foo.com
host5.foo.com
host6.foo.com
host7.foo.com
host8.foo.com
host9.foo.com

```

3. hbase-env.sh

修改 `hbase-env.sh` 文件内的配置只需要修改内容中的默认信息即可。下面是对 `hbase-env.sh` 中默认内容做的修改，我们将 **HBase** 的堆设置为 4 GB，以替代默认的 1 GB。

```

...
# export HBASE_HEAPSIZE=1000
export HBASE_HEAPSIZE=4096
...

```

一旦用户编辑完配置文件,就需要将文件同步到集群的所有服务器上。用户可以在 Unix 或类 Unix 平台中使用 *rsync* 完成这项工作,编辑完配置文件后通过 *rsync* 就可以将 *conf* 目录同步到集群其他节点中,详情见 2.7 节。



2.6 节介绍了用户需要扩展集群时所需修改的配置。

2.6.6 客户端配置

HBase Master 能够在物理机器上自由移动(详情见 12.1.3 节的“添加本地冗余的 master”),客户端启动时通过 ZooKeeper 来获取关键信息(见 8.5 节),因此客户端必须在 *hbase-site.xml* 文件中配置 ZooKeeper 的连接地址,同时要将 *hbase-site.xml* 配置到启动 Java 进程的 CLASSPATH 中。



用户还可以在代码中设置关键配置属性 *hbase.zookeeper.quorum*。这种做法可以保证客户端不依赖额外的配置文件。详情见 3.2.1 节。

如果用户通过 IDE 运行 HBase 客户端,必须保证 *conf* 目录已经在 classpath 中,这样才能保证客户端代码找到配置文件。

HBase 的 Java 客户端需要在 CLASSPATH 中指出依赖的 JAR 文件: *hbase*、*hadoop-core*、*zookeeper*、*log4j*、*commons-logging* 和 *commons-lang*。这些 JAR 文件都以特定发布版本号作为后缀。理想情况下,用户只能使用 HBase 提供的 JAR,而不能随意使用其他版本,因为即使轻微的版本变化都有可能客户端远程访问 HBase 集群时出现问题。

客户端使用的 *hbase-site.xml* 配置的例子如下:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>hbase.zookeeper.quorum</name>
    <value>zk1.foo.com,zk2.foo.com,zk3.foo.com</value>
  </property>
</configuration>
```

2.7 部署

一旦用户配置好 HBase 之后,接下来就应该考虑在集群上部署 HBase。由于 Hadoop 和 HBase 是 Java 语言编写的,尽管有少量的特殊要求,但还是有许多方法可以进行集群部署。最简单的就是在服务器间复制所有文件,因为这样可以使服务器共享相

同的配置文件。有很多关于部署的做法可以尝试，首先必须确认所有建议的配置和在 2.2 节中讨论过的调整已经被应用，或者说在用户添加新服务器时需要同时应用这些配置。

2.7.1 基于脚本

与下面列出的更先进的方式相比，基于脚本的方法似乎已经过时了。基于脚本模式比较适合小型甚至中等规模的集群。与其说是集群的规模不如说是运维人员的数量，在一个更大的操作组中，管理员不得不重复部署，而不是通过脚本来更新集群。

脚本利用配置文件 *regionserver*s 作为集群服务器列表。例 2.2 展示了一个非常简单的脚本，这个脚本可以实现从 master 节点复制 HBase 发布目录到其他 slave 节点的功能。

例 2.2 示例脚本在 HBase 集群中复制文件

```
#!/bin/bash
# Rsyncs HBase files across all slaves. Must run on master. Assumes
# all files are located in /usr/local

if [ "$#" != "2" ];then
    echo "usage: $(basename $0)<dir-name> <ln-name>"
    echo "    example: $(basename $0)hbase-0.1 hbase"
    exit 1
fi

SRC_PATH="/usr/local/$1/conf/regionserver"
for srv in $(cat $SRC_PATH);do
    echo "Sending command to $srv...";
    rsync -vaz --exclude='logs/*' /usr/local/$1 $srv:/usr/local/
    ssh $srv "rm -fR /usr/local/$2;ln -s /usr/local/$1 /usr/local/$2"
done

echo "done."
```

例 2.3 展示了另一个简单的脚本，该脚本用于从 master 节点向其他所有 slave 节点复制 HBase 配置文件。在 master 服务器上编辑过配置文件后，通过这个脚本可以将配置文件同步到所有 region 服务器中。

例 2.3 示例脚本在 HBase 集群中复制文件

```
#!/bin/bash
# Rsync's HBase config files across all region servers. Must run on master.

for srv in $(cat /usr/local/hbase/conf/regionserver);do
    echo "Sending command to $srv...";
    rsync -vaz --delete --exclude='logs/*' /usr/local/hadoop/ $srv:/usr/ local/hadoop/
    rsync -vaz --delete --exclude='logs/*' /usr/local/hbase/ $srv:/usr/ local/hbase/
```

```
done
```

```
echo "done."
```

这个脚本与第一个脚本类似,都使用了 *rsync*,不同的是增加了 *--delete* 选项以确保 *region* 服务器不会保留任何旧文件,但是旧文件的副本会保留在原始服务器中。

很显然,还有很多方法可以做到这一点,上述的例子仅仅是为了方便阅读。用户需要管理员的协助,一起建立一套机制来保证文件的同步。许多初学者常常碰到的问题都是由于集群内的配置不一致引起的,此外,更新配置后一定要重启才能使配置生效。如果想要不停机就使更新的配置生效,用户可以查阅 12.1.3 节。

2.7.2 Apache Whirr

越来越多的用户希望能够在动态环境中运行 HBase 集群,如公开云服务平台 Amazon 的 EC2 或 Rackspace 云服务,或者是使用类似于 Eucalyptus 的开源工具的私有服务器。

这种方式的优点是能够快速地从集群中获取负载并加以分析汇总,一旦结果检索完毕,立即关闭集群,或者集群资源还可以被其他服务所利用。由于这并不是普通的程序,其需要为很多 API 提供动态集群架构,其工作量可能很大,所以有必要抽象出一个集群管理层。一旦集群部署好后,用户就可以像在本地静态集群上一样执行 MapReduce 作业。以上就是 Whirr 的作用。

Whirr (详情见 <http://incubator.apache.org/whirr/>^①) 支持各种公有和私有云的 API,并提供一个拥有一定范围服务的集群。其中的功能之一就是需要支持 HBase,并能够快速地在动态环境中部署完整的 HBase 集群。

用户可以通过上述地址下载 Whirr 的最新版本,并能够在 *recipes* 目录中找到预定义的配置文件,可以直接使用并动态部署集群。

Whirr 的基本原理是使用操作系统提供的简单机器镜像和 SSH 访问。其他步骤则由 Whirr 针对需求执行,例如,按 Hadoop 或 HBase 服务的方式处理。每个远程服务器上的服务都会执行一些必要的步骤,例如,创建用户账户、下载和安装所需要的软件包、为服务编写配置文件等。这些步骤是高度可定制的,用户可以根据自己的需求添加需要的步骤。

2.7.3 Puppet 与 Chef

类似于 Whirr,还有其他的专用部署框架。Puppet 出自 Puppet 实验室 (<http://www.puppetlabs.com>)。

① 请注意 Whirr 是一个 Apache 基金会孵化的项目,一旦被 Apache 社区接受并晋升为社区正式成员,就会被转移到一个受 Apache 保护的、完整且永久有效的网址中。

puppetlabs.com/), Chef 出自 Opscode (<http://www.opscode.com/chef/>)。

两者相似的地方是, 所有配置文件都保存在中央配置服务器中, 每台客户端服务器都与中央配置服务器连接, 通过客户端软件监听配置的更新并合并到本地。

类似于 Whirr, 上述两者都有 *recipes* 的概念, 本质上会转化成脚本或命令在节点上执行。^①事实上, 基于 Puppet 或 Chef 的进程极有可能取代 Whirr 的脚本。

Whirr 仅用于引导, Puppet 和 Chef 还可用于变更集群, master 进程监控配置库, 并检查远程是否更新, 一旦发生更新则触发远程操作。这可以协助用户重新配置集群或升级新版本, 并进行滚动重启等操作。Whirr 可以概括为配置管理, 但功能并不局限于配置管理。



想必读者以前听过: 选择一个自己熟悉的做法或选择一个自己喜欢的做法。两者的目标是一样的, 即安装集群节点中所需要的所有东西。如果用户需要一个完整的、实时更新的、基于 Puppet 或 Chef 的配置管理解决方案, 那么与 Whirr 结合使用是正确的选择。

2.8 操作集群

如果用户现在准备第一次启动集群, 一定要确认服务器已经安装好, 并配置好了操作系统与文件系统, 此外还要确认集群所需要的属性在配置文件中都已经配置完成。

2.8.1 确定安装运行

首先启动 HDFS。运行 `HADOOP_HOME` 目录下的 `bin/start-dfs.sh` 与 `bin/stop-dfs.sh` 可以启动与关闭 HDFS。之后用户可以通过 `put` 与 `get` 文件来验证 Hadoop 文件系统是否已经启动成功。HBase 本身并不依赖 MapReduce 守护进程, 只有在需要运行 MapReduce 作业的时候才需要启动 MapReduce 框架, 具体细节见第 7 章。

如果不依赖 HBase 管理 ZooKeeper, 则需要确保独立的 ZooKeeper 已经运行; 否则 HBase 会把 ZooKeeper 作为 HBase 进程的一部分启动。

如果用户需要启动单机模式请查阅 2.1 节, 启动完全分布式模式需要如下命令:

```
bin/start-hbase.sh
```

以上命令需要在 `HBASE_HOME` 目录下运行。如果 HBase 已经运行, 可以在 `logs` 目录的子目录中找到 HBase 运行日志文件。用户如果发现 HBase 没有按照预期运行, 请参考 12.5.2 节寻找帮助以便分析和查找问题原因。

① 一些可用的配置包是为了适应早期的 EC2 脚本创建的, 被用于部署 HBase 到动态云平台中。有关 Chef 的详情用户可以查阅网页 <http://cookbooks.opscode.com/cookbooks/hbase>。有关 Puppet 用户可以查阅 <http://hstack.org/hstack-automated-deployment-using-puppet/> 和 <http://github.com/hstack/puppet>。

一旦 HBase 启动, 关于如何建表、添加数据、扫描已插入的数据、禁用表和删除表等操作都可以在 2.1 节中找到帮助信息。

2.8.2 Web UI 介绍

HBase Master 默认基于 Web 的 UI 服务端口为 60010, HBase region 服务器默认基于 Web 的 UI 服务端口为 60030。如果 master 运行在名为 master.foo.com 的主机中, master 的主页地址就是 http://master.foo.com:60010, 用户可以通过 Web 浏览器输入这个地址查看该页面。图 2-2 展示了页面显示结果, 更多信息见 6.5 节。

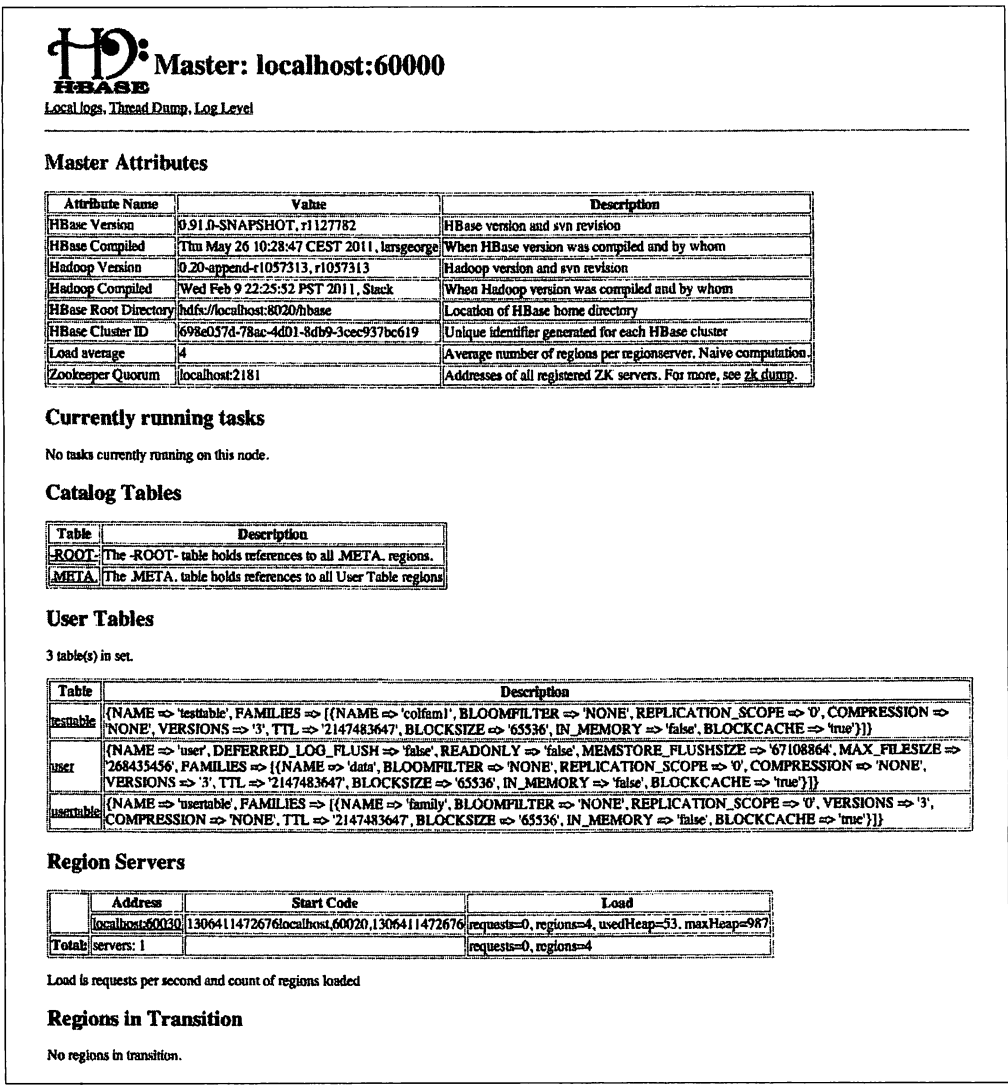


图 2-2 HBase Master 的用户界面

这个页面中可以查看 HBase 集群的当前状态。这些信息分成了若干块，页面顶部展示了集群的安装信息，还可以看到当前运行的任务（如果有的话）。Catalog 与 User 表展示了所有可用的表，从这里可以看到完整的表结构。

页面底部显示了 Region Servers 表，展示了当前可用的 region 服务器。最底部是处于事务状态下的 region，这些 region 当前可能处于系统维护状态。

集群启动后，用户不仅可以通过页面检查 region 服务器是否都已经正常注册到 master，并以期望的主机名显示在页面中（客户端能够连接），还可以检查当前启动的 HBase 与 Hadoop 版本是否正确。

2.8.3 Shell 介绍

通过 2.1 节，你可以使用 HBase 提供的 Shell 命令行，其中介绍了通过命令行模式创建表、新增和更新数据，以及删除表。

HBase Shell 是使用 (J) Ruby (<http://jruby.org>) 的 IRB 实现的命令行脚本，IRB 中可做的事情在 HBase Shell 中也可以完成。通过以下命令就可以运行 Shell：

```
$ $HBASE_HOME/bin/hbase shell
HBase Shell;enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.91.0-SNAPSHOT,r1130916,Sat Jul 23 12:44:34 CEST 2011

hbase(main):001:0>
```

输入 help 并按回车键能够得到所有 Shell 命令和选项。浏览帮助文档可以看到每个具体的命令参数的用法（变量、命令参数）；特别注意怎样引用表名、行键、列名等。6.4 节详细介绍了 Shell 的相关内容。

由于 HBase Shell 是基于 Ruby 实现的，因此在使用过程中可以将 HBase 命令与 Ruby 代码混合使用，你可以按照如下方式使用：

```
hbase(main):001:0> create 'testtable','colfam1'
hbase(main):002:0> for i in 'a'..'z' do for j in 'a'..'z' do \
  put 'testtable', "row-#{i}#{j}", "colfam1:#{j}", " #{j}" end end
```

第一行命令是创建一张叫 testtable 的表，并创建一个叫 colfam1 的列族名，列族的属性都使用默认属性（详情见 5.1.3 节）。第二行命令是用 Ruby 代码循环插入多行数据到刚刚创建的表中。行键开始于 row-aa、row-ab，终止于 row-zz。

2.8.4 关闭集群

运行以下命令可以停止 HBase 集群。一旦启动了这个脚本，你将会看到一条描述集群正在停止的信息，该信息会周期性地打印 “.” 字符（这仅仅表明脚本正在运行，并不是运行进度的反馈或隐藏的有用信息）。

```
$ ./bin/stop-hbase.sh
stopping hbase.....
```

关闭脚本大概需要几分钟完成。如果集群中机器数量很多，那么执行时间可能更长。如果用户运行的是分布式模式，在关闭 Hadoop 集群之前一定要确认 HBase 已经被正常关闭了。

第 12 章中有许多更高级的管理功能，如滚动重启、增加 master 节点等。如果集群无法启动或关闭，第 12 章里也有相关信息，说明如何分析和解决这些问题。

第 3 章

客户端 API：基础知识

本章将会介绍 HBase 提供的客户端 API。在前文提到过，HBase 是使用 Java 编写的，所以原生的 API 也是 Java 开发的，不过这并不意味着必须通过 Java 访问 HBase。我们会在第 6 章介绍如何通过其他编程语言使用 HBase。

3.1 概述

HBase 的主要客户端接口是由 `org.apache.hadoop.hbase.client` 包中的 `HTable` 类提供的，通过这个类，用户可以完成向 HBase 存储和检索数据，以及删除无效数据之类的操作。在介绍这个类的各个方法之前，让我们先了解一下它的大体功能。

所有修改数据的操作都保证了行级别的原子性，这会影响到这一行数据所有的并发读写操作。换句话说，其他客户端或线程对同一行的读写操作都不会影响该行数据的原子性：要么读到最新的修改，要么等待系统允许写入该行修改。更多内容请参考第 8 章。^①

通常，在正常负载和常规操作下，客户端读操作不会受到其他修改数据的客户端影响，因为它们之间的冲突可以忽略不计。但是，当许多客户端需要同时修改同一行数据时就会产生问题。所以，用户应当尽量使用批量处理（batch）更新来减少单独操作同一行数据的次数。

写操作中涉及的列的数目不会影响该行数据的原子性，行原子性会同时保护到所有列。

最后，创建 `HTable` 实例是有代价的。每个实例都需要扫描 `.META.` 表，以检查该表是否存在、是否可用，此外还要执行一些其他操作，这些检查和操作导致实例调用非常耗时。因此，推荐用户只创建一次 `HTable` 实例，而且是每个线程创建一个，然后在客户

^① `region` 服务器采用了一种多版本并发控制机制，具体实现在 `ReadWriteConsistencyControl`（简称为 `RWCC`）类中。这种机制保证了读程序读取数据时可以不用等待写程序完成写操作。写程序则需要等待其他写程序完成写操作之后才能继续执行。

端应用的生存期内复用这个对象。

如果用户需要使用多个 HTable 实例，应考虑使用 HTablePool 类（详情见 4.4 节），它为用户提供了—个复用多个实例的便捷方式。



以下是我们刚才讨论内容的几点总结。

- 只创建—次 HTable 实例，—般在应用程序开始时创建。
- 为执行的每一个线程（或者所使用的 HTablePool）创建独立的 HTable 实例。
- 所有的修改操作只保证行级别的原子性。

3.2 CRUD 操作

数据库的初始基本操作通常被称为 CRUD（Create, Read, Update, Delete），具体指增、查、改、删。HBase 中有与之相对应的一组操作，随后我们会依次介绍。这些方法都由 HTable 类提供，本章后面将直接引用这个类的方法，不再特别提到这个包含类。

接下来介绍的操作大多都能不言自明，但本书有一些细节需要大家注意。这意味着，对于书中出现的一些重复的模式，我们不会多次赘述。



你所看到的示例源代码都可以从 GitHub 的公用源中下载，具体地址为 <https://github.com/larsgeorge/hbase-book>。如果需要了解源码编译的细节，请参考前言中的“编译示例程序”—节。

读者—开始会在—些程序的开头看到 import 语句，但为了简洁，后续将会省略 import 语句。同时，—些与主题不太相关的代码部分也会被省略。如有疑问，请到上面的地址中查阅完整源代码。

3.2.1 put 方法

下面介绍的这组操作可以被分为两类：—类操作用于单行，另—类操作用于多行。鉴于后面有一些内容比较复杂，我们会分开介绍这两类操作。同时，我们还会介绍—些衍生的客户端 API 特性。

1. 单行 put

也许你现在最想了解的就是如何向 HBase 中存储数据，下面就是实现这个功能的调用：

```
void put(Put put) throws IOException
```

这个方法以单个 Put 或存储在列表中的—组 Put 对象作为输入参数，其中 Put 对象是由

以下几个构造函数创建的：

```
Put(byte[] row)
Put(byte[] row, RowLock rowLock)
Put(byte[] row, long ts)
Put(byte[] row, long ts, RowLock rowLock)
```

创建 Put 实例时用户需要提供一个行键 row，在 HBase 中每行数据都有唯一的行键(row key) 作为标识，跟 HBase 的大多数数据类型一样，它是一个 Java 的 byte[] 数组。用户可以按自己的需求来指定每行的行键，且可以参考第 9 章，其中专门有一节详细讨论了行键的设计（见 9.1 节）。现在我们假设用户可以随意设置行键，通常情况下，行键的含义与真实场景相关，例如，它的含义可以是一个用户名或者订单号，它的内容可以是简单的数字，也可以是较复杂的 UUID^① 等。

HBase 非常友好地为用户提供了一个包含很多静态方法的辅助类，这个类可以把许多 Java 数据类型转换为 byte[] 数组。例 3.1 提供了方法的部分清单。

例 3.1 Bytes 类所提供的方法

```
static byte[] toBytes(ByteBuffer bb)
static byte[] toBytes(String s)
static byte[] toBytes(boolean b)
static byte[] toBytes(long val)
static byte[] toBytes(float f)
static byte[] toBytes(int val)
...
```

创建 Put 实例之后，就可以向该实例添加数据了，添加数据的方法如下：

```
Put add(byte[] family, byte[] qualifier, byte[] value)
Put add(byte[] family, byte[] qualifier, long ts, byte[] value)
Put add(KeyValue kv) throws IOException
```

每一次调用 add() 都可以特定地添加一系列数据，如果再加一个时间戳选项，就能形成一个数据单元格。注意，当不指定时间戳调用 add() 方法时，Put 实例会使用来自构造函数的可选时间戳参数（也称作 ts），如果用户在构造 Put 实例时也没有指定时间戳，则时间戳将会由 region 服务器设定。

系统为一些高级用户提供了 KeyValue 实例的变种，这里所说的高级用户是指知道怎样检索或创建这个内部类的用户。KeyValue 实例代表了一个唯一的数据单元格，类似于一个协调系统，该系统使用行键、列族、列限定符、时间戳指向一个单元格的值，像一个三维立方体系统（其中，时间成为了第三维度）。

获取 Put 实例内部添加的 KeyValue 实例需要调用与 add() 相反的方法 get()：

① 全局唯一标识符 (Universally Unique Identifier) 细节请参阅 http://en.wikipedia.org/wiki/Universally_unique_identifier。

```
List<KeyValue> get(byte[] family, byte[] qualifier)
Map<byte[], List<KeyValue>> getFamilyMap()
```

以上两个方法可以查询用户之前添加的内容，同时将特定单元格的信息转换成 KeyValue 实例。用户可以选择获取整个列族（column family）的全部数据单元，一个列族中的特定列或是全部数据。后面的 getFamilyMap() 方法可以遍历 Put 实例中每一个可用的 KeyValue 实例，检查其中包含的详细信息。



每一个 KeyValue 实例包含其完整地址（行键、列族、列限定符及时间戳）和实际数据。KeyValue 是 HBase 在存储架构中最底层的类。8.2 节将会详细介绍相关内容。对于客户端 API 所用到的 KeyValue 类中的方法参见 3.2.1 节。

用户可以采用以下这些方法检查是否存在特定的单元格，而不需要遍历整个集合：

```
boolean has(byte[] family, byte[] qualifier)
boolean has(byte[] family, byte[] qualifier, long ts)
boolean has(byte[] family, byte[] qualifier, byte[] value)
boolean has(byte[] family, byte[] qualifier, long ts, byte[] value)
```

随着以上方法所使用参数的逐步细化，获得的信息也越详细，当找到匹配的列时返回 true。第一个方法仅检查一个列是否存在，其他的方法则增加了检查时间戳、限定值的选项。

Put 类还提供了很多的其他方法，在表 3-1 中进行了概括。

表 3-1 Put 类提供的其他方法一览表

方法	描述
getRow()	返回创建 Put 实例时所指定的行键
getRowLock()	返回当前 Put 实例的行 RowLock 实例
getLockId()	返回使用 rowlock 参数传递给构造函数的可选的锁 ID，当未被指定时返回 -1L
setWriteToWAL()	允许关闭默认启用的服务器端预写日志（WAL）功能
getWriteToWAL()	返回代表是否启用了 WAL 的值
getTimeStamp()	返回相应 Put 实例的时间戳，该值可在构造函数中由 ts 参数传入。当未被设定时返回 Long.MAX_VALUE
heapSize()	计算当前 Put 实例所需的堆大小，既包含其中的数据，也包含内部数据结构所需的 空间
isEmpty()	检测 FamilyMap 是否含有任何 KeyValue 实例
numFamilies()	查询 FamilyMap 的大小，即所有的 KeyValue 实例中列族的个数
size()	返回本次 Put 会添加的 KeyValue 实例的数量



注意，表 3-1 所列的 Put 类中的那些 `getter` 函数仅能够获取用户预先设定的内容，实际应用中很少用到它们，仅当用户在代码的私有方法中准备实现一个 Put 实例，并在其他地方检查其内容时，才会用到它们。

例 3.2 展示了如何在一个简单的程序里使用上述方法。



本章的示例使用了一个非常有限的精确数据集。当读者查看整个源代码时，会注意到源代码使用了一个名叫 `HBaseHelper` 的内部类。该内部类会创建一个有特定行和列数量的数据测试表。这让我们更容易对比处理前后的差异。

读者可以将代码直接放到本地主机上的独立 HBase 实例中来测试，也可以放到 HBase 集群上测试。前言中的“编译示例程序”一节解释了如何编译这些例子。读者可以大胆地修改这部分代码，以便更好地体会各部分功能。

为了清除前一步示例程序执行时产生的数据，示例代码通常会删除前一步执行时所创建的表。如果你在生产集群上运行示例，请先确保表名无冲突。通常示例代码创建的表为 `testtable`，这个名称容易让人联想到表的用途。

例 3.2 向 HBase 插入数据的示例应用

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.util.Bytes;

import java.io.IOException;

public class PutExample {

    public static void main(String[] args) throws IOException {
        Configuration conf = HBaseConfiguration.create(); ❶

        HTable table = new HTable(conf, "testtable"); ❷

        Put put = new Put(Bytes.toBytes("row1")); ❸

        put.add(Bytes.toBytes("colfam1"), Bytes.toBytes("qual1"),
            Bytes.toBytes("val1")); ❹
        put.add(Bytes.toBytes("colfam1"), Bytes.toBytes("qual2"),
            Bytes.toBytes("val2")); ❺

        table.put(put); ❻
    }
}
```

❶ 创建所需的配置。

❷ 实例化一个新的客户端。

- ③ 指定一行来创建一个 Put。
- ④ 向 Put 中添加一个名为“colfam1:qual1”的列。
- ⑤ 向 Put 中添加另一个名为“colfam1:qual2”的列。
- ⑥ 将这一行存储到 HBase 表中。

这个示例代码（几乎）十分完整，并且每一行都进行了解释。以后的示例会逐渐减少样板代码，以便读者能将注意力集中到重要的部分。

通过客户端代码访问配置文件

2.6.7 节介绍了 HBase 客户端应用程序使用的配置文件。应用程序需要通过默认位置（classpath）下的 *hbase-site.xml* 文件来获知如何访问集群，此外也可以在代码里指定集群的地址。

无论哪种方式，都需要在代码中使用一个 HBaseConfiguration 类来处理配置的属性。可以使用该类提供的以下静态方法构建 Configuration 实例：

```
static Configuration create()
static Configuration create(Configuration that)
```

例 3.2 中使用了 create() 来获得 Configuration 实例。第二个方法允许你使用一个已存在的配置，该配置会融合并覆盖 HBase 默认配置。

当你调用任何一个静态 create() 方法时，代码会尝试使用当前的 Java classpath 来载入两个配置文件：*hbase-default.xml* 和 *hbase-site.xml*。

如果使用 create(Configuration that) 方法指定一个已存在的配置，那么与所有从 classpath 载入的配置相比，用户指定的配置优先级最高。

HBaseConfiguration 类继承自 Hadoop 的 Configuration 类，但是 HBaseConfiguration 类仍然和 Configuration 类兼容：用户可以提交一个 Hadoop 的 Configuration 实例，它们的内容可以很好地合并。

当用户获得了一个 HBaseConfiguration 实例之后，其实已获得了一个已经合并过的配置，其中包括默认值和在 *hbase-site.xml* 配置文件中重写的属性，以及一些用户提交的可选配置。在使用 HTable 实例之前，用户可以任意地修改配置。例如，可以重写 ZooKeeper 的可用连接地址来定位到另一个集群：

```
Configuration config = HBaseConfiguration.create();
config.set("hbase.zookeeper.quorum", "zk1.foo.com,zk2.foo.com");
```

换句话说，可以简单地忽略任何外部的客户端配置文件，而直接在代码中设置 hbase.zookeeper.quorum 属性。这样就创建了一个不需要额外配置的客户端。

同时应该共享配置实例，4.6 节解释了这样做的原因。

现在又可以使用 Shell 命令行（详情见 2.1 节）来验证插入是否成功了：

```
hbase(main):001:0>list
TABLE
testtable
1 row(s) in 0.0400 seconds

hbase(main):002:0>scan 'testtable'
ROW          COLUMN+CELL
row1         column=colfam1:qual1,timestamp=1294065304642,value=val1
1 row(s) in 0.2050 seconds
```

在创建 Put 实例时用到的另一个可选参数是 ts，即时间戳（timestamp）。在 HBase 表中，时间戳使用户可以在 HBase 表中将数据存储为一个特定版本。

数据的版本化

HBase 的一个特殊功能是，能为一个单元格（一个特定列的值）存储多个版本的数据。这是通过每个版本使用一个时间戳。并且按照降序存储来实现的。每个时间戳是一个长整型值，以毫秒为单位。它表示自世界标准时间（UTC）1970 年 1 月 1 日 0 时以来所经过的时间，这个时间又称为 Unix 时间^①或 Unix 纪元。大多数操作系统都有一个时钟获取函数来读取这个时间。例如，在 Java 中可以使用 System.currentTimeMillis() 函数。

将数据存入 HBase 时，要么显式地提供一个时间戳，要么忽略该时间戳。如果用户忽略该时间戳的话，RegionServer 会在执行 put 操作的时候填充该时间戳。

如 2.2 节所述，必须确保服务器的时间是正确的，并且互相之间是同步的。用户可能无法控制客户端时间，所以很可能会与服务器时间不同，可能会相差几小时甚至相差几年。

如果用户不指定时间，那么客户端 API 调用会以服务器端时间为准。一旦需要使用并指定明确的时间戳，用户需要确保不受一些可能发生的意外的影响。客户端可能会使用意想不到的时间戳来插入数据，从而产生看似无序的版本历史。

虽然大多数程序并不担心版本化问题，并且依赖于 HBase 内置的处理时间戳的方法，但是如果使用自定义的时间戳，就应该了解这些特性。

下面展示了如何向一个单元格插入并且获取多版本数据。

```
hbase(main):001:0> create 'test','cf1'
0 row(s) in 0.9810 seconds

hbase(main):002:0> put 'test','row1','cf1','val1'
```

① 见维基百科的“Unix time”（http://en.wikipedia.org/wiki/Unix_epoch）。

```

0 row(s) in 0.0720 seconds

hbase(main):003:0> put 'test','row1','cf1','val2'
0 row(s) in 0.0520 seconds

hbase(main):004:0> scan 'test'
ROW          COLUMN+CELL
 row1        column=cf1:,timestamp=1297853125623,value=val2
1 row(s) in 0.0790 seconds

hbase(main):005:0> scan 'test',{ VERSIONS => 3 }
ROW          COLUMN+CELL
 row1        column=cf1:,timestamp=1297853125623,value=val2
 row1        column=cf1:,timestamp=1297853122412,value=val1
1 row(s) in 0.0640 seconds

```

该示例在 test 表中创建了一个名为 cf1 的列族。两个 put 命令使用了相同的行键和列键，但它们的值不同：分别为 val1 和 val2。然后使用 scan 操作查看了这张表的所有内容。你可能并不惊讶于只看到了 val2，因为你可能已经假设第二次 put 操作覆盖了 val1。

但是在 HBase 中并不是这样的。默认情况下，HBase 会保留 3 个版本的数据，用户可以利用这种特性，稍稍修改 scan 操作以便获取所有可获得的数据（即版本）。示例中的最后一个命令列出了所有存储的数据版本。注意，即使所有输出的行键都是相同的，在 Shell 的输出中，所有的单元格都是以单独的一行输出的。

scan 操作和 get 操作只会返回最后的（也叫最新的）版本，这是因为 HBase 默认按照版本的降序存储，并且只返回一个版本。在调用中加入最大版本（maximum version）参数就可以获得多个版本的数据，如果将参数值设定为 Integer.MAX_VALUE，就可以获得所有的版本。

正如最大版本的术语所表现出来的意思一样，对于一个特定的单元格，有可能只有少于最大版本数个版本。示例将 VERSIONS（MAX_VERSIONS 的缩写）设为 3，但是该单元格只存储了两个版本的数据，所以就列出了两个。

另一个获取多个版本数据的方法是，使用时间范围参数。只需要设置开始时间和结束时间，就能获得所有满足时间范围的版本数据。更多有关这一方面的内容，请参考 3.2.2 节和 3.5 节。

关于版本化，有很多细小（有些也不算小）的问题，将在 8.4 节继续讨论，而且还会在 9.6 节重新讨论更高级的概念，以及不标准的行为。

如果读者不指定该参数，当数据存储到底层文件系统时，RegionServer 会将当前行的时间戳隐式地设定为系统当前时间。

Put 类的构造函数还有一个名为 rowlock 的可选参数，它允许提交一个额外的行锁（row

lock), 详见 3.4 节。最后还要说一句, 若需要频繁地重复修改某些行, 用户有必要创建一个 RowLock 实例来防止其他客户端访问这些行。

2. KeyValue 类

在代码中有时需要直接处理 KeyValue 实例。你可能还记得之前讨论过的那些实例, 它们都含有一个特定单元格的数据以及坐标 (coordinate)。坐标包括行键、列族名、列限定符以及时间戳。该类提供了特别多的构造器, 允许以各种方式组合这些参数。下面展示了包括所有参数的构造器:

```
KeyValue(byte[] row, int roffset, int rlength,
         byte[] family, int foffset, int flength, byte[] qualifier, int goffset,
         int qlength, long timestamp, Type type, byte[] value, int voffset,
         int vlength)
```



建议将 KeyValue 类和它的比较器都设计为 HBase 内部使用。只在客户端 API 的几个地方出现, 以使用户访问原始数据, 这样可以避免额外的复制操作。还可以允许基于字节的比较, 而不是依靠比较慢的基于类的比较。

数据和坐标都是以 Java 的 byte[] 形式存储的, 即以字节数组的形式存储的。使用这种底层存储类型的目的是, 允许存储任意类型的数据, 并且可以有效地只存储所需的字节, 这保证了最少的内部数据结构开销。另一个原因是, 每一个字节数组都有一个 offset 参数和一个 length 参数, 它们允许用户提交一个已存在的字节数组, 并进行效率很高的字节级别的操作。

坐标中任意一个成员都有一个 getter 方法, 可以获得字节数组以及它们的参数 offset 和 length。不过也可以在最顶层访问它们, 即直接读取底层字节缓冲区:

```
byte[] getBuffer()
int getOffset()
int getLength()
```

它们返回当前 KeyValue 实例中字节数组完整信息。用户用到这些方法的场景很少, 但是在需要的时候, 还是可以使用这些方法的。

还有两个有意思的方法:

```
byte [] getRow()
byte [] getKey()
```

读者也许会问这样一个问题: 行 (row) 和键 (key) 有什么区别? 关于它们的区别将在 8.2 节中描述。行目前来说指的是行键, 即 Put 构造器里的 row 参数。而在之前介绍的内容中, 键是一个单元格的坐标, 用的是原始的字节数组格式。在实践中, 几乎用不到 getKey(), 但有可能会用到 getRow()。

KeyValue 类还提供一系列实现了 Comparator 接口的内部类，可以在代码里使用它们来实现与 HBase 内部一样的比较器。当需要使用 API 获取 KeyValue 实例时，并进一步排序或按顺序处理时，就要用到这些比较器。表 3-2 列出了这些比较器。

表 3-2 KeyValue 类提供的比较器的简要概述

比较器	描述
KeyComparator	比较两个 KeyValue 实例的字节数组格式的行键，即 getKey() 方法的返回值
KVComparator	是 KeyComparator 的封装，基于两个给定的 KeyValue 实例，提供与 KeyComparator 一样的功能
RowComparator	比较两个 KeyValue 实例的行键 (getRow() 的返回值)
MetaKeyComparator	比较两个以字节数组格式表示的 .META. 条目的行键
MetaComparator	KVComparator 类的一个特别版本，用于比较 .META. 目录表中的条目，是 MetaKeyComparator 的封装
RootKeyComparator	比较两个以字节数组格式表示的 -ROOT- 条目的行键
RootComparator	KVComparator 类的一个特别版本，用于比较 -ROOT- 目录表中的条目，是 RootKeyComparator 的封装

KeyValue 类将大部分的比较器按照静态实例提供给其他类使用。例如，有一个公有变量 KEY_COMPARATOR，让用户可以访问 KeyComparator 实例。COMPARATOR 变量指向使用更频繁的 KVComparator 实例。所以可以不用创建自己的实例，而是使用提供的实例。例如，可以按照以下方法创建一个 KeyValue 实例的集合，这个集合可以按照 HBase 内部使用的顺序来排序：

```
TreeSet<KeyValue> set =
    new TreeSet<KeyValue>(KeyValue.COMPARATOR)
```

KeyValue 实例还有一个变量（一个额外的属性），代表该实例的唯一坐标：类型。表 3-3 列出了所有可能的值。

表 3-3 KeyValue 实例所有可能的类型值

类型	描述
Put	KeyValue 实例代表一个普通的 Put 操作
Delete	KeyValue 实例代表一个 Delete 操作，也称为墓碑标记
DeleteColumn	与 Delete 相同，但是会删除一整列
DeleteFamily	与 Delete 相同，但是会删除整个列族，包括该列族的所有列

可以通过使用另外一个方法来查看一个 KeyValue 实例的类型，例如：

```
String toString()
```

该方法会按照以下格式打印出当前 KeyValue 实例的元信息：

<row-key>/<family>:<qualifier>/<version>/<type>/<value-length>

这个方法会用在本书的一些示例代码中，用于检查数据是否被标记或者被恢复，同时也可以查看元信息。

该类有很多更便捷的方法：允许对存储数据的其中一部分进行比较，检查实例的类型是什么，获得它已经计算好的堆大小，克隆或者复制该类等。有一些静态方法可以创建一些特殊的 KeyValue 实例，用以在 HBase 内更底层地比较或者操作数据。可以参考 Java 文档来了解更多的内容^①。还可以查看 8.2 节，该节详细地解释了 KeyValue 原始的二进制格式内容。

3. 客户端的写缓冲区

每一个 put 操作实际上都是一个 RPC^②操作，它将客户端数据传送到服务器然后返回。这只适合小数据量的操作，如果有个应用程序需要每秒存储上千行数据到 HBase 表中，这样的处理就不太合适了。



减少独立 RPC 调用的关键是限制往返时间（round-trip time），往返时间就是客户端发送一个请求到服务器，然后服务器通过网络进行响应的时间。这个时间不包含数据实际传输的时间，它其实就是通过线路传送网络包的开销。一般情况下，在 LAN 网络中大约要花 1 毫秒的时间，这意味着在 1 秒钟的时间内只能完成 1000 次 RPC 往返响应。

另一个重要的因素就是消息大小。如果通过网络发送的请求内容较大，那么需要请求返回的次数相应较少，这是因为时间主要花费在数据传递上。不过如果传送的数据量很小，比如一个计数器递增操作，那么用户把多次修改的数据批量提交给服务器并减少请求次数，性能会有相应提升。

HBase 的 API 配备了一个客户端的写缓冲区（write buffer），缓冲区负责收集 put 操作，然后调用 RPC 操作一次性将 put 送往服务器。全局交换机控制着该缓冲区是否在使用，以下是其方法：

```
void setAutoFlush(boolean autoFlush)
boolean isAutoFlush()
```

默认情况下，客户端缓冲区是禁用的。可以通过将自动刷写（autoflush）设置为 false 来激活缓冲区，调用如下：

```
table.setAutoFlush(false)
```

① 完整的描述请参阅 API 文档中 KeyValue 类的介绍（<http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/KeyValue.html>）。

② 参见维基百科的“Remote procedure call”（http://en.wikipedia.org/wiki/Remote_procedure_call）。

启用客户端缓冲机制后，用户可以通过 `isAutoFlush()` 方法检查标识的状态。当用户初始化创建一个 `HTable` 实例时，这个方法将返回 `true`，如果有用户修改过缓冲机制，它会返回用户当前所设定的状态。

激活客户端缓冲区之后，用户可以像 3.2.1 节“单行 put”中介绍的那样，将数据存储到 `HBase` 中。此时的操作不会产生 RPC 调用，因为存储的 `Put` 实例保存在客户端进程的内存中。当需要强制把数据写到服务端时，可以调用另外一个 API 函数：

```
void flushCommits() throws IOException
```

`flushCommits()` 方法将所有的修改传送到远程服务器。被缓冲的 `Put` 实例可以跨多行。客户端能够批量处理这些更新，并把它们传送到对应的 `region` 服务器。和调用单行 `put()` 方法一样，用户不需要担心数据分配到了哪里，因为对于用户来说，`HBase` 客户端对这个方法的处理是透明的。图 3-1 展示了在客户端请求传送到服务器之前，是怎样按 `region` 服务器排序分组，并通过每个 `region` 服务器的 RPC 请求将数据传送到服务器的。

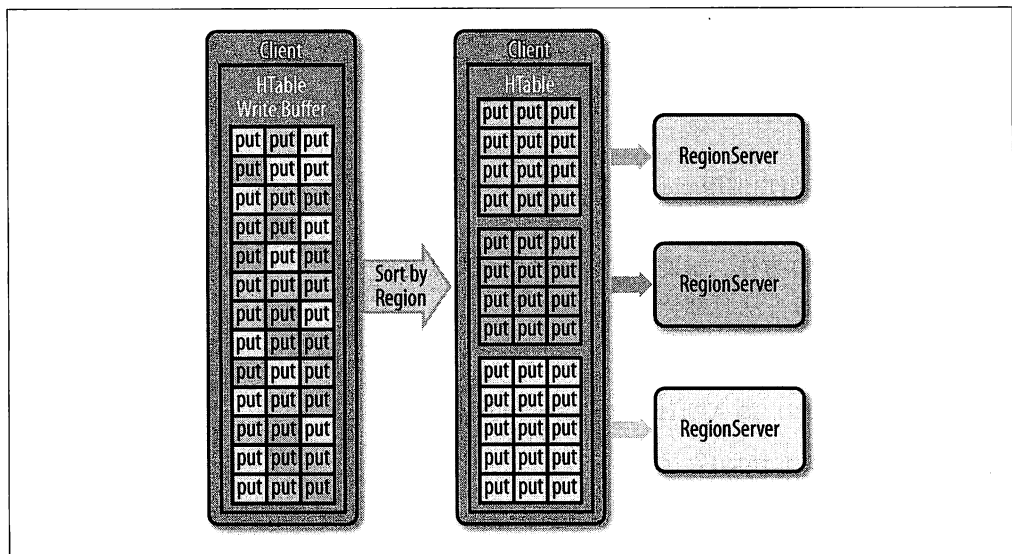


图 3-1 客户端 `put` 操作按所属 `region` 服务器排序和分组

用户可以强制刷写缓冲区，不过这通常不是必要的，因为 API 会追踪统计每个用户添加的实例的堆大小，从而计算出缓冲的数据量。除了追踪所有的数据开销，还会追踪必要的内部数据结构，一旦超出缓冲指定的大小限制，客户端就会隐式地调用刷写命令。用户可以通过以下调用来配置客户端写缓冲区的大小：

```
long getWriteBufferSize()
void setWriteBufferSize(long writeBufferSize) throws IOException
```

默认的大小是 2 MB（即 2 097 152 字节），这个大小比较适中，一般用户插入 `HBase` 中

的数据都相当小，即每次插入的数据都远小于缓冲区的大小。如果需要存储较大的数据，可能就需要考虑增大这个数值，从而允许客户端更高效地将一定数量的数据组成一组，通过一个 RPC 请求来执行。



给每一个用户创建的 HTable 实例都设定缓冲区大小十分麻烦，为了避免这个麻烦，用户可以在 *hbase-site.xml* 配置文件中添加一个较大的预设值。例如：

```
<property>
  <name>hbase.client.write.buffer</name>
  <value>20971520</value>
</property>
```

这会将缓冲区大小增加到 20 MB。

缓冲区仅在以下两种情况下会刷写。

显式刷写

用户调用 `flushCommits()` 方法，把数据发送到服务器做永久存储。

隐式刷写

隐式刷写会在用户调用 `put()` 或 `setWriteBufferSize()` 方法时触发。这两个方法都会将目前占用的缓冲区大小与用户配置的大小做比较，如果超出限制则会调用 `flushCommits()` 方法。如果缓冲区被禁用，可以设置 `setAutoFlush(true)`，这样用户每次调用 `put()` 方法时都会触发刷写。

此外调用 HTable 类的 `close()` 方法时也会无条件地隐式触发刷写。

例 3.3 展示了客户端 API 如何控制写缓冲区。

例 3.3 使用客户端写缓冲区

```
HTable table = new HTable(conf, "testtable");
System.out.println("Auto flush: " + table.isAutoFlush()); ❶

table.setAutoFlush(false); ❷

Put put1 = new Put(Bytes.toBytes("row1"));
put1.add(Bytes.toBytes("colfam1"), Bytes.toBytes("qual1"),
    Bytes.toBytes("val1"));
table.put(put1); ❸

Put put2 = new Put(Bytes.toBytes("row2"));
put2.add(Bytes.toBytes("colfam1"), Bytes.toBytes("qual1"),
    Bytes.toBytes("val2"));
table.put(put2);
```



```

Put put3 = new Put(Bytes.toBytes("row3"));
put3.add(Bytes.toBytes("colfam1"), Bytes.toBytes("qual1"),
    Bytes.toBytes("val3"));
table.put(put3);

Get get = new Get(Bytes.toBytes("row1"));
Result res1 = table.get(get);
System.out.println("Result: " + res1);❶

table.flushCommits();❷

Result res2 = table.get(get);
System.out.println("Result: " + res2);❸

```

- ❶ 检查自动刷写标识位的设置，应该会打印出 “Auto flush: true”。
- ❷ 设置自动刷写为 false，启用客户端写缓冲区。
- ❸ 将一些行和列数据存入 HBase。
- ❹ 试图加载先前存储的行，结果会打印出 “Result: keyvalues=NONE”。
- ❺ 强制刷写缓冲区，会导致产生一个 RPC 请求。
- ❻ 现在，这一行被持久化了，可以被读取了。

这个例子展示了一个用户之前意想不到的，使用缓冲区之后产生的现象。让我们看看当执行它时会打印出什么：

```

Auto flush: true
Result: keyvalues=NONE
Result: keyvalues={row1/colfam1:qual1/1300267114099/Put/vlen=4}

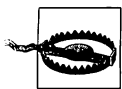
```

虽然还没有介绍过 `get()` 操作，但你应该能够正确地推断出它是用于从服务器读取数据的。例子中的第一个 `get()` 操作返回了一个 `NONE`，这是什么意思呢？这是由于客户端的写缓冲区是一个内存结构，存储了所有未刷写的记录，这些数据记录尚未发送到服务器，因此用户无法访问它。



用户可以使用以下方法访问客户端写缓冲区的内容：`ArrayList<Put> getWriteBuffer()`。这个方法可以获取 `table.put(put)` 添加到缓冲区中的 `Put` 实例列表。

前面提到过，正是由于该列表，使得 `HTable` 类被多个线程操作时不安全。直接操作那个列表的时候要非常小心，因为这将绕过堆大小的检查，同时还有可能遇到缓冲区正在刷写其内容。



由于客户端缓冲区是一个简单的保存在客户端进程内存里的列表，用户需要注意不能在运行时终止程序。如果发生这种情况，那些尚未刷写的的数据就会丢失！服务器将无法收到数据，因此这些数据没有任何副本可以用来做数据恢复。

另外请注意，一个更大的缓冲区需要客户端和服务端消耗更多的内存，因为服务器端也需要先将数据写入到服务器的写缓冲区中，然后再处理它。另一方面，一个大的缓冲区减少了 RPC 请求的次数。估算服务器端内存的占用可使用 `hbase.client.write.buffer × hbase.regionserver.handler.count × region` 服务器的数量。

再次提到往返时间，如果用户只存储大单元格，客户端缓冲区的作用就不大了，因为传输时间占用了大部分的请求时间。在这种情况下，建议最好不要增加客户端缓冲区大小。

4. Put 列表

客户端的 API 可以插入单个 Put 实例，同时也有批量处理操作的高级特性。其调用形式如下：

```
void put(List<Put> puts) throws IOException
```

用户需要建立一个 Put 实例的列表。例 3.4 修改了之前的例子，创建了一个列表保存所有的修改，最后调用了以列表为参数的 `put()` 方法。

例 3.4 使用列表向 HBase 中添加数据

```
List<Put> puts = new ArrayList<Put>();❶

Put put1 = new Put(Bytes.toBytes("row1"));
put1.add(Bytes.toBytes("colfam1"), Bytes.toBytes("qual1"),
    Bytes.toBytes("val1"));
puts.add(put1);❷

Put put2 = new Put(Bytes.toBytes("row2"));
put2.add(Bytes.toBytes("colfam1"), Bytes.toBytes("qual1"),
    Bytes.toBytes("val2"));
puts.add(put2);❸

Put put3 = new Put(Bytes.toBytes("row2"));
put3.add(Bytes.toBytes("colfam1"), Bytes.toBytes("qual2"),
    Bytes.toBytes("val3"));
puts.add(put3);❹

table.put(puts);❺
```

❶ 创建一个列表用于存储 Put 实例。

- ❷ 将一个 Put 实例添加到列表中。
- ❸ 将另外一个 Put 实例添加到列表中。
- ❹ 将第三个 Put 实例添加到列表中。
- ❺ 向 HBase 中存入多行多列数据。

用 HBase Shell 可以快速查看存入的数据是否与预期一致。请注意，例 3.4 实际上修改了三列，不过它们只属于两行。有两列内容存入了键为 row2 的行中，这两列使用了两个不同的列名，qual1 和 qual2，在同一行创建了两个不同名称的列。

```
hbase(main):001:0>scan 'testtable'
ROW                COLUMN+CELL
 row1              column=colfam1:qual1,timestamp=1300108258094,value=val1
 row2              column=colfam1:qual1,timestamp=1300108258094,value=val2
 row2              column=colfam1:qual2,timestamp=1300108258098,value=val3
2 row(s) in 0.1590 seconds
```

由于用户提交的修改行数据的列表可能涉及多行，所以有可能会有部分修改失败。造成修改失败的原因有很多，例如，一个远程的 region 服务器出现了问题，导致客户端的重试次数超过了配置的最大值，因此不得不放弃当前操作。如果远程服务器的 put 调用出现问题，错误会通过随后的一个 IOException 异常反馈给客户端。

例 3.5 使用了一个错误的（bogus）列族名来插入列。由于客户端不知道远程表的结构（可能在本次操作之前，实际的表结构已经有所变化），因此对列族的检查会在服务器端完成。

例 3.5 向 HBase 中插入一个错误的列族

```
Put put1 = new Put(Bytes.toBytes("row1"));
put1.add(Bytes.toBytes("colfam1"),Bytes.toBytes("qual1"),
    Bytes.toBytes("val1"));
puts.add(put1);
Put put2 = new Put(Bytes.toBytes("row2"));
put2.add(Bytes.toBytes("BOGUS"),Bytes.toBytes("qual1"),
    Bytes.toBytes("val2"));❶
puts.add(put2);
Put put3 = new Put(Bytes.toBytes("row2"));
put3.add(Bytes.toBytes("colfam1"),Bytes.toBytes("qual2"),
    Bytes.toBytes("val3"));
puts.add(put3);

table.put(puts);❷
```

- ❶ 将使用不存在列族的 Put 实例加入列表。

❷ 将多行多列数据存储到 HBase 中。

那个插入错误列族的 put() 调用失败，会返回如下的（或类似的）错误信息：

```
org.apache.hadoop.hbase.client.RetriesExhaustedWithDetailsException:  
Failed 1 action: NoSuchColumnFamilyException: 1 time,  
servers with issues: 10.0.0.57:51640,
```

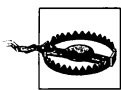
用户可能想知道列表中没有发生异常的 put 的情况如何。再次使用命令行工具，应该可以看见两个正确的 put 的数据已经被添加到 HBase 中了：

```
hbase(main):001:0>scan 'testtable'  
ROW          COLUMN+CELL  
  row1             column=colfam1:qual1,timestamp=1300108925848,value=val1  
  row2             column=colfam1:qual2,timestamp=1300108925848,value=val3  
2 row(s) in 0.0640 seconds
```

服务器遍历所有的操作并设法执行它们，失败的会返回，然后客户端会使用 RetriesExhaustedWithDetailsException 报告远程错误，这样用户可以查询有多少个操作失败、出错的原因以及重试的次数。要注意的是，对于错误列族，服务器端的重试次数会自动设为 1（见 NoSuchColumnFamilyException: 1 time），因为这是一个不可恢复的错误类型。

这些在服务器上失败的 Put 实例会被保存在本地写缓冲区中，下一次缓冲区刷写的时候会重试。用户也可以通过 HTable 的 getWriteBuffer() 方法访问它们，并对它们做一些处理，例如，清除操作。

有一些检查是在客户端完成的，例如，确认 Put 实例的内容是否为空或是否指定了列。在这种情况下，客户端会抛出异常，同时将出错的 Put 留在客户端缓冲区中不做处理。



调用基于列表的 put() 时，客户端会先把所有的 Put 实例插入到本地写缓冲区中，然后隐式地调用 flushCache()。在插入每个 Put 实例的时候，客户端 API 都会执行之前提到过的检查。如果检查失败，例如，5 个 Put 中的第 3 个失败了，则前两个会被添加到缓冲区中，最后两个则不会，同时也不会触发刷写命令。

用户可以捕获异常并手动刷写写缓冲区来执行已经添加的操作。例 3.6 展示了如何处理这个异常。

例 3.6 向 HBase 中插入一个空的 Put 实例

```
Put put1 = new Put(Bytes.toBytes("row1"));  
put1.add(Bytes.toBytes("colfam1"), Bytes.toBytes("qual1"),  
    Bytes.toBytes("val1"));  
puts.add(put1);
```

```

Put put2 = new Put(Bytes.toBytes("row2"));
put2.add(Bytes.toBytes("BOGUS"), Bytes.toBytes("qual1"),
    Bytes.toBytes("val2"));
puts.add(put2);
Put put3 = new Put(Bytes.toBytes("row2"));
put3.add(Bytes.toBytes("colfam1"), Bytes.toBytes("qual2"),
    Bytes.toBytes("val3"));
puts.add(put3);
Put put4 = new Put(Bytes.toBytes("row2"));
puts.add(put4); ❶

try {
    table.put(puts);
} catch(Exception e) {
    System.err.println("Error: " + e);
    table.flushCommits(); ❷
}

```

❶ 将没有内容的 Put 添加到列表中。

❷ 捕获本地异常然后提交更新。

这个例子会抛出两个异常，异常信息如下：

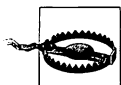
```

Error: java.lang.IllegalArgumentException: No columns to insert
Exception in thread "main"
    org.apache.hadoop.hbase.client.RetriesExhaustedWithDetailsException:
    Failed 1 action: NoSuchColumnFamilyException: 1 time,
    servers with issues: 10.0.0.57:51640,

```

第一个错误 (Error) 是客户端检查发现的，第二个错误是在 try/catch 代码块中调用下面的函数引起的远程异常：

```
table.flushCommits()
```



如果用户激活了客户端写缓冲区功能（请参考 3.2.1 节中“客户端的写缓冲区”）就会发现没有马上报告异常，而是延迟到了缓冲区刷写的时候才抛出。

当使用基于列表的 put 调用时，用户需要特别注意：用户无法控制服务器端执行 put 的顺序，这意味着服务器被调用的顺序也不受用户控制。如果要保证写入的顺序，需要小心地使用这个操作，最坏的情况是，要减少每一批量处理的操作数，并显示地刷写客户端写缓冲区，强制把操作发送到远程服务器。

5. 原子性操作 compare-and-set

有一种特别的 put 调用，其能保证自身操作的原子性：检查写（check and put）。该方法的签名如下：

```
boolean checkAndPut(byte[] row,byte[] family,byte[] qualifier,  
    byte[] value,Put put) throws IOException
```

有了这种带有检查功能的方法，就能保证服务器端 put 操作的原子性。如果检查成功通过，就执行 put 操作，否则就彻底放弃修改操作。这种方法可以用于需要检查现有相关值，并决定是否修改数据的操作。

这种有原子性保证的操作经常被用于账户结余、状态转换或数据处理等场景。这些应用场景的共同点是，在读取数据的同时需要处理数据。一旦你想把一个处理好的结果写回 HBase，并保证没有其他客户端已经做了同样的事情，你就可以使用这个有原子性保证的操作，先比较原值，再做修改。



有一种特别的检查通过 checkAndPut() 调用来完成，即只有在另外一个值不存在的情况下，才执行这个修改。要执行这种操作只需要将参数 value 设置为 null 即可，只要指定列不存在，就可以成功执行修改操作。

这个方法返回一个布尔类型的值，表示 put 操作成功执行还是失败，对应的值分别是 true 和 false。例 3.7 展示了客户端与服务器端不同操作返回值的交互过程。

例 3.7 使用原子性操作 compare-and-set

```
Put put1 = new Put(Bytes.toBytes("row1"));  
put1.add(Bytes.toBytes("colfam1"),Bytes.toBytes("qual1"),  
    Bytes.toBytes("val1"));❶  
  
boolean res1 = table.checkAndPut(Bytes.toBytes("row1"),  
    Bytes.toBytes("colfam1"),Bytes.toBytes("qual1"),null,put1);❷  
System.out.println("Put applied: " + res1);❸  
  
boolean res2 = table.checkAndPut(Bytes.toBytes("row1"),  
    Bytes.toBytes("colfam1"),Bytes.toBytes("qual1"),null,put1);❹  
System.out.println("Put applied: " + res2);❺  
  
Put put2 = new Put(Bytes.toBytes("row1"));  
put2.add(Bytes.toBytes("colfam1"),Bytes.toBytes("qual2"),  
    Bytes.toBytes("val2"));❻  
  
boolean res3 = table.checkAndPut(Bytes.toBytes("row1"),  
    Bytes.toBytes("colfam1"),Bytes.toBytes("qual1"),❷  
    Bytes.toBytes("val1"),put2);
```

```

System.out.println("Put applied: " + res3);❸

Put put3 = new Put(Bytes.toBytes("row2"));
put3.add(Bytes.toBytes("colfam1"),Bytes.toBytes("qual1"),
    Bytes.toBytes("val3"));❹

boolean res4 = table.checkAndPut(Bytes.toBytes("row1"),
    Bytes.toBytes("colfam1"),Bytes.toBytes("qual1"),❺
    Bytes.toBytes("val1"),put3);
System.out.println("Put applied: " + res4);❻

```

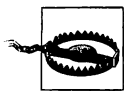
- ❶ 创建一个新的 Put 实例。
- ❷ 检查指定列是否存在，按检查的结果决定是否执行 put 操作。
- ❸ 输出结果，此处应为：“Put applied: true”。
- ❹ 再次向同一个单元格写入数据。
- ❺ 因为那个列的值已经存在，此时的输出结果应为 “Put applied: false”。
- ❻ 创建另一个新的 Put 实例，这次使用一个不同的列限定符。
- ❼ 当上一次的 put 值存在时，写入新的值。
- ❽ 因为已经存在，所以输出的结果应当为 “Put applied: true”。
- ❾ 再创建一个 Put 实例，这回使用一个不同的行键。
- ❿ 检查一个不同行的值是否相等，然后写入另一行。
- ⓫ 程序执行不到这里，因为在❿处会抛出异常。

例子中最后一次调用会抛出以下异常：

```

Exception in thread "main" org.apache.hadoop.hbase.DoNotRetryIOException:
    Action's getRow must match the passed row

```



HBase 提供的 **compare-and-set** 操作，只能检查和修改同一行数据。与其他的许多操作一样，这个操作只提供同一行数据的原子性保证。检查和修改分别针对不同行数据时会抛出异常。

compare-and-set（CAS）操作十分强大，尤其是在分布式系统中，且有多多个独立的客户端同时操作数据时。通过这个方法，HBase 与其他复杂的设计结构区分了开来，提供了使不同客户端可以并发修改数据的功能。

3.2.2 get 方法

下面我们将介绍从客户端 API 中获取已存储数据的方法。HTable 类中提供了 get() 方法，同时还有与之对应的 Get 类。get 方法分为两类：一类是一次获取一行数据；另一类是

一次获取多行数据。

1. 单行 get

这种方法可以从 HBase 表中取一个特定的值：

```
Result get(Get get) throws IOException
```

与 put() 方法有对应的 Put 类相似，get() 方法也有对应的 Get 类，此外还有一个相似之处，那就是在使用下面的方法构造 Get 实例时，也需要设置行键：

```
Get(byte[] row)
Get(byte[] row, RowLock rowLock)
```



虽然一次 get() 操作只能取一行数据，但不会限制一行当中取多少列或者多少单元格。

这两个 Get 实例都通过 row 参数指定了要获取的行，其中第二个 Get 实例还增加了一个可选的 rowLock 参数，允许用户设置行锁。与 put 操作一样，用户有许多方法可用，可以通过多种标准筛选目标数据，也可以指定精确的坐标获取某个单元格的数据：

```
Get addFamily(byte[] family)
Get addColumn(byte[] family, byte[] qualifier)
Get setTimeRange(long minStamp, long maxStamp) throws IOException
Get setTimeStamp(long timestamp)
Get setMaxVersions()
Get setMaxVersions(int maxVersions) throws IOException
```

addFamily() 方法限制 get 请求只能取得一个指定的列族，要取得多个列族时需要多次调用。addColumn() 的使用方式与之类似，用户通过它可以指定 get 取得哪一列的数据，从而进一步缩小地址空间。有一些方法允许用户设定要获取的数据的时间戳，或通过设定一个时间段来取得时间戳属于该时间段内的数据。

最后，如果用户没有设定时间戳的话，也有方法允许用户设定要获取的数据的版本数目。默认情况下，版本数为 1，即 get() 请求返回最新的匹配版本。如果有疑问，可以使用 getMaxVersions() 来检查这个 Get 实例所要取出的最大版本数。不带参数的 setMaxVersions() 方法会把要取出的最大版本数设为 Integer.MAX_VALUE，这是用户在列族描述符（column family descriptor）中可配置的最大版本数，此时系统会返回这个单元格中所有的版本，换句话说，此时系统会返回用户在列族中已配置的最大版本数以内的所有数据。

表 3-4 列出了 Get 类中其他方法的介绍。

表 3-4 Get 类提供的其他方法概览

方法	描述
<code>getRow()</code>	返回创建 Get 实例时指定的行键
<code>getRowLock()</code>	返回当前 Get 实例的 RowLock 实例
<code>getLockId()</code>	返回创建时指定 rowLock 的锁 ID，如果没有指定则返回 -1L
<code>getTimeRange()</code>	返回指定的 Get 实例的时间戳范围。注意，Get 类中已经没有了 <code>getTimeStamp()</code> 方法了，因为 API 会在内部将 <code>setTimeStamp()</code> 赋的值转换成 TimeRange 实例，设定给定时间戳的最大值和最小值
<code>setFilter()/getFilter()</code>	用户可以使用一个特定的过滤器实例，通过多种规则和条件来筛选列和单元格。使用这些方法，用户可以设定或查看 Get 实例的过滤器成员，详情参见 4.1 节
<code>setCacheBlocks()/getCacheBlocks()</code>	每个 HBase 的 region 服务器都有一个块缓存来有效地保存最近存取过的数据，并以此来加速之后的相邻信息的读取。不过在某些情况下，例如完全随机读取时，最好能避免这种机制带来的扰动。这些方法能够控制当次读取的块缓存机制是否启用
<code>numFamilies()</code>	快捷地获取列族 FamilyMap 大小的方法，包括用 <code>addFamily()</code> 方法和 <code>addColumn()</code> 方法添加的列族
<code>hasFamilies()</code>	检查列族或列是否存在于当前的 Get 实例中
<code>familySet()/getFamilyMap()</code>	这些方法能够让用户直接访问 <code>addFamily()</code> 和 <code>addColumn()</code> 添加的列族和列。FamilyMap 列族中键是列族的名称，键对应的值是指定列族的限定符列表。familySet() 方法返回一个所有已存储列族的 Set，即一个只包含列族名的集合



表 3-4 中所列的 getter 函数只能得到所属的 Get 实例中用户预先设定好的筛选条件。它们的应用场景很少，而且只能在类似如下的场景中使用，例如，用户的一个私有方法中有一个 Get 实例，需要在其他地方检查 Get 实例中各筛选条件是否正确。

以前提到过，HBase 为用户提供了 Bytes 这个工具类，该类有许多可以把 Java 的常用数据类型转化为 byte[] 数组的静态方法。同时，它也可以做一些反向转化的工作：例如当用户从 HBase 中取得一行数据时，可使用 Bytes 对应的方法把 byte[] 的内容转化为 Java 的数据类型。下面是它提供的相关方法的列表：

```
static String toString(byte[] b)
static boolean toBoolean(byte[] b)
static long toLong(byte[] bytes)
static float toFloat(byte[] bytes)
static int toInt(byte[] bytes)
...
```

例 3.8 展示了从 HBase 中获取数据的完整过程。

例 3.8 从 HBase 中获取数据的应用

```
Configuration conf = HBaseConfiguration.create();❶
HTable table = new HTable(conf,"testtable");❷
Get get = new Get(Bytes.toBytes("row1"));❸
get.addColumn(Bytes.toBytes("colfam1"),Bytes.toBytes("qual1"));❹
Result result = table.get(get);❺
byte[] val = result.getValue(Bytes.toBytes("colfam1"),
    Bytes.toBytes("qual1"));❻
System.out.println("Value: " + Bytes.toString(val));❼
```

- ❶ 创建配置实例。
- ❷ 初始化一个新的表引用。
- ❸ 使用一个指定的行键构建一个 Get 实例。
- ❹ 向 Get 实例中添加一个列。
- ❺ 从 HBase 中获取指定列的行数据。
- ❻ 从返回的结果中获取对应列的数据。
- ❼ 将数据转化为字符串打印输出。

如果用户在执行这个例子之前执行过前面的例子，如例 3.2，那么会有如下的输出结果：

```
Value: val1
```

虽然上面的这个输出结果很普通，但却展示了一次完整的读取数据的过程。这个例子只添加并取回了一个特定的列，取回的版本数为默认值 1。get() 方法调用后返回一个 Result 类的实例，这个类将在下面介绍。

2. Result 类

当用户使用 get() 方法获取数据时，HBase 返回的结果包含所有匹配的单元格数据，这些数据将被封装在一个 Result 实例中返回给用户。用它提供的方法，可以从服务器端获取匹配指定行的特定返回值，这些值包括列族、列限定符和时间戳等。

以下是一些获取特定返回值的工具方法，和前面使用的例 3.8 一样，可以设定一些具体的查询维度。如果用户之前要求服务器端返回一个列族下的所有列，现在就可以从返回值中取得这个列族下所需的任意列。换句话说，用户使用 get() 方法时需要提供一些具体的信息，以便数据取回之后客户端可以筛选出对应的数据。Result 类提供的方法如下：

```
byte[] getValue(byte[] family,byte[] qualifier)
byte[] value()
```

```

byte[] getRow()
int size()
boolean isEmpty()
KeyValue[] raw()
List<KeyValue> list()

```

`getValue()` 方法允许用户取得一个 **HBase** 中存储的特定单元格的值。因为该方法不能指定时间戳（或者说版本），所以用户只能获得数据最新的版本。`value()` 方法的使用更简单，它会返回第一个列对应的最新单元格的值。因为列在服务器端是按字典序存储的，所以会返回名称（包括列族和列限定符）排在首位的那一列的值。

之前我们介绍过 `getRow()` 方法：它返回创建 `Get` 类当前实例时使用的行键。`size()` 方法返回服务器端返回值中键值对（`KeyValue` 实例）的数目。用户可以使用 `size()` 方法或者 `isEmpty()` 方法查看键值对的数目是否大于 0，这样可以检查服务器端是否找到了与查询相对应的结果。

`raw()` 方法返回原始的底层 `KeyValue` 的数据结构，具体来说，是基于当前的 `Result` 实例返回 `KeyValue` 实例的数组。`list()` 调用则把 `raw()` 中返回的数组转化为一个 `List` 实例，并返回给用户，创建的 `List` 实例由原始返回结果中的 `KeyValue` 数组成员组成，用户可以方便地迭代存取数据。



`raw()` 方法返回的数组已经按字典序排列，排列时考虑了 `KeyValue` 实例的所有坐标。先按列族排序，列族内再按列限定符排序，此后再按时间戳排序，最后按类型排序。

另外还有一些面向列的存取函数如下：

```

List<KeyValue> getColumn(byte[] family,byte[] qualifier)
KeyValue getColumnLatest(byte[] family,byte[] qualifier)
boolean containsColumn(byte[] family,byte[] qualifier)

```

这个方法返回一个特定列的多个值，解答了前文提出的一个问题：如何获得一个列的多个版本。返回值中的版本数取决于用户调用 `get()` 方法之前，创建 `Get` 实例时设置的最大版本数，默认是 1。换句话说，`getColumn()` 返回的列表中包括 0（当本行没有该列值时）或 1 个条目，这一条目是该列最新版本的值。如果用户指定了一个比默认值 1 大的版本数（可以是最大值范围内的任意值），返回的列表中就可能会有多个条目。

`getColumnLatest()` 方法返回对应列的最新版本值，不过与 `getValue()` 不同，它不返回值的原始字节数组，而是返回一个 `KeyValue` 实例。如果用户需要的不仅仅是数据，那么这个方法将会非常有用。`containsColumn()` 是一个十分简便的方法，它检查返回值中是否有对应的列。



这些方法也可以不指定列限定符，即将列限定符设置为 `null`，这样方法会匹配没有列限定符的特殊列。

不使用限定符就意味着这一列没有标签。当查询表时，例如，用户通过命令行查询时，需要自己明白数据所表示的具体含义。可能只有一种情况能用到空限定符：即一个列族下只有一列，同时列族名就能够表示数据的含义及目的。

下面是第三类取值函数，以映射形式返回结果：

```
NavigableMap<byte[], NavigableMap<byte[],  
    NavigableMap<Long, byte[]>>> getMap()  
NavigableMap<byte[],  
    NavigableMap<byte[], byte[]>> getNoVersionMap()  
NavigableMap<byte[], byte[]> getFamilyMap(byte[] family)
```

最常用的方法是 `getMap()`，它把所有 `get()` 请求返回的内容都装入一个 Java 的 `Map` 类实例中，这样用户可以使用该方法遍历所有结果。`getNoVersionMap()` 与 `getMap()` 形式上相似，不过它只返回每个列的最新版本。`getFamilyMap()` 允许用户指定一个特定的列族，返回这次结果中这个列族下的所有版本。

不论用户使用什么方法获取 `Result` 中的数据，都不会产生额外的性能和资源消耗，因为这些数据都已经通过网络从服务器端传输到了客户端。

转储内容

所有的 Java 对象都有 `toString()` 方法，这个方法通常会被重载，用于将实例数据转化为文本表示。这样做一般不是为了序列化对象，而是为了方便调试程序。

同样 `Result` 类也有 `toString()` 方法的实现，它把实例的内容转储为一个可读的字符串，下面是输出的例子：

```
keyvalues={row-2/colfam1:col-5/1300802024293/Put/vlen=7,  
    row-2/colfam2:col-33/1300802024325/Put/vlen=8}
```

这个方法只是简单地打印实例所包含的 `KeyValue` 实例，也就是逐个调用 `KeyValue.toString()` 方法。如果 `Result` 的实例为空，则返回结果如下：

```
keyvalues=NONE
```

这种情况表示查询没有 `KeyValue` 实例返回。本书的代码示例使用 `toString()` 方法来打印之前读取操作的结果。

3. Get 列表

使用列表参数的 `get()` 方法与使用列表参数的 `put()` 方法对应，用户可以用一次请求获取多行数据。它允许用户快速高效地从远程服务器获取相关的或完全随机的多行数据。



如图 3-1 所示，实际上，请求有可能被发往多个不同的服务器，但这部分逻辑已经被封装起来，因此对于客户端代码来说，还是表现为一次请求。

API 提供的方法签名如下：

```
Result[] get(List<Get> gets) throws IOException
```

这个方法含义十分直白，跟之前介绍的类似方法一样：用户需要创建一个列表，并把之前准备好的 `Get` 实例添加到其中。然后将这个列表传给 `get()`，会返回一个与列表大小相等的 `Result` 数组。例 3.9 展示了用两种方法获取数据的整个流程。

例 3.9 使用 `Get` 实例的列表从 HBase 中获取数据

```
byte[] cf1 = Bytes.toBytes("colfam1");
byte[] qf1 = Bytes.toBytes("qual1");
byte[] qf2 = Bytes.toBytes("qual2");❶
byte[] row1 = Bytes.toBytes("row1");
byte[] row2 = Bytes.toBytes("row2");

List<Get> gets = new ArrayList<Get>();❷

Get get1 = new Get(row1);
get1.addColumn(cf1, qf1);
gets.add(get1);

Get get2 = new Get(row2);
get2.addColumn(cf1, qf1);❸
gets.add(get2);

Get get3 = new Get(row2);
get3.addColumn(cf1, qf2);
gets.add(get3);

Result[] results = table.get(gets);❹

System.out.println("First iteration...");
for(Result result : results){
    String row = Bytes.toString(result.getRow());
    System.out.print("Row: " + row + " ");
    byte[] val = null;
    if(result.containsColumn(cf1, qf1)){❺
        val = result.getValue(cf1, qf1);
```

```

        System.out.println("Value: " + Bytes.toString(val));
    }
    if(result.containsColumn(cf1,qf2)){
        val = result.getValue(cf1,qf2);
        System.out.println("Value: " + Bytes.toString(val));
    }
}

System.out.println("Second iteration...");
for(Result result : results){
    for(KeyValue kv : result.raw()){
        System.out.println("Row: " + Bytes.toString(kv.getRow()) + ❶
            " Value: " + Bytes.toString(kv.getValue()));
    }
}

```

- ❶ 准备好共用的字节数组。
- ❷ 准备好要存放 Get 实例的列表。
- ❸ 将 Get 实例添加到列表中。
- ❹ 从 HBase 中获取这些行和选定的列。
- ❺ 遍历结果并检查哪些行中包含选定的列。
- ❻ 再次遍历，打印所有结果。

如果先运行例 3.4，然后再运行例 3.9，可能会看到如下结果：

```

First iteration...
Row: row1 Value: val1
Row: row2 Value: val2
Row: row2 Value: val3
Second iteration...
Row: row1 Value: val1
Row: row2 Value: val2
Row: row2 Value: val3

```

两次遍历返回了同样的值，说明从服务器端得到结果之后，用户可以有多种方式访问结果。现在就差查询时出现的异常如何反馈没有介绍了。get() 返回异常的方法与 3.2.1 节中的“Put 列表”不同。get() 方法要么返回与给定列表大小一致的 Result 数组，要么抛出一个异常。例 3.10 展示了这个行为。

例 3.10 尝试读取一个错误的列族

```

List<Get> gets = new ArrayList<Get>();

Get get1 = new Get(row1);
get1.addColumn(cf1,qf1);

```

```

gets.add(get1);

Get get2 = new Get(row2);
get2.addColumn(cf1,qf1);❶
gets.add(get2);

Get get3 = new Get(row2);
get3.addColumn(cf1,qf2);
gets.add(get3);

Get get4 = new Get(row2);
get4.addColumn(Bytes.toBytes("BOGUS"),qf2);
gets.add(get4);❷

Result[] results = table.get(gets);❸
System.out.println("Result count: " + results.length);❹

```

- ❶ 将 Get 实例添加到列表中。
- ❷ 添加包含有错的（bogus）列族的 Get。
- ❸ 抛出异常，操作停止。
- ❹ 不会执行到此行。

执行这个例子会导致整个 get() 操作终止，程序会抛出类似下面这样的异常，并且没有返回值：

```

org.apache.hadoop.hbase.client.RetriesExhaustedWithDetailsException:
Failed 1 action: NoSuchColumnFamilyException: 1 time,
servers with issues: 10.0.0.57:51640,

```

对于批量操作中的局部错误，有一种更为精细的处理方法，即使用 batch() 方法，这部分内容将在 3.3 节详细介绍。

4. 获取数据的相关方法

还有一些方法可以用来获取或检查存储的数据，第一个是：

```
boolean exists(Get get) throws IOException
```

可以和使用 HTable 的 get() 方法一样，先创建一个 Get 类的实例。exists() 方法通过 RPC 验证请求的数据是否存在，但不会从远程服务器返回请求的数据，只返回一个布尔值表示这个结果。

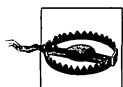


exists() 方法会引发 region 服务器端查询数据的操作，包括加载文件块来检查某行或某列是否存在。用户通过这种方法只能避免网络数据传输的开销，不过在需要检查或频繁检查一个比较大的列时，这种方法还是十分实用的。

某些情况下，用户在检索数据时可能需要查找一个特定的行，或者某个请求行之前的一行。下面的方法可以帮助用户实现这种查找：

```
Result getRowOrBefore(byte[] row,byte[] family) throws IOException
```

用户需要指定要查找的行键和列族。指定后者的原因是，HBase 是一个列式存储的数据库，不存在没有列的行数据。设定列族之后，服务器端会检查要查找的那一行里是否有任何属于指定列族的列值。



请注意，在使用 `getRowOrBefore()` 方法时，需要指定一个已经存在的列族，否则服务端会因为要访问一个不存在的存储文件而抛出一个 Java 的 `NullPointerException` 异常。

可以从 `getRowOrBefore()` 返回的 `Result` 实例中得到要查找的行键。这个行键要么与用户设定的行一致，要么刚好是设定行键之前的一行。如果没有匹配的结果，本方法返回 `null`。

例 3.11 使用 `getRowOrBefore()` 方法查找用户之前使用 `put` 示例存入的数据。

例 3.11 使用特殊检索方法

```
Result result1 = table.getRowOrBefore(Bytes.toBytes("row1"), ❶  
    Bytes.toBytes("colfam1"));  
System.out.println("Found: " + Bytes.toString(result1.getRow())); ❷  
  
Result result2 = table.getRowOrBefore(Bytes.toBytes("row99"), ❸  
    Bytes.toBytes("colfam1"));  
System.out.println("Found: " + Bytes.toString(result2.getRow())); ❹  
  
for(KeyValue kv : result2.raw()){  
    System.out.println(" Col: " + Bytes.toString(kv.getFamily())+ ❺  
        "/" + Bytes.toString(kv.getQualifier())+  
        ",Value: " + Bytes.toString(kv.getValue()));  
}  
  
Result result3 = table.getRowOrBefore(Bytes.toBytes("abc"), ❻  
    Bytes.toBytes("colfam1"));  
System.out.println("Found: " + result3); ❼
```

- ❶ 尝试查找已经存在的行。
- ❷ 打印查找结果。
- ❸ 尝试查找不存在的行。
- ❹ 返回已排好序的表中的最后一条结果。
- ❺ 打印返回结果。
- ❻ 尝试查找测试行之前的一行。
- ❼ 由于没有匹配的结果，返回 `null`。

假如已经执行过例 3.4，那上面的代码将会输出如下结果：

```
Found: row1
Found: row2
  Col: colfam1/qual1,Value: val2
  Col: colfam1/qual2,Value: val3
Found: null
```

第一次调用找到一个匹配的行，成功返回。第二次调用使用了一个大数字作为后缀来查找表的最后一行。从 row-前缀开始，查找到对应的 row-2 行。最后一个例子要查找 abc 这一行或这行之前的一行，不过之前插入数据的前缀都是 row-，所以 abc 以及之前的行不存在。因此返回值为 null，表示查找失败。

令人感兴趣的是，这个循环打印出了与匹配条件的行一起返回的数据。返回了指定列族的所有列，包括这些列的最新版本。用户可以使用这种方法快速取回特定排序规则下一个列族中所有列的最新值。例如，假设像 Put 示例一样，所有的行键都使用 row- 作为前缀，调用 `getRowOrBefore()` 时送入 row-999999999 作为 row 参数，返回的结果将总是按字典序排在表尾的那一行。

3.2.3 删除方法

此前介绍了 HBase 表的创建、读取和更新，就剩如何从表中删除数据没讲了。HTable 提供了删除的方法，同时与之前的方法一样有一个相应的类命名为 Delete。

1. 单行删除

`delete()` 方法有许多变体，其中一个只需要一个 Delete 实例：

```
void delete(Delete delete) throws IOException
```

与前面讲过的 `get()` 方法和 `put()` 方法一样，用户必须先创建一个 Delete 实例，然后再添加你想要删除的数据的详细信息。构造函数是：

```
Delete(byte[] row)
Delete(byte[] row,long timestamp,RowLock rowLock)
```

用户需要提供要修改的行，如果要多次频繁地修改同一行的话，还可以提供 rowLock 参数 (RowLock 类的一个实例)，以指定自己的 RowLock。此外，最好缩小要删除的给定行中涉及数据的范围，可使用下列方法：

```
Delete deleteFamily(byte[] family)
Delete deleteFamily(byte[] family,long timestamp)
Delete deleteColumns(byte[] family,byte[] qualifier)
Delete deleteColumns(byte[] family,byte[] qualifier,long timestamp)
Delete deleteColumn(byte[] family,byte[] qualifier)
Delete deleteColumn(byte[] family,byte[] qualifier,long timestamp)
void setTimestamp(long timestamp)
```

有 4 种调用可缩小删除所涉及的数据范围。首先，用户可以使用 `deleteFamily()` 方法来删除一整个列族，包括其下所有的列。用户也可以指定一个时间戳，触发针对单元格数据版本的过滤，从所有的列中删除与这个时间戳相匹配的版本和比这个时间戳旧

的版本。

另一种方法是 `deleteColumns()`，它作用于特定的一列，如果用户没有指定时间戳，这个方法会删除该列的所有版本，如果用户指定了时间戳，这个方法会删除所有与这个时间戳相匹配的版本和更旧的版本。

第三种方法与第二种类似，使用 `deleteColumn()`，它也操作一个具体的列，但是只删除最新的版本或者指定的版本，即用一个精确匹配的时间戳执行删除操作。

最后一个方法是 `setTimestamp()`，这个方法在调用其他 3 种方法时经常被忽略。但是，如果不指定列族或列，则此调用与删除整行不同，它会删除匹配时间戳的或者比给定时间戳旧的所有列族中的所有列。表 3-5 以表格的形式展示了 `delete()` 的功能，可读性更强。

表 3-5 `delete()` 功能表

方法	无时间戳的删除	有时间戳的删除
<code>none</code>	删除整行，即所有列的所有版本	从所有列族的所有列中删除与给定时间戳相同或更旧的版本
<code>deleteColumn()</code>	只删除给定列的最新版本，保留旧版本	只删除与时间戳匹配的给定列的指定版本，如果不存在，则不删除
<code>deleteColumns()</code>	删除给定列的所有版本	从给定列中删除与给定时间戳相等或更旧的版本
<code>deleteFamily()</code>	删除给定列族中的所有列（包括所有版本）	从给定列族下的所有列中删除与给定时间戳相等或更旧的版本

表 3-6 列举了 `Delete` 类提供的其他方法，以供用户查阅。

表 3-6 `Delete` 类提供的其他方法概览

方法	描述
<code>getRow()</code>	返回创建 <code>Delete</code> 实例时指定的行键
<code>getRowLock()</code>	返回当前 <code>Delete</code> 实例的 <code>RowLock</code> 实例
<code>getLockId()</code>	返回使用 <code>raulk</code> 参数创建实例时可选参数锁 ID 的值，如果没有指定则返回 <code>-1L</code>
<code>getTimeStamp()</code>	检索 <code>Delete</code> 实例相关的时间戳
<code>isEmpty()</code>	检查 <code>FamilyMap</code> 是否含有任何条目，即用户所指定的想要删除的列族或者列
<code>getFamilyMap()</code>	这个方法可以获取用户通过 <code>deleteFamily()</code> 以及 <code>deleteColumn()</code> / <code>deleteColumns()</code> 添加的要删除的列和列族，返回的 <code>FamilyMap</code> 是使用列族名作为键，它的值是当前列族下要删除的列限定符的列表

例 3.12 展示了怎样在客户端代码中调用 `delete()` 函数。

例 3.12 从 HBase 中删除数据的应用示例

```
Delete delete = new Delete(Bytes.toBytes("row1"));❶

delete.setTimestamp(1);❷

delete.deleteColumn(Bytes.toBytes("colfam1"), Bytes.toBytes("qual1"), 1);❸

delete.deleteColumns(Bytes.toBytes("colfam2"), Bytes.toBytes("qual1"));❹
delete.deleteColumns(Bytes.toBytes("colfam2"), Bytes.toBytes("qual3"), 15);❺

delete.deleteFamily(Bytes.toBytes("colfam3"));❻
delete.deleteFamily(Bytes.toBytes("colfam3"), 3);❼

table.delete(delete);❽

table.close();
```

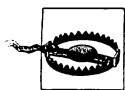
- ❶ 创建针对特定行的 `Delete` 实例。
- ❷ 设置时间戳。
- ❸ 删除一行中的特定版本。
- ❹ 删除一行中的所有版本。
- ❺ 删除一行中的给定版本和所有更旧的版本。
- ❻ 删除整个列族，包括所有的列和版本。
- ❼ 删除给定列族中所有列的给定版本和所有更旧的版本。
- ❽ 从 `HBase` 表中删除数据。

这个例子列举出了用户通过设定不同参数操作 `delete()` 方法的方法。像这样一个接着一个调用没有太大实际意义，读者可以随意注释掉一些删除调用，观察控制台上显示结果的变化。

删除操作所设定的时间戳只对匹配的单元格有影响，即匹配给定时间戳的列和值。另一方面，如果不设定时间戳，服务器会强制检索服务器端最新的时间戳，这比执行一个具有明确时间戳的删除要慢。

如果尝试删除未设置时间戳的单元格，什么都不会发生。例如，某一列有两个版本，版本 10 和版本 20，删除版本 15 将不会影响现存的任何版本。

这个例子同时展示了用户自定义数据版本的使用。它使用从 1 开始自增的序号，不依靠隐式或显式的时间戳。这种方式非常有用，用户必须按需求自己设置版本，因为服务器并不知道客户端的使用模式，只会使用 `Unix` 时间戳来代替。



就本文而言，不建议用户自定义版本号。自定义版本号可能会起作用，但是没有经过很好的测试。请确保使用这项技术之前仔细评估过你的选择。

使用自定义版本的另一个例子可以在 9.4 节中找到。

2. Delete 的列表

基于列表的 `delete()` 调用与基于列表的 `put()` 调用非常相似，需要创建一个包含 `Delete` 实例的列表，对其进行配置，并调用下面的方法：

```
void delete(List<Delete> deletes) throws IOException
```

例 3.13 展示了影响三个不同行的删除操作，删除了它们所包含的各种细节。当运行这个例子时，你会看到打印输出的删除前后的状态，还能看到使用 `KeyValue.toString()` 打印输出的原始的 `KeyValue` 实例。



正如其他基于列表的操作，用户不能对删除操作在远程服务器上的执行顺序做任何假设。API 会重新排列它们，并将同一个 `region` 服务器的操作集中到一个 RPC 请求中以提升性能。如果需要确保执行的顺序，用户需要把这些调用分成更小的组，并控制组之间的执行顺序。在最坏的情况下，需要发送单独的 `Delete` 调用以保证顺序。

例 3.13 删除值列表的应用示例

```
List<Delete> deletes = new ArrayList<Delete>();❶

Delete delete1 = new Delete(Bytes.toBytes("row1"));
delete1.setTimestamp(4);❷
deletes.add(delete1);

Delete delete2 = new Delete(Bytes.toBytes("row2"));
delete2.deleteColumn(Bytes.toBytes("colfam1"), Bytes.toBytes("qual1"));❸
delete2.deleteColumns(Bytes.toBytes("colfam2"), Bytes.toBytes("qual3"), 5);❹
deletes.add(delete2);

Delete delete3 = new Delete(Bytes.toBytes("row3"));
delete3.deleteFamily(Bytes.toBytes("colfam1"));❺
delete3.deleteFamily(Bytes.toBytes("colfam2"), 3);❻
deletes.add(delete3);

table.delete(deletes);❼

table.close();
```

- ❶ 创建一个列表，保存 Delete 实例。
- ❷ 为删除行的 Delete 实例设置时间戳。
- ❸ 删除一列的最新版本。
- ❹ 在另一列中删除给定的版本及所有更旧的版本。
- ❺ 删除整个列族，包括所有的列和版本。
- ❻ 在整个列族中，删除给定的版本以及所有更旧的版本。
- ❼ 删除 HBase 表中的多行。

你会看到如下输出^❶：

```
Before delete call...
KV: row1/colfam1:qual1/2/Put/vlen=4,Value: val2
KV: row1/colfam1:qual1/1/Put/vlen=4,Value: val1
KV: row1/colfam1:qual2/4/Put/vlen=4,Value: val4
KV: row1/colfam1:qual2/3/Put/vlen=4,Value: val3
KV: row1/colfam1:qual3/6/Put/vlen=4,Value: val6
KV: row1/colfam1:qual3/5/Put/vlen=4,Value: val5

KV: row1/colfam2:qual1/2/Put/vlen=4,Value: val2
KV: row1/colfam2:qual1/1/Put/vlen=4,Value: val1
KV: row1/colfam2:qual2/4/Put/vlen=4,Value: val4
KV: row1/colfam2:qual2/3/Put/vlen=4,Value: val3
KV: row1/colfam2:qual3/6/Put/vlen=4,Value: val6
KV: row1/colfam2:qual3/5/Put/vlen=4,Value: val5

KV: row2/colfam1:qual1/2/Put/vlen=4,Value: val2
KV: row2/colfam1:qual1/1/Put/vlen=4,Value: val1
KV: row2/colfam1:qual2/4/Put/vlen=4,Value: val4
KV: row2/colfam1:qual2/3/Put/vlen=4,Value: val3
KV: row2/colfam1:qual3/6/Put/vlen=4,Value: val6
KV: row2/colfam1:qual3/5/Put/vlen=4,Value: val5

KV: row2/colfam2:qual1/2/Put/vlen=4,Value: val2
KV: row2/colfam2:qual1/1/Put/vlen=4,Value: val1
KV: row2/colfam2:qual2/4/Put/vlen=4,Value: val4
KV: row2/colfam2:qual2/3/Put/vlen=4,Value: val3
KV: row2/colfam2:qual3/6/Put/vlen=4,Value: val6
KV: row2/colfam2:qual3/5/Put/vlen=4,Value: val5

KV: row3/colfam1:qual1/2/Put/vlen=4,Value: val2
KV: row3/colfam1:qual1/1/Put/vlen=4,Value: val1
KV: row3/colfam1:qual2/4/Put/vlen=4,Value: val4
```

❶ 为了方便阅读，相关细节被分为多组，组之间用空行隔开。

```
KV: row3/colfam1:qual2/3/Put/vlen=4,Value: val3
KV: row3/colfam1:qual3/6/Put/vlen=4,Value: val6
KV: row3/colfam1:qual3/5/Put/vlen=4,Value: val5
```

```
KV: row3/colfam2:qual1/2/Put/vlen=4,Value: val2
KV: row3/colfam2:qual1/1/Put/vlen=4,Value: val1
KV: row3/colfam2:qual2/4/Put/vlen=4,Value: val4
KV: row3/colfam2:qual2/3/Put/vlen=4,Value: val3
KV: row3/colfam2:qual3/6/Put/vlen=4,Value: val6
KV: row3/colfam2:qual3/5/Put/vlen=4,Value: val5
```

After delete call...

```
KV: row1/colfam1:qual3/6/Put/vlen=4,Value: val6
KV: row1/colfam1:qual3/5/Put/vlen=4,Value: val5
```

```
KV: row1/colfam2:qual3/6/Put/vlen=4,Value: val6
KV: row1/colfam2:qual3/5/Put/vlen=4,Value: val5
```

```
KV: row2/colfam1:qual1/1/Put/vlen=4,Value: val1
KV: row2/colfam1:qual2/4/Put/vlen=4,Value: val4
KV: row2/colfam1:qual2/3/Put/vlen=4,Value: val3
KV: row2/colfam1:qual3/6/Put/vlen=4,Value: val6
KV: row2/colfam1:qual3/5/Put/vlen=4,Value: val5
```

```
KV: row2/colfam2:qual1/2/Put/vlen=4,Value: val2
KV: row2/colfam2:qual1/1/Put/vlen=4,Value: val1
KV: row2/colfam2:qual2/4/Put/vlen=4,Value: val4
KV: row2/colfam2:qual2/3/Put/vlen=4,Value: val3
KV: row2/colfam2:qual3/6/Put/vlen=4,Value: val6
```

```
KV: row3/colfam2:qual2/4/Put/vlen=4,Value: val4
KV: row3/colfam2:qual3/6/Put/vlen=4,Value: val6
KV: row3/colfam2:qual3/5/Put/vlen=4,Value: val5
```

“Before delete call...” 中突出显示（粗体）的部分是将要被删除的原始数据。这 3 行包含同样的数据，由两个列族组成，每个列族下有 3 列，每个列有两个版本。

示例代码先删除了整行数据中版本小于等于 4 的数据，这次操作留下了版本为 5 和 6 的数据。

之后，使用两个指定了列的删除操作，先后清除了 row2 上 colfam1:qual1 列的最新单元格，以及 colfam1:qual3 中小于等于 5 的所有单元格。这两个删除操作都只有一个单元格满足条件，它们将依次被删除。

最后，在 row-3 上，示例代码删除了列族 colfam1 下的全部数据，然后又删除了 colfam2 中版本小于等于 3 的所有数据。在示例代码执行的过程中，使用以下的方法可以看到 KeyValue 的详细情况：

```
System.out.println("KV: " + kv.toString() +  
    ",Value: " + Bytes.toString(kv.getValue()))
```

现在，我们熟悉了 Bytes 类的使用，可以用它打印由 getValue() 返回的 KeyValue 实例的值。这样做是有必要的，因为 KeyValue.toString() 方法（见 3.2.1 节）无法打印出实际的值，只能打印出关键的部分。toString() 无法打印出值的原因是，值的内容可能非常大。

示例代码插入的列值非常短，并且内容是可读的，所以在控制台上把结果打印出来是安全的。用户也可以在调试时使用类似的方法。

请参阅本书的源代码库，那里有例子中全部的源码。用户可以从其中查看数据如何插入，并最终形成前面示例代码中的输出。

最后要介绍的是基于列表的 delete() 操作的异常处理。下面对传入的 deletes 参数（即 Delete 实例的列表）做一下修改，使得在调用返回时，还有一个错误的 Delete 实例。换句话说，如果所有的操作都成功了，这个列表会为空。但是，如果最后还有一个实例的话，远程服务器会报告这个错误，这个调用也要抛出异常。用户需要用 try/catch 语句捕获异常，并做相应处理。例 3.14 是一个简单的示范。

例 3.14 从 HBase 中删除错误数据

```
Delete delete4 = new Delete(Bytes.toBytes("row2"));  
delete4.deleteColumn(Bytes.toBytes("BOGUS"),Bytes.toBytes("qual1"));❶  
deletes.add(delete4);  
  
try {  
    table.delete(deletes);❷  
} catch(Exception e){  
    System.err.println("Error: " + e);❸  
}  
table.close();  
  
System.out.println("Deletes length: " + deletes.size());❹  
for(Delete delete : deletes){  
    System.out.println(delete);❺  
}
```

- ❶ 添加一个错误的列族来触发错误。
- ❷ 从 HBase 表中删除多行数据。
- ❸ 捕获远程异常。
- ❹ 检查调用之后列表的长度。
- ❺ 把失败的 delete 操作打印出来用于调试。

这个例子修改了例 3.13，添加了一个出错的删除细节，即添加了一个出错的（BOGUS）列族。输出结果与例 3.13 相似，只是中间有一些其他的信息：

```
Before delete call...
KV: row1/colfam1:qual1/2/Put/vlen=4,Value: val2
KV: row1/colfam1:qual1/1/Put/vlen=4,Value: val1
...
KV: row3/colfam2:qual3/6/Put/vlen=4,Value: val6
KV: row3/colfam2:qual3/5/Put/vlen=4,Value: val5

Error: org.apache.hadoop.hbase.client.RetriesExhaustedWithDetailsException:
  Failed 1 action: NoSuchColumnFamilyException: 1 time,
  servers with issues: 10.0.0.43:59057,
Deletes length: 1
row=row2, ts=9223372036854775807,families={(family=BOGUS,keyvalues=\
(row2/BOGUS:qual1/9223372036854775807/Delete/vlen=0)}

After delete call...
KV: row1/colfam1:qual3/6/Put/vlen=4,Value: val6
KV: row1/colfam1:qual3/5/Put/vlen=4,Value: val5
...
KV: row3/colfam2:qual3/6/Put/vlen=4,Value: val6
KV: row3/colfam2:qual3/5/Put/vlen=4,Value: val5
```

如预期的一样，列表中还剩下一个 Delete 实例，就是包含错误列族的那个。打印这个实例（Java 默认使用 `toString()` 方法来打印一个对象），结果展示了这个出错的实例的内部细节。失败的主要原因是，列族名明显是错误的。读者可以用这个方法检查一个操作出错的原因，出错原因通常都十分明显。

最后，注意一下例子中 `catch` 语句捕获的异常 `RetriesExhaustedWithDetailsException`，它在之前的例子中已经出现过两次了。这个异常报告出错的操作个数，报告重试的次数及对应的服务器。更进一步的处理方法包括检查和监控服务器，这些细节将在后面的章节中提到，这里返回的出错的服务器地址可以帮助我们定位错误的根源。

3. 原子性操作 `compare-and-delete`

前文已经在“原子性操作 `compare-and-set`”一节介绍过，如何使用原子性的条件操作向表中插入数据。有一个相似的删除操作，提供了让用户可以在服务器端读取并修改（`read-and-modify`）的功能：

```
boolean checkAndDelete(byte[] row,byte[] family,byte[] qualifier,
    byte[] value,Delete delete) throws IOException
```

用户必须指定行键、列族、列限定符和值来执行删除操作之前的检查。如果检查失败，则不执行删除操作，调用返回 `false`。如果检查成功，则执行删除操作，调用返回 `true`。例 3.15 展示了这里介绍的内容。

例 3.15 使用原子操作 compare-and-delete 删除值的应用示例

```
Delete delete1 = new Delete(Bytes.toBytes("row1"));
delete1.deleteColumns(Bytes.toBytes("colfam1"), Bytes.toBytes("qual3")); ❶

boolean res1 = table.checkAndDelete(Bytes.toBytes("row1"),
    Bytes.toBytes("colfam2"), Bytes.toBytes("qual3"), null, delete1); ❷
System.out.println("Delete successful: " + res1); ❸

Delete delete2 = new Delete(Bytes.toBytes("row1"));
delete2.deleteColumns(Bytes.toBytes("colfam2"), Bytes.toBytes("qual3")); ❹
table.delete(delete2);

boolean res2 = table.checkAndDelete(Bytes.toBytes("row1"),
    Bytes.toBytes("colfam2"), Bytes.toBytes("qual3"), null, delete1); ❺
System.out.println("Delete successful: " + res2); ❻

Delete delete3 = new Delete(Bytes.toBytes("row2"));
delete3.deleteFamily(Bytes.toBytes("colfam1")); ❼

try{
    boolean res4 = table.checkAndDelete(Bytes.toBytes("row1"),
        Bytes.toBytes("colfam1"), Bytes.toBytes("qual1"), ❽
        Bytes.toBytes("val1"), delete3);
    System.out.println("Delete successful: " + res4); ❾
} catch(Exception e){
    System.err.println("Error: " + e);
}
```

- ❶ 创建一个 Delete 实例。
- ❷ 检查指定列是否不存在，依检查结果执行删除操作。
- ❸ 打印结果，结果应当为 “Delete successful: false”。
- ❹ 手工删除已经检查过的列。
- ❺ 尝试再一次删除同一个单元格。
- ❻ 打印结果，应当为 “Delete successful: true”，因为这个列之前存在所以成功删除。
- ❼ 创建另一个 Delete 实例，这次使用一个不同的行。
- ❽ 检查这个不同的行，并执行删除操作。
- ❾ 执行不到这里，在此行之前有异常抛出。

示例的全部输出如下：

```

Before delete call...
KV: row1/colfam1:qual1/2/Put/vlen=4,Value: val2
KV: row1/colfam1:qual1/1/Put/vlen=4,Value: val1
KV: row1/colfam1:qual2/4/Put/vlen=4,Value: val4
KV: row1/colfam1:qual2/3/Put/vlen=4,Value: val3
KV: row1/colfam1:qual3/6/Put/vlen=4,Value: val6
KV: row1/colfam1:qual3/5/Put/vlen=4,Value: val5
KV: row1/colfam2:qual1/2/Put/vlen=4,Value: val2
KV: row1/colfam2:qual1/1/Put/vlen=4,Value: val1
KV: row1/colfam2:qual2/4/Put/vlen=4,Value: val4
KV: row1/colfam2:qual2/3/Put/vlen=4,Value: val3
KV: row1/colfam2:qual3/6/Put/vlen=4,Value: val6
KV: row1/colfam2:qual3/5/Put/vlen=4,Value: val5
Delete successful: false
Delete successful: true
After delete call...
KV: row1/colfam1:qual1/2/Put/vlen=4,Value: val2
KV: row1/colfam1:qual1/1/Put/vlen=4,Value: val1
KV: row1/colfam1:qual2/4/Put/vlen=4,Value: val4
KV: row1/colfam1:qual2/3/Put/vlen=4,Value: val3
KV: row1/colfam2:qual1/2/Put/vlen=4,Value: val2
KV: row1/colfam2:qual1/1/Put/vlen=4,Value: val1
KV: row1/colfam2:qual2/4/Put/vlen=4,Value: val4
KV: row1/colfam2:qual2/3/Put/vlen=4,Value: val3
Error: org.apache.hadoop.hbase.DoNotRetryIOException:
    org.apache.hadoop.hbase.DoNotRetryIOException:
        Action's getRow must match the passed row
...

```

将 null 作为 value 参数的传入值，将会触发一次“不存在”（nonexistence）检查，即所指定的列只要不存在，检查就会成功。因为上面这个例子在检查之前插入了检查过的列，所以第一次检查会失败，调用会返回 false 并放弃删除操作。

之后这一列被手工删除，第二次执行检查并修改（check-and-modify）操作。这次检查成功，删除了数据，最后返回 true。

跟之前关于 Put 的 CAS 调用一样，用户只能对同一行数据进行检查并修改。例子中检查的行键与 Delete 实例自身的行键不同，所以在执行检查时会相应地抛出异常。这个方法允许用户做跨列族检查，例如，用户可以使用一组列控制筛选其他列。

这个例子还不足以证明检查并删除（check-and-delete）操作的重要性。在分布式系统中，很难可靠地完成这种操作，并且很难避免由外部锁带来的性能上的损失。换句话说，如果原子性由客户端保证，那么就要对整行数据加排他锁。如果在已加锁的情况下客户端崩溃，那么服务器端必须通过超时机制来对数据解锁。这样也需要额外的 RPC 请求，这比一次服务器端的操作要慢很多。

3.3 批量处理操作

现在我们已经介绍过添加、检索和删除表中数据的操作了，不过前面介绍的操作都是基于单个实例或基于列表的操作。这一节将会介绍一些 API 调用，这些调用可以批量处理跨多行的不同操作。

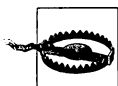


事实上，许多基于列表的操作，如 `delete(List<Delete> deletes)` 或者 `get(List<Get> gets)`，都是基于 `batch()` 方法实现的。它们都是一些为了方便用户使用而保留的方法。如果你是新手，推荐使用 `batch()` 方法进行所有操作。

下面的客户端 API 方法提供了批量处理操作。用户可能注意到这里引入了一个新的名为 `Row` 的类，它是 `Put`、`Get` 和 `Delete` 的祖先，或者说是父类。

```
void batch(List<Row> actions, Object[] results)
    throws IOException, InterruptedException
Object[] batch(List<Row> actions)
    throws IOException, InterruptedException
```

使用同样的父类允许在列表中实现多态，即放入以上 3 种不同的子类。这种调用跟之前介绍的基于列表的调用方法一样简单易用。下面的例 3.16 展示了如何把不同的操作融合为一个服务器端调用。



请注意，不可以将针对同一行的 `Put` 和 `Delete` 操作放在同一个批量处理请求中。为了保证最好的性能，这些操作的处理顺序可能不同，但是这样会产生不可预料的结果。由于资源竞争，某些情况下，用户会看到波动的结果。

例 3.16 使用批量处理操作的应用示例

```
private final static byte[] ROW1 = Bytes.toBytes("row1");
private final static byte[] ROW2 = Bytes.toBytes("row2");
private final static byte[] COLFAM1 = Bytes.toBytes("colfam1");❶
private final static byte[] COLFAM2 = Bytes.toBytes("colfam2");
private final static byte[] QUAL1 = Bytes.toBytes("qual1");
private final static byte[] QUAL2 = Bytes.toBytes("qual2");

List<Row> batch = new ArrayList<Row>();❷

Put put = new Put(ROW2);
put.add(COLFAM2, QUAL1, Bytes.toBytes("val5"));❸
```

```

batch.add(put);

Get get1 = new Get(ROW1);
get1.addColumn(COLFAM1,QUAL1);❶
batch.add(get1);

Delete delete = new Delete(ROW1);
delete.deleteColumns(COLFAM1,QUAL2);❷
batch.add(delete);

Get get2 = new Get(ROW2);
get2.addFamily(Bytes.toBytes("BOGUS"));❸
batch.add(get2);

Object[] results = new Object[batch.size()];❹
try {
    table.batch(batch,results);
} catch(Exception e){
    System.err.println("Error: " + e);❺
}

for(int i = 0;i < results.length;i++){
    System.out.println("Result[" + i + "]: " + results[i]);❻
}

```

- ❶ 使用常量可以方便重用。
- ❷ 创建列表存放所有操作。
- ❸ 添加一个 Put 实例。
- ❹ 添加一个针对不同行的 Get 实例。
- ❺ 添加一个 Delete 实例。
- ❻ 添加一个会失败的 Get 实例。
- ❼ 创建一个结果数组。
- ❽ 打印捕获的异常。
- ❾ 打印所有结果。

从控制台上可以看到以下结果：

```

Before batch call...
KV: row1/colfam1:qual1/1/Put/vlen=4,Value: val1
KV: row1/colfam1:qual2/2/Put/vlen=4,Value: val2
KV: row1/colfam1:qual3/3/Put/vlen=4,Value: val3

Result[0]: keyvalues=NONE

```

```
Result[1]: keyvalues={row1/colfam1:qual1/1/Put/vlen=4}
Result[2]: keyvalues=NONE
Result[3]: org.apache.hadoop.hbase.regionserver.NoSuchColumnFamilyException:
    org.apache.hadoop.hbase.regionserver.NoSuchColumnFamilyException:
        Column family BOGUS does not exist in ...
```

```
After batch call...
KV: row1/colfam1:qual1/1/Put/vlen=4,Value: val1
KV: row1/colfam1:qual3/3/Put/vlen=4,Value: val3
KV: row2/colfam2:qual1/1308836506340/Put/vlen=4,Value: val5
```

```
Error: org.apache.hadoop.hbase.client.RetriesExhaustedWithDetailsException:
    Failed 1 action: NoSuchColumnFamilyException: 1 time,
    servers with issues: 10.0.0.43:60020,
```


与之前的例子一样，在执行批量处理之前，由于插入了测试行的数据，因此先打印了测试行的相关输出。首先输出的是表的原内容，然后是示例代码产生的输出，最后输出的是操作以后的表的内容。由输出结果可见，要删除的列被删除了，新添加的列也成功添加了。

Get 操作的结果需要观察输出结果的中间部分，即示例代码产生的输出。那些以 Result[n] (n 从 0 到 3) 开头的输出是 actions 参数中对应操作的结果。示例中第一个操作是 Put，对应的结果是一个空的 Result 实例，其中没有 KeyValue 实例。这是批量处理调用返回值的通常规则，它们给每个输入操作返回一个最佳匹配的结果，可能的返回值如表 3-7 所示。

表 3-7 batch() 调用可能的返回结果

结果	描述
null	操作与远程服务器的通信失败
EmptyResult	Put 和 Delete 操作成功后的返回结果
Result	Get 操作成功的返回结果，如果没有匹配的行或列，会返回空的 Result
Throwable	当服务器端返回一个异常时，这个异常会按原样返回给客户端。用户可以使用这个异常检查哪里出了错，也许可以在自己的代码中自动处理异常

更进一步地观察控制台上输出的返回结果数组，你会发现空的 Result 实例打印出来是：keyvalues=NONE。Get 请求成功找到了相匹配的结果，返回一个对应的 KeyValue 实例。最后，有一个 BOGUS 列族的操作请求并返回一个异常供用户参考。



当用户使用 batch() 功能时，Put 实例不会被客户端写入缓冲区缓冲。batch() 请求是同步的，会把操作直接发送到服务器端，这个过程没有什么延迟或其他中间操作。这与 put() 调用明显不同，所以请慎重挑选需要的方法。

有两种不同的批量处理操作看起来非常相似。不同之处在于，一个需要用户输入包含返回结果的 `Object` 数组，而另一个由函数帮助用户创建这个数组。为什么需要两个方法呢？它们的语义有什么不同吗（如果有不同的话）？它们都可能抛出之前见到过的 `RetriesExhaustedWithDetailsException`，所以关键的不同在于：

```
void batch(List<Row> actions, Object[] results)
    throws IOException, InterruptedException
```

上面这个方法让用户可以访问部分结果，而下面这个方法不行：

```
Object[] batch(List<Row> actions)
    throws IOException, InterruptedException
```

后面这个方法如果抛出异常的话，不会有任何返回结果，因为新结果数组返回之前，控制流就中断了。

而之前的方法会先向用户提供的数组中填充数据，然后再抛出异常。例 3.16 的代码就采用了前一种方法，并提交了结果数组。下面是一些 `batch()` 方法特性的汇总。

两种方法的共同点

`get`、`put` 和 `delete` 都支持。如果执行时出现问题，客户端将抛出异常并报告问题。它们都不使用客户端写缓冲区。

```
void batch(actions, results)
```

能够访问成功操作的结果，同时也可以获取远程失败时的异常。

```
Object[] batch(actions)
```

只返回客户端异常，不能访问程序执行中的部分结果。



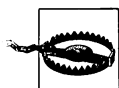
在检查结果之前，所有的批量处理操作都被执行了：即使用户收到一个操作的异常，其他操作也都已经执行了。不过，在最坏的情况下，可能所有操作都会返回异常。

另外，批量处理可以感知暂时性错误，例如 `NotServingRegionException`（表明一个 `region` 已经被移动）会多次重试这个操作。用户可以通过调整 `hbase.client.retries.number` 配置项（默认是 10）来增加或减少重试次数。

3.4 行锁

像 `put()`、`delete()`、`checkAndPut()` 这样的修改操作是独立执行的，这意味着在一个串行方式的执行中，对于每一行必须保证行级别的操作是原子性的。`region` 服务器提供了

一个行锁 (row lock) 的特性, 这个特性保证了只有一个客户端能获取一行数据相应的锁, 同时对该行进行修改。在实践中, 大部分客户端应用程序都没有提供显式的锁, 而是使用这个机制来保障每个操作的独立性。



用户应该尽可能地避免使用行锁。就像在 RDBMS 中, 两个客户端很可能在拥有对方要请求的锁时, 又同时请求对方已拥有的锁, 这样便形成了一个死锁。

锁超时之前, 两个被阻塞的客户端会占用一个服务器端的处理线程 (handler), 而这个线程是一种十分稀缺的资源。如果在一个频繁操作的行上发生了这种情况, 那么很多其他的客户端会占用掉其所有的处理线程, 阻塞所有其他客户端访问这台服务器, 导致这个 region 服务器将不能为其负责的 region 内的行提供服务。

重申一下: 在不必要的情况下, 尽量不要使用行锁。如果必须使用, 那么一定要节约占用锁的时间!

比如, 当使用 put() 访问服务器时, Put 实例可以通过以下构造函数生成:

```
Put(byte[] row)
```

这个构造函数就没有 RowLock 实例参数, 所以服务器会在调用期间创建一个锁。实际上, 通过客户端的 API, 得不到这个生存期短暂的服务器端的锁的实例。

除了服务器端隐式加锁之外, 客户端也可以显式地对单行数据的多次操作进行加锁, 通过以下调用便可以做到:

```
RowLock lockRow(byte[] row) throws IOException  
void unlockRow(RowLock rl) throws IOException
```

第一个调用 lockRow() 需要一个行键作为参数, 返回一个 RowLock 的实例, 这个实例可以供后续的 Put 或者 Delete 的构造函数使用。一旦不再需要锁时, 必须通过 unlockRow() 调用来释放它。

每一个排他锁 (unique lock), 无论是由服务器提供的, 还是通过客户端 API 传入的, 都能保护这一行不被其他锁锁定。换句话说, 锁必须针对整个行, 并且指定其行键, 一旦它获得锁定权就能防止其他的并发修改。

当一个锁被服务器端或客户端显式获取之后, 其他所有想要对这行数据加锁的客户端将会等待, 直到当前锁被释放, 或者锁的租期超时。后者是为了确保错误进程不会占用锁太长时间或无限期占用。



默认的锁超时时间是一分钟，但是可以在 *hbase-site.xml* 文件中添加以下配置项来修改这个默认值，时间以毫秒为单位：

```
<property>
  <name>hbase.regionserver.lease.period</name>
  <value>120000</value>
</property>
```

通过添加以上代码，超时时间被设置为原来的两倍——120 秒也就是 2 分钟。小心不要将这个值设得太大，因为每一个想获取被锁住的行的客户端都会阻塞并等待锁的恢复。

例 3.17 展示了如何在行上创建一个锁，该锁阻塞所有的并发读取。

例 3.17 显式使用行锁

```
static class UnlockedPut implements Runnable { ❶
    @Override
    public void run() {
        try {
            HTable table = new HTable(conf, "testtable");
            Put put = new Put(ROW1);
            put.add(COLFAM1, QUAL1, VAL3);
            long time = System.currentTimeMillis();
            System.out.println("Thread trying to put same row now...");
            table.put(put); ❷
            System.out.println("Wait time: " +
                (System.currentTimeMillis() - time) + "ms");
        } catch (IOException e) {
            System.err.println("Thread error: " + e);
        }
    }
}

System.out.println("Taking out lock...");
RowLock lock = table.lockRow(ROW1); ❸
System.out.println("Lock ID: " + lock.getLockId());

Thread thread = new Thread(new UnlockedPut()); ❹
thread.start();

try {
    System.out.println("Sleeping 5secs in main()..."); ❺
    Thread.sleep(5000);
} catch (InterruptedException e) {
    // ignore
}

try {
```



```

Put put1 = new Put(ROW1,lock);❶
put1.add(COLFAM1,QUAL1,VAL1);
table.put(put1);

Put put2 = new Put(ROW1,lock);❷
put2.add(COLFAM1,QUAL1,VAL2);
table.put(put2);
} catch(Exception e){
    System.err.println("Error: " + e);
} finally {
    System.out.println("Releasing lock...");❸
    table.unlockRow(lock);
}

```

- ❶ 使用一个异步的线程更新同一个行，但是不显式加锁。
- ❷ put() 调用会阻塞，直到锁被释放。
- ❸ 给整行加锁。
- ❹ 启动那个会阻塞的异步线程。
- ❺ 休眠一会儿，以阻塞其他写入操作。
- ❻ 在拥有锁的情况下创建 Put。
- ❼ 在拥有锁的情况下创建另外一个 Put。
- ❽ 释放锁，让阻塞线程继续执行。

执行这个例子代码时，应该能在控制台看到以下输出：

```

Taking out lock...
Lock ID: 4751274798057238718
Sleeping 5secs in main()...
Thread trying to put same row now...
Releasing lock...
Wait time: 5007ms
After thread ended...
KV: row1/colfam1:qual1/1300775520118/Put/vlen=4,Value: val2
KV: row1/colfam1:qual1/1300775520113/Put/vlen=4,Value: val1
KV: row1/colfam1:qual1/1300775515116/Put/vlen=4,Value: val3

```

从这个例子能看出，一个显示的锁是如何阻塞另一个使用隐式锁的线程的。主线程休眠了 5 秒，一醒来就调用了两次 put()，分别将同一列设置为两个不同的数值。

主线程的锁一释放，阻塞线程的 run() 方法就继续执行并调用了第三个 put。观察 put 操作在服务器端的执行情况，会觉得很有意思。读者可能注意到了，KeyValue 实例的时间戳显示第三个 put 拥有最小的时间戳，虽然这个 put 表面上是最后执行的。这是因为线程中的 put() 调用是在两个主线程中的 put() 之前执行的，这之后主

线程休眠了 5 秒。当 put 被发送到服务器时，如果它的时间戳没有被显式指定，服务器端会帮它设定时间戳，同时试图获得这一行的锁。但是示例代码中主线程已经获得了该行的锁，因此服务器端的处理一直等待了 5 秒多，锁被释放才得以继续。从上面的输出可以看出，主线程中两个 put 调用的执行以及行的解锁只花费了 7 毫秒的时间。

Get 需要锁吗？

修改行时锁定行是有意义的，那么获取数据时是否需要加锁呢？Get 类有一个构造器允许用户指定一个显式的锁：

```
Get(byte[] row, RowLock rowLock)
```

这是遗留的方法，但服务器端根本用不着这种方法，因为在获取数据的过程中，服务器根本不需要任何锁，而是应用了一个多版本的并发控制（*multiversion concurrency control-style*）^①机制来保证行级读操作。例如，get()调用永远不会返回写了一半的数据，比如当这些数据是另一个线程或者客户端写的。

这个就像是小规模的事务系统：只有当一个变动被应用到整个行之后，客户端才能读出这个改动。当改动在进行中时，所有的客户端读取操作得到的都将是所有列以前的状态。

当用户试图使用之前申请的显式锁，但锁的租约已经超时并恢复，用户将会从服务器得到一个以 UnknownRowLockException 形式报告的错误。这个异常告诉用户服务器已经废弃了用户尝试使用的锁。用户应该在代码中丢弃这个锁，然后请求一个新的锁再试图恢复锁定状态。

3.5 扫描

在讨论过基本的 CRUD 类型的操作之后，现在来看一下扫描（scan）技术，这种技术类似于数据库系统中的游标（cursor），并利用到了 HBase 提供的底层顺序存储的数据结构。^②

3.5.1 介绍

扫描操作的使用跟 get() 方法非常类似。同样，和其他函数类似，这里也提供了 Scan 类。但是由于扫描操作的工作方式类似于迭代器，所以用户无需调用 scan() 方法创建

① 见链接 http://en.wikipedia.org/wiki/Multiversion_concurrency_control。

② 扫描操作与不可回滚的游标操作相似。用户需要先声明，然后打开，并获取数据，最后关闭数据库游标。不过扫描操作没有声明这一步，否则扫描操作和游标操作的使用方法就是一样的了。参见维基百科中的 Cursors。

实例，只需调用 HTable 的 `getScanner()` 方法，此方法在返回真正的扫描器（`scanner`）实例的同时，用户也可以使用它迭代获取数据。可用方法如下：

```
ResultScanner getScanner(Scan scan) throws IOException
ResultScanner getScanner(byte[] family) throws IOException
ResultScanner getScanner(byte[] family, byte[] qualifier)
              throws IOException
```

后两个为了方便用户，隐式地帮用户创建了一个 `Scan` 实例，逻辑中最后调用 `getScanner(Scan scan)` 方法。

`Scan` 类拥有以下构造器：

```
Scan()
Scan(byte[] startRow, Filter filter)
Scan(byte[] startRow)
Scan(byte[] startRow, byte[] stopRow)
```

这与 `Get` 类的不同点是显而易见的：用户可以选择性地提供 `startRow` 参数，来定义扫描读取 `HBase` 表的起始行键，即行键不是必须指定的。同时可选 `stopRow` 参数用来限定读取到何处停止。



起始行包括在内，而终止行是不包括在内的。一般用区间表示法表示为 `[startRow, stopRow)`。

扫描操作有一个特点：用户提供的参数不必精确匹配这两行。扫描会匹配相等或大于给定的起始行的行键。如果没有显式地指定起始行，它会从表的起始位置开始获取数据。

当遇到了与设置的终止行相同或大于终止行的行键时，扫描也会停止。如果没有指定终止行键，会扫描到表尾。

另一个可选参数叫做过滤器（`filter`），可直接指向 `Filter` 实例。尽管 `Scan` 实例通常由空白构造器构造，但其所有可选参数都有对应的 `getter` 方法和 `setter` 方法。

创建 `Scan` 实例之后，用户可能还要给它增加更多限制条件。这种情况下，用户仍然可以使用空白参数的扫描，它可以读取整个表格，包括所有列族以及它们的所有列。可以用多种方法限制所要读取的数据：

```
Scan addFamily(byte [] family)
Scan addColumn(byte[] family, byte[] qualifier)
```

这里有很多与 `Get` 类相似的功能：可以使用 `addFamily()` 方法限制返回数据的列族，或者通过 `addColumn()` 方法限制返回的列。



如果用户只需要数据的子集，那么限制扫描的范围就能发挥 HBase 的优势。因为 HBase 中的数据是按列族存储的，如果扫描不读取某个列族，那么整个列族文件就都不会被读取，这就是列式存储架构的优势。

```
Scan setTimeRange(long minStamp,long maxStamp) throws IOException
Scan setTimeStamp(long timestamp)
Scan setMaxVersions()
Scan setMaxVersions(int maxVersions)
```

用户可以通过 `setTimeStamp()` 设置详细的时间戳，或者通过 `setTimeRange()` 设置时间范围，进一步对结果进行限制。还可以使用 `setMaxVersions()` 方法，让扫描只返回每一列的一些特定版本，或者全部的版本。

```
Scan setStartRow(byte[] startRow)
Scan setStopRow(byte[] stopRow)
Scan setFilter(Filter filter)
boolean hasFilter()
```

还可以使用 `setStartRow()`、`setStopRow()` 以及 `setFilter()`，进一步限定返回的数据。这 3 个方法中的参数可以与构造器中的一样。附加的 `hasFilter()` 方法可以检查是否已经设定过滤器。

还有一些相关的方法，见表 3-8。

表 3-8 Scan 类的其他方法概览

方法	描述
<code>getStartRow()/getStopRow()</code>	查询当前设定的值
<code>getTimeRange()</code>	检索 Get 实例指定的时间范围或相关时间戳。注意，当需要指定单个时间戳时，API 会在内部通过 <code>setTimeStamp()</code> 将 <code>TimeRange</code> 实例的起止时间戳设为传入值，所以 <code>Get</code> 类中此时已经没有 <code>getTimeStamp()</code> 方法了
<code>getMaxVersions()</code>	返回当前配置下应该从表中获取的每列的版本数
<code>getFilter()</code>	可以使用特定的过滤器实例，通过多种规则来筛选列和单元格。使用这个方法，用户可以设定或查看 <code>Scan</code> 实例的过滤器成员。没有设置的话则返回 <code>null</code> ，详情参见 4.1 节
<code>setCacheBlocks()/getCacheBlocks()</code>	每个 HBase 的 <code>region</code> 服务器都有一个块缓存，可以有效地保存最近访问过的数据，并以此来加速之后相邻信息的读取。不过在某些情况下，例如全表扫描，最好能避免这种机制带来的扰动。这个方法能够控制本次读取的块缓存机制是否启效
<code>numFamilies()</code>	快捷地获取 <code>FamilyMap</code> 大小的方法，包括用 <code>addFamily()</code> 和 <code>addColumn()</code> 方法添加的列族和列

方法	描述
<code>hasFamilies()</code>	检查是否有添加过列族或列
<code>getFamilies()</code> <code>setFamilyMap()</code> <code>getFamilyMap()</code>	这些方法能够让用户直接访问 <code>addFamily()</code> 和 <code>addColumn()</code> 添加的列族和列。 FamilyMap 中键是列族的名称，键对应的值是特定列族下列限定符的列表。 <code>getFamilies()</code> 方法返回一个只包含列族名的数组

一旦设置好了 Scan 实例，就可以调用 Htable 的 `getScanner()` 方法，获得用于检索数据的 `ResultScanner` 实例。我们将在下一节中详细讨论这个类。

3.5.2 ResultScanner 类

扫描操作不会通过一次 RPC 请求返回所有匹配的行，而是以行为单位进行返回。很明显，行的数目很大，可能有上千条甚至更多，同时在一次请求中发送大量数据，会占用大量的系统资源并消耗很长时间。

`ResultScanner` 把扫描操作转换为类似的 `get` 操作，它将每一行数据封装成一个 `Result` 实例，并将所有的 `Result` 实例放入一个迭代器中。`ResultScanner` 的一些方法如下：

```
Result next() throws IOException
Result[] next(int nbRows) throws IOException
void close()
```

有两种类型的 `next()` 调用供用户选择。调用 `close()` 方法会释放所有由扫描控制的资源。

扫描器租约

要确保尽早释放扫描器实例，一个打开的扫描器会占用不少服务端资源，累积多了会占用大量的堆空间。当使用完 `ResultScanner` 之后应调用它的 `close()` 方法，同时应当把 `close()` 方法放到 `try/finally` 块中，以保证其在迭代获取数据过程中出现异常和错误时，仍然能执行 `close()`。

注意为了简洁，示例代码并未遵循这个建议。

就像行锁一样，扫描器也使用同样的租约超时机制，保护其不被失效的客户端阻塞太久。用户可以使用修改锁租约处提到的那个配置属性来修改超时时间（单位为毫秒）：

```
<property>
  <name>hbase.regionserver.lease.period</name>
  <value>120000</value>
</property>
```

用户需要确保该属性值适当，这个值要同时适用于锁租约和扫描器租约。

`next()` 调用返回一个单独的 `Result` 实例，这个实例代表了下一个可用的行。此外，用

户可以使用 `next(int nbRows)` 一次获取多行数据，它返回一个数组，数组中包含的 `Result` 实例最多可达 `nbRows` 个，每个实例代表唯一的一行。当用户扫描到表尾或到终止行时，由于没有足够的行来填充数据，返回的结果数组可能会小于既定长度。有关如何使用 `Result` 实例的问题请参阅前面介绍的“`Result` 类”，更详细的内容请参阅 3.2.2 节。

例 3.18 集中使用了之前解释过的功能，扫描了一张表，逐行处理了其中的列数据。

例 3.18 使用扫描器获取表中数据

```
Scan scan1 = new Scan();❶
ResultScanner scanner1 = table.getScanner(scan1);❷
for(Result res : scanner1){
    System.out.println(res);❸
}
scanner1.close();❹

Scan scan2 = new Scan();
scan2.addFamily(Bytes.toBytes("colfam1"));❺
ResultScanner scanner2 = table.getScanner(scan2);
for(Result res : scanner2){
    System.out.println(res);
}
scanner2.close();

Scan scan3 = new Scan();
scan3.addColumn(Bytes.toBytes("colfam1"), Bytes.toBytes("col-5")).
    addColumn(Bytes.toBytes("colfam2"), Bytes.toBytes("col-33"));❻
scan3.setStartRow(Bytes.toBytes("row-10"));
scan3.setStopRow(Bytes.toBytes("row-20"));
ResultScanner scanner3 = table.getScanner(scan3);
for(Result res : scanner3){
    System.out.println(res);
}
scanner3.close();
```

- ❶ 创建一个空的 `Scan` 实例。
- ❷ 取得一个扫描器迭代访问所有的行。
- ❸ 打印行内容。
- ❹ 关闭扫描器释放远程资源。
- ❺ 只添加一个列族，这样可以禁止获取“`colfam2`”的数据。
- ❻ 使用 `builder` 模式将详细限制条件添加到 `Scan` 中。

代码插入了 100 行数据，每行有两个列族，每个列族下包含 100 个列。第一个扫描操作扫描全表内容，第二个扫描操作只扫描一个列族，最后一个扫描操作有严

格的限制条件，其中包括对行范围的限制，同时还要求只扫描两个特定的列。输出如下：

```
Scanning table #3...
keyvalues={row-10/colfam1:col-5/1300803775078/Put/vlen=8,
           row-10/colfam2:col-33/1300803775099/Put/vlen=9}
keyvalues={row-100/colfam1:col-5/1300803780079/Put/vlen=9,
           row-100/colfam2:col-33/1300803780095/Put/vlen=10}
keyvalues={row-11/colfam1:col-5/1300803775152/Put/vlen=8,
           row-11/colfam2:col-33/1300803775170/Put/vlen=9}
keyvalues={row-12/colfam1:col-5/1300803775212/Put/vlen=8,
           row-12/colfam2:col-33/1300803775246/Put/vlen=9}
keyvalues={row-13/colfam1:col-5/1300803775345/Put/vlen=8,
           row-13/colfam2:col-33/1300803775376/Put/vlen=9}
keyvalues={row-14/colfam1:col-5/1300803775479/Put/vlen=8,
           row-14/colfam2:col-33/1300803775498/Put/vlen=9}
keyvalues={row-15/colfam1:col-5/1300803775554/Put/vlen=8,
           row-15/colfam2:col-33/1300803775582/Put/vlen=9}
keyvalues={row-16/colfam1:col-5/1300803775665/Put/vlen=8,
           row-16/colfam2:col-33/1300803775687/Put/vlen=9}
keyvalues={row-17/colfam1:col-5/1300803775734/Put/vlen=8,
           row-17/colfam2:col-33/1300803775748/Put/vlen=9}
keyvalues={row-18/colfam1:col-5/1300803775791/Put/vlen=8,
           row-18/colfam2:col-33/1300803775805/Put/vlen=9}
keyvalues={row-19/colfam1:col-5/1300803775843/Put/vlen=8,
           row-19/colfam2:col-33/1300803775859/Put/vlen=9}
keyvalues={row-2/colfam1:col-5/1300803774463/Put/vlen=7,
           row-2/colfam2:col-33/1300803774485/Put/vlen=8}
```

再强调一次，匹配的行键都是按词典序排列的，这使得结果非常有趣。用户可以简单地用 0 把行键补齐，这样扫描出来结果顺序更有可读性。这些都是在你的控制下完成的，所以请仔细设计行键。

3.5.3 缓存与批量处理

到目前为止，每一个 `next()` 调用都会为每行数据生成一个单独的 RPC 请求，即使使用 `next(int nbRows)` 方法，也是如此，因为该方法仅仅是在客户端循环地调用 `next()` 方法。很显然，当单元格数据较小时，这样做的性能不会很好（参见 3.2.1 节中“客户端的写缓冲区”的讨论）。因此，如果一次 RPC 请求可以获取多行数据，这样会更有意义。这样的方法可以由扫描器缓存（scanner caching）实现，默认情况下，这个缓存是关闭的。

可以在两个层面上打开它：在表的层面，这个表所有扫描实例的缓存都会生效；也可以在扫描层面，这样便只会影响当前的扫描实例。用户可以使用以下的 `HTable` 方法设置表级的扫描器缓存：

```
void setScannerCaching(int scannerCaching)
int getScannerCaching()
```



用户可以修改整个 HBase 集群的默认值 1。只要把下面的配置项添加到 *hbase-site.xml* 中即可：

```
<property>
  <name>hbase.client.scanner.caching</name>
  <value>10</value>
</property>
```

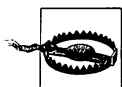
这样所有 Scan 实例的扫描器缓存大小就都被设置为 10 了。用户还可以从表或扫描两个层面覆盖默认配置，但是需要明确这样做的目的。

`setScannerCaching()` 可以设置缓存大小，`getScannerCaching()` 可以返回当前缓存大小的值。每次用户调用 `getScanner(scan)` 之后，API 都会把设定值配置到扫描实例中——除非用户使用了扫描层面的配置并覆盖了表层面的配置，扫描层面的配置优先级最高。可以使用下列 Scan 类的方法设置扫描级的缓存：

```
void setCaching(int caching)
int getCaching()
```

这两个方法的作用和表层面的方法一样，能控制每次 RPC 调用取回的行数。两种 `next()` 方法都会受这些配置影响。

用户需要为少量的 RPC 请求次数和客户端以及服务器端的内存消耗找到平衡点。很多时候，将扫描器缓存设得比较高能提高扫描的性能，不过设得太高就会产生不良影响：每次 `next()` 调用将会占用更长的时间，因为要获取更多的文件并传输到客户端，如果返回给客户端的数据超出了其堆的大小，程序就会终止并抛出 `OutOfMemoryException` 异常。



当传输和处理数据的时间超过配置的扫描器租约超时时间时，用户将会收到一个以 `ScannerTimeoutException` 形式抛出的租约过期（`lease expired`）错误。

例 3.19 展示了扫描器超时的情况。

例 3.19 使用扫描器时超时

```
Scan scan = new Scan();
ResultScanner scanner = table.getScanner(scan);

int scannerTimeout = (int)conf.getLong(
    HConstants.HBASE_REGIONSERVER_LEASE_PERIOD_KEY, -1);❶
try {
    Thread.sleep(scannerTimeout + 5000);❷
} catch (InterruptedException e) {
    // ignore
}
```



```

while(true){
    try {
        Result result = scanner.next();
        if(result == null)break;
        System.out.println(result);❸
    } catch(Exception e){
        e.printStackTrace();
        break;
    }
}
scanner.close();

```

- ❶ 得到当前配置的租约超时时间。
- ❷ 休眠的时间比租约超时时间再长一点。
- ❸ 打印行的内容。

这段代码得到当前配置的租约时间，休眠了比这个时间更长的时间，然后服务器端感知租约超时并触发租约恢复操作。控制台输出的结果与如下结果类似（为了方便阅读做了精简）：

```

Adding rows to table...
Current(local)lease period: 60000
Sleeping now for 65000ms...
Attempting to iterate over scanner...
Exception in thread "main" java.lang.RuntimeException:
    org.apache.hadoop.hbase.client.ScannerTimeoutException: 65094ms passed
        since the last invocation,timeout is currently set to 60000
        at org.apache.hadoop.hbase.client.HTable$ClientScanner$1.hasNext
        at ScanTimeoutExample.main
Caused by: org.apache.hadoop.hbase.client.ScannerTimeoutException: 65094ms
        passed since the last invocation,timeout is currently set to 60000
        at org.apache.hadoop.hbase.client.HTable$ClientScanner.next
        at org.apache.hadoop.hbase.client.HTable$ClientScanner$1.hasNext
        ... 1 more
Caused by: org.apache.hadoop.hbase.UnknownScannerException:
    org.apache.hadoop.hbase.UnknownScannerException: Name: -315058406354472427
        at org.apache.hadoop.hbase.regionserver.HRegionServer.next
...

```

示例代码打印了执行的进度，在休眠一定的时间之后，尝试迭代获取扫描器提供的行。由于租约超时，这个操作触发服务器端超时异常，同时返回的异常信息中还包括了当前配置的超时时间。



用户可能会尝试向配置中添加如下信息:

```
Configuration conf = HBaseConfiguration.create()
conf.setLong(HConstants.HBASE_REGIONSERVER_LEASE_PERIOD_KEY, 120000)
```

假设这个修改把超时时间延长了(在这个例子里,延长到了2分钟)。由于这个值是在客户端应用中配置的,不会被传递到远程 region 服务器,所以这样的修改是无效的。

如果用户要修改之前讨论的超时时间,用户必须修改服务器端(region 服务器)的配置文件 *hbase-site.xml*,修改完之后别忘了重启服务器使配置生效!

从上面打印出的堆栈追踪信息中还可以看出: `ScannerTimeoutException` 异常是如何包装在 `UnknownScannerException` 异常的外面抛出的。以上信息表明扫描器的 `next()` 方法使用扫描器 ID 在服务器端查找已经建立的扫描器,但由于这个扫描器 ID 的租约超时,已经被删除了。换句话说,客户端缓存的扫描器 ID 在 region 服务器上已经查找不到了,这与这个异常名称所表达的含义相符。

到目前为止,我们已经介绍了如何使用客户端的扫描器缓存来从远程 region 服务器向客户端整批传输数据。不过还有之前提到过的一件事需要注意:数据量非常大的行,这些行有可能超过客户端进程的内存容量。HBase 和它的客户端 API 对这个问题有一个解决方法:批量。用户可以使用以下方法控制批量获取操作:

```
void setBatch(int batch)
int getBatch()
```

缓存是面向行一级的操作,而批量则是面向列一级的操作。批量可以让用户选择每一次 `ResultScanner` 实例的 `next()` 操作要取回多少列。例如,在扫描中设置 `setBatch(5)`,则一次 `next()` 返回的 `Result` 实例会包括 5 列。



如果一行包括的列数超过了批量中设置的值,则可以将这一行分片,每次 `next` 操作返回一片。

当一行的列数不能被批量中设置的值整除时,最后一次返回的 `Result` 实例会包含比较少的列,例如,如果一行有 17 列,用户把 `batch` 值设为 5,则一共会返回 4 个 `Result` 实例,这 4 个实例中包括的列数应当分别为 5、5、5 和 2。

组合使用扫描器缓存和批量大小,可以让用户方便地控制扫描一个范围内的行键时所需要的 RPC 调用次数。例 3.20 为了控制 RPC 请求的次数,使用了这两个参数来调节每次 `Result` 实例的大小。

例 3.20 在扫描中使用缓存和批量参数

```
private static void scan(int caching,int batch)throws IOException {
    Logger log = Logger.getLogger("org.apache.hadoop");
    final int[] counters = {0,0};
    Appender appender = new AppenderSkeleton() {
        @Override
        protected void append(LoggingEvent event){
            String msg = event.getMessage().toString();
            if(msg != null && msg.contains("Call: next")){
                counters[0]++;
            }
        }
        @Override
        public void close() {}
        @Override
        public boolean requiresLayout() {
            return false;
        }
    };
    log.removeAllAppenders();
    log.setAdditivity(false);
    log.addAppender(appender);
    log.setLevel(Level.DEBUG);

    Scan scan = new Scan();
    scan.setCaching(caching); ❶
    scan.setBatch(batch);
    ResultScanner scanner = table.getScanner(scan);
    for(Result result : scanner){
        counters[1]++; ❷
    }
    scanner.close();
    System.out.println("Caching: " + caching + ",Batch: " + batch +
        ",Results: " + counters[1] + ",RPCs: " + counters[0]);
}

public static void main(String[] args) throws IOException {
    scan(1,1);
    scan(200,1);
    scan(2000,100); ❸
    scan(2,100);
    scan(2,10);
    scan(5,100);
    scan(5,20);
    scan(10,10);
}
```

- ❶ 设置缓存和批量处理两个参数。
- ❷ 对返回的 Result 实例计数。
- ❸ 用不同的参数组合测试。

代码打印出了这两个参数的值、服务器返回的 Result 实例数目以及获取数据过程所发起的 RPC 请求的数目。结果如下：

```
Caching: 1,Batch: 1,Results: 200,RPCs: 201
Caching: 200,Batch: 1,Results: 200,RPCs: 2
Caching: 2000,Batch: 100,Results: 10,RPCs: 1
Caching: 2,Batch: 100,Results: 10,RPCs: 6
Caching: 2,Batch: 10,Results: 20,RPCs: 11
Caching: 5,Batch: 100,Results: 10,RPCs: 3
Caching: 5,Batch: 20,Results: 10,RPCs: 3
Caching: 10,Batch: 10,Results: 20,RPCs: 3
```

用户可以修改调整这两个参数来查看它们对输出结果的影响。表 3-9 展示了一些组合。这些组合与例 3.20 相关，例 3.20 中我们建立了一张有两个列族的表，添加了 10 行数据，每个行的每个列族下有 10 列。这意味着整个表一共有 200 列（或单元格，因为每个列只有一个版本），其中每行有 20 列。

表 3-9 示例设置及其影响

缓存	批量处理	Result 个数	RPC 次数	说明
1	1	200	201	每个列都作为一个 Result 实例返回。最后还多一个 RPC 确认扫描完成
200	1	200	2	每个 Result 实例都只包含一列的值，不过它们都被一次 RPC 请求取回（加一次完成检查）
2	10	20	11	批量参数是一行所包含的列数的一半，所以 200 列除以 10，需要 20 个 Result 实例。同时需要 10 次 RPC 请求取回（加一次完成检查）
5	100	10	3	对于一行来讲，这个批量参数太大了，所以一行的 20 列都被放入了一个 Result 实例中。同时缓存为 5，所以 10 个 Result 实例被两次 RPC 请求取回（加一次完成检查）
5	20	10	3	同上，不过这次的批量值与一行的列数正好相同，所以输出与上面一种情况相同
10	10	20	3	这次把表分成了较小的 Result 实例，但使用了较大的缓存值，所以也是只用了两次 RPC 请求就取回了数据



要计算一次扫描操作的 RPC 请求的次数，用户需要先计算出行数 and 每行列数的乘积（至少了解大概情况）。然后用这个值除以批量大小和每行列数中较小的那个值。最后再用除得的结果除以扫描器缓存值。用数学公式表示如下：

$$\text{RPC 请求的次数} = (\text{行数} \times \text{每行的列数}) / \text{Min}(\text{每行的列数}, \text{批量大小}) / \text{扫描器缓存}$$

此外，还需要一些请求来打开和关闭扫描器。用户或许需要把这两次请求也考虑在内。

图 3-2 展示了缓存和批量两个参数如何联动。图 3-2 中有一个包含 9 行数据的表，每行都包含一些列。使用了一个缓存为 6、批量大小为 3 的扫描器，读者可以观察到需要 3 个 PRC 请求来传送数据（虚线圆角方框）。

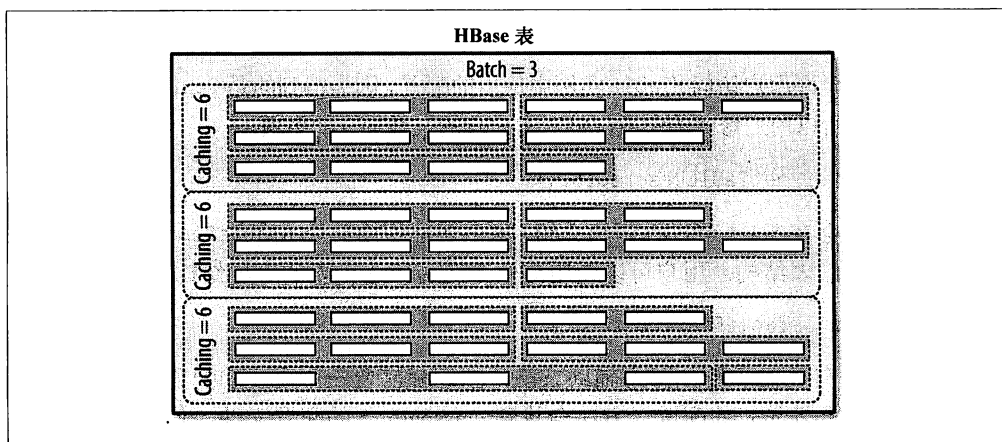


图 3-2 扫描器缓存和批量两个参数控制 RPC 的次数

小的批量值使服务端把 3 个列装入一个 Result 实例，同时扫描器缓存为 6，使每次 RPC 请求传输 6 行，即 6 个被批量封装的 Result 实例。如果没有指定批量大小，但指定了扫描器缓存，那么一个调用结果就能包含所有的行，因为每一行都包含在一个 Result 实例中。只有当用户使用批量模式之后，行内（intra-row）扫描功能才会启用。

最初，用户可能不必为扫描器缓存和批量模式的使用操心，但当用户想尽量提高和利用系统性能时，可能就需要为这两个参数选择一个合适的组合了。

3.6 各种特性

在深入介绍客户端可以利用的特性之前，让我们先介绍一下 HBase 和其客户端 API 提供的各种特性或功能。

3.6.1 HTable 的实用方法

客户端 API 是由 HTable 类的实例提供的，用户可以用它来操作 HBase 表。除了之前提到过的一些主要特性外，还有以下一些值得注意的方法。

```
void close()
```

这个方法之前提到过，不过为了它的完整性和重要性，我们在这儿要重新讨论一下。用户使用完一个 HTable 实例之后，需要调用一次 close()。这个方法会刷写

所有客户端缓冲的写操作：`close()`方法会隐式调用`flushCache()`方法。

```
byte[] getTableName()
```

这是一个获取表名称的快捷方法。

```
Configuration getConfiguration()
```

这个方法允许用户访问 HTable 实例中使用的配置。因为得到的是 Configuration 实例的引用，所以用户修改的参数（针对客户端的）都会立即生效。

```
HTableDescriptor getTableDescriptor()
```

这个方法会在 5.1.1 节进行详细介绍，每个表都使用一个 HTableDescriptor 实例来定义自己的表结构。用户可以使用这个方法访问这个表的底层结构定义。

```
static boolean isTableEnabled(table)
```

HTable 类有 4 个不同的静态辅助方法，它们都需要一个显式的配置和一个表名。如果没有提供显式的配置，方法会找到 `classpath` 下的配置文件，使用默认值创建一个隐式的配置。这个方法可以检查表在 ZooKeeper 中是否被标志为启用。

```
byte[][] getStartKeys()
```

```
byte[][] getEndKeys()
```

```
Pair<byte[][], byte[]]> getStartEndKeys()
```

这些方法让用户可以查看当前表的物理分布情况，不过这个分布很可能在添加一些数据之后发生变化。这些方法返回了表的所有 **region** 的起始行键或/和终止行键。它们以二维字节数组形式返回，用户可以使用 `Bytes.toStringBinary()` 方法打印这些键。

```
void clearRegionCache()
```

```
HRegionLocation getRegionLocation(row)
```

```
Map<HRegionInfo, HServerAddress> getRegionsInfo()
```

这些方法可以帮助用户获取某一行数据的具体位置信息，或者说这行数据所在的 **region** 信息，以及所有 **region** 的分布信息。用户也可以在必要的时候清空缓存的 **region** 位置信息。这些方法可以让高级用户了解并使用这些信息，例如，控制集群流量或者调整数据的位置。

```
void prewarmRegionCache(Map<HRegionInfo, HServerAddress> regionMap)
```

```
static void setRegionCachePrefetch(table, enable)
```

```
static boolean getRegionCachePrefetch(table)
```

这又是一组高级方法。在 1.4.5 节提到过，可以先预取 **region** 位置信息来避免耗时较多的操作（查找每行数据所属 **region** 地址直到本地 **region** 地址缓存相对稳定）。使用这些方法，用户可以先获取一个 **region** 的信息表来预热一下 **region** 缓存（例

如，用户可以使用 `getRegionsInfo()` 访问这个 `region` 的信息表，然后再处理它），也可以把整张表的预取功能打开。

3.6.2 Bytes 类

前面介绍过，如何使用这个类转化 Java 的数据类型，例如，将 `String` 或 `long` 转化为 `HBase` 原生支持的原始字节数组。关于这个类和它的功能还有一些值得一提。

大部分方法都有 3 种形式，例如：

```
static long toLong(byte[] bytes)
static long toLong(byte[] bytes,int offset)
static long toLong(byte[] bytes,int offset,int length)
```

用户可以输入一个字节数组，或者一个字节数组再加一个偏移值，或者一个字节数组、一个偏移值和一个长度值。具体使用哪一种方法取决于最初这个字节数组是怎么生成的。如果这个字节数组之前是由 `toBytes()` 方法生成的，用户就可以安全地使用第一种形式的方法将其转化回来，只需送入这个字节数组而不需要其他额外信息。整个数组的内容都是转换过来的值。

不过，`API` 和 `HBase` 内部都把数据存储为一个较大的数组，例如，使用下面的方法：

```
static int putLong(byte[] bytes,int offset,long val)
```

这个方法允许用户把一个 `long` 值写入一个字节数组的特定偏移位置。用户可以使用后面的两种 `toLong()` 方法存取这种较大的字节数组的数据。

`Bytes` 类支持以下原生 Java 类型到字节数组的互转：`String`、`boolean`、`short`、`int`、`long`、`double` 和 `float`。除了之前介绍的方法外，还有一些值得一提的方法列举在表 3-10 中。

表 3-10 Bytes 类提供的其他方法概述

方法	描述
<code>toStringBinary()</code>	与 <code>toString()</code> 方法非常相似，这个方法可以安全地把不能打印的信息转换为人工可读的十六进制数。如果用户不清楚字节数组中的内容，就可以使用这个方法把内容打印出来，比如打印到控制台或日志文件中
<code>compareTo()/equals()</code>	这个方法让用户可以对两个 <code>byte[]</code> （即字节数组）进行比较。前者返回一个比较结果，后者返回一个布尔值，表示两个数组是否相等
<code>add()/head()/tail()</code>	用这些方法可以把两个字节数组连接在一起形成一个新的数组，或者可以取到字节数组头或尾的一部分
<code>binarySearch()</code>	在用户给定的字节数组中二分查找一个目标值，该方法可以在字节数组上对用户要查找的值和键进行操作
<code>incrementBytes()</code>	一个 <code>long</code> 类型数据转化成的字节数组与 <code>long</code> 的数据相加并返回字节数组，用户可以使用负数参数进行减法

Bytes 类与 Java 提供的 ByteBuffer 类的功能有所重叠。不同的是，前者所有的操作都不需要创建一个新的实例。这是考虑到某些情况下的性能优化，因为 HBase 内部使用了其中的许多方法并多次调用，不创建实例也就避免了许多不必要的垃圾回收。

完整的文档请参阅基于 JavaDoc 的 API 文档^①。

^① 参阅在线文档 Bytes 一节 (<http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/util/Bytes/html>)。

客户端 API：高级特性

在了解 HBase 的基本客户端 API 之后，我们将进一步讨论 HBase 提供给客户端的高级特性。

4.1 过滤器

HBase 过滤器（filter）提供了非常强大的特性来帮助用户提高其处理表中数据的效率。用户不仅可以使用 HBase 中预定义好的过滤器，而且可以实现自定义的过滤器。下面将介绍这两类过滤器。

4.1.1 过滤器简介

HBase 中两种主要的数据读取函数是 `get()` 和 `scan()`，它们都支持直接访问数据和通过指定起止行键访问数据的功能。读者可以在查询中添加更多的限制条件来减少查询得到的数据量，这些限制可以是指定列族、列、时间戳以及版本号。

这些方法可以帮助用户控制哪些数据在查询时被包含其中，但是它们缺少一些细粒度的筛选功能，比如基于正则表达式对行键或值进行筛选。`Get` 和 `Scan` 两个类都支持过滤器，理由如下：这类对象提供的基本 API 不能对行键、列名或列值进行过滤，但是通过过滤器可以达到这个目的。过滤器最基本的接口叫 `Filter`，除此之外，还有一些由 HBase 提供的无需编程就可以直接使用的类。

同时用户还可以通过继承 `Filter` 类来实现自己的需求。所有的过滤器都在服务端生效，叫做谓词下推（predicate push down）。这样可以保证被过滤掉的数据不会被传送到客户端。用户可以在客户端代码中实现过滤的功能（但会影响系统性能），因为在这种情况下服务器端需要传输更多的数据到客户端，用户应当尽量避免这种情况。

图 4-1 描述了过滤器怎样在客户端进行配置，怎样在网络传输中被序列化，怎样在服务器端执行。

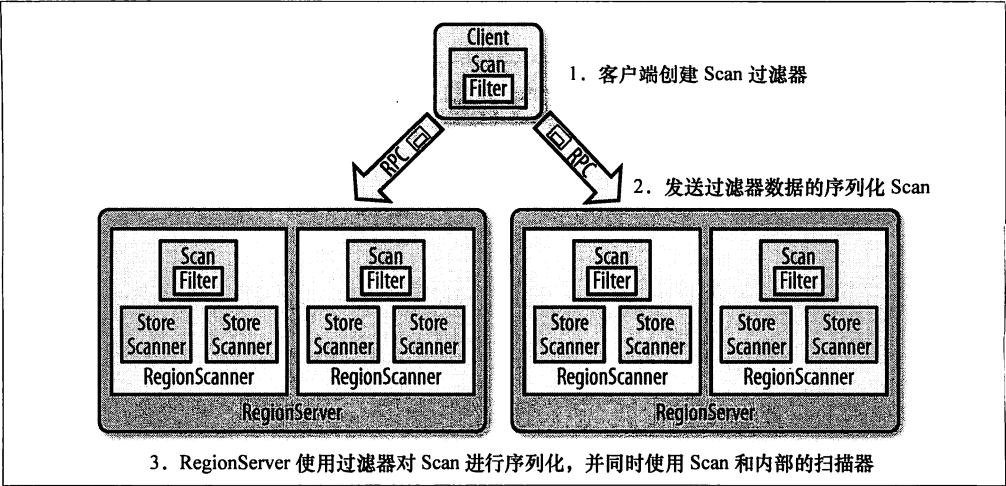


图 4-1 过滤器在客户端创建，通过 RPC 传送到服务器端，然后在服务器端执行过滤操作

1. 过滤器层次结构

在过滤器层次结构的最底层是 Filter 接口和 FilterBase 抽象类，它们实现了过滤器的空壳和骨架，这使得实际的过滤器类可以避免许多重复的结构代码。

大部分实体过滤器类一般都直接继承自 FilterBase，也有一些间接继承自该类。不过它们的使用流程是相同的，用户定义一个所需要的过滤器实例，同时把定义好的过滤器实例传递给 Get 或 Scan 实例：

```
setFilter(filter)
```

在实例化过滤器的时候，用户需要提供一些参数来设定过滤器的用途。其中有一组特殊的过滤器，它们继承自 CompareFilter，需要用户同时提供至少两个特定的参数，这两个参数会被基类用于执行它的任务。下面用户会学习到这两个参数的类型以方便使用。



过滤器可以作用于用户插入的整行数据，所以用户可以基于任意可用的信息来决定如何处理这一行。这些信息包括行键、列名、实际的列值和时间戳等。

我们将简短讨论下涉及值或者比较这两种情况，下面介绍的各种方法都可以使用。特定过滤器的实现可能只考虑了其中一条标准。

2. 比较运算符

因为继承自 `CompareFilter` 的过滤器比基类 `FilterBase` 多了一个 `compare()` 方法，它需要使用传入参数定义比较操作的过程。可用值已经列在表 4-1 中。

表 4-1 `CompareFilter` 中的比较运算符

操作	描述
LESS	匹配小于设定值的值
LESS_OR_EQUAL	匹配小于或等于设定值的值
EQUAL	匹配等于设定值的值
NOT_EQUAL	匹配与设定值不相等的值
GREATER_OR_EQUAL	匹配大于或等于设定值的值
GREATER	匹配大于设定值的值
NO_OP	排除一切值

当过滤器被应用时，比较运算符可以决定什么被包含，什么被排除。这样可以帮助用户筛选数据的一段子集或一些特定数据。

3. 比较器

`CompareFilter` 所需要的第二类类型是比较器 (comparator)，比较器提供了多种方法来比较不同的键值。比较器都继承自 `WritableByteArrayComparable`，`WritableByteArrayComparable` 实现了 `Writable` 和 `Comparable` 接口。如果您仅仅想使用 `HBase` 原生提供的比较器实现则可以不必过度关注细节，`HBase` 提供的比较器罗列于表 4-2，这些比较器构造时通常只需要提供一个阈值，这个值将会与表中的实际值进行比较。

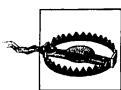
表 4-2 `HBase` 对基于 `CompareFilter` 的过滤器提供的比较器

比较器	描述
<code>BinaryComparator</code>	使用 <code>Bytes.compareTo()</code> 比较当前值与阈值
<code>BinaryPrefixComparator</code>	与上面的相似，使用 <code>Bytes.compareTo()</code> 进行匹配，但是是从左端开始前缀匹配
<code>NullComparator</code>	不做匹配，只判断当前值是不是 <code>null</code>
<code>BitComparator</code>	通过 <code>BitwiseOp</code> 类提供的按位与 (AND)、或 (OR)、异或 (XOR) 操作执行位级比较
<code>RegexStringComparator</code>	根据一个正则表达式，在实例化这个比较器的时候去匹配表中的数据
<code>SubstringComparator</code>	把阈值和表中数据当作 <code>String</code> 实例，同时通过 <code>contains()</code> 操作匹配字符串



后面的 3 种比较器，即 `BitComparator`、`RegexStringComparator` 和 `SubstringComparator`，只能与 `EQUAL` 和 `NOT_EQUAL` 运算符搭配使用，因为这些比较器的 `compareTo()` 方法匹配时返回 0，不匹配时返回 1。如果和 `LESS` 或 `GREATER` 运算符搭配使用，会产生错误结果。

通常每个比较器都有一个带比较值参数的构造函数。换句话说，用户需要定义一个值来跟每个单元格做比较。一些构造函数使用字节数组作为参数，并通过二进制进行比较，还有一些使用 `String` 参数（当需要比较的参数是可读的有序文本时）进行比较。例 4.1 展示了这些功能。



基于字符串的比较器，如 `RegexStringComparator` 和 `SubstringComparator`，比基于字节的比较器更慢，更消耗资源。因为每次比较时它们都需要将给定的值转化为 `String`。截取字符串子串和正则式的处理也需要花费额外的时间。

4.1.2 比较过滤器

HBase 提供的第一种过滤器实现就是比较过滤器（`comparison filter`）。如前所述，用户创建实例时需要一个比较运算符和一个比较器实例。每个比较过滤器的构造方法都有一个从 `CompareFilter` 继承来的签名方法。

```
CompareFilter(CompareOp valueCompareOp,  
              WritableByteArrayComparable valueComparator)
```

用户需要提供比较运算符和比较类来让过滤器工作。后面读者会看到实现了特殊比较的实际过滤器。



用户需要了解，HBase 中过滤器本来的目的是为了筛掉无用的信息。被过滤掉的信息不会被传送到客户端。过滤器不能用来指定用户需要哪些信息，而是在读取数据的过程中不返回用户不想要的信息。

正好相反，所有基于 `CompareFilter` 的过滤处理过程与上面所描述的恰好相反，它们返回匹配的值。换句话说，用户需要根据过滤器的不同规则来小心地挑选过滤器。例如，如果用户需要返回大于或等于某值的数据，就不应当使用 `LESS` 来跳过不需要的数据，而应当使用 `GREATER_OR_EQUAL` 来包括所有符合条件的数据。

1. 行过滤器（`RowFilter`）

行过滤器基于行键来过滤数据。例 4.1 展示了如何通过使用不同的过滤器来获取需要的行。使用多种比较运算符来返回符合条件的行键，同时会过滤不符合条件的行键。用

户可以随意地修改代码来改变返回结果。

例 4.1 使用过滤器来挑选特定的行

```
Scan scan = new Scan();
scan.addColumn(Bytes.toBytes("colfam1"), Bytes.toBytes("col-0"));

Filter filter1 = new RowFilter(CompareFilter.CompareOp.LESS_OR_EQUAL, ❶
    new BinaryComparator(Bytes.toBytes("row-22")));
scan.setFilter(filter1);
ResultScanner scanner1 = table.getScanner(scan);
for(Result res : scanner1){
    System.out.println(res);
}
scanner1.close();

Filter filter2 = new RowFilter(CompareFilter.CompareOp.EQUAL, ❷
    new RegexStringComparator(".*-.5"));
scan.setFilter(filter2);
ResultScanner scanner2 = table.getScanner(scan);
for(Result res : scanner2){
    System.out.println(res);
}
scanner2.close();

Filter filter3 = new RowFilter(CompareFilter.CompareOp.EQUAL, ❸
    new SubstringComparator("-5"));
scan.setFilter(filter3);
ResultScanner scanner3 = table.getScanner(scan);
for(Result res : scanner3){
    System.out.println(res);
}
scanner3.close();
```

❶ 创建一个过滤器，指定比较运算符和比较器，这里需要精确匹配。

❷ 创建另一个过滤器，使用正则表达式来匹配行键。

❸ 创建第三个过滤器，使用子串匹配方法。

终端的输出如下：

```
Adding rows to table...
Scanning table #1...
keyvalues={row-1/colfam1:col-0/1301043190260/Put/vlen=7}
keyvalues={row-10/colfam1:col-0/1301043190908/Put/vlen=8}
keyvalues={row-100/colfam1:col-0/1301043195275/Put/vlen=9}
keyvalues={row-11/colfam1:col-0/1301043190982/Put/vlen=8}
keyvalues={row-12/colfam1:col-0/1301043191040/Put/vlen=8}
keyvalues={row-13/colfam1:col-0/1301043191172/Put/vlen=8}
keyvalues={row-14/colfam1:col-0/1301043191318/Put/vlen=8}
```

```

keyvalues={row-15/colfam1:col-0/1301043191429/Put/vlen=8}
keyvalues={row-16/colfam1:col-0/1301043191509/Put/vlen=8}
keyvalues={row-17/colfam1:col-0/1301043191593/Put/vlen=8}
keyvalues={row-18/colfam1:col-0/1301043191673/Put/vlen=8}
keyvalues={row-19/colfam1:col-0/1301043191771/Put/vlen=8}
keyvalues={row-2/colfam1:col-0/1301043190346/Put/vlen=7}
keyvalues={row-20/colfam1:col-0/1301043191841/Put/vlen=8}
keyvalues={row-21/colfam1:col-0/1301043191933/Put/vlen=8}
keyvalues={row-22/colfam1:col-0/1301043191998/Put/vlen=8}
Scanning table #2...
keyvalues={row-15/colfam1:col-0/1301043191429/Put/vlen=8}
keyvalues={row-25/colfam1:col-0/1301043192140/Put/vlen=8}
keyvalues={row-35/colfam1:col-0/1301043192665/Put/vlen=8}
keyvalues={row-45/colfam1:col-0/1301043193138/Put/vlen=8}
keyvalues={row-55/colfam1:col-0/1301043193729/Put/vlen=8}
keyvalues={row-65/colfam1:col-0/1301043194092/Put/vlen=8}
keyvalues={row-75/colfam1:col-0/1301043194457/Put/vlen=8}
keyvalues={row-85/colfam1:col-0/1301043194806/Put/vlen=8}
keyvalues={row-95/colfam1:col-0/1301043195121/Put/vlen=8}
Scanning table #3...
keyvalues={row-5/colfam1:col-0/1301043190562/Put/vlen=7}
keyvalues={row-50/colfam1:col-0/1301043193332/Put/vlen=8}
keyvalues={row-51/colfam1:col-0/1301043193514/Put/vlen=8}
keyvalues={row-52/colfam1:col-0/1301043193603/Put/vlen=8}
keyvalues={row-53/colfam1:col-0/1301043193654/Put/vlen=8}
keyvalues={row-54/colfam1:col-0/1301043193696/Put/vlen=8}
keyvalues={row-55/colfam1:col-0/1301043193729/Put/vlen=8}
keyvalues={row-56/colfam1:col-0/1301043193766/Put/vlen=8}
keyvalues={row-57/colfam1:col-0/1301043193802/Put/vlen=8}
keyvalues={row-58/colfam1:col-0/1301043193842/Put/vlen=8}
keyvalues={row-59/colfam1:col-0/1301043193889/Put/vlen=8}

```

可以看到第一个过滤器精确匹配了所需要的行键。返回的结果中包括了所有行键等于或小于给定值的行。用户需要注意筛选行键的过程，行键是按照字典序排列的。第二个过滤器使用正则表达式筛选，第三个使用子串筛选结果。结果表明过滤器工作正常。

2. 列族过滤器 (FamilyFilter)

这个过滤器与行过滤器 (RowFilter) 相似，不过它是通过比较列族而不是比较行键来返回结果的。通过使用不同组合的运算符和比较器，用户可以在列族一级筛选所需的数据。例 4.2 展示了如何使用它们。

例 4.2 使用过滤器来返回特定的列族

```

Filter filter1 = new FamilyFilter(CompareFilter.CompareOp.LESS, ❶
    new BinaryComparator(Bytes.toBytes("colfam3")));

```

```

Scan scan = new Scan();
scan.setFilter(filter1);
ResultScanner scanner = table.getScanner(scan);❷
for(Result result : scanner){
    System.out.println(result);
}
scanner.close();

Get get1 = new Get(Bytes.toBytes("row-5"));
get1.setFilter(filter1);
Result result1 = table.get(get1);❸
System.out.println("Result of get(): " + result1);

Filter filter2 = new FamilyFilter(CompareFilter.CompareOp.EQUAL,
    new BinaryComparator(Bytes.toBytes("colfam3")));
Get get2 = new Get(Bytes.toBytes("row-5"));❹
get2.addFamily(Bytes.toBytes("colfam1"));
get2.setFilter(filter2);
Result result2 = table.get(get2);❺
System.out.println("Result of get(): " + result2);

```

- ❶ 创建一个过滤器，指定比较运算符和比较器。
- ❷ 使用过滤器扫描表。
- ❸ 使用相同的过滤器获取一行数据。
- ❹ 在一个列族上创建过滤器，同时获取另一行数据。
- ❺ 使用新的过滤器获取同一行数据，此时返回结果为“NONE”。

输出结果（为了方便阅读稍作整理）表明了过滤器的作用。存入表的数据有 4 个列族，每个列族有两列，同时一共有 10 行数据。

```

Adding rows to table...
Scanning table...
keyvalues={row-1/colfam1:col-0/1303721790522/Put/vlen=7,
           row-1/colfam1:col-1/1303721790574/Put/vlen=7,
           row-1/colfam2:col-0/1303721790522/Put/vlen=7,
           row-1/colfam2:col-1/1303721790574/Put/vlen=7}
keyvalues={row-10/colfam1:col-0/1303721790785/Put/vlen=8,
           row-10/colfam1:col-1/1303721790792/Put/vlen=8,
           row-10/colfam2:col-0/1303721790785/Put/vlen=8,
           row-10/colfam2:col-1/1303721790792/Put/vlen=8}
...
keyvalues={row-9/colfam1:col-0/1303721790778/Put/vlen=7,
           row-9/colfam1:col-1/1303721790781/Put/vlen=7,
           row-9/colfam2:col-0/1303721790778/Put/vlen=7,

```

```
row-9/colfam2:col-1/1303721790781/Put/vlen=7}
```

```
Result of get(): keyvalues={row-5/colfam1:col-0/1303721790652/Put/vlen=7,  
    row-5/colfam1:col-1/1303721790664/Put/vlen=7,  
    row-5/colfam2:col-0/1303721790652/Put/vlen=7,  
    row-5/colfam2:col-1/1303721790664/Put/vlen=7}
```

```
Result of get(): keyvalues=NONE
```

最后的 `get()` 调用说明用户可以（非故意）通过使用过滤器指定一个列族，同时使用 `addFamily()` 指定另一列族，从而得到一个空的结果集。

3. 列名过滤器 (QualifierFilter)

例 4.3 展示了使用列名进行筛选的类似逻辑。这种操作可以帮助用户筛选特定的列。

例 4.3 使用过滤器筛选列

```
Filter filter = new QualifierFilter(CompareFilter.CompareOp.LESS_OR_EQUAL,  
    new BinaryComparator(Bytes.toBytes("col-2")));  
  
Scan scan = new Scan();  
scan.setFilter(filter);  
ResultScanner scanner = table.getScanner(scan);  
for(Result result : scanner){  
    System.out.println(result);  
}  
scanner.close();  
  
Get get = new Get(Bytes.toBytes("row-5"))  
get.setFilter(filter);  
Result result = table.get(get);  
System.out.println("Result of get(): " + result);
```

4. 值过滤器 (ValueFilter)

这个过滤器可以帮助用户筛选某个特定值的单元格。与 `RegexStringComparator` 配合使用，可以使用功能强大的表达式来进行筛选。例 4.4 展示了这项功能。需要注意的是，在使用特定比较器的时候，只能与部分运算符搭配。在本例中，我们使用了子字符串匹配，这种匹配只能使用 `EQUAL` 和 `NOT_EQUAL` 运算符。

例 4.4 使用值过滤器的例子

```
Filter filter = new ValueFilter(CompareFilter.CompareOp.EQUAL, ❶  
    new SubstringComparator(".4"));
```



```

Scan scan = new Scan();
scan.setFilter(filter);❷
ResultScanner scanner = table.getScanner(scan);
for(Result result : scanner){
    for(KeyValue kv : result.raw()){
        System.out.println("KV: " + kv + ",Value: " + ❸
            Bytes.toString(kv.getValue()));
    }
}
scanner.close();

Get get = new Get(Bytes.toBytes("row-5"));
get.setFilter(filter);❹
Result result = table.get(get);
for(KeyValue kv : result.raw()){
    System.out.println("KV: " + kv + ",Value: " +
        Bytes.toString(kv.getValue()));
}

```

- ❶ 创建过滤器，指定比较运算符和比较器。
- ❷ 设置扫描过程中的过滤器。
- ❸ 打印结果并检查。
- ❹ 将同样的过滤器应用于 Get 实例。

5. 参考列过滤器 (DependentColumnFilter)

这里有一种更复杂的过滤器，这种过滤器不仅仅简单地通过用户指定的信息筛选数据。这种过滤器允许用户指定一个参考列或是引用列，并使用参考列控制其他列的过滤。参考列过滤器使用参考列的时间戳，并在过滤时包括所有与引用时间戳相同的列。下面是它的构造方法：

```

DependentColumnFilter(byte[] family,byte[] qualifier)
DependentColumnFilter(byte[] family,byte[] qualifier,
    boolean dropDependentColumn)
DependentColumnFilter(byte[] family,byte[] qualifier,
    boolean dropDependentColumn,CompareOp valueCompareOp,
    WritableByteArrayComparable valueComparator)

```

由于参考过滤器也是继承自 CompareFilter，所以它也可以帮助用户筛选列，不过这个过滤器是基于这些列值进行筛选的。用户可以把它理解为一个 ValueFilter 和一个时间戳过滤器的组合。用户可以传入比较运算符和基准值来启用 ValueFilter 的功能。这个过滤器的构造函数默认允许用户在所有列上忽略运算符和比较器，以及屏蔽按值筛选的功能，也就是说整个过滤器只基于参考列的时间戳进行筛选。

例 4.5 展示了过滤器的用法，例子展示了各个可选参数的作用。dropDependentColumn 参数

可以帮助用户操作参考列: 该参数设为 false 或 true 决定了参考列可以被返回还是被丢弃。

例 4.5 使用过滤器返回一些特定的列

```
private static void filter(boolean drop,
    CompareFilter.CompareOp operator,
    WritableByteArrayComparable comparator)
    throws IOException {
    Filter filter;
    if(comparator != null){
        filter = new DependentColumnFilter(Bytes.toBytes("colfam1"), ❶
            Bytes.toBytes("col-5"), drop, operator, comparator);
    } else {
        filter = new DependentColumnFilter(Bytes.toBytes("colfam1"),
            Bytes.toBytes("col-5"), drop);
    }

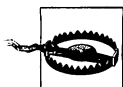
    Scan scan = new Scan();
    scan.setFilter(filter);
    ResultScanner scanner = table.getScanner(scan);
    for(Result result : scanner){
        for(KeyValue kv : result.raw()){
            System.out.println("KV: " + kv + ", Value: " +
                Bytes.toString(kv.getValue()));
        }
    }
    scanner.close();

    Get get = new Get(Bytes.toBytes("row-5"));
    get.setFilter(filter);
    Result result = table.get(get);
    for(KeyValue kv : result.raw()){
        System.out.println("KV: " + kv + ", Value: " +
            Bytes.toString(kv.getValue()));
    }
}

public static void main(String[] args) throws IOException {
    filter(true, CompareFilter.CompareOp.NO_OP, null);
    filter(false, CompareFilter.CompareOp.NO_OP, null); ❷
    filter(true, CompareFilter.CompareOp.EQUAL,
        new BinaryPrefixComparator(Bytes.toBytes("val-5")));
    filter(false, CompareFilter.CompareOp.EQUAL,
        new BinaryPrefixComparator(Bytes.toBytes("val-5")));
    filter(true, CompareFilter.CompareOp.EQUAL,
        new RegexStringComparator(".*\\.5"));
    filter(false, CompareFilter.CompareOp.EQUAL,
```

```
new RegexStringComparator(".*\\.5");
}
```

- ❶ 使用多个参数创建过滤器。
- ❷ 使用多个参数调用过滤器方法。



这种过滤器与扫描操作的批量处理功能不兼容，换句话说，需要使用 `Scan.setBatch()` 方法将 `batch` 值设为一个比 0 大的数。然而，过滤器需要查看整行数据来决定哪些数据被过滤，使用批量处理可能会导致取到的数据中不包括参考列，因此结果有错。

如果启用批量处理模式，用户会遇到以下错误：

```
Exception org.apache.hadoop.hbase.filter.IncompatibleFilter Exception:
    Cannot set batch on a scan using a filter that returns true for filter.
    hasFilterRow
```

这个例子不同于之前提到的过滤器，因为它控制列的版本来得到时间戳相符的结果。服务器端使用隐式时间戳作为版本可能导致结果出现波动，主要原因是用户不能保证使用的时间精确到毫秒。

例子中 `filter()` 方法被多种不同的组合参数调用，展示了如何使用内建值过滤器和终止标志位来操纵返回的数据集。

4.1.3 专用过滤器

HBase 提供的第二类过滤器直接继承自 `FilterBase`，同时用于更特定的使用场景。其中的一些过滤器只能做行筛选，因此只适用于扫描操作。对 `get()` 方法来说，这些过滤器限制得过于苛刻：要么包括整行，要么什么都不包括。

1. 单列值过滤器 (`SingleColumnValueFilter`)

用户针对如下情况时可以使用该过滤器：用一列的值决定是否一行数据被过滤。首先设定待检查的列，然后设置待检查的列的对应值。具体构造函数如下：

```
SingleColumnValueFilter(byte[] family,byte[] qualifier,
    CompareOp compareOp,byte[] value)
SingleColumnValueFilter(byte[] family,byte[] qualifier,
    CompareOp compareOp,WritableByteArrayComparable comparator)
```

第一个构造函数较为简单，因为它只在内部创建一个 `BinaryComparator` 实例。第二个构造函数中所需的参数与用户一直在使用的基于 `CompareFilter` 的类相同，尽管 `SingleColumnValueFilter` 并不是直接继承自 `CompareFilter`，但还是使用了相同的参数类型。

同时，过滤器还提供了一些辅助方法帮助用户微调过滤行为。

```

boolean getFilterIfMissing()
void setFilterIfMissing(boolean filterIfMissing)
boolean getLatestVersionOnly()
void setLatestVersionOnly(boolean latestVersionOnly)

```

前者决定了当参考列不存在时如何处理这一行。默认的这一行是被包含在结果中的，用户可以使用 `setFilterIfMissing(true)` 来过滤这些行，也就是说，在这样设置以后，所有不包含参考列的行都可以被过滤掉。



用户在扫描时必须包括参考列，用户使用类似于 `addColumn()` 的方法把参考列添加到查询中。如果用户没有这么做，也就是说扫描结果中没有包括参考列，那么结果可能为空或包含所有行，至于具体情况会根据 `getFilterIfMissing()` 的设定值来返回。

使用 `setLatestVersionOnly(false)` 可以改变过滤器的行为，默认值为 `true`，此时过滤器只检查参考列的最新版本，设为 `false` 之后会检查所有版本。例 4.6 组合了这些特性来获取一组特定的行。

例 4.6 使用过滤器返回包含特定列中特定值的行

```

SingleColumnValueFilter filter = new SingleColumnValueFilter(
    Bytes.toBytes("colfam1"),
    Bytes.toBytes("col-5"),
    CompareFilter.CompareOp.NOT_EQUAL,
    new SubstringComparator("val-5"));
filter.setFilterIfMissing(true);

Scan scan = new Scan();
scan.setFilter(filter);
ResultScanner scanner = table.getScanner(scan);
for(Result result : scanner){
    for(KeyValue kv : result.raw()){
        System.out.println("KV: " + kv + ", Value: " +
            Bytes.toString(kv.getValue()));
    }
}
scanner.close();

Get get = new Get(Bytes.toBytes("row-6"));
get.setFilter(filter);
Result result = table.get(get);
System.out.println("Result of get: ");
for(KeyValue kv : result.raw()){

```

```

        System.out.println("KV: " + kv + ",Value: " +
            Bytes.toString(kv.getValue()));
    }
}

```

2. 单列排除过滤器 (SingleColumnValueExcludeFilter)

单列排除过滤器继承自 SingleColumnValueFilter，经过拓展后提供一种略微不同的语义：参考列不被包括到结果中。换句话说，用户可以使用与之前相同的特性和方法来控制过滤器的工作，唯一的区别是，客户端 Result 实例中用户永远不会获得作为检查目标的参考列。

3. 前缀过滤器 (PrefixFilter)

在构造当前过滤器时传入一个前缀，所有与前缀匹配的行都会被返回到客户端。构造函数如下：

```
public PrefixFilter(byte[] prefix)
```

例 4.7 在常用测试数据集中验证了这个过滤器的功能。

例 4.7 使用前缀过滤器

```

Filter filter = new PrefixFilter(Bytes.toBytes("row-1"));

Scan scan = new Scan();
scan.setFilter(filter);
ResultScanner scanner = table.getScanner(scan);
for(Result result : scanner){
    for(KeyValue kv : result.raw()){
        System.out.println("KV: " + kv + ",Value: " +
            Bytes.toString(kv.getValue()));
    }
}
scanner.close();

Get get = new Get(Bytes.toBytes("row-5"));
get.setFilter(filter);
Result result = table.get(get);
for(KeyValue kv : result.raw()){
    System.out.println("KV: " + kv + ",Value: " +
        Bytes.toString(kv.getValue()));
}

```

需要用户注意的是，get() 方法并没有返回任何结果，因为它请求的行与过滤器的前缀不匹配。这个过滤器在使用 get() 方法时作用不大，但是在扫描操作中非常有用。

扫描操作以字典序查找，当遇到比前缀大的行时，扫描操作就结束了。通过与起始行配合使用，过滤器的扫描性能大大提高，原因是当它发现后面的行不符合要求时会全部跳过。

4. 分页过滤器 (PageFilter)

用户可以使用这个过滤器对结果按行分页。当用户创建当前过滤器实例时需要指定 `pageSize` 参数，这个参数可以控制每页返回的行数。



在物理上分离的服务器中并行执行过滤操作时，需要注意以下几个事项。在不同的 `region` 服务器上并行执行的过滤器不能共享它们现在的状态和边界，因此，每个过滤器都会在完成扫描前获取 `pageCount` 行的结果，这种情况使得分页过滤器可能失效，极有可能返回的比所需要的多。最终客户端在合并结果时可以选择返回所有结果，也可以使用 API 根据需求筛选结果。

客户端代码会记录本次扫描的最后一行，并在下一次获取数据时把记录的上次扫描的最后一行设为这次扫描的起始行，同时保留相同的过滤属性，然后依次进行迭代。

分页时对一次返回的行数设定了严格的限制，一次扫描所覆盖的行数很可能是多于分页大小的，一旦这种情况发生，过滤器有一种机制通知 `region` 服务器停止扫描。

例 4.8 把以上介绍的机制放在了一起，并展示了如何在客户端连续的迭代中重置扫描的起始行。

例 4.8 使用过滤器按行分页

```
Filter filter = new PageFilter(15);

int totalRows = 0;
byte[] lastRow = null;
while(true){
    Scan scan = new Scan();
    scan.setFilter(filter);
    if(lastRow != null){
        byte[] startRow = Bytes.add(lastRow, POSTFIX);
        System.out.println("start row: " +
            Bytes.toStringBinary(startRow));
        scan.setStartRow(startRow);
    }
    ResultScanner scanner = table.getScanner(scan);
    int localRows = 0;
    Result result;
    while((result = scanner.next()) != null){
        System.out.println(localRows++ + ": " + result);
    }
}
```

```

        totalRows++;
        lastRow = result.getRow();
    }
    scanner.close();
    if(localRows == 0)break;
}
System.out.println("total rows: " + totalRows);

```

HBase 中的行键是按字典序排列的，因此返回的结果也是如此排序的，并且起始行是被包括在结果中的。用户需要拼接一个零字节（一个长度为零的字节数组）到之前的行键，这样可以保证最后返回的行在本轮扫描时不被包括。当重置扫描的边界时，零字节是最聪明可靠的方式，因为零字节是最小的增幅。即使有一行的行键正好与之前一行加零字节相同，在这一轮循环时也不会有问题，因为起始行在扫描时是被包括在内的。

5. 行键过滤器 (KeyOnlyFilter)

在一些应用中只需要将结果中 KeyValue 实例的键返回，而不需要返回实际的数据。KeyOnlyFilter 提供了可以修改扫描出的列和单元格的功能。这个过滤器通过 KeyValue.convertToKeyOnly(boolean) 方法帮助调用只返回键不返回值。

这个过滤器的构造函数中需要一个叫 lenAsVal 的布尔参数。这个参数会被传入 convertToKeyOnly() 方法中，它可以控制 KeyValue 实例中值的处理。默认值为 false，设置为 false 时，值被设为长度为 0 的字节数组，设置为 true 时，值被设为原值长度的字节数组。

键虽然包含了有意义的信息，但值的长度可能用来做二次排序并需要快速迭代所有列，此时前段描述中的后者对当前的应用非常有用。在 11.7 节中有相关示例。

6. 首次行键过滤器 (FirstKeyOnlyFilter)

如果用户需要访问一行中的第一列（HBase 隐式排序），则这种过滤器可以满足需求。这种过滤器通常在行数统计（row counter）的应用场景中使用，这种场景只需要检查这一行是否存在。在列式存储数据库中如果某一行存在，则行中必然有列。

由于列也按字典序排列，因此其他可能用到的场景是按照时间先后生成列名，这样最旧的列就会排在最前面，因此时间戳最久的列会最先被检索到。这个场景与当前的过滤器结合使用时，通过单一的扫描操作就可以得到每行中最早创建的列。

这个类使用了过滤器框架提供的另一个优化特性：它在检查完第一列之后会通知 region 服务器结束对当前行的扫描，并跳到下一行，与全表扫描相比，其性能得到了提升。

7. 包含结束的过滤器 (InclusiveStopFilter)

扫描操作中的开始行被包含到结果中，但终止行被排除在外。使用这个过滤器时，用户也可以将结束行包括到结果中。例 4.9 从 row-3 开始扫描，到 row-5 结束扫描。

例 4.9 使用包含结束行的过滤器取回结束行

```
Filter filter = new InclusiveStopFilter(Bytes.toBytes("row-5"));

Scan scan = new Scan();
scan.setStartRow(Bytes.toBytes("row-3"));
scan.setFilter(filter);
ResultScanner scanner = table.getScanner(scan);
for(Result result : scanner){
    System.out.println(result);
}
scanner.close();
```

示例运行时控制台的输出结果如下所示：

```
Adding rows to table...
Results of scan:
keyvalues={row-3/colfam1:col-0/1301337961569/Put/vlen=7}
keyvalues={row-30/colfam1:col-0/1301337961610/Put/vlen=8}
keyvalues={row-31/colfam1:col-0/1301337961612/Put/vlen=8}
keyvalues={row-32/colfam1:col-0/1301337961613/Put/vlen=8}
keyvalues={row-33/colfam1:col-0/1301337961614/Put/vlen=8}
keyvalues={row-34/colfam1:col-0/1301337961615/Put/vlen=8}
keyvalues={row-35/colfam1:col-0/1301337961616/Put/vlen=8}
keyvalues={row-36/colfam1:col-0/1301337961617/Put/vlen=8}
keyvalues={row-37/colfam1:col-0/1301337961618/Put/vlen=8}
keyvalues={row-38/colfam1:col-0/1301337961619/Put/vlen=8}
keyvalues={row-39/colfam1:col-0/1301337961620/Put/vlen=8}
keyvalues={row-4/colfam1:col-0/1301337961571/Put/vlen=7}
keyvalues={row-40/colfam1:col-0/1301337961621/Put/vlen=8}
keyvalues={row-41/colfam1:col-0/1301337961622/Put/vlen=8}
keyvalues={row-42/colfam1:col-0/1301337961623/Put/vlen=8}
keyvalues={row-43/colfam1:col-0/1301337961624/Put/vlen=8}
keyvalues={row-44/colfam1:col-0/1301337961625/Put/vlen=8}
keyvalues={row-45/colfam1:col-0/1301337961626/Put/vlen=8}
keyvalues={row-46/colfam1:col-0/1301337961627/Put/vlen=8}
keyvalues={row-47/colfam1:col-0/1301337961628/Put/vlen=8}
keyvalues={row-48/colfam1:col-0/1301337961629/Put/vlen=8}
keyvalues={row-49/colfam1:col-0/1301337961630/Put/vlen=8}
keyvalues={row-5/colfam1:col-0/1301337961573/Put/vlen=7}
```

8. 时间戳过滤器 (TimestampsFilter)

当用户需要在扫描结果中对版本进行细粒度的控制时，这个过滤器可以满足需求。用

户需要传入一个装载了时间戳的 List 实例：

```
TimestampsFilter(List<Long> timestamps)
```



如前文所述，一个版本（version）是指一个列在一个特定时间的值，因此用一个时间戳（timestamp）来表示。当过滤器请求一系列的时间戳时，它会找到与其中时间戳精确匹配的列版本。

例 4.10 在第一个扫描中使用了一个包括 3 个时间戳的过滤器，在第二个扫描中增加了一个时间范围限制。

例 4.10 使用时间戳过滤数据

```
List<Long> ts = new ArrayList<Long>();
ts.add(new Long(5));
ts.add(new Long(10)); ❶
ts.add(new Long(15));
Filter filter = new TimestampsFilter(ts);

Scan scan1 = new Scan();
scan1.setFilter(filter); ❷
ResultScanner scanner1 = table.getScanner(scan1);
for(Result result : scanner1){
    System.out.println(result);
}
scanner1.close();

Scan scan2 = new Scan();
scan2.setFilter(filter);
scan2.setTimeRange(8,12); ❸
ResultScanner scanner2 = table.getScanner(scan2);
for(Result result : scanner2){
    System.out.println(result);
}
scanner2.close();
```

- ❶ 向列表中添加时间戳。
- ❷ 向 Scan 实例中添加过滤器。
- ❸ 添加时间范围限制，看它如何影响过滤器的行为。

以下是节选的输出：

```
Adding rows to table...
Results of scan #1:
keyvalues={row-1/colfam1:col-10/10/Put/vlen=8,
           row-1/colfam1:col-15/15/Put/vlen=8,
           row-1/colfam1:col-5/5/Put/vlen=7}
keyvalues={row-10/colfam1:col-10/10/Put/vlen=9,
```

```

        row-10/colfam1:col-15/15/Put/vlen=9,
        row-10/colfam1:col-5/5/Put/vlen=8}
keyvalues={row-100/colfam1:col-10/10/Put/vlen=10,
        row-100/colfam1:col-15/15/Put/vlen=10,
        row-100/colfam1:col-5/5/Put/vlen=9}
...
Results of scan #2:
keyvalues={row-1/colfam1:col-10/10/Put/vlen=8}
keyvalues={row-10/colfam1:col-10/10/Put/vlen=9}
keyvalues={row-100/colfam1:col-10/10/Put/vlen=10}
keyvalues={row-11/colfam1:col-10/10/Put/vlen=9}
...

```

第一个只使用过滤器的扫描操作返回了这个列的 3 个版本，它们的时间戳与过滤器预期的一致。第二个扫描操作只返回了符合时间范围限制的时间戳的结果，时间范围与过滤器都起了作用，输出结果同时被它们限制。

9. 列计数过滤器 (ColumnCountGetFilter)

用户可以使用这个过滤器来限制每行最多取回多少列。你可以使用以下构造器来设置这个数字：

```
ColumnCountGetFilter(int n)
```

当一行的列数达到设定的最大值时，这个过滤器会停止整个扫描操作，所以它不太适合扫描操作，反而比较适合在 `get()` 方法中使用。

10. 列分页过滤器 (ColumnPaginationFilter)

与 `PageFilter` 相似，这个过滤器可以对一行的所有列进行分页。它的构造器需要两个参数：

```
ColumnPaginationFilter(int limit,int offset)
```

它将跳过所有偏移量小于 `offset` 的列，并包括之后所有偏移量在 `limit` 之前（包含 `limit`）的列。例 4.11 在一次扫描操作中使用了这个过滤器。

例 4.11 对一行中的所有列分页

```

Filter filter = new ColumnPaginationFilter(5,15);

Scan scan = new Scan();
scan.setFilter(filter);
ResultScanner scanner = table.getScanner(scan);
for(Result result : scanner){
    System.out.println(result);
}
scanner.close();

```

示例代码的输出如下：

```
Adding rows to table...
Results of scan:
keyvalues={row-01/colfam1:col-15/15/Put/vlen=9,
           row-01/colfam1:col-16/16/Put/vlen=9,
           row-01/colfam1:col-17/17/Put/vlen=9,
           row-01/colfam1:col-18/18/Put/vlen=9,
           row-01/colfam1:col-19/19/Put/vlen=9}
keyvalues={row-02/colfam1:col-15/15/Put/vlen=9,
           row-02/colfam1:col-16/16/Put/vlen=9,
           row-02/colfam1:col-17/17/Put/vlen=9,
           row-02/colfam1:col-18/18/Put/vlen=9,
           row-02/colfam1:col-19/19/Put/vlen=9}
...
```



这个例子通过在行和列的后面添加了一段用作记数的部分改变了它们的计数方式。例如第一行被补位成了 row-01。这也展示了如何使用补全的字段让排序的结果更加可读易懂，换句话说，让输出结果像字典或电话簿。

整个 10 行数据都被打印出来，并且返回的列都是在 offset=15 和 limit=5 的范围内，每个结果集有 5 列。

11. 列前缀过滤器 (ColumnPrefixFilter)

类似于 PrefixFilter，这个过滤器通过对列名称进行前缀匹配过滤。用户需要指定一个前缀来创建过滤器。

```
ColumnPrefixFilter(byte[] prefix)
```

所有与设定前缀匹配的列都被包含在结果中。

12. 随机行过滤器 (RandomRowFilter)

最后，有一种过滤器可以让结果中包含随机行。构造函数需要传入参数 chance，chance 取值区间在 0.0 到 1.0 之间。

```
RandomRowFilter(float chance)
```

在过滤器内部会使用 Java 方法 Random.nextFloat() 来决定一行是否被过滤，使用这个方法的结果会与用户设定的 chance 进行比较。如果用户为 chance 赋一个负值会导致所有结果都被过滤掉，相反地，如果 chance 大于 1.0 则结果集中包含所有行。

4.1.4 附加过滤器

目前 HBase 提供的过滤器已经十分强大，这些过滤器可以提供修改、扩展和对返回结

果的行为进行控制等功能。一些额外的控制不依赖于这些过滤器本身，但却可以应用在其他过滤器上。这正是附加过滤器（decorating filter）想要提供的功能。

1. 跳转过滤器（SkipFilter）

这个过滤器包装了一个用户提供的过滤器，当被包装的过滤器遇到一个需要过滤的 Key-Value 实例时，用户可以拓展并过滤整行数据。换句话说，当过滤器发现某一行中的一列需要过滤时，那么整行数据都将被过滤掉。



被包装的过滤器^①必须实现 filterKeyValue() 方法，否则 SkipFilter 无法正常工作。这是因为 SkipFilter 只通过检查这个方法的返回结果来决定如何处理这一行。参考表 4-5 来查看兼容的过滤器概述。

例 4.12 将 SkipFilter 和 ValueFilter 组合起来获取不包含空列值的行，同时过滤掉其他不符合条件的行。

例 4.12 根据另一个过滤器的结果使用 SkipFilter 过滤整行数据

```
Filter filter1 = new ValueFilter(CompareFilter.CompareOp.NOT_EQUAL,
    new BinaryComparator(Bytes.toBytes("val-0")));

Scan scan = new Scan();
scan.setFilter(filter1);①
ResultScanner scanner1 = table.getScanner(scan);
for(Result result : scanner1){
    for(KeyValue kv : result.raw()){
        System.out.println("KV: " + kv + ",Value: " +
            Bytes.toString(kv.getValue()));
    }
}
scanner1.close();

Filter filter2 = new SkipFilter(filter1);

scan.setFilter(filter2);②
ResultScanner scanner2 = table.getScanner(scan);
for(Result result : scanner2){
    for(KeyValue kv : result.raw()){
        System.out.println("KV: " + kv + ",Value: " +
            Bytes.toString(kv.getValue()));
    }
}
scanner2.close();
```

^① 各种过滤器方法在 4.1.6 节中讨论。

- ❶ 添加 ValueFilter 到第一个扫描中。
- ❷ 在第二个扫描中添加一个用 SkipFilter 装饰过的过滤器。

运行时示例代码应当有如下输出，每次调用所得结果的顺序有可能不同：

```
Adding rows to table...
Results of scan #1:
KV: row-01/colfam1:col-00/0/Put/vlen=5,Value: val-4
KV: row-01/colfam1:col-01/1/Put/vlen=5,Value: val-2
KV: row-01/colfam1:col-02/2/Put/vlen=5,Value: val-4
KV: row-01/colfam1:col-03/3/Put/vlen=5,Value: val-3
KV: row-01/colfam1:col-04/4/Put/vlen=5,Value: val-1
KV: row-02/colfam1:col-00/0/Put/vlen=5,Value: val-3
KV: row-02/colfam1:col-01/1/Put/vlen=5,Value: val-1
KV: row-02/colfam1:col-03/3/Put/vlen=5,Value: val-4
KV: row-02/colfam1:col-04/4/Put/vlen=5,Value: val-1
...
Total KeyValue count for scan #1: 122

Results of scan #2:
KV: row-01/colfam1:col-00/0/Put/vlen=5,Value: val-4
KV: row-01/colfam1:col-01/1/Put/vlen=5,Value: val-2
KV: row-01/colfam1:col-02/2/Put/vlen=5,Value: val-4
KV: row-01/colfam1:col-03/3/Put/vlen=5,Value: val-3
KV: row-01/colfam1:col-04/4/Put/vlen=5,Value: val-1
KV: row-07/colfam1:col-00/0/Put/vlen=5,Value: val-4
KV: row-07/colfam1:col-01/1/Put/vlen=5,Value: val-1
KV: row-07/colfam1:col-02/2/Put/vlen=5,Value: val-1
KV: row-07/colfam1:col-03/3/Put/vlen=5,Value: val-2
KV: row-07/colfam1:col-04/4/Put/vlen=5,Value: val-4
...
Total KeyValue count for scan #2: 50
```

第一次扫描返回了所有非空的列。由于这个值是之前赋的随机值，所以在一行中可能有一个或多个列的值为空。一些行的某些列因为值为空而不被返回。

第二次扫描由于包装了第一个过滤器，则所有包括空列的行都会被丢弃。用户可以从终端输出中看到完整的多行数据，即最初创建的 5 列。最后扫描中输出的 KeyValue 总数说明了 SkipFilter 筛掉的数据更多。

2. 全匹配过滤器 (WhileMatchFilter)

第二个附加过滤器与之前的附加过滤器相似，不过当一条数据被过滤掉时，它就会直接放弃这次扫描操作。它使用其封装的过滤器来检查 KeyValue，并确认是否有一行数据因为行键或是列被跳过而过滤。^①

^① 见表 4-5 中有关过滤器兼容的概述。

例 4.13 与之前的例子稍有不同，它使用了不同的过滤器来展示附加类的功能。

例 4.13 基于另一个过滤器输出的结果使用过滤器来跳过整行结果

```
Filter filter1 = new RowFilter(CompareFilter.CompareOp.NOT_EQUAL,
    new BinaryComparator(Bytes.toBytes("row-05")));

Scan scan = new Scan();
scan.setFilter(filter1);
ResultScanner scanner1 = table.getScanner(scan);
for(Result result : scanner1){
    for(KeyValue kv : result.raw()){
        System.out.println("KV: " + kv + ",Value: " +
            Bytes.toString(kv.getValue()));
    }
}
scanner1.close();

Filter filter2 = new WhileMatchFilter(filter1);

scan.setFilter(filter2);
ResultScanner scanner2 = table.getScanner(scan);
for(Result result : scanner2){
    for(KeyValue kv : result.raw()){
        System.out.println("KV: " + kv + ",Value: " +
            Bytes.toString(kv.getValue()));
    }
}
scanner2.close();
```

运行示例代码，可以在控制台得到如下输出：

```
Adding rows to table...
Results of scan #1:
KV: row-01/colfam1:col-00/0/Put/vlen=9,Value: val-01.00
KV: row-02/colfam1:col-00/0/Put/vlen=9,Value: val-02.00
KV: row-03/colfam1:col-00/0/Put/vlen=9,Value: val-03.00
KV: row-04/colfam1:col-00/0/Put/vlen=9,Value: val-04.00
KV: row-06/colfam1:col-00/0/Put/vlen=9,Value: val-06.00
KV: row-07/colfam1:col-00/0/Put/vlen=9,Value: val-07.00
KV: row-08/colfam1:col-00/0/Put/vlen=9,Value: val-08.00
KV: row-09/colfam1:col-00/0/Put/vlen=9,Value: val-09.00
KV: row-10/colfam1:col-00/0/Put/vlen=9,Value: val-10.00
Total KeyValue count for scan #1: 9
Results of scan #2:
KV: row-01/colfam1:col-00/0/Put/vlen=9,Value: val-01.00
KV: row-02/colfam1:col-00/0/Put/vlen=9,Value: val-02.00
KV: row-03/colfam1:col-00/0/Put/vlen=9,Value: val-03.00
KV: row-04/colfam1:col-00/0/Put/vlen=9,Value: val-04.00
Total KeyValue count for scan #2: 4
```

第一个扫描使用 `RowFilter` 跳过十行中的一行，其余都被返回到了客户端。第二个扫描使用了全匹配过滤器 (`WhileMatchFilter`)，在遇到行 `row-05` 后结束了整个扫

描过程。



附加过滤器与其他过滤器都实现了 `Filter` 接口。这样可以把自身的功能与其他过滤器加以组合，以变成一个有新功能的过滤器。

4.1.5 FilterList

到目前为止，我们已经向用户展示了包装过的或没有包装过的过滤器，以及如何通过某种维度（行、列、版本号）使用过滤器来对表进行筛选。实际应用中，用户可能需要多个过滤器共同限制返回到客户端的结果，`FilterList`（过滤器列表）提供了这项功能。



与其他单一功能过滤器一样，`FilterList` 类实现了 `Filter` 接口。所以它可以通过组合多个过滤器的功能来实现某种效果，从而代替提供这类效果的过滤器。

用户使用以下构造器创建 `FilterList` 实例时，需要提供多种不同的参数：

```
FilterList(List<Filter> rowFilters)
FilterList(Operator operator)
FilterList(Operator operator, List<Filter> rowFilters)
```

参数 `rowFilters` 以列表的形式组合过滤器，参数 `operator` 决定了组合它们的结果。表 4-3 提供了一些可选的操作符，默认值是 `MUST_PASS_ALL`，当用户不需要其他操作符时可以在构造器中省略该参数。

表 4-3 `FilterList.Operator` 的可选枚举值

操作	描述
<code>MUST_PASS_ALL</code>	当所有过滤器都允许包含这个值时，这个值才会被包含在结果中，也就是说没有过滤器会忽略这个值
<code>MUST_PASS_ONE</code>	只要有一个过滤器允许包括这个值，那这个值就会包含在结果中

当创建 `FilterList` 实例之后，可以用以下方法添加过滤器：

```
void addFilter(Filter filter)
```

每个 `FilterList` 只能添加一个操作符，但用户可以随意地向已经存在的 `FilterList` 实例中添加 `FilterList` 实例，这样可以构造一组多级的过滤器，同时它们可以与用户需要的操作符进行组合。

用户也可以通过控制 `List` 中过滤器的顺序来进一步精确地控制过滤器的执行顺序。例如，使用 `ArrayList` 可以保证过滤器的执行顺序与它们添加到列表中的顺序一致。具体见例 4.14。

例 4.14 使用过滤器列表组合单一功能的过滤器

```
List<Filter> filters = new ArrayList<Filter>();

Filter filter1 = new RowFilter(CompareFilter.CompareOp.GREATER_OR_EQUAL,
    new BinaryComparator(Bytes.toBytes("row-03")));
filters.add(filter1);

Filter filter2 = new RowFilter(CompareFilter.CompareOp.LESS_OR_EQUAL,
    new BinaryComparator(Bytes.toBytes("row-06")));
filters.add(filter2);

Filter filter3 = new QualifierFilter(CompareFilter.CompareOp.EQUAL,
    new RegexStringComparator("col-0[03]"));
filters.add(filter3);

FilterList filterList1 = new FilterList(filters);

Scan scan = new Scan();
scan.setFilter(filterList1);
ResultScanner scanner1 = table.getScanner(scan);
for(Result result : scanner1){
    for(KeyValue kv : result.raw()){
        System.out.println("KV: " + kv + ",Value: " +
            Bytes.toString(kv.getValue()));
    }
}
scanner1.close();

FilterList filterList2 = new FilterList(
    FilterList.Operator.MUST_PASS_ONE, filters);

scan.setFilter(filterList2);
ResultScanner scanner2 = table.getScanner(scan);
for(Result result : scanner2){
    for(KeyValue kv : result.raw()){
        System.out.println("KV: " + kv + ",Value: " +
            Bytes.toString(kv.getValue()));
    }
}
scanner2.close();
```

第一个扫描中的过滤器过滤了许多数据，正是由于列表中任意一个过滤器过滤了该数据，该数据都会被丢弃。只有当数据经过了所有过滤器的筛选才会被传回客户端。

相反，第二个扫描结果中包含了所有的行和列。这是因为 `FilterList` 操作符是 `MUST_PASS_ONE`，只要数据通过了一个过滤器的过滤就会被返回，再则由于所有的行都至少符合一个过滤器的要求，所以所有行都被返回了。

4.1.6 自定义过滤器

最后，用户可能需要按各自的需求实现自定义过滤器。用户可以实现 `Filter` 接口或者直接继承 `FilterBase` 类，后者已经为接口中所有成员方法提供了默认实现。

`Filter` 接口的结构如下：

```
public interface Filter extends Writable {
    public enum ReturnCode {
        INCLUDE, SKIP, NEXT_COL, NEXT_ROW, SEEK_NEXT_USING_HINT
    }
    public void reset()
    public boolean filterRowKey(byte[] buffer, int offset, int length)
    public boolean filterAllRemaining()
    public ReturnCode filterKeyValue(KeyValue v)
    public void filterRow(List<KeyValue> kvs)
    public boolean hasFilterRow()
    public boolean filterRow()
    public KeyValue getNextKeyHint(KeyValue currentKV)
```

接口中有一个公有的枚举类型，叫做 `ReturnCode`，它被 `filterKeyValue()` 方法用于通知执行框架，进而决定如何进行下一步的工作。过滤器可以跳过一个值、一列的剩余部分或一行的剩余部分，而不用遍历所有数据。因此获取数据的效率会大大提升。



服务器端可能还是需要遍历整行数据来查找匹配数据，不过使用 `filterKeyValue()` 提供的返回值优化后还是可以减少许多工作量。

表 4-4 展示了所有的取值以及每个值对应的含义。

表 4-4 `Filter.ReturnCode` 的值类型

返回值	描述
INCLUDE	在结果中包括这个 <code>KeyValue</code> 实例
SKIP	跳过这个 <code>KeyValue</code> 实例并继续处理接下来的工作
NEXT_COL	跳过当前列并继续处理后面的列。例如， <code>TimestampsFilter</code> 使用了这个返回值
NEXT_ROW	与上面的行为相似，跳过当前行并继续处理下一行。例如， <code>RowFilter</code> 使用了这个返回值
SEEK_NEXT_USING_HINT	一些过滤器需要跳过一系列的值，此时需要使用这个返回值通知执行框架使用 <code>getNextKeyHint()</code> 来决定跳到什么位置。例如， <code>ColumnPrefixFilter</code> 使用了这个功能

上面的大多数方法都在客户端获取检索操作（如扫描操作）的不同阶段时被调用。调整调用次序，用户可以以如下的预期顺序执行。

```
filterRowKey(byte[] buffer, int offset, int length)
```

本方法使用 Filter 的实现方法检查行键，用户可以跳过整行数据以避免之后的处理。RowFilter 使用这个方法筛选符合条件的行并返回给客户端。

```
filterKeyValue(KeyValue v)
```

如果本行数据没有被之前的方法过滤掉，那么执行框架会调用这个方法检查这一行中每个 KeyValue 实例。同时按照 ReturnCode 处理当前值。

```
filterRow(List<KeyValue> kvs)
```

一旦所有行和列经过之前两个方法的检查之后，这个方法就会被调用。本方法让用户可以访问之前两个方法筛选出来的 KeyValue 实例。DependentColumnFilter 过滤器使用这个方法过滤与参考列不匹配的数据。

```
filterRow()
```

以上所有方法调用完成之后，filterRow() 方法会被执行。PageFilter 使用当前方法来检查在一次迭代分页中返回的行数是否达到预期的页大小，如果达到页大小则返回 true。默认返回值是 false，此时结果中包含当前行。

```
reset()
```

在迭代扫描中为每个新行重置过滤器。服务器端读取一行数据后，这个方法会被隐式地调用。这个方法在 get 和 scan 中都会被按以上规则调用，很明显它对于前者没有任何效果，因为 get 只取单独的一行。

```
filterAllRemaining()
```

当这个方法返回 true 时，可以用于结束整个扫描操作。用户使用这个方法被过滤器用于提供上述的优化——提前结束。在一个过滤中，如果这个方法返回 false，扫描操作会继续执行，同时前面介绍的方法会被再次调用。显然，这个方法对 get 操作也没有用。

filterRow() 和批量处理模式

一个过滤器使用 filterRow() 过滤行或使用 filterRow(List) 修改返回结果列表时，必需重载 hasRowFilter() 方法让其返回 true。

框架使用这个标志位来保证过滤器与扫描操作的各个参数兼容。特别是当这些过滤器与扫描的批量处理模式冲突时：当扫描使用批量处理模式传送数据时，之前介绍的方法不会在每次批量操作时调用，而是在当前行数据结束时被调用。

图 4-2 展示了处理一行数据时，过滤器中各方法的逻辑流程。还有一些更细化的流程是

关于过滤器如何在列级别中起作用的，但这些内容并没有被包含到这张图中。

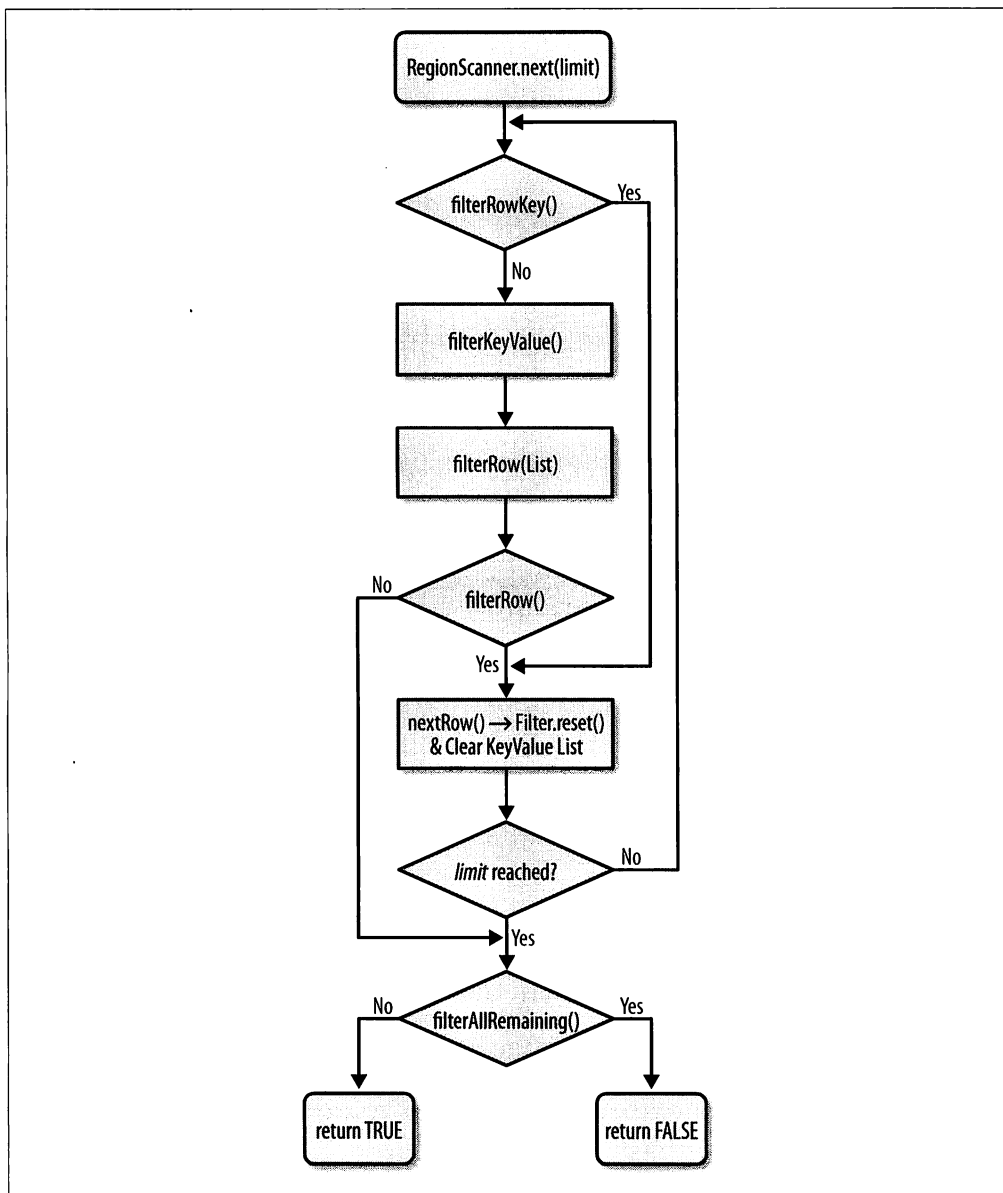


图 4-2 过滤器处理一行数据的逻辑流程

例 4.15 基于 `FilterBase` 提供的方法实现了一个自定义过滤器，并重载了需要修改的方法。过滤器首先假设所有行都要被过滤掉，也就是说，所有行都不在结果中返回。只有当有任意一列中的值与设定值相等时，这一行才会被返回到客户端。

例 4.15 实现一个只让一些特定行通过的过滤器

```
public class CustomFilter extends FilterBase{

    private byte[] value = null;
    private boolean filterRow = true;

    public CustomFilter() {
        super();
    }

    public CustomFilter(byte[] value){
        this.value = value;❶
    }

    @Override
    public void reset() {
        this.filterRow = true;❷
    }

    @Override
    public ReturnCode filterKeyValue(KeyValue kv){
        if(Bytes.compareTo(value,kv.getValue())== 0){
            filterRow = false;❸
        }
        return ReturnCode.INCLUDE;❹
    }

    @Override
    public boolean filterRow() {
        return filterRow;❺
    }

    @Override
    public void write(DataOutput dataOutput)throws IOException {
        Bytes.writeByteArray(dataOutput,this.value);❻
    }

    @Override
    public void readFields(DataInput dataInput)throws IOException {
        this.value = Bytes.readByteArray(dataInput);❼
    }
}
```

- ❶ 设置要比较的值。
- ❷ 每当有新行时重置过滤器的标志位。
- ❸ 当有值匹配设定值时，让这一行通过过滤。

- ④ 总是先包含 KeyValue 实例，直到 filterRow() 决定是否过滤这一行。
- ⑤ 这是实际上决定数据是否被返回的一行代码，其基于标志位判断。
- ⑥ 把设定值写到 DataOutput 中，服务器端实例化过滤器时可以读到设定值。
- ⑦ 服务端使用这个方法来自初始化过滤器实例，所以客户端的设定值可以被读到。

用户自定义过滤器部署

一旦用户完成过滤器的编写就需要将其部署到 HBase 上。用户首先需要编译好这个类，同时打成 JAR 包，并保证可以被 region 服务器调用。

用户可以使用编译系统来准备配置用的 JAR 文件，同时使用配置管理系统把文件分发到每个 region 服务器中。文件分发完成后用户需要修改 *hbase-env.sh* 文件，如下：

```
# Extra Java CLASSPATH elements. Optional.
# export HBASE_CLASSPATH=
export HBASE_CLASSPATH="/hbase-book/ch04/target/hbase-book-ch04-1.0.jar"
```

使用 Maven 从本书的源码中编译出 JAR 文件。因为在单机模式下安装，所以这里使用绝对路径，也就是说开发环境和 HBase 在同一个物理机上。

注意用户需要重启 HBase 的守护进程，才能使配置生效。完成这些工作之后就可以测试过滤器的功能了。

例 4.16 使用了新的用户自定义过滤器来查找包含特定值的行，同时还使用了 FilterList。

例 4.16 使用用户自定义过滤器

```
List<Filter> filters = new ArrayList<Filter>();

Filter filter1 = new CustomFilter(Bytes.toBytes("val-05.05"));
filters.add(filter1);

Filter filter2 = new CustomFilter(Bytes.toBytes("val-02.07"));
filters.add(filter2);

Filter filter3 = new CustomFilter(Bytes.toBytes("val-09.00"));
filters.add(filter3);

FilterList filterList = new FilterList(
    FilterList.Operator.MUST_PASS_ONE, filters);

Scan scan = new Scan();
scan.setFilter(filterList);
```

```

ResultScanner scanner = table.getScanner(scan);
for(Result result : scanner){
    for(KeyValue kv : result.raw()){
        System.out.println("KV: " + kv + ",Value: " +
            Bytes.toString(kv.getValue()));
    }
}
scanner.close();

```

使用以上例子，应当会有如下输出：

```

Adding rows to table...
Results of scan:
KV: row-02/colfam1:col-00/1301507323088/Put/vlen=9,Value: val-02.00
KV: row-02/colfam1:col-01/1301507323090/Put/vlen=9,Value: val-02.01
KV: row-02/colfam1:col-02/1301507323092/Put/vlen=9,Value: val-02.02
KV: row-02/colfam1:col-03/1301507323093/Put/vlen=9,Value: val-02.03
KV: row-02/colfam1:col-04/1301507323096/Put/vlen=9,Value: val-02.04
KV: row-02/colfam1:col-05/1301507323104/Put/vlen=9,Value: val-02.05
KV: row-02/colfam1:col-06/1301507323108/Put/vlen=9,Value: val-02.06
KV: row-02/colfam1:col-07/1301507323110/Put/vlen=9,Value: val-02.07
KV: row-02/colfam1:col-08/1301507323112/Put/vlen=9,Value: val-02.08
KV: row-02/colfam1:col-09/1301507323113/Put/vlen=9,Value: val-02.09
KV: row-05/colfam1:col-00/1301507323148/Put/vlen=9,Value: val-05.00
KV: row-05/colfam1:col-01/1301507323150/Put/vlen=9,Value: val-05.01
KV: row-05/colfam1:col-02/1301507323152/Put/vlen=9,Value: val-05.02
KV: row-05/colfam1:col-03/1301507323153/Put/vlen=9,Value: val-05.03
KV: row-05/colfam1:col-04/1301507323154/Put/vlen=9,Value: val-05.04
KV: row-05/colfam1:col-05/1301507323155/Put/vlen=9,Value: val-05.05
KV: row-05/colfam1:col-06/1301507323157/Put/vlen=9,Value: val-05.06
KV: row-05/colfam1:col-07/1301507323158/Put/vlen=9,Value: val-05.07
KV: row-05/colfam1:col-08/1301507323158/Put/vlen=9,Value: val-05.08
KV: row-05/colfam1:col-09/1301507323159/Put/vlen=9,Value: val-05.09
KV: row-09/colfam1:col-00/1301507323192/Put/vlen=9,Value: val-09.00
KV: row-09/colfam1:col-01/1301507323194/Put/vlen=9,Value: val-09.01
KV: row-09/colfam1:col-02/1301507323196/Put/vlen=9,Value: val-09.02
KV: row-09/colfam1:col-03/1301507323199/Put/vlen=9,Value: val-09.03
KV: row-09/colfam1:col-04/1301507323201/Put/vlen=9,Value: val-09.04
KV: row-09/colfam1:col-05/1301507323202/Put/vlen=9,Value: val-09.05
KV: row-09/colfam1:col-06/1301507323203/Put/vlen=9,Value: val-09.06
KV: row-09/colfam1:col-07/1301507323204/Put/vlen=9,Value: val-09.07
KV: row-09/colfam1:col-08/1301507323205/Put/vlen=9,Value: val-09.08
KV: row-09/colfam1:col-09/1301507323206/Put/vlen=9,Value: val-09.09

```

与预期的一样，一行中有一列的值与预定值匹配时，这一行数据就会被包含在返回结果中。

4.1.7 过滤器总结

表 4-5 展示了默认提供的过滤器中的一些属性和兼容性。标注 ✓ 说明属性可用，标注 × 表示属性缺失。

表 4-5 过滤器属性和它们之间的兼容性

Filter	Batch ^a	Skip ^b	While-Match ^c	List ^d	Early Out ^e	Gets ^f	Scans ^g
RowFilter	✓	✓	✓	✓	✓	×	✓
FamilyFilter	✓	✓	✓	✓	×	✓	✓
QualifierFilter	✓	✓	✓	✓	×	✓	✓
ValueFilter	✓	✓	✓	✓	×	✓	✓
DependentColumnFilter	×	✓	✓	✓	×	✓	✓
SingleColumnValueFilter	✓	✓	✓	✓	×	×	✓
SingleColumnValueExcludeFilter	✓	✓	✓	✓	×	×	✓
PrefixFilter	✓	×	✓	✓	✓	×	✓
PageFilter	✓	×	✓	✓	✓	×	✓
KeyOnlyFilter	✓	✓	✓	✓	×	✓	✓
FirstKeyOnlyFilter	✓	✓	✓	✓	×	✓	✓
InclusiveStopFilter	✓	×	✓	✓	✓	×	✓
TimestampsFilter	✓	✓	✓	✓	×	✓	✓
ColumnCountGetFilter	✓	✓	✓	✓	×	✓	×
ColumnPaginationFilter	✓	✓	✓	✓	×	✓	✓
ColumnPrefixFilter	✓	✓	✓	✓	×	✓	✓
RandomRowFilter	✓	✓	✓	✓	×	×	✓
SkipFilter	✓	✓/× ^h	✓/× ^h	✓	×	×	✓
WhileMatchFilter	✓	✓/× ^h	✓/× ^h	✓	✓	×	✓
FilterList	✓/× ^h	✓/× ^h	✓/× ^h	✓	✓/× ^h	✓	✓

a. 过滤器可以支持 `Scan.setBatch()` 方法，也就是说可以支持扫描的批量模式。

b. 过滤器可以被 `SkipFilter` 类包装。

c. 过滤器可以被 `WhileMatchFilter` 类包装。

d. 过滤器可以被合并到 `FilterList` 类中。

e. 过滤器为了优化性能可以在发现没有符合条件的行之后提前停止扫描操作。

f. 过滤器可以在 `Get` 实例中发挥作用。

g. 过滤器可以在 `Scan` 实例中发挥作用。

h. 依赖于被包含的过滤器。

4.2 计数器

除了以上讨论的功能之外，HBase 还有一个高级功能：计数器（counter）。许多收集统计信息的应用有点击流或在线广告意见，这些应用需要被收集到日志文件中用于后续的分析。用户可以使用计数器做实时统计，从而放弃延时较高的批量处理操作。

4.2.1 计数器简介

与之前介绍的原子操作检查并修改（check-and-modify）一样，HBase 也有一种机制可以将列当作计数器。否则，如果用户需要对一行数据加锁，然后读取数据，再对当前数据做加法，最后写回 HBase 并释放该行锁，从而其他写程序可以访问该行数据。这样做会引起大量的资源竞争问题，尤其是当客户端进程崩溃之后，尚未释放的锁需要等待超时恢复——这会在一个高负载的系统中引起灾难性的后果。

客户端 API 提供了专门的方法来完成这种读取并修改（read-and-modify）操作，同时在单独一次客户端的调用过程中保证原子性。早期的 HBase 版本只会在每次计数器更新操作中使用一个 RPC 请求，不过新版本的 HBase 中 CRUD 操作（参考阅读 3.2 节）开始使用与此相同的机制，让许多更新计数器的请求都可以在一次 RPC 中完成。



虽然用户可以一次更新多个计数器，但它们都必须属于同一行。更新多行的计数器需要通过独立的 API 调用，即多个 RPC 请求。batch() 方法调用目前并不支持 Increment 实例，不过这种情况今后可能会改变。

用户分别了解每个类型之前，还需要了解一些计数器在列一级的工作细节。以下是用 Shell 创建表之后，两次增加同一个计数器的值，最后查询计数器当前值的例子。

```
hbase(main):001:0> create 'counters','daily','weekly','monthly'
0 row(s) in 1.1930 seconds

hbase(main):002:0> incr 'counters','20110101','daily:hits',1
COUNTER VALUE = 1

hbase(main):003:0> incr 'counters','20110101','daily:hits',1
COUNTER VALUE = 2

hbase(main):004:0> get_counter 'counters','20110101','daily:hits'
COUNTER VALUE = 2
```

每次 incr 调用返回这个计数器的新值。最后检查时使用了 get_counter，并显示当前计数器值与所预期的一致。



终端的 `incr` 命令格式如下:

```
incr '<table>','<row>','<column>', [<increment-value>]
```

初始化计数器

用户不用初始化计数器, 当用户第一次使用计数器时, 计数器将被自动设为 0, 也就是说当用户创建一个新列时, 计数器的值是 0。第一次增加操作会返回 1 或增加设定的值。用户也可以直接读写一个计数器, 不过需要使用以下方法来解码:

```
Bytes.toLong()
```

并使用以下方法来编码:

```
Bytes.toBytes(long)
```

特别是对于后一种情况, 需要保证参数是长整型的 (`long`)。用户也可以直接将变量或数字类型转换为长整型, 如下所示:

```
byte[] b1 = Bytes.toBytes(1L)
byte[] b2 = Bytes.toBytes((long)var)
```

如果用户使用 `put` 方法错误地初始化了一个计数器值, 用户可能会经历如下过程:

```
hbase(main):001:0> put 'counters','20110101','daily:clicks','1'
0 row(s) in 0.0540 seconds
```

当用户增加这个计数器的值时会得到如下结果:

```
hbase(main):013:0> incr 'counters','20110101','daily:clicks',1
COUNTER VALUE = 3530822107858468865
```

这个结果不是预期中的 2! 这是因为 `put` 方法按错误的格式存储了计数器: 值是字符 1, 同时这个值是一个字节, 并不是表示 Java 语言中长度为 8 的 `long` 类型值的字节数组

注意: 这一个字节被 Shell 当做字节数组存储时, 最高位被设为 ASCII 码的字母 1 的值 49, 这是基于 Ruby 的 Shell 脚本接收到的用户输入值。增加这个值的最低位字节同时将其转化为 `long` 类型, 就会得到非常大并且难以预期的数值, 如前面代码中的 `COUNT VALUE` 所示:

```
hbase(main):001:0> include_class org.apache.hadoop.hbase.util.Bytes
=> Java::OrgApacheHadoopHbaseUtil::Bytes
hbase(main):002:0> Bytes::toLong([49,0,0,0,0,0,0,1].to_java :byte)
=> 3530822107858468865
```

用户可以使用 `get` 请求访问这个计数器，结果如下：

```
hbase(main):005:0> get 'counters','20110101'
COLUMN      CELL
daily:hits timestamp=1301570823471,value=\x00\x00\x00\x00\x00\x00\x00\x02
1 row(s) in 0.0600 seconds
```

这样得到的结果可读性较差，但是这表明了一个计数器就是一个与其他列类似的简单列。用户也可以指定一个更大的递增值：

```
hbase(main):006:0> incr 'counters',
'20110101','daily:hits',20
COUNTER VALUE = 22

hbase(main):007:0> get 'counters','20110101'
COLUMN      CELL
daily:hits timestamp=1301574412848,value=\x00\x00\x00\x00\x00\x00\x00\x16
1 row(s) in 0.0400 seconds

hbase(main):008:0> get_counter 'counters',
'20110101','daily:hits'
COUNTER VALUE = 22
```

用户直接读取计数器时得到的是字节数组，Shell 把每个字节按十六进制数打印。使用 `get_counter` 可以以可读格式返回数据，并确认递增数据可行，并且与预期一致。

最后，用户不只可以用 `incr` 命令来对一个计数器加值，也可以取回计数器当前值或者减少当前值。实际上，用户也可以完全忽略初始值，默认情况下是 1。

```
hbase(main):004:0> incr 'counters','20110101',
'daily:hits'
COUNTER VALUE = 3

hbase(main):005:0> incr 'counters','20110101','daily:hits'
COUNTER VALUE = 4

hbase(main):006:0> incr 'counters','20110101','daily:hits',0
COUNTER VALUE = 4

hbase(main):007:0> incr 'counters','20110101','daily:hits',-1
COUNTER VALUE = 3

hbase(main):008:0> incr 'counters','20110101','daily:hits',-1
COUNTER VALUE = 2
```

使用增加值，即 `incr` 命令的最后一个参数，用户可以在表 4-6 中观察不同值带来的行为影响。

表 4-6 增加值和对计数器产生的作用

值	作用
比零大的值	按给定值增加计数器中的数值
零	得到计数器的当前值，与 Shell 命令 <code>get_counter</code> 的返回值相同
比零小的值	减少计数器的当前值

显然，使用 `incr` 命令只能一次操作一个计数器。用户也可以使用后面介绍的客户端 API 操作计数器。

4.2.2 单计数器

第一种增加操作只能操作一个计数器：用户需要自己设定列，方法由 `HTable` 提供，如下所示：

```
long incrementColumnValue(byte[] row,byte[] family,byte[] qualifier,
    long amount) throws IOException
long incrementColumnValue(byte[] row,byte[] family,byte[] qualifier,
    long amount,boolean writeToWAL) throws IOException
```

这两种方法都需要提供列的坐标（`coordinates`）和增加值，除此之外这两种方法只在参数 `writeToWAL` 上有差别，这个参数的作用与 `Put.setWriteToWAL()` 方法一致。

忽略该参数会直接使用默认值 `true`，也就是说，**WAL** 是有效的。

抛开这个参数，用户可以按照下面的示例轻松地使用这些方法。

例 4.17 使用单计数器自增方法

```
HTable table = new HTable(conf,"counters");

long cnt1 = table.incrementColumnValue(Bytes.toBytes("20110101"), ❶
    Bytes.toBytes("daily"),Bytes.toBytes("hits"),1);
long cnt2 = table.incrementColumnValue(Bytes.toBytes("20110101"), ❷
    Bytes.toBytes("daily"),Bytes.toBytes("hits"),1);

long current = table.incrementColumnValue(Bytes.toBytes("20110101"), ❸
    Bytes.toBytes("daily"),Bytes.toBytes("hits"),0);

long cnt3 = table.incrementColumnValue(Bytes.toBytes("20110101"), ❹
    Bytes.toBytes("daily"), Bytes.toBytes("hits"), -1);
```

- ❶ 计数器值加 1。
- ❷ 第二次给计数器值加 1。
- ❸ 得到计数器当前值，不做自增操作。

④ 计数器值减 1。

对应的输出如下：

```
cnt1: 1,cnt2: 2,current: 2,cnt3: 1
```

与之前使用的 Shell 命令一样，API 调用也有相同的作用：使用正值时增加了计数器的值，使用 0 时可以得到当前计数器的值，使用负值时可以减少当前计数器的值。

4.2.3 多计数器

另一个增加计数器值的途径是 HTable() 的方法 increment()。工作模式与 CRUD 操作类似，请使用以下方法完成该功能：

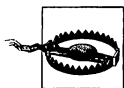
```
Result increment(Increment increment) throws IOException
```

用户需要创建一个 Increment 实例，同时需要填充一些相应的细节到该实例中，例如，计数器的坐标。构造器如下：

```
Increment() {}  
Increment(byte[] row)  
Increment(byte[] row, RowLock rowLock)
```

用户构造 Increment 实例时需要传入行键，此行应当包含此实例需要通过 increment() 方法修改的所有计数器。

可选参数 rowLock 设置了用户自定义锁实例，这样可以使本次操作完全在用户的控制下完成，例如，当用户需要多次修改同一行时，可以保证其间此行不被其他写程序修改。



虽然用户可以限制其他写程序修改此行的值，但是用户无法限制读操作。事实上，这里并没有保证读操作的原子性。

因为读操作不需要获取锁，所以它可能读到一行中被修改到一半的数据！scan 和 get 操作同样会出现这种情况。

一旦用户使用行键创建了一个 Increment 实例，就需要向其中加入实际的计数器，也就是说，用户需要增加列，使用方法如下：

```
Increment addColumn(byte[] family, byte[] qualifier, long amount)
```

与 Put 方法不同的地方是没有选项来设置版本或时间戳：当做增加操作时，版本都被隐式处理了。同样，这里没有 addFamily() 方法，因为计数器都是特定的列，所以需要特定如此，因此去添加一个列族是没有意义的。

Increment 类的特别功能是可以添加一个时间范围：

```
Increment setTimeRange(long minStamp, long maxStamp)
```

throws IOException

设定计数器的时间范围与之前提到的版本被隐式处理相比有些奇怪。时间范围被送到服务器端来限制内部的 `get` 操作来取得当前这些计数器的值。用户可以使用它来使计数器过期 (`expire`)，例如，用时间划分一行的计数器：用户限制时间范围，可以用来屏蔽比较老的计数器，使它们看上去不存在。一次增加操作会认为这此较老的计数器不存在，并把它们重置为 1。

`Increment` 类提供的其他方法见表 4-7。

表 4-7 `Increment` 类的附加方法概览

方法	描述
<code>getRow()</code>	返回创建 <code>Increment</code> 实例时指定的行键值
<code>getRowLock()</code>	返回当前 <code>Increment</code> 实例中的 <code>RowLock</code> 实例
<code>getLockId()</code>	返回构造器中可选参数 <code>rowLock</code> 的锁 ID，如果没有设置的话，默认该值为 <code>-1L</code>
<code>setWriteToWAL()</code>	允许用户禁用本次操作服务器端默认的 <code>WAL</code> 功能
<code>getWriteToWAL()</code>	返回是否在本次操作中启用 <code>WAL</code> 功能
<code>getTimeRange()</code>	返回与 <code>Increment</code> 实例相关的时间范围，可以使用 <code>setTimeStamp()</code> 方法进行设定
<code>numFamilies()</code>	便捷地取回 <code>FamilyMap</code> 的大小，其中包括添加的所有列的列族
<code>numColumns()</code>	返回将要被处理的列的数目
<code>hasFamilies()</code>	检查是否有列或列族被添加到这个 <code>Increment</code> 实例中
<code>familySet()/getFamilyMap()</code>	使用户可以访问 <code>addColumn()</code> 方法添加的列。 <code>FamilyMap</code> 的键存储的是列族名称，相应的值是添加过的列族下列的列表。 <code>familySet()</code> 方法返回一个列族的 <code>Set</code> 实例，换句话说就是一个只包括列族名的集合

与上面命令行的例子相似，例 4.18 使用了多个增加值来增加、获取或减少一个计数器的值。

例 4.18 增加一行中多个计数器的计数

```
Increment increment1 = new Increment(Bytes.toBytes("20110101"));

increment1.addColumn(Bytes.toBytes("daily"),Bytes.toBytes("clicks"), 1);
increment1.addColumn(Bytes.toBytes("daily"),Bytes.toBytes("hits"), 1);❶
increment1.addColumn(Bytes.toBytes("weekly"),Bytes.toBytes("clicks"), 10);
increment1.addColumn(Bytes.toBytes("weekly"),Bytes.toBytes("hits"), 10);

Result result1 = table.increment(increment1);❷
```

```

for (KeyValue kv : result1.raw()){
    System.out.println("KV: " + kv +
        " Value: " + Bytes.toLong(kv.getValue()));❶
}

Increment increment2 = new Increment(Bytes.toBytes("20110101"));

increment2.addColumn(Bytes.toBytes("daily"),Bytes.toBytes("clicks"),5);
increment2.addColumn(Bytes.toBytes("daily"),Bytes.toBytes("hits"), 1);❷
increment2.addColumn(Bytes.toBytes("weekly"),Bytes.toBytes("clicks"), 0);
increment2.addColumn(Bytes.toBytes("weekly"),Bytes.toBytes("hits"), -5);

Result result2 = table.increment(increment2);

for(KeyValue kv : result2.raw()){
    System.out.println("KV: " + kv +
        " Value: " + Bytes.toLong(kv.getValue()));
}

```

- ❶ 使用不同的增加值增加计数器的计数。
- ❷ 使用上述的计数器更新值调用实际的增加方法，并得到返回结果。
- ❸ 打印 KeyVaule 和返回的计数器计数结果。
- ❹ 使用正、负和零增加值来修改计数器值。

运行上面例子的输出如下：

```

KV: 20110101/daily:clicks/1301948275827/Put/vlen=8 Value: 1
KV: 20110101/daily:hits/1301948275827/Put/vlen=8 Value: 1
KV: 20110101/weekly:clicks/1301948275827/Put/vlen=8 Value: 10
KV: 20110101/weekly:hits/1301948275827/Put/vlen=8 Value: 10

KV: 20110101/daily:clicks/1301948275829/Put/vlen=8 Value: 6
KV: 20110101/daily:hits/1301948275829/Put/vlen=8 Value: 2
KV: 20110101/weekly:clicks/1301948275829/Put/vlen=8 Value: 10
KV: 20110101/weekly:hits/1301948275829/Put/vlen=8 Value: 5

```

比较两次得到的结果，用户可以发现达到的效果如所预料的一样。

4.3 协处理器

到目前为止，用户已经掌握了如何使用过滤器来减少服务器端通过网络返回到客户端的数据量。HBase 中还有一些特性让用户甚至可以把一部分计算也移动到数据的存放端：协处理器（coprocessor）。

4.3.1 协处理器简介

使用客户端 API，配合筛选机制，例如，使用过滤器或限制列族的范围，都可以控制被返回到客户端的数据量。如果可以更进一步优化会更好，例如，数据的处理流程直接放到服务器端执行，然后仅返回一个小的处理结果集。这类似于一个小型的 MapReduce 框架，该框架将工作分发到整个集群。

协处理器允许用户在 **region** 服务器上运行自己的代码，更准确地说是允许用户执行 **region** 级的操作，并且可以使用与 **RDBMS** 中触发器（**trigger**）类似的功能。在客户端，用户不用关心操作具体在哪里执行，HBase 的分布式框架会帮助用户把这些工作变得透明。

这里用户可以监听一些隐式的事件，并利用其来完成一些辅助任务。如果这些还不够，用户还可以自己扩展现有的 **RPC** 协议来引入自己的调用，这些调用由客户端触发，并在服务器端执行。

例如，用户自定义过滤器（参考 4.1.6 节），用户需要编写一些特定的 **Java** 类来实现特定接口。用户需要将其编译成 **JAR** 文件，并使服务器端可以加载。**region** 服务器进程会实例化这些类，并在正确的系统环境中运行。与过滤器不同的是，协处理器可以动态加载。这一点可以使用户在 **HBase** 集群运行中扩展其功能。

有很多可以使用协处理器的场景，例如，使用钩子关联行修改操作来维护一个辅助索引，或维护一些数据间的引用完整性。过滤器也可以被增强为有状态的，因此它们可以做一些跨行级的决策。如 **RDBMS** 中常见的 **sum()**、**avg()** 等聚合函数，以及 **SQL** 也可以在服务器端完成，服务器端只需在本地扫描并统计数据，然后把数值结果返回给客户端。



另一个适合使用协处理器的场景是权限控制。**HBase 0.92** 的授权认证和审查功能就是基于协处理器完成的。它们在系统启动时被加载，并提供与触发器类似的钩子函数来检查一个用户是否通过了认证，以及是否有权限访问表中的某些数据。

协处理器框架已经提供了一些类，用户可以通过继承这些类来扩展自己的功能。这些类主要分为两大类，即 **observer** 和 **endpoint**。以下是各个有功能的简要介绍。

observer

这一类协处理器与触发器（**trigger**）类似：回调函数（也被称作钩子函数，**hook**）在一些特定事件发生时被执行。这些事件包括一些用户产生的事件，也包括服务器端内部自动产生的事件。

协处理器框架提供的接口如下所示。

- **RegionObserver**：用户可以用这种的处理器处理数据修改事件，它们与表的

region 联系紧密。

- MasterObserver: 可以被用作管理或 DDL 类型的操作, 这些是集群级事件。
- WALObserver: 提供控制 WAL 的钩子函数。

observer 提供了一些设计好的回调函数, 每个操作在集群服务器端都可以被调用。

endpoint

除了事件处理之外还需要将用户自定义操作添加到服务器端。用户代码可以被部署到管理数据的服务器端, 例如, 做一些服务器端计算的工作。

endpoint 通过添加一些远程过程调用来动态扩展 RPC 协议。可以把它们理解为与 RDBMS 中类似的存储过程。endpoint 可以与 observer 的实现组合起来直接作用于服务器端的状态。

这些接口都基于 Coprocessor 框架的接口, 以获取一些共有的特性, 同时也可以实现自己特有的功能。

最后, 协处理器可以被链接起来使用, 这个特点与 Java Servlet API 的过滤器请求相似。下面介绍一些协处理器框架中可用的类型。

4.3.2 Coprocessor 类

所有协处理器的类都必须实现这个接口。它定义了协处理器的基本约定, 并使得框架本身的管理变得容易。这里提供了两个被应用于框架的枚举类——Priority 和 State。表 4-8 解释了这些值的含义。

表 4-8 Coprocessor.Priority 枚举类定义的优先级

值	说明
SYSTEM	高优先级, 定义最先被执行的协处理器
USER	定义其他的协处理器, 按顺序执行

协处理器的优先级决定了执行的顺序: 系统 (system) 级协处理器在用户 (user) 级协处理器之前执行。



在同一个优先级中还有一个序号 (sequence number) 的概念, 用来维护协处理器的加载顺序。序号从 0 开始依次增加。

这个数字作用并不大, 但用户可以依靠它们来为同一优先级的协处理器排序: 在同一优先级下, 它们按照其序号递增的顺序执行, 即定义了执行顺序。

在协处理器的生命周期中，它们由框架管理。Coprocessor 接口提供了如下两个方法：

```
void start(CoprocessorEnvironment env) throws IOException;
void stop(CoprocessorEnvironment env) throws IOException;
```

这两个方法在协处理器开始和结束时被调用。CoprocessorEnvironment 用来在协处理器的生命周期中保持其状态。协处理器实例一直被保存在提供的环境中。

表 4-9 列举了它提供的方法。

表 4-9 CoprocessorEnvironment 类提供的方法

方法	说明
String getHBaseVersion()	以字符串的格式返回 HBase 版本
int getVersion()	返回 Coprocessor 接口的版本
Coprocessor getInstance()	返回加载的协处理器实例
Coprocessor.Priority getPriority()	返回协处理器的优先级
int getLoadSequence()	协处理器的序号，当协处理器加载时被设置，这反映了它的执行顺序
HTableInterface getTable(byte[] tableName)	返回与传入的表名参数对应的 HTable 实例，这允许协处理器访问实际的表数据

协处理器应当只与提供给它们的环境进行交互。这样的好处是可以保证没有会被恶意代码用来破坏数据的后门。



协处理器应当使用 getTable() 方法访问表数据。注意这个方法实际上在默认的 HTable 类上添加了特定的安全措施。例如，协处理器不可以对一行数据加锁。

目前为止，还没有阻止用户在协处理器代码中创建 HTable 实例的办法，这些漏洞可能会在今后的版本中被检查出来，并被阻止。

在协处理器实例的生命周期中，Coprocessor 接口的 start() 和 stop() 方法会被框架隐式调用，处理过程中的每一步都有一个状态。表 4-10 列举了协处理器提供的生命周期状态。

表 4-10 Coprocessor.State 枚举类定义的状态

值	说明
UNINSTALLED	协处理器最初的状态，没有环境，也没有被初始化
INSTALLED	实例装载了它的环境参数
STARTING	协处理器将要开始工作，也就是说 start() 方法将要被调用
ACTIVE	一旦 start() 方法被调用，当前状态设置为 active

值	说明
STOPPING	stop()方法被调用之前的状态
STOPPED	一旦 stop()方法将控制权交给框架, 协处理器会被设置为状态 stopped

最后的迷团是 CoprocessorHost 类, 它维护所有协处理器实例和它们专用的环境。它有一些子类, 这些子类应用在不同的使用环境, 例如, master 和 region 服务器等环境。

Coprocessor、CoprocessorEnvironment 和 CoprocessorHost 这 3 个类形成了协处理器类的基础, 基于这 3 个类能够实现更高级的功能。它们支持协处理器的生命周期, 管理协处理器的状态, 同时提供了执行时的环境参数, 以保证协处理器正确执行。此外, 这些类也提供了一个抽象层方便用户更简单的构建自己的实现。

图 4-3 展示了客户端的调用如何与一系列的协处理器进行交互。需要注意的是, 执行顺序在输入输出阶段都是相同的: 首先执行系统级协处理器, 然后是用户级协处理器, 并按它们加载时的顺序执行。

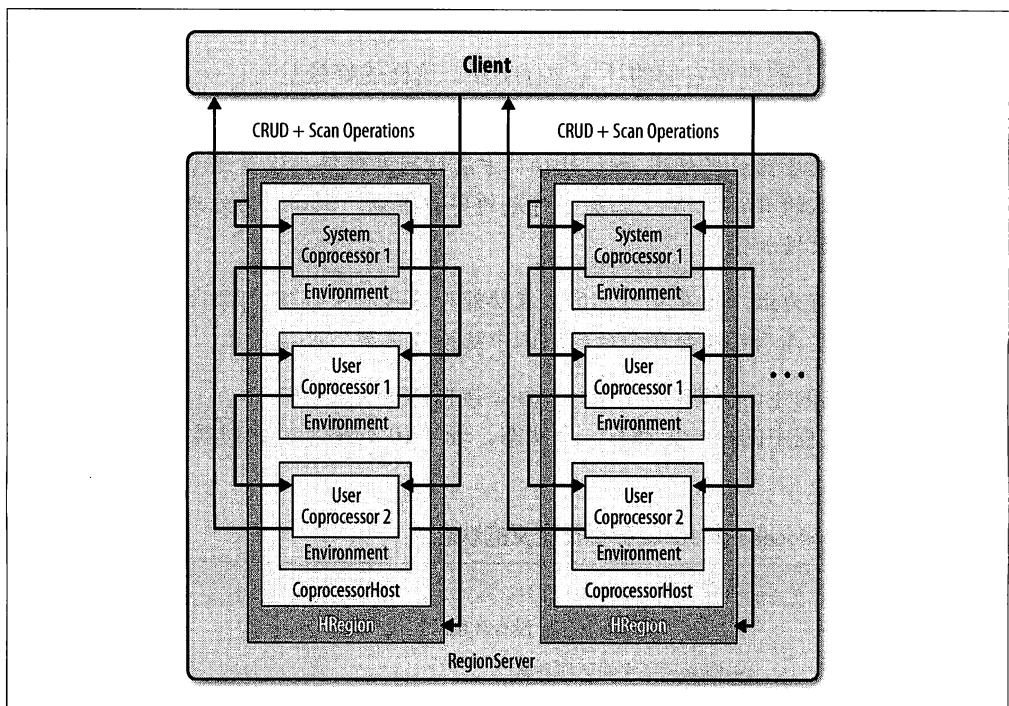


图 4-3 协处理器在它们每个 region 环境中按顺序执行

4.3.3 协处理器加载

加载协处理器的方式有许多种。在用户了解实际的协处理器类型和如何实现自定义类型之前，首先需要了解如何部署它们，这样才能尝试我们提供的例子。

用户可以将协处理器配置为使用静态方式加载，也可以在集群运行时动态加载协处理器。使用配置文件和表模式加载是静态加载方法，接下来会介绍这种方法。不幸的是，现在还没有开放用户动态加载协处理器的 API。^①

1. 从配置中加载

用户可以在 HBase 启动时通过配置文件控制在全局中加载哪些协处理器。用户通过添加以下配置中的一条或几条到 *hbase-site.xml* 文件中来添加协处理器。

```
<property>
  <name>hbase.coprocessor.region.classes</name>
  <value>coprocessor.RegionObserverExample,coprocessor.AnotherCoproprocessor </value>
</property>
<property>
  <name>hbase.coprocessor.master.classes</name>
  <value>coprocessor.MasterObserverExample</value>
</property>
<property>
  <name>hbase.coprocessor.wal.classes</name>
  <value>coprocessor.WALObserverExample,bar.foo.MyWALObserver</value>
</property>
```



将以上例子中的类名替换为用户自己的类名。

配置文件中配置项的顺序非常重要，这个顺序决定了执行顺序。所有协处理器都是以系统级优先级进行加载的。用户应当将全局类配置到这里，这样它们会被优先执行，还可以执行权限操作。安全相关的协处理器都是这样加载的。



配置文件首先在 HBase 启动时被检查。虽然用户可以在其他地方增加系统级优先级的协处理器，但是在配置文件中配置的协处理器是被最先执行的。

这 3 项配置中只有一个会被与之相对应的 CoprocessorHost 的实现加载。例如，`hbase.coprocessor.master.classes` 中定义的协处理器会被 `MasterCoproprocessorHost` 类加载。

表 4-11 展示了每个配置项的作用。

^① 协处理器是最近添加到 HBase 中的，因此最近还有变化。在线的文档和问题跟踪系统都不完善，但在计划增加中。

表 4-11 配置项和它们的作用

属性	协处理器主机	服务类型
hbase.coprocessor.master.classes	MasterCoprocessorHost	master 服务器
hbase.coprocessor.region.classes	RegionCoprocessorHost	region 服务器
hbase.coprocessor.wal.classes	WALCoprocessorHost	region 服务器

当一张表的 **region** 被打开时, `hbase.coprocessor.region.classes` 定义的协处理器会被加载。注意用户不能指定具体是哪张表或哪个 **region** 加载这个类: 默认的协处理器会被每张表和每个 **region** 加载。用户在设计自己的协处理器时要注意这一点。

2. 从表描述符中加载

另一个决定哪些协处理器被加载的选项是表描述符。因为这是针对特定表的, 所以加载的协处理器只针对这个表的 **region**, 同时也只被这些 **region** 的 **region** 服务器使用。换句话说, 用户只能在与 **region** 相关的协处理器上使用这种方法, 而不能在 **master** 或 **WAL** 相关的协处理器上使用。

由于它们是使用表的上下文加载的, 所以与配置文件中加载的协处理器影响所有表相比, 这种方法加载的协处理器更具有针对性。用户需要在表描述符中利用 `HTableDescriptor.setValue()` 方法定义它们。键必须以 **COPROCESSOR** 开头, 值必须符合以下格式:

```
<path-to-jar>|<classname>|<priority>
```

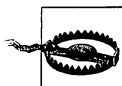
以下是一个定义了两个协处理器的例子, 一个使用系统优先级, 另一个使用用户优先级:

```
'COPROCESSOR$1' => \
    'hdfs://localhost:8020/users/leon/test.jar|coprocessor.Test|SYSTEM'
'COPROCESSOR$2' => \
    '/Users/laura/test2.jar|coprocessor.AnotherTest|USER'
```

`path-to-jar` 可以是一个完整的 **HDFS** 地址或其他 **Hadoop FileSystem** 类支持的地址。第二个协处理器就使用了本地路径。

`classname` 定义了具体的实现类。由于 **JAR** 可能包含许多协处理器类, 但只能给一张表设定一个协处理器。用户应该使用标准的 **Java** 包命名规则来命名指定类。

`priority` 只能是 **SYSTEM** 或 **USER**。



不要在协处理器定义中添加空格。解释十分严格, 添加头尾或间隔字符会使整个配置条目无效。

使用`$<number>`后缀可以改变定义中的顺序,即协处理器的加载顺序。虽然只有 `COPROCESSOR` 的前缀会被检查,但是还是推荐大家使用数字后缀定义顺序。例 4.19 展示了如何使用管理 API 实现这些功能。

例 4.19 检查特定 `get` 请求的 `region observer`

```
public class LoadWithTableDescriptorExample {

    public static void main(String[] args) throws IOException {
        Configuration conf = HBaseConfiguration.create();

        FileSystem fs = FileSystem.get(conf);
        Path path = new Path(fs.getUri() + Path.SEPARATOR + "test.jar");❶

        HTableDescriptor htd = new HTableDescriptor("testtable");❷
        htd.addFamily(new HColumnDescriptor("colfam1"));
        htd.setValue("COPROCESSOR$1", path.toString() +
            "|" + RegionObserverExample.class.getCanonicalName() + ❸
            "|" + Coprocessor.Priority.USER);

        HBaseAdmin admin = new HBaseAdmin(conf);❹
        admin.createTable(htd);

        System.out.println(admin.getTableDescriptor(Bytes.toBytes("testtable")));❺
    }
}
```

- ❶ 得到包含协处理器实现的 JAR 文件的地址。
- ❷ 定义表描述符。
- ❸ 将协处理器定义添加到表描述符中。
- ❹ 创建集群的管理 API 并添加这个表。
- ❺ 检查定义的协处理器是否被正确添加。

当运行一个本地单机的 HBase 集群时,最后的检查会有以下输出:

```
{NAME => 'testtable',COPROCESSOR$1 => \
'file:/test.jar|coprocessor.RegionObserverExample|USER',FAMILIES => \
[{{NAME => 'colfam1',BLOOMFILTER => 'NONE',REPLICATION_SCOPE => '0',\
COMPRESSION => 'NONE',VERSIONS => '3',TTL => '2147483647',BLOCKSIZE \
=> '65536',IN_MEMORY => 'false',BLOCKCACHE => 'true'}}]}
```

协处理器的定义被成功地添加到了表定义中。一旦表被启用,且 `region` 被打开,框架

会首先加载配置文件中定义的协处理器，然后再加载表描述符中的协处理器。

4.3.4 RegionObserver 类

在 region 级别中介绍的 Coprocessor 第一个子类是 RegionObserver 类。从名字中可以看出它属于 observer 协处理器：当一个特定的 region 级别的操作发生时，它们的钩子函数会被触发。

这些操作可以被分为两类：region 生命周期变化和客户端 API 调用。我们会在后面介绍这两类协处理器。

1. 处理 region 生命周期事件

8.6 节将介绍 region 的生命周期，图 4-4 简单展示了原理。

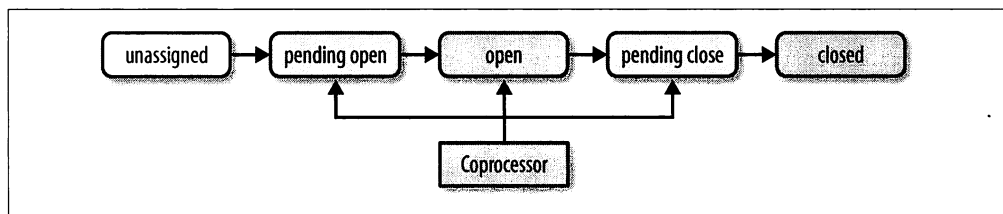


图 4-4 在 region 生命周期状态变化时起作用的协处理器

这些 observer 可以与 pending open、open 和 pending close 状态通过钩子链接。每一个钩子都被框架隐式地调用。



为了简洁，在介绍 observer 调用时，所有参数和异常都被忽略了。读者可以从线上文档中得到完整的定义^①。需要注意的是，所有的调用都有一个特定的第一参数：

```
ObserverContext<RegionCoprocessorEnvironment> c
```

特殊的 CoprocessorEnvironment 包装让用户可以控制在钩子执行之后会发生什么。参阅“RegionCoprocessorEnvironment 类”和“ObserverContext 类”两节中的详细介绍。

状态：pending open。 region 将要被打开时会处于这个状态。监听的协处理器可以搭载这个过程或阻止这个过程。以下几个调用可以完成这些功能：

```
void preOpen(...)/ void postOpen(...)
```

这些方法会在 region 被打开前或刚刚打开后被调用。用户可以在自己的协处理器实现中使用这两个方法，例如，使用 preOpen() 方法告知框架这次打开操作应当被放弃，或

^① 见链接 <http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/coprocessor/RegionObserver.html>。

勾住 `postOpen()` 方法来触发一次缓存预热或其他一些操作。

region 经过 **pending open**，且在打开状态之前，**region** 服务器可能需要从 **WAL** 中应用一些记录到 **region** 中，这时会触发以下方法：

```
void preWALRestore(...)/ void postWALRestore(...)
```

这个方法让用户可以细粒度地控制在 **WAL** 重做时哪些修改需要被实施。用户可以访问修改记录，因此用户就可以监督哪些记录被实施了。

状态：open。当一个 **region** 被部署到一个 **region** 服务器中，并可以正常工作时，这个 **region** 会被认为处于 **open** 状态。此前本书中提到的那些方法就可以被应用在这个 **region** 上了，例如，**region** 的内存存储可以被持久化到磁盘，当它变得非常大时，**region** 也可以被拆分。可用的钩子函数如下：

```
void preFlush(...) / void postFlush(...)
void preCompact(...) / void postCompact(...)
void preSplit(...) / void postSplit(...)
```

现在我们只能简单直观地介绍一下：**pre** 方法在事件执行前被调用，**post** 方法在事件执行后被调用。例如，用户使用 `preSplit()` 钩子函数可以有效地禁用 **region** 拆分，然后手动执行这些操作。

状态：pending close。最后一组 **region** 监听器的钩子函数可以监听 **pending close** 状态。这个状态在 **region** 状态从 **open** 到 **closed** 转变时发生。在 **region** 被关闭之前和之后，以下钩子函数将被执行：

```
void preClose(..., boolean abortRequested)/
void postClose(..., boolean abortRequested)
```

`abortRequested` 参数包含了 **region** 被关闭的原因。通常情况下，**region** 会在正常操作中被关闭，例如，**region** 由于负载均衡被移动到其他 **region** 服务器时被关闭。也有可能是由于 **region** 服务器被撤销，且需避免一些副作用。当这些情况发生时，所有它管理的 **region** 都会被撤销，同时用户从这个参数中可以看到是否符合这种情况。

2. 处理客户端 API 事件

与生命周期事件相比，所有的客户端 **API** 调用都显式地从客户端应用中传输到 **region** 服务器。用户可以在这些调用执行前或刚刚执行后拦截它们。以下是可用的方法。

```
void preGet(...)/ void postGet(...)
```

在客户端 `HTable.get()` 请求执行之前和之后调用。

```
void prePut(...)/ void postPut(...)
```

在客户端 HTable.put() 请求执行之前和之后调用。

```
void preDelete(...)/ void postDelete(...)
```

在客户端 HTable.delete() 请求执行之前和之后调用。

```
boolean preCheckAndPut(...)/ boolean postCheckAndPut(...)
```

在客户端调用 HTable.checkAndPut() 之前和之后调用。

```
boolean preCheckAndDelete(...)/ boolean postCheckAndDelete(...)
```

在客户端调用 HTable.checkAndDelete() 之前和之后调用。

```
void preGetClosestRowBefore(...)/ void postGetClosestRowBefore(...)
```

在客户端调用 HTable.getClosestRowBefore() 之前和之后调用。

```
boolean preExists(...)/ boolean postExists(...)
```

在客户端调用 HTable.exists() 之前和之后调用。

```
long preIncrementColumnValue(...)/ long postIncrementColumnValue(...)
```

在客户端调用 HTable.incrementColumnValue() 之前和之后调用。

```
void preIncrement(...)/ void postIncrement(...)
```

在客户端调用 HTable.increment() 之前和之后调用。

```
InternalScanner preScannerOpen(...)/ InternalScanner postScannerOpen(...)
```

在客户端调用 HTable.getScanner() 之前和之后调用。

```
boolean preScannerNext(...)/ boolean postScannerNext(...)
```

在客户端调用 ResultScanner.next() 之前和之后调用。

```
void preScannerClose(...)/ void postScannerClose(...)
```

在客户端调用 ResultScanner.close() 之前和之后调用。

3. RegionCoprocessorEnvironment 类

实现 RegionObserver 类的协处理器环境的实例是基于 RegionCoprocessor Environment 类的, RegionCoprocessorEnvironment 实现了 Coprocessor Environment 接口。后者在 4.3 节中介绍过。

除了已提供的方法, 一些更特别的、面向 region 的子类还添加了一些方法, 具体描述见表 4-12。

表 4-12 RegionCoprocessorEnvironment 类提供的方法及子类方法

方法	描述
HRegion getRegion()	返回监听器监听的 region 的引用
RegionServerServices getRegionServerServices()	返回共享的 RegionServerServices 实例

getRegion() 方法可以用于得到目前正在管理的 HRegion 实例，同时可以执行这个类提供的方法。此外用户代码可以访问共享的 RegionServerServices 实例，这会在表 4-13 中进行说明。

表 4-13 RegionServerServices 类提供的方法

方法	描述
boolean isStopping()	当 region 服务器正在停止服务时，返回 true
HLog getWAL()	提供访问 WAL 实例的功能
CompactionRequestor getCompactionRequestor()	提供访问共享的 CompactionRequestor 实例的功能，可以在协处理器内部发起合并
FlushRequester getFlushRequester()	提供访问共享的 FlushRequester 实例功能，可以用于发起 memstore 刷写
RegionServerAccounting getRegionServerAccounting()	提供访问共享 RegionServerAccounting 实例的功能。用户可以利用它得到当前服务进程资源的占用状态，例如当前 memstore 的大小
postOpenDeployTasks(Hregion r, CatalogTracker ct,final boolean daughter)	这是一个内部调用，在 region 服务器内部使用
HBaseRpcMetrics getRpcMetrics()	提供访问共享 HBaseRpcMetrics 实例的功能，包含当前服务端 RPC 统计信息

这里不详细讨论所有功能，不过用户可以参考 Java API 文档。^①

4. ObserverContext 类

RegionObserver 类提供的所有回调函数都需要一个特殊的上下文作为共同的参数：ObserverContext 类，它不仅提供了访问当前系统环境的入口，同时也添加了一些关键功能用以通知协处理器框架在回调函数完成时需要做什么。



所有的协处理器在执行时共用一个上下文实例，并会随着环境一起变化。

^① API 文档见 <http://hbase.apache.org/apidocs/>。

表 4-14 展示了上下文类提供的方法。

表 4-14 `ObserverContext` 类提供的方法

方法	描述
<code>E getEnvironment()</code>	返回当前协处理器环境的引用
<code>void bypass()</code>	当用户代码调用此方法时，框架将使用用户提供的值，而不使用框架通常使用的值
<code>void complete()</code>	通知框架后续的处理可以被跳过，剩下没有被执行的协处理器也会被跳过。这意味着当前协处理器的响应是最后的一个协处理器
<code>boolean shouldBypass()</code>	框架内部用来检查标志位
<code>boolean shouldComplete()</code>	框架内部用来检查标志位
<code>void prepare(E env)</code>	使用特定的环境准备上下文。这个方法只供内部使用。它被静态方法 <code>createAndPrepare()</code> 使用
<code>static <T extends CoprocessorEnvironment> ObserverContext<T> createAndPrepare(T env, ObserverContext<T> context)</code>	初始化上下文的静态方法。当提供的 <code>context</code> 参数是 <code>null</code> 时，它会创建一个新实例

两个重要的上下文方法是 `bypass()` 和 `complete()`。它们为用户的协处理器实现提供了选择，以控制框架后续行为。`complete()` 的调用会影响后面执行的协处理器，同时 `bypass()` 方法可以停止当前服务进程的处理过程。通过之前的例子，用户可以使用它停止 `region` 的自动拆分：

```
@Override
public void preSplit(ObserverContext<RegionCoprocessorEnvironment> e){
    e.bypass();
}
```

与基于接口实现自己的 `RegionObserver` 相比，用户可以使用基类修改自己需要的部分。

5. `BaseRegionObserver` 类

这个类可以作为所有用户实现监听类型协处理器的基类。它实现了所有 `RegionObserver` 接口的空方法，所以在默认情况下继承这个类的协处理器没有任何功能。用户需要重载他们感兴趣的方法来实现自己的功能。

例 4.20 实现了一个 `observer` 处理特殊行键的请求。

例 4.20 检查特殊 `get` 请求的 `region observer`

```
public class RegionObserverExample extends BaseRegionObserver {
    public static final byte[] FIXED_ROW = Bytes.toBytes("@@GETTIME@@");

    @Override
    public void preGet(final ObserverContext<RegionCoprocessorEnvironment> e,
```

```

        final Get get,final List<KeyValue> results)throws IOException {
    if(Bytes.equals(get.getRow(),FIXED_ROW)){ ❶
        KeyValue kv = new KeyValue(get.getRow(),FIXED_ROW,FIXED_ROW,
            Bytes.toBytes(System.currentTimeMillis()));
        results.add(kv);❷
    }
}
}
}

```

❶ 检查请求的行键是否匹配。

❷ 创建一个特殊的 KeyValue 实例，只包含服务器的当前时间。



将下面的配置项添加到 *hbase-site.xml* 中可以启动协处理器：

```

<property>
  <name>hbase.coprocessor.region.classes</name>
  <value>coprocessor.RegionObserverExample</value>
</property>

```

由于已经把编译过的包含这个类的 JAR 添加到了 *hbase-env.sh* 的 HBASE_CLASSPATH 中，region 服务器在 JRE 中可以加载这个类。请用户参考 4.1.6 节。

部署完成之后需要重启 HBase 来使配置生效。

行键@@@GETTIME@@@被 observer 的 preGet() 捕获，然后添加当前服务器端时间。部署完成之后，使用 HBase Shell 可以看到如下输出：

```

hbase(main):001:0> get 'testtable','@@@GETTIME@@@'
COLUMN                                CELL
@@@GETTIME@@@:@@@GETTIME@@@          timestamp=9223372036854775807,\
                                         value=\x00\x00\x01/s@3\xD8
1 row(s) in 0.0410 seconds

hbase(main):002:0> Time.at(Bytes.toLong(\
  "\x00\x00\x01/s@3\xD8".to_java_bytes)/ 1000)
=> Wed Apr 20 16:11:18 +0200 2011

```

这些请求都针对一个已经存在的表，因为 get 请求一张不存在的表时会产生错误。由于示例没有设置 *bypass* 标志位，所以会发生下面的情况：

```

hbase(main):003:0> create 'testtable2','colfam1'
0 row(s) in 1.3070 seconds

hbase(main):004:0> put 'testtable2','@@@GETTIME@@@',\
  'colfam1:qual1','Hello there!'
0 row(s) in 0.1360 seconds

hbase(main):005:0> get 'testtable2','@@@GETTIME@@@'

```

COLUMN	CELL
@@@GETTIME@@@:@@@GETTIME@@@	timestamp=9223372036854775807,\ value=\x00\x00\x01/sJ\xBC\xEC
colfam1:qual1	timestamp=1303309353184,value=Hello there!

2 row(s) in 0.0450 seconds

新表被创建之后，向表中添加一行数据，这一行数据随后也被检索出来了。用户可以观察到用户添加的列与表中实际的数据混在了结果中。为了避免这种情况，例 4.21 添加了必要的 `e.bypass()` 调用。

例 4.21 region observer 检查特殊的 get 请求并跳过之后的处理过程

```
if(Bytes.equals(get.getRow(),FIXED_ROW)){
    KeyValue kv = new KeyValue(get.getRow(),FIXED_ROW,FIXED_ROW,
        Bytes.toBytes(System.currentTimeMillis()));
    results.add(kv);
    e.bypass();❶
}
```

❶ 一旦特殊的 KeyValue 被添加，之后的操作都会被跳过。



用户需要把以下配置添加到 `hbase-site.xml` 中：

```
<property>
  <name>hbase.coprocessor.region.classes</name>
  <value>coprocessor.RegionObserverWithBypassExample</value>
</property>
```

与之前的示例一样，请重启 HBase 使改动生效。

与之前设想的一样，使用命令行查看结果，如下所示：

```
hbase(main):069:0> get 'testtable2','@@@GETTIME@@@'
COLUMN                                CELL
@@@GETTIME@@@:@@@GETTIME@@@          timestamp=9223372036854775807,\
                                         value=\x00\x00\x01/sj\x1D4
1 row(s) in 0.0470 seconds
```

由于默认的 `get` 操作被跳过，只有人工添加的一列被返回，并且是返回的唯一一列数据。同时注意返回列的时间戳是 9223372036854775807，这个值是 `Long.MAX_VALUE` 预计得到的值。因为示例代码创建 `KeyValue` 实例时并没有指定时间戳，所以被默认设为 `HConstants.LATEST_TIMESTAMP`，即 `Long.MAX_VALUE`。用户可以修改示例代码并使用 Shell 查看修改后的返回结果。

4.3.5 MasterObserver 类

讨论的 `Coprocessor` 的第二个子类是为了处理 `master` 服务器的所有回调函数。这些操作和 API 调用会在第 5 章介绍，与关系型数据库中 DDL 类似，它们可以被归类到数据处理操作中。基于上述原因 `MasterObserver` 类提供如下钩子函数。

`void preCreateTable(...)/ void postCreateTable(...)`

在表创建前后被调用。

`void preDeleteTable(...)/ void postDeleteTable(...)`

在表删除前后被调用。

`void preModifyTable(...)/ void postModifyTable(...)`

在表修改前后被调用。

`void preAddColumn(...)/ void postAddColumn(...)`

在表中添加列前后被调用。

`void preModifyColumn(...)/ void postModifyColumn(...)`

在表中列被修改前后被调用。

`void preDeleteColumn(...)/ void postDeleteColumn(...)`

在表中列被删除前后被调用。

`void preEnableTable(...)/ void postEnableTable(...)`

在表启用前后被调用。

`void preDisableTable(...)/ void postDisableTable(...)`

在表禁用前后被调用。

`void preMove(...)/ void postMove(...)`

在 **region** 被移动前后被调用。

`void preAssign(...)/ void postAssign(...)`

在 **region** 分配前后被调用。

`void preUnassign(...)/ void postUnassign(...)`

在 **region** 未分配前后被调用。

`void preBalance(...)/ void postBalance(...)`

在 **region** 负载均衡操作前后被调用。

`boolean preBalanceSwitch(...)/ void postBalanceSwitch(...)`

在修改自动负载均衡标志位前后被调用。

`void preShutdown(...)`

在集群关闭工作开始前被调用。没有 **post** 钩子函数，因为集群关闭之后没有进程

可以执行 post 函数。

```
void preStopMaster(...)
```

在 master 进程停止工作开始前被调用。没有 post 钩子函数，因为 master 停止工作之后没有进程可以执行 post 函数。

1. MasterCoproprocessorEnvironment 类

与 RegionCoproprocessorEnvironment 包括一个 RegionObserver 协处理器类似，MasterCoproprocessorEnvironment 封装了一个 MasterObserver 实例，它同样实现了 CoprocessorEnvironment 接口，因此它也能提供 getTable() 之类的方法帮助用户在自己的实现中访问数据。

面向 master 的子类添加了表 4-15 中描述的方法。

表 4-15 MasterCoproprocessorEnvironment 类提供的非继承方法

方法	说明
MasterServices getMasterServices()	提供可访问的共享 MasterServices 实例

用户代码可以访问共享的 MasterServices 实例，表 4-16 介绍了它的方法。

表 4-16 MasterServices 类提供的方法

方法	说明
AssignmentManager getAssignmentManager()	使用户可以访问 AssignmentManager 实例，它负责为所有的 region 分配操作，例如分配、卸载和负载均衡等
MasterFileSystem getMasterFileSystem()	提供一个与 master 操作相关的文件系统抽象层，例如，创建表或日志文件的目录
ServerManager getServerManager()	返回 ServerManager 实例。它可以访问所有的服务器进程，无论进程处于存活、死亡或其他状态
ExecutorService getExecutorService()	执行服务被 master 用来调度系统级事件
void checkTableModifiable(byte[] tableName)	检查表是否已经存在以及是否已经离线，如果是就可以修改它

在这里不会介绍所有的细节，更多信息请参考 Java API 文档^①。

2. BaseMasterObserver 类

用户可以直接实现 MasterObserver 接口，或扩展 BaseMasterObserver 类来实现自己的功能。BaseMasterObserver 为接口的每个方法完成了一个空的实现。用户不做任何改变直接使用这个类不会有任何反馈。

① API 文档见 <http://hbase.apache.org/apidocs/>。

用户可以通过重载合适的事件函数来实现自己的功能。用户可以选择对应的 `pre` 或 `post` 方法。

例 4.22 使用了 `post` 钩子函数在建表完成后添加了其他操作。

例 4.22 创建新表时创建一个单独的目录

```
public class MasterObserverExample extends BaseMasterObserver {

    @Override
    public void postCreateTable(
        ObserverContext<MasterCoproprocessorEnvironment> env,
        HRegionInfo[] regions, boolean sync)
        throws IOException {
        String tableName = regions[0].getTableDesc().getNameAsString();❶

        MasterServices services = env.getEnvironment().getMasterServices();
        MasterFileSystem masterFileSystem = services.getMasterFileSystem();❷
        FileSystem fileSystem = masterFileSystem.getFileSystem();

        Path blobPath = new Path(tableName + "-blobs");❸
        fileSystem.mkdirs(blobPath);

    }
}
```

- ❶ 从表描述符中得到表名。
- ❷ 获取可用的服务，同时取得真实文件系统的引用。
- ❸ 创建新目录用来存储客户端应用的二进制数据。



用户需要将以下配置项添加到 `hbase-site.xml` 文件中，然后协处理器将被 `master` 进程加载：

```
<property>
  <name>hbase.coprocessor.master.classes</name>
  <value>coprocessor.MasterObserverExample</value>
</property>
```

运行例子之前，请重启 HBase 使修改生效。

一旦用户成功启动了协处理器，它就开始监听事件并在事件发生时自动触发用户添加的代码。示例代码使用了已提供的服务建立目录，一个假想的应用可以使用这个目录在 HBase 外部存储大的二进制对象（称为 `blob`）。

使用 Shell 触发事件如下所示：

```
hbase(main):001:0> create 'testtable','colfam1'
0 row(s) in 0.4300 seconds
```

这个操作创建了一张表，然后调用了协处理器的 `postCreateTable()` 方法。用户可以用 Hadoop 的命令行工具来检验结果：

```
$ bin/hadoop dfs -ls
Found 1 items
drwxr-xr-x - larsgeorge supergroup 0 ... /user/larsgeorge/testtable- blobs
```

用户可以使用 `MasterObserver` 协处理器做许多事情。由于用户可以通过 `MasterServices` 实例得到许多共享的 `master` 资源，所以用户需要小心操作以避免造成严重破坏。

最后，因为 `Environment` 实例被 `ObserverContext` 封装过，用户也可以调用流程控制函数 `bypass()` 和 `complete()`。用户可以使用它们显式地禁用一些操作，或跳过后续要执行的协处理器。

4.3.6 endpoint

在之前 `RegionObserver` 的示例中，我们使用了一个已知的行键，并在 `get` 请求中添加了一个计算好的列。这看起来足以让我们实现其他一些功能了，例如，使用聚合函数来计算一个特定列所有值的和。

不幸的是，这种方式行不通，因为行键决定了哪一个 `region` 处理这个请求，所以计算请求只会送到这个 `region` 所在的服务器上。而我们需要的是向所有 `region` 发送请求，即所有 `region` 服务器，这样它们就能在本地计算这个特定列的所有值之和。一旦所有 `region` 返回了它们的计算结果，我们就可以在客户端收集这些结果并计算出最终结果。如果数据有 1000 个 `region` 和 100 万列，用户会在客户端得到 1000 个十进制的计算结果，每个结果对应一个 `region`。用户使用这样的形式计算最终结果的速度会快很多。

如果用户使用普通的客户端 API 来遍历整个表，最坏的情况下，用户可能需要将 100 万列的数据全传到客户端来计算最终结果。所以将计算转移到服务器端显然是一个更好的选择。不过 HBase 可能不知道用户具体需要做什么，为了克服这些问题，协处理器提供了以 `endpoint` 概念为代表的动态调用实现。

1. CoprocessorProtocol 接口

为了给客户端提供自定义的 RPC 协议，系统提供了一个协处理器实现来定义扩展 `CoprocessorProtocol` 协议的接口。通过这个接口可以定义协处理器希望暴露给用户的任意方法。通过以下 `HTable` 提供的调用方法，使用这个协议可以和协处理器实例之间通信。


```

<T extends CoprocessorProtocol> T coprocessorProxy(
    Class<T> protocol,byte[] row)
<T extends CoprocessorProtocol,R> Map<byte[],R> coprocessorExec(
    Class<T> protocol,byte[] startKey,byte[] endKey,
    Batch.Call<T,R> callable)
<T extends CoprocessorProtocol,R> void coprocessorExec(
    Class<T> protocol,byte[] startKey,byte[] endKey,
    Batch.Call<T,R> callable,Batch.Callback<R> callback)

```

由于 CoprocessorProtocol 实例和表中单个 **region** 联系在一起，所以客户端的 RPC 调用必须定义 **region**，这个 **region** 会在 CoprocessorProtocol 方法的调用中被使用到。虽然客户端代码很少直接对 **region** 进行操作，而且 **region** 的名字经常变化，然而协处理器 RPC 调用会通过行键来查找涉及的 **region**。客户端可以调用如下 CoprocessorProtocol 方法。

单个 region

此方法使用单个行键调用 coprocessorProxy()。返回一个 CoprocessorProtocol 接口的动态代理，它使用包含给定行键的 **region** 作为 RPC endpoint，即使给空行键对应的行不存在也不影响。

一段范围的 region

此方法通过使用起始行键和终止行键来调用 coprocessorExec()。表中包含在起始行键到终止行键（不包含终止行键）范围内的所有 **region** 都将作为 PRC endpoint。



作为参数被传入到 HTable 的方法中的行键但不会传入 CoprocessorProtocol 的实现中，而仅仅被用于确定远端调用的 endpoint 的 **region**。

Batch 类为 CoprocessorProtocol 中涉及多个 **region** 方法的调用定义了两个接口：客户端实现了 Batch.Call 方法来调用 CoprocessorProtocol 实例的方法。每个选中的 **region** 将会调用一次这个接口的 call() 方法，并将 CoprocessorProtocol 实例作为 **region** 的参数。

在调用完成时，客户端可以选择实现 Batch.Callback 来传回每次 **region** 调用的结果。

```
void update(byte[] region,byte[] row,R result)
```

以上方法在被调用时将使用以下函数 R call(T instance) 返回的值作为参数，并且每个 **region** 都会调用并返回。

2. BaseEndpointCoproprocessor 类

实现一个 endpoint 涉及以下两个步骤。

1. 扩展 CoprocessorProtocol 接口。

这一步将设定与新 endpoint 的通信细节，即定义了客户端和服务端端的 PRC 协议。

2. 扩展 BaseEndpointCoproprocessor 类。

用户必须实现 endpoint 涉及的方法，包括抽象类 BaseEndpointCoproprocessor，以及之前定义的 endpoint 协议接口。

例 4.23 实现了 CoprocessorProtocol，并为 HBase 添加了自定义的方法。客户端可以远程调用该方法来统计每个 region 中的行数目的和 KeyValue 数目。

例 4.23 endpoint 协议，添加一个行和 KeyValue 的计数方法

```
public interface RowCountProtocol extends CoprocessorProtocol {
    long getRowCount() throws IOException;

    long getRowCount(Filter filter) throws IOException;

    long getKeyValueCount() throws IOException;
}
```

第二步是将新协议的接口和继承自 BaseEndpointCoproprocessor 的类结合起来。例 4.24 使用环境提供的 InternalScanner 实例来访问数据。

例 4.24 endpoint 实现增加了行和 KeyValue 实例的统计方法

```
public class RowCountEndpoint extends BaseEndpointCoproprocessor
    implements RowCountProtocol {

    private long getCount(Filter filter, boolean countKeyValues)
        throws IOException {
        Scan scan = new Scan();
        scan.setMaxVersions(1);
        if (filter != null) {
            scan.setFilter(filter);
        }
        RegionCoproprocessorEnvironment environment =
            (RegionCoproprocessorEnvironment) getEnvironment();
        // use an internal scanner to perform scanning.
        InternalScanner scanner = environment.getRegion().getScanner(scan);
        int result = 0;
        try {
            List<KeyValue> curVals = new ArrayList<KeyValue>();
```

```

        boolean done = false;
        do {
            curVals.clear();
            done = scanner.next(curVals);
            result += countKeyValues ? curVals.size() : 1;
        } while(done);
    } finally {
        scanner.close();
    }
    return result;
}

@Override
public long getRowCount() throws IOException {
    return getRowCount(new FirstKeyOnlyFilter());
}

@Override
public long getRowCount(Filter filter) throws IOException {
    return getCount(filter, false);
}

@Override
public long getKeyValueCount() throws IOException {
    return getCount(null, true);
}
}

```

注意例子中如何使用 `FirstKeyOnlyFilter` 来减少扫描的列数。



用户需要向 `hbase-site.xml` 中添加（或者修改以前的 `hbase-site.xml`）如下配置来确保你的 `endpoint` 协处理器可以被 `region` 服务器加载：

```

<property>
  <name>hbase.coprocessor.region.classes</name>
  <value>coprocessor.RowCountEndpoint</value>
</property>

```

同以往一样，需要重启 HBase 来保证这些设置生效。

例 4.25 展示了客户端如何使用 `HTable` 的调用来执行部署好的协处理器 `endpoint` 函数。由于统计请求被单独分发到每个 `region` 上，最后需要对结果进行归并处理。

例 4.25 使用自定义行计数 `endpoint`

```

public class EndpointExample {

    public static void main(String[] args) throws IOException {
        Configuration conf = HBaseConfiguration.create();

```

```

HTable table = new HTable(conf, "testtable");
try {
    Map<byte[], Long> results = table.coprocessorExec(
        RowCountProtocol.class, ❶
        null, null, ❷
        new Batch.Call<RowCountProtocol, Long>() { ❸

            @Override
            public Long call(RowCountProtocol counter) throws IOException {
                return counter.getRowCount(); ❹
            }
        });

    long total = 0;
    for(Map.Entry<byte[], Long> entry : results.entrySet()){ ❺
        total += entry.getValue().longValue();
        System.out.println("Region: " + Bytes.toString(entry.getKey()) +
            ", Count: " + entry.getValue());
    }
    System.out.println("Total Count: " + total);
} catch(Throwable throwable){
    throwable.printStackTrace();
}
}
}

```

- ❶ 定义将要被调用的协议接口。
- ❷ 设置起始行键和终止行键为 `null`，来统计所有的行。
- ❸ 创建一个发往所有 `region` 服务器的匿名类。
- ❹ `call()` 方法将会执行 `endpoint` 功能。
- ❺ 遍历返回的键值映射结果，其中包含了每个 `region` 的结果。

这段代码返回了每个 `region` 的名字，每个 `region` 包含了多少行和最终计数：

```

Region: testtable,,1303417572005.51f9e2251c29ccb2...cbcb0c66858f., Count: 2
Region: testtable,row3,1303417572005.7f3df4dcba3f...dbc99fce5d87., Count: 3
Total Count: 5

```

`Batch` 类提供了另一种更加便捷的方法来访问远程 `endpoint`，使用 `Batch.forMethod()` 会得到一个已经配置好的 `Batch.Call` 实例，并准备好将其发送到所有 `region` 服务器。例 4.26 使用了这个方法修改上一个例子。

例 4.26 使用 `Batch.forMethod()` 来减少代码数目

```
Batch.Call call = Batch.forMethod(RowCountProtocol.class,
    "getKeyValueCount");
Map<byte[],Long> results = table.coprocessorExec(
    RowCountProtocol.class,null,null,call);
```

`forMethod()` 方法使用 Java 的反射机制来获取给定的方法, 返回的 `Batch.Call` 实例将会执行 `endpoint` 的功能, 并且返回此方法的协议定义的应返回的数据类型。

然而, 如果通过直接扩展 `Batch.Call` 实例, 可以对这些结果执行额外的处理, 这样处理会更加方便和灵活。例 4.27 展示了批量处理多个 `endpoint` 请求, 并统计每个 `region` 的行数和 `KeyValue` 数。

例 4.27 扩展批量调用来执行多个 `endpoint` 的调用

```
Map<byte[],Pair<Long,Long>> results = table.coprocessorExec(
    RowCountProtocol.class,
    null,null,
    new Batch.Call<RowCountProtocol,Pair<Long,Long>>() {
        public Pair<Long,Long> call(RowCountProtocol counter)
            throws IOException {
            return new Pair(counter.getRowCount(),
                counter.getKeyValueCount());
        }
    });

long totalRows = 0;
long totalKeyValues = 0;
for(Map.Entry<byte[],Pair<Long,Long>> entry : results.entrySet()){
    totalRows += entry.getValue().getFirst().longValue();
    totalKeyValues += entry.getValue().getSecond().longValue();
    System.out.println("Region: " + Bytes.toString(entry.getKey())+
        ",Count: " + entry.getValue());
}
System.out.println("Total Row Count: " + totalRows);
System.out.println("Total KeyValue Count: " + totalKeyValues);
```

运行这个例子, 输出如下:

```
Region: testtable,,1303420252525.9c336bd2b294a...0647alf2d13b.,Count: {2,4}
Region: testtable,row3,1303420252525.6d7c95de8a7...386cfec7f2.,Count: {3,6}
Total Row Count: 5
Total KeyValue Count: 10
```

到目前为止, 示例都使用 `coprocessorExec()` 请求来集中所有 `region` 的请求, 这些请求通过起始和终止行键确定 `region`。例 4.28 中使用 `coprocessorProxy()` 获取了一个

endpoint 的本地客户端代理。由于行键被指定了，不管这个行键在 region 中是否存在，只要这个行键在 region 的起始行键和终止行键之间，客户端 API 都会通过行键路由该代理调用到包含这个行键的 region。

例 4.28 使用 HTable 的代理调用单个 region 的 endpoint

```
RowCountProtocol protocol = table.coprocessorProxy(  
    RowCountProtocol.class, Bytes.toBytes("row4"));  
long rowsInRegion = protocol.getRowCount();  
System.out.println("Region Row Count: " + rowsInRegion);
```

通过使用代理引用，客户端代码可以调用任何 CoprocessorProtocol 中描述的服务器端函数，同时可以返回对应 region 的处理结果。图 4-5 说明了两种方法的差异。

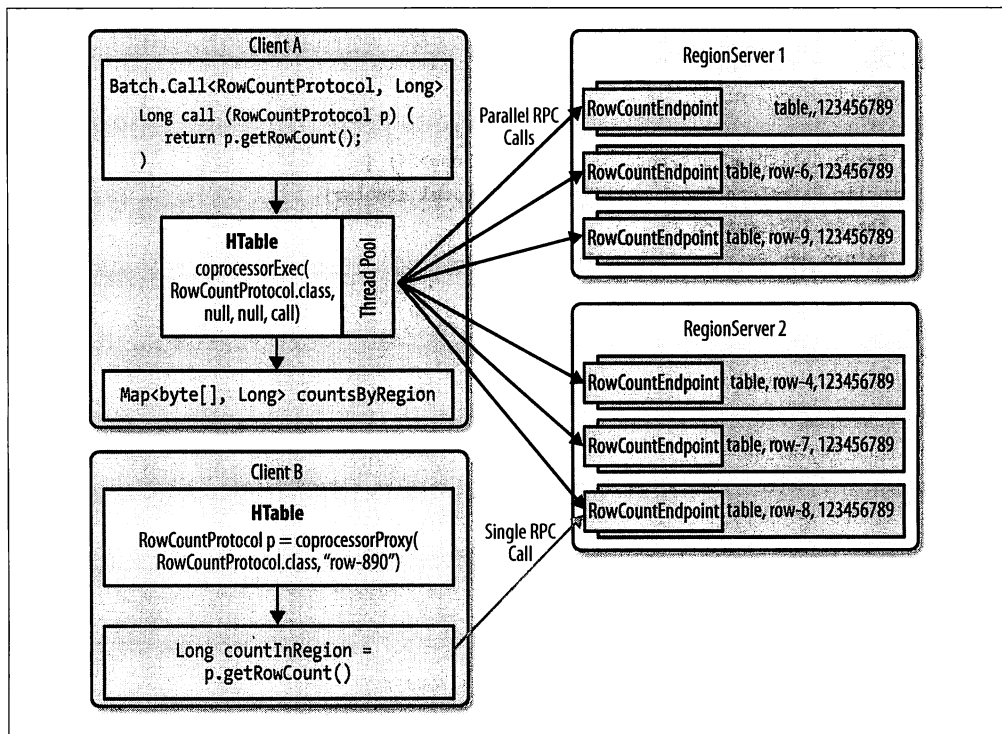


图 4-5 协处理器的请求方式分为两种:批量处理并且在多个 region 上并行执行,或者只涉及单个 region

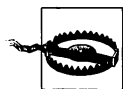
4.4 HTablePool

与其为客户端的每个请求创建一个 HTable 实例，不如创建一个实例，然后不断地复用这个实例。

按以上方式操作的主要原因，是创建 HTable 实例是一项非常耗时的操作，通常耗时数

秒才能完成。在资源高度紧张的环境中，每秒都有几千个请求，为每个请求单独创建 HTable 实例根本行不通，这种方式速度太慢了以至于无法调用方法。用户应当在一开始创建实例，然后在客户端生命周期内不断复用他们。

但是，在多线程环境中重用 HTable 实例会出现其他问题。



HTable 类不是线程安全的，本地的写缓冲区并不能保证一致性。即使使用 `setAutoFlush(true)`（默认设置，详见 3.2.1 节的“客户端的写缓冲区”）也无济于事。你必须为每个线程创建一个 HTable 实例。

客户端可以通过 HTablePool 类来解决这个问题。它只有一个目的，那就是为 HBase 集群提供客户端连接池。用户可以通过以下其中一个构造器来创建池：

```
HTablePool()
HTablePool(Configuration config,int maxSize)
HTablePool(Configuration config,int maxSize,
    HTableInterfaceFactory tableFactory)
```

默认构造器，即没有参数的那个，使用 classpath 中的配置创建一个表实例池，设定大小为无限。这等价于使用第二个构造函数编写以下例子：

```
Configuration conf = HBaseConfiguration.create()
HTablePool pool = new HTablePool(conf,Integer.MAX_VALUE)
```

maxSize 参数是连接池中允许的最大 HTable 实例数目。可选参数 tableFactory 将会被用来操作自定义工厂类，以创建实际的 HTable 实例。

HTableInterfaceFactory 接口

用户可以创建自定义的工厂类，例如，为 HTable 实例使用特定的配置。或者可以让实例完成一些初始化操作，比如添加行，或者更新计数器。如果用户想自己扩展 HTableInterfaceFactory，则必须实现下面两个方法。

```
HTableInterface createHTableInterface(Configuration config,
    byte[] tableName)
void releaseHTableInterface(HTableInterface table)
```

第一个方法用于创建 HTable 实例，第二个方法用于释放实例。用户可以在调用前准备好实例，并在使用完成之后进行一些相应的清除操作。用户要特别注意在共享表引用时对客户端写缓冲区的处理。releaseHTableInterface() 是完成一些隐式的操作的理想方法，比如写缓冲区刷写、调用 flushCommits() 请求。

工厂类有个默认的实现，叫 HTableFactory，当工厂调用 create 方法时创建 HTable 实例，当调用 release 方法时调用 HTable.close()。

如果用户不指定自己的工厂类，系统会默认使用 HTableFactory。

可以用如下调用方式来使用表实例池：

```
HTableInterface getTable(String tableName)
HTableInterface getTable(byte[] tableName)
void putTable(HTableInterface table)
```

getTable() 方法从表实例池中获取 HTable 实例，使用之后通过 putTable() 方法放回。以上两种方法把一些工作迁移到了表实例池配置的 HTableInterfaceFactory 接口。



maxSize 参数并不强行限制用户所能得到的 HTableInterface 实例的上界，用户可以使用 getTable() 方法访问尽可能多得 Table 实例。

这个参数仅仅设置表实例池中能够存放的 HTableInterface 实例的数目。例如，当用户将这个值设置为 5 时，调用 10 次 getTable() 会创建 10 个 HTable 实例。不过之后只有 5 次 putTable() 方法发挥作用，后面 5 次会被直接忽略。更重要的是，工厂的释放（release）机制也不会被调用。

这些是关闭表实例池中特定表实例的方法：

```
void closeTablePool(String tableName)
void closeTablePool(byte[] tableName)
```

很明显，这两个方法的功能是相同的，一个方法的参数是 String，另一个方法的参数是 byte[]，用户可以按自己的需要使用其中任意一个。

close 方法会遍历所有保存在列表中与参数对应的表引用，然后使用工厂的释放机制。这对于释放一张表的所有资源，并重头再来非常有用。请记住，所有的资源都需要释放，用户需要对自己使用过的所有表都调用这个方法。

例 4.29 展示了如何创建和使用 HTablePool。

例 4.29 使用 HTablePool 来共享 HTable 实例

```
Configuration conf = HBaseConfiguration.create();
HTablePool pool = new HTablePool(conf, 5); ❶

HTableInterface[] tables = new HTableInterface[10];
for(int n = 0; n < 10; n++){
    tables[n] = pool.getTable("testtable"); ❷
    System.out.println(Bytes.toString(tables[n].getTableName()));
}

for(int n = 0; n < 5; n++){
    pool.putTable(tables[n]); ❸
}

pool.closeTablePool("testtable"); ❹
```


- ❶ 创建表实例池，并允许保留 5 个实例。
- ❷ 获取 10 个实例，超出容量 5 个。
- ❸ 向表实例池返还 HTable 实例，其中的 5 个会被保留，多余的会被丢弃。
- ❹ 关闭整个表实例池，释放其中保留的表实例引用。

控制台展示结果如下：

```
Acquiring tables...
testtable
testtable
testtable
testtable
testtable
testtable
testtable
testtable
testtable
testtable
testtable
Releasing tables...
Closing pool...
```

之前已经讨论过了使用多于配置数目的 HTable 实例的问题。虽然将实例返回到 TablePool 时没有相应日志和打印输出，但释放资源的相关操作还是会在后台完成。

使用场景：Hush

Hush 中所有的表都是使用表实例池来获取的。下面是使用表实例池共享表实例的代码。

```
private ResourceManager(Configuration conf)throws IOException {
    this.conf = conf;
    this.pool = new HTablePool(conf,10);
    /* ... */
}

public HTable getTable(byte[] tableName)throws IOException {
    return(HTable)pool.getTable(tableName);
}

public void putTable(HTable table)throws IOException {
    if(table != null){
        pool.putTable(table);
    }
}
```

下面的代码演示了如何在上下文中调用这些代码，如何从表实例池中获取表引用，以及如何使用完成之后再交还表实例池中。

```

public void createUser(String username,String firstName,String lastName,
    String email,String password,String roles)throws IOException {
    HTable table = rm.getTable(UserTable.NAME);
    Put put = new Put(Bytes.toBytes(username));
    put.add(UserTable.DATA_FAMILY,UserTable.FIRSTNAME,
        Bytes.toBytes(firstName));
    put.add(UserTable.DATA_FAMILY,UserTable.LASTNAME,Bytes.toBytes (lastName));
    put.add(UserTable.DATA_FAMILY,UserTable.EMAIL,Bytes.toBytes (email));
    put.add(UserTable.DATA_FAMILY,UserTable.CREDENTIALS,
        Bytes.toBytes(password));
    put.add(UserTable.DATA_FAMILY,UserTable.ROLES,Bytes.toBytes (roles));
    table.put(put);
    table.flushCommits();
    rm.putTable(table);
}

```

4.5 连接管理

每个 HTable 实例都需要建立和远程主机的连接。这些连接在内部使用 HConnection 类表示，更重要的是，其被 HConnectionManager 类管理并共享。用户没有必要同时和这两个类打交道，只需要创建一个 Configuration 实例，然后利用客户端 API 使用这些类。

HBase 内部使用键值映射来存储连接，使用 Configuration 实例作为键值映射的键。换句话说，当你创建很多 HTable 实例时，如果你提供了相同的 Configuration 引用，那么它们都共享同一个底层的 HConnection 实例。有关细节如下所示。

共享 ZooKeeper 连接

因为每个客户端最终都需要 ZooKeeper 连接来完成表的 region 地址初始寻址。连接一旦建立后，共享就变得很有意义，这使得之后的客户端实例可以共用。

缓存通用资源

通过 ZooKeeper 查询到的 -ROOT- 和 .META. 的地址，以及 region 的地址定位都需要网络传输开销。这些地址将被缓存在客户端来减少网络的调用次数，因此达到加速寻址的目的。

对于每个连接到远程集群的本地客户端来说，它们的地址表都是相同的，因此运行相同进程的客户端共享连接非常有用，这是通过共享 HConnection 实例来实现的。另外，当寻址失败时（如 region 拆分时），连接有内置的重试机制来刷新缓存，对于其他所有共享相同连接引用的客户端来说，这项更改立即生效，因此这更加减少了

客户端初始化连接的开销。

另一个受益的类是 `HTablePool`，所有连接池中的 `HTable` 实例都自动共用一个提供的 `Configuration` 实例，因此它们也共享连接。因为当用户想创建多个 `HTable` 实例时，最好先创建一个共用的 `Configuration` 实例。

```
HTable table1 = new HTable("table1");
//...
HTable table2 = new HTable("table2");
```

上述代码不如以下代码有效：

```
Configuration conf = HBaseConfiguration.create();
HTable table1 = new HTable(conf,"table1");
//...
HTable table2 = new HTable(conf,"table2");
```

后者隐式共享 `HBase` 客户端 `API` 类提供的连接。



目前没有明显的证据表明共享连接会带来性能问题，即使在繁忙的多线程环境下也是如此。

共享连接的缺点在于释放，如果用户不显式关闭连接，它将一直存在，直到客户端退出。这样可能导致很多 `ZooKeeper` 连接都保持打开状态，尤其是在大型分布式环境下，比如执行 `MapReduce` 作业的 `HBase` 程序，这样可能会产生一些问题。最坏的情况是耗尽所有的连接句柄或内存，并导致 `I/O` 异常。

用户可通过显式关闭连接来避免这种情况。建议用户不再需要 `HTable` 时主动调用 `HTable` 的 `close()` 方法，调用这个方法将释放所有共享资源，其中包括 `ZooKeeper` 连接，同时移除内部列表中的连接引用。

每次用户重用 `Configuration` 实例时，连接管理器都会增加引用计数。因此用户必须调用 `close()` 来触发清除工作。以下是用显式的方法来清理一个连接或所有连接。

```
static void deleteConnection(Configuration conf,boolean stopProxy)
static void deleteAllConnections(boolean stopProxy)
```

所有的共享连接都是按照 `Configuration` 实例作为键，因此用户需要提供这个实例来关闭相应的连接。布尔类型参数 `stopProxy` 保证强制清除整个客户端的 `RPC` 栈，因此不再需要远程连接服务器时，应该将这个参数设置为 `true`。

`deleteAllConnections()` 函数只需要 `stopProxy` 参数，它将遍历整个连接管理器注册过的共享连接列表，然后逐一释放连接。

如果用户需要显式地使用某个连接，可以通过如下方式使用 `getConnection()` 方法。

```
Configuration newConfig = new Configuration(originalConf);
HConnection connection = HConnectionManager.getConnection(newConfig);
// Use the connection to your hearts' delight and then when done...
HConnectionManager.deleteConnection(newConfig,true);
```

这样操作的好处是可以保证这个连接的用户唯一，但是，切记必须要在调用结束后关闭它。

客户端 API：管理功能

除了进行数据处理的客户端 API，HBase 还提供了数据描述的 API，类似于传统 RDBMS 中的 DDL 与 DML。首先我们来看看数据模式定义的类，然后探讨如何使用这些 API，例如，建表。

5.1 模式定义

在 HBase 中建表涉及到表结构以及所有涉及的列族列族结构的定义，这些定义关系到表和列族内的数据如何存储以及何时存储。

5.1.1 表

在 HBase 中数据最终会存储在一张表或多张表中，使用表的主要原因是控制表中的所有列以达到共享表内的某些特性的目的，一个典型的例子是定义表的列族。下面是表描述符的构造函数。

```
HTableDescriptor();  
HTableDescriptor(String name);  
HTableDescriptor(byte[] name);  
HTableDescriptor(HTableDescriptor desc);
```

Writable 和无参数的构造函数

API 提供的大多数类和本章中讨论到的类都拥有一个特殊的构造函数，即一个不带任何参数的构造函数。因为这些类都实现了 Hadoop Writable 接口。

任意不相交系统间的远程通信：例如，客户端与服务器端进行交互，或者服务器端彼此进行内部通信，都使用到了 Hadoop RPC 框架。这个框架中需要远程方法中的参数都实现 Writable 接口，进而能够序列化对象并进行远程传输。Writable 接口有

两个必须实现的方法:

```
void write(DataOutput out)throws IOException;  
void readFields(DataInput in)throws IOException;
```

框架通过调用这两个方法把对象序列化输出流, 通信接收端读取字节流, 并将其反序列化成对象。框架在数据发送端调用 `write()` 方法, 并序列化对象的字段, 同时会在字节流中添加类名以及其他一些信息。

数据接收服务器先读取元数据信息, 并创建类的无参数实例, 然后调用新创建对象的 `readFields()` 方法, 将字节流中的信息读取到对应对象的字段中, 用户可以理解为这是数据发送端对象的一个可运行的, 已经初始化的完整副本。

因为数据接收端首先读取元数据并创建该类的空实例, 并通过 `readFields()` 方法及序列化字段信息到新创建的实例中。通常客户端与服务端需要使用相同的 HBase JAR。

如果用户开发并扩展了 HBase 的基础实现, 例如, 过滤器和协处理器 (见第 4 章), 用户必须确保满足以下条件。

1. 在 RPC 通信两端都必须可用, 即在数据发送进程与数据接收进程均可用。
2. 实现 `Writable` 接口, 并实现了 `write()` 与 `readFields()` 方法。
3. 拥有无参数的构造函数, 即一个没有任何输入参数的构造函数。

如果运行时无法使用这个特殊的构造函数生成实例, 则会报运行时错误, 并且从代码中显式调用该构造函数也是徒劳的, 因为留下了不同于预期定义且未初始化的变量。

客户端 API 开发人员必须了解 RPC 的潜在依赖, 以及怎样部署 API。在扩展 HBase 代码时, 高级开发人员需要适当地实现和部署自定义代码。详情见 4.1.6 节的相关例子。

用户可以通过表名或已有的表描述符来创建表。没有任何参数的构造函数仅仅是为了反序列化, 并且不应被直接使用。表名通常使用 Java String 类型或 `byte[]` (二进制数组) 表示, HBase 中的很多功能都提供以上两种参数类型的选择。使用字符串明显是为了方便使用, 在 HBase 内部通常会将字符串转化为字节数组处理, HBase 也是这样的处理模式。HBase 中提供的 `Bytes` 类有转化功能, 示例如下:

```
byte[] name = Bytes.toBytes("test");  
HTableDescriptor desc = new HTableDescriptor(name);
```

表名会作为存储系统中存储路径的一部分来使用, 因此必须要符合文件名规范, 因此构成表名的字符是有限制的。用户可以直接查看低级别存储系统, 例如, 用户在 HDFS 中可以看到每张表的表名都作为独立的目录结构——在某些情况下, 用户可能会需要查看这部分信息。

HBase 列式存储格式允许用户存储大量的信息到相同的表中，而在 RDBMS 模型中，大量信息则需要切分成多张表存储。通常的数据库范式化^①规则不适合 HBase，因此 HBase 中表的数量相对较少。更多详情在 1.3.3 节。

虽然理论上 HBase 的表是由行和列组成的，但是从物理结构上看，表存储在不同的分区，即不同的 region。图 5-1 展示了数据存储逻辑与物理上的不同。每个 region 只在一个 region 服务器中提供服务，而 region 直接向客户端提供存储服务。

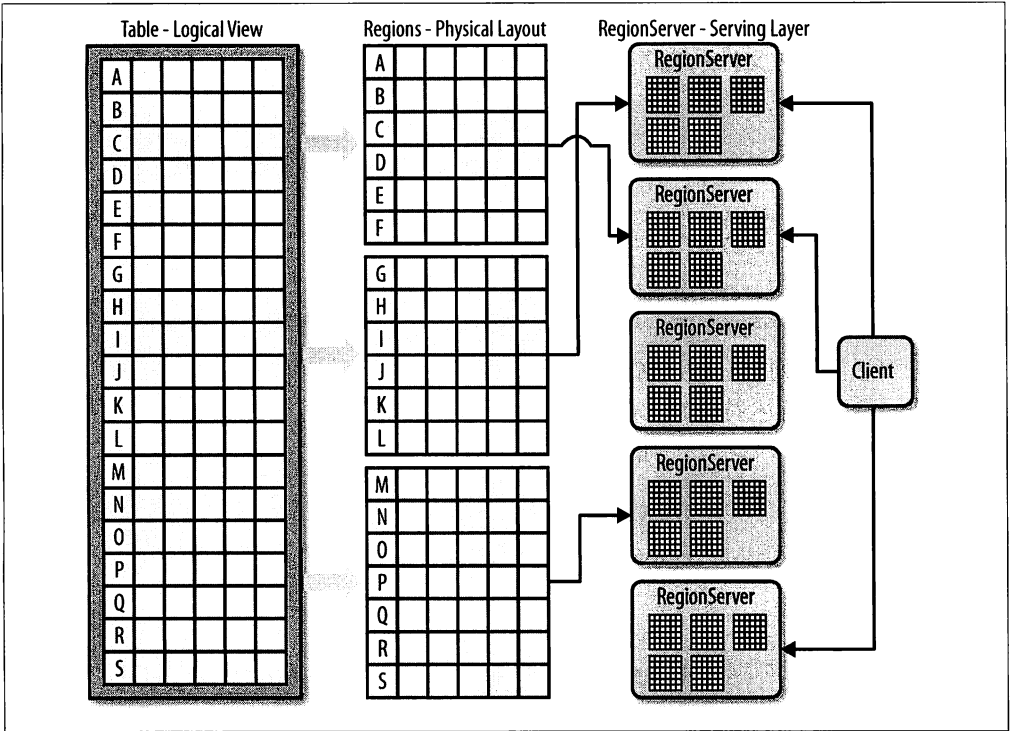


图 5-1 region 中行的逻辑与物理视图

5.1.2 表属性

表提供了 `getter` 与 `setter`^② 方法来设置表的其他属性。实际上，这些属性中的大部分都很少用到，但是这些属性非常重要，用户可以使用这些方法来微调表的性能，因此用户需要了解它们。

名

构造函数中已经有设置表名的参数，Java API 也提供了显式设置表名的方法。

① 见维基百科中的“Database Normalization” (http://en.wikipedia.org/wiki/Database_normalization)。
② 在 Java 中 `getter` 和 `setter`，可以使一个类的内部变量变得可控，方法通常以这个变量的名字加上 `get` 和 `set` 为前缀来命名，如 `getName()` 和 `setName()` 方法。

```
byte[] getName();
String getNameAsString();
void setName(byte[] name);
```



表名一定不能以“.”（结束符）或“-”（连字符）开头。表名只能包含拉丁字母或数字，以及“_”（下划线）、“-”（连字符）或“.”（结束符）。按照正则表达式语法，其规则可表示为[\[a-zA-Z_0-9-.\]](#)。

例如，`.testtable` 是错误的表名，`test.table` 是正确的表名。

有关表名如何在文件系统路径中使用的内容在 5.1.3 节和图 5-2 中。

列族

列族是表中非常重要的一部分，用户需要在表中指定将要使用的列族。

```
void addFamily(HColumnDescriptor family);
boolean hasFamily(byte[] c);
HColumnDescriptor[] getColumnFamilies();
HColumnDescriptor getFamily(byte[] column);
HColumnDescriptor removeFamily(byte[] column);
```

用户可以添加列族、通过列族名来检查列族是否存在、获取存在的列族的列表、获取某个列族描述符（`HColumnDescriptor`）或删除某个列族等操作。更多有关 `HColumnDescriptor` 的信息在 5.1.3 节中。

文件大小限制

这个参数限制了表中 **region** 的大小。通过以下方法可以显式地获取和设置该参数的值：

```
long getMaxFileSize();
void setMaxFileSize(long maxFileSize);
```



文件大小限制并不能表达其真正的含义，其真正含义应该是每个存储单元的大小限制，即一个列族有若干个存储单元，而其中每个存储单元会包含若干个文件。如果一个列族的存储单元已使用的存储空间超过了大小限制，**region** 将发生拆分操作。所以这个参数被称为 `maxStoreSize` 更合适。

当 **region** 的大小达到配置的大小时，文件大小限制会帮助 HBase 拆分 **region**，1.4 节详细讨论了如何通过 **region** 进行线性拓展和负载均衡。因此用户需要了解这个参数应该设置为多少合适，这个参数的默认值是 256 MB，设置为多少关键要看使用场景，例如，表数据量非常的大情况下，用户可以酌情考虑将这个值调高。

请注意，这个参数是个大致的预期值，而在某种特定条件下，文件大小可能会超过这个参数的大小，并且不会带来影响。例如，用户将这个参数设置为 10 MB，然后插入了一行大小为 20 MB 的值，由于一行数据不能跨 **region** 存储，因此系统

也就不会拆分该行数据。

只读

默认所有的表都可写，但是，对一些特殊的表来说，只读参数有特殊的用途。如果该参数设置为 `true`，这个表只能读而不能修改数据。通过以下方法可获取和设置该参数：

```
boolean isReadOnly();  
void setReadOnly(boolean readOnly);
```

memstore 刷写大小

早期我们讨论到 HBase 内存中预留了写缓冲区，写操作会写入到写缓冲区中，然后按照合适的条件顺序写入到磁盘的一个新存储文件中，这个过程称为刷写 (flush)。这个参数能够控制何时触发将内存中的数据写入到磁盘的事件。示例方法如下：

```
long getMemStoreFlushSize();  
void setMemStoreFlushSize(long memstoreFlushSize);
```

上面提到的文件大小限制跟需求相关，该参数的默认值是 64 MB。该参数越大，可以生成的存储文件越大，文件数量会越少，同时也会导致更长的阻塞时间问题。这种情况下，**region** 服务器不能持续接收新增加的数据，请求被阻塞的时间也就随之增加了，此外，一旦服务器崩溃，系统通过 WAL 恢复数据的时间也相应增加了，且更新的内存丢失。

延时日志刷写

日志刷写的细节可查阅 8.3 节，那里有这个选项的详细解释。现在我们需要注意的是，HBase 有两种将 WAL 保存到磁盘的方式，一种是延时日志刷写 (deferred log flushing)，另一种则不是。这个参数是一个布尔类型变量，默认设置为 `false`。下面是通过 Java API 模式设置参数的方法：

```
synchronized boolean isDeferredLogFlush();  
void setDeferredLogFlush(boolean isDeferredLogFlush);
```

其他选项

除了已经提到的属性，还有一些提供给用户设置任意键值对的方法：

```
byte[] getValue(byte[] key){  
String getValue(String key)  
Map<ImmutableBytesWritable,ImmutableBytesWritable> getValues()  
void setValue(byte[] key,byte[] value)  
void setValue(String key,String value)  
void remove(byte[] key)
```

以上方法中的数据均存储在预定义的表中，并在必要的时候能被检索。例如，在

HBase 中，用户可以通过加载协处理器来进行检索，详情见 4.4.2 节。用户设置键与值时有多种选择，如 String 或 byte 数组，内部数据使用 ImmutableBytesWritable 类型存储，目的是为了能够序列化（见 5.1.1 节中“Writable 和无参数的构造函数”部分）。

5.1.3 列族

在前面的描述中，我们看到了如何通过 HTableDescriptor 类为一张表增加列族，与此类似，这里提供了一个专门的 Java 类来实现对每个列族的设置。在其他编程语言中，用户也能够发现同样的设置列族属性的方法。



在 Java 中，这个类的命名并没有完全体现它的真正含义，更适合的命名或许应该是 HColumnFamilyDescriptor，这个类名能够表达定义列族参数的含义。

列族定义了所有列的共享信息，并且可以通过客户端创建任意数量的列，列常用的变量名为 *column qualifiers*。定位某一具体列需要列族名与列名合并在一起，以“:”分割：

```
family:qualifier
```

列族名字必须是可见字符，列名可以由任意二进制字符组成。前面提到的工具类 Bytes 提供了将 Java 语言中基本类型转化为字节数组的方法。列族名必须为可见字符是因为这个名字将会在底层存储系统中使用，图 5-2 概述了列族映射到存储系统的过程。列族名需要添加到文件系统路径名中使用，优点是用户可以通过格式可读的名称轻松访问文件系统层面的列族。



用户需要注意空列名的使用，也可以只使用列族名而忽略列名，用户的体验是一样的。读写操作与其他列没有区别，但是最好增加列名来加以区分。

用户可以不指定简单应用的列名，只是使用 HBase Shell 查看数据时看不到有价值的结果。从规范的角度来讲，用户也应该指定列名，或者说从一开始就指定列名，因为后期无法简单地重命名空列。

用户创建列族时，可以通过不同的构造函数实现所有属性的设置。Java 类提供了许多不同参数的构造函数，这些函数便于用户有针对性地选择使用，并定制需要的参数。下面是含不同参数的构造函数：

```
HColumnDescriptor();  
HColumnDescriptor(String familyName),  
HColumnDescriptor(byte[] familyName);  
HColumnDescriptor(HColumnDescriptor desc);  
HColumnDescriptor(byte[] familyName,int maxVersions,String compression,  
    boolean inMemory,boolean blockCacheEnabled,int timeToLive,
```

```
String bloomFilter);
HColumnDescriptor(byte [] familyName,int maxVersions,String compression,
    boolean inMemory,boolean blockCacheEnabled,int blocksize,
    int timeToLive,String bloomFilter,int scope);
```

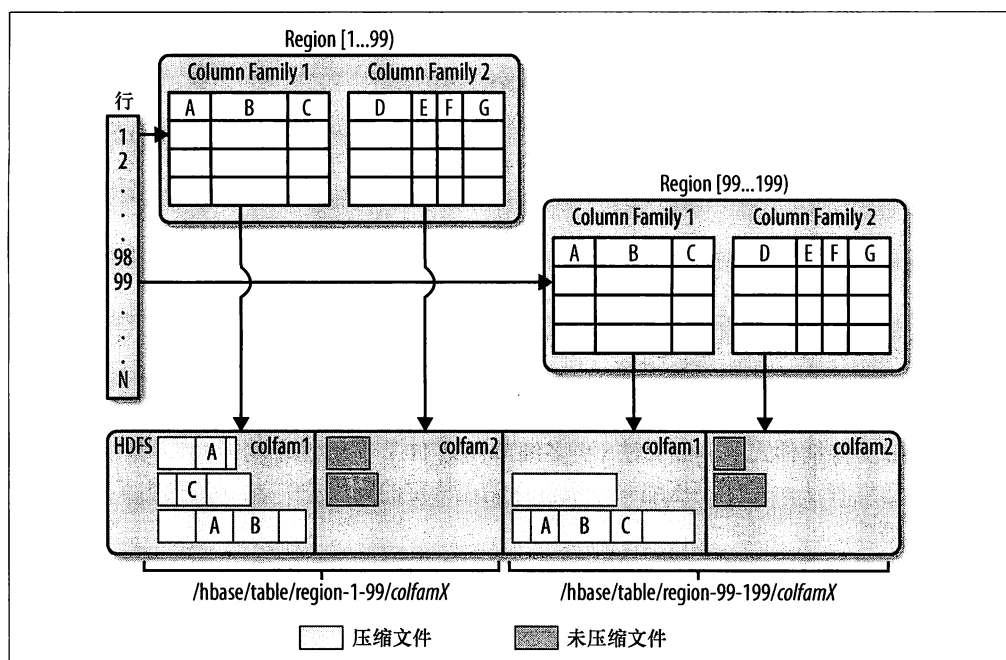


图 5-2 列族映射到独立的存储文件

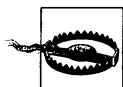
第一个构造函数用于内部反序列化。接下来的两个构造函数使用了 `String` 或 `byte[]` (字节数组已经在文中出现多次了)。另外一个构造函数复制了已存在的 `HColumnDescriptor` 对象属性，最后两个则给出了所有可用参数。

除了构造函数，用户还可以通过 `getter` 与 `setter` 方法设置参数。接下来将讨论相关细节。

名字

每个列族都有名字，用户可以通过 `HColumnDescriptor` 实例的以下方法获取列族名。

```
byte[] getName();
String getNameAsString();
```



列族不能被重命名。通常的做法是新建一个列族，然后使用 API 从旧列族中复制数据到新列族。

用户除了通过构造函数，没有其他重命名列族的途径。此外一定要谨记，列族名必

须是可见字符。



列族名不能以 “.” 开头，也不能包含 “:”、“/” 或 ISO 控制符，即字符不能在 \u0000 到 \u001F 的范围内或 \u007F 到 \u009F 的范围内。

最大版本数

所有列族都限定了每个值能够保留的最大版本数量。在前面提到的断言删除过程中，HBase 会移除超过最大版本数的数据。下面是该参数的 getter 与 setter 方法：

```
int getMaxVersions();
void setMaxVersions(int maxVersions);
```

该参数默认值是 3，用户也可以设置为 1，例如，在不访问旧数据的情况下。

压缩

HBase 支持插件式的压缩算法（更多细节参见 11.3 节），这个功能允许用户选择最适合的压缩算法——或不压缩——数据存储在特定的列族中。表 5-1 列出了目前支持的压缩算法。

表 5-1 支持的压缩算法

算法	描述
NONE	不压缩（默认）
GZ	使用 Java 提供的或本地库提供的 GZIP
LZO	启用 LZO 压缩，需要安装 LZO 的类库
SNAPPY	启用 Snappy 压缩，需要独立安装

默认值是 NONE，简言之，就是创建不压缩直接存储的列族。用户如果需要 Java API 和列描述符，可能会用以下方法来修改相应的值：

```
Compression.Algorithm getCompression();
Compression.Algorithm getCompressionType();
void setCompressionType(Compression.Algorithm type);
Compression.Algorithm getCompactionCompression();
Compression.Algorithm getCompactionCompressionType();
void setCompactionCompressionType(Compression.Algorithm type);
```

注意，压缩类型不是 String 而是 Compression.Algorithm 枚举，枚举中的类型与上述表格列出的类型列表一致。HColumnDescriptor 的构造函数也同样需要字符串的参数形式。

获取压缩信息有两种方法，一种是一般压缩设置，另一种是合并压缩设置。此外，每种方法分别有相应的 `getCompression()` 与 `getCompressionType()` 或 `getCompactionCompression()` 与 `getCompactionCompressionType()`，这两种方法返回同一类型的值，都可以用于检查当前压缩算法的类型。^③

更多相关信息在 11.3 节中有详细介绍。

块大小

在 HBase 中，所有的存储文件都被划分成了若干个小存储块，这些小存储块在 `get` 或 `scan` 操作时会加载到内存中，它们类似于 RDBMS 中的存储单元页。这个参数的默认大小是 64 KB，并通过以下方法设置和获取：

```
synchronized int getBlocksize();  
void setBlocksize(int s);
```

这个参数用于指定 HBase 在一次读取过程中顺序读取多少数据到内存缓冲区，更多细节见 11.8 节。



注意，这里有一个重要的不同点：列族的块，或者说 HFile 的块不同于 HDFS 层面的块。HDFS 层面提到的块是用于拆分大文件以提供分布式存储，且便于 MapReduce 框架进行并行计算的存储单元——默认是 64 MB。HBase 中的 HFile 块大小默认是 64 KB，是 HDFS 中块大小的 1024 分之一，主要用于高效加载和缓存数据，并不依赖于 HDFS 的块大小，并且只用于 HBase 内部。更多内容见 8.2 节，其中图 8-3 展示了两者的不同。

缓存块

HBase 顺序地读取一个数据块到内存缓存中，其读取相邻的数据时就可以在内存中读取而不需要从磁盘中再次读取，有效地减少了磁盘 I/O 的次数，提高了 I/O 效率。这个参数默认为 `true`，这意味着每次读取的块都会缓存到内存中。但是，如果用户要顺序读取某个特定列族，最好将这个属性设置为 `false`，从而禁止其使用块缓存。以下是可以改变该标识的 API：

```
boolean isBlockCacheEnabled();  
void setBlockCacheEnabled(boolean blockCacheEnabled);
```

还有其他参数可以影响块缓存的使用，例如，`scan` 的过程涉及众多参数，这些参数最终都可能影响到缓存。这些参数在全表扫描时非常有用，它们能够减少在全表扫描过程中缓存的大量流失。更多有关信息见 2.6 节。

③ 这是因为开源和代码遗留问题造成的冗余，请帮忙清理并贡献到 HBase 中。

生存期 TTL

HBase 不仅支持断言删除每个值能保存的版本数，也支持处理版本数据保存时间。生存期（TTL）设置了一个基于时间戳的临界值，内部的管理会自动检查 TTL 值是否达到上限，在 major 合并过程中时间戳被判定为超过 TTL 的数据会被删除。以下是读写 TTL 的 getter 与 setter 的 API。

```
int getTimeToLive();
void setTimeToLive(int timeToLive);
```

TTL 参数的单位是秒，默认值是 Integer.MAX_VALUE，即 2 147 483 647 秒。使用 TTL 默认值的数据可以理解为永久保留，即生产过程中产生的任意正数都永远小于当前默认值。

在内存中

我们提到了缓存块和怎样使用缓存块来提高连续访问的效率。这里还有一个在内存中（in-memory）标志，默认值为 false，以下方法可以修改此属性。

```
boolean isInMemory();
void setInMemory(boolean inMemory);
```

将这个参数设置为 true 并不意味着将整个列族的所有存储块都加载到内存中，也不意味着它们会被长期保存在那儿，而是一种承诺，或者说是高优先级。在正常的数据读取过程中，块数据被加载到缓存区中并长期驻留在内存中，除非堆压力过大，这个时候才会强制从内存卸载这部分数据。

这个参数通常适合数据量较小的列族，例如，保存登录账号和密码的用户表，将这个参数设置为 true 有利于提升这个环节的处理速度。

布隆过滤器

布隆过滤器^④是 HBase 系统中的高级功能，它能够减少特定访问模式下的查询时间（细节见 9.5 节）。但是，由于这种模式增加了内存和存储的负担，这个模式被默认为关闭状态。表 5-2 展示了布隆过滤器的类型。

表 5-2 支持的布隆过滤器类型

类型	描述
NONE	不使用布隆过滤器
ROW	行键使用布隆过滤器过滤
ROWCOL	列键使用布隆过滤器过滤

由于列的数量远多于行的数量（除非每行只有一列），使用最后一个选项 ROWCOL

④ 见维基百科中的“Bloom filter”（http://en.wikipedia.org/wiki/Bloom_filter）。

会占用大量的空间。与仅仅只有行的模式相比，行/列组合形式的粒度无疑更细。以下 API 可以设置布隆过滤器。

```
StoreFile.BloomType getBloomFilterType();
void setBloomFilterType(StoreFile.BloomType bt);
```

虽然这两个方法提供了 StoreFile.BloomType 类型，但列族描述符也可以直接使用字符串进行描述，所以参数类型不是关键因素，例如，用户可以使用 “row”。9.5 节有详细的相关信息，从中可以了解怎样使用才能有最佳效果。

复制范围

HBase 的另一个高级功能是复制（replication）。它提供了跨集群同步的功能，本地集群的数据更新可以及时同步到其他集群。复制范围（replication scope）的参数默认为 0，意味着这个功能默认处于关闭状态。以下方法可以改变其参数。

```
int getScope();
void setScope(int scope);
```

另一个可设置的值是 1，这个参数可以开启本地集群向远程集群实时同步的功能。将来这个参数有可能支持其他可用值。表 5-3 列出了目前已支持的值。

表 5-3 支持的复制范围

范围	描述
0	本地范围，即关闭实时同步（默认）
1	全局范围，即开启实时同步

更多相关信息可在 8.8 节中进行查阅。

最后，Java 类还提供了检查列族是否存在的帮助方法。

```
static byte[] isLegalFamilyName(byte[] b);
```

用户在程序中可以通过以上方法验证列族名是否合法。这个方法并不返回布尔类型标志位，而是当检查到不合法的输入时会抛出 IllegalArgumentException 异常，否则会将输入直接返回。这个方法会在构造函数中调用，不需要用户提前特殊调用。

5.2 HBaseAdmin

客户端提供了 API 的模式来管理集群，与 RDBMS 中的 DDL 相比——客户端提供的具有管理功能的 API 更像是 DML。

HBaseAdmin 提供了建表、创建列族、检查表是否存在、修改表结构和列族结构和删除表等功能。下面我们对这些功能按操作关联性分组进行介绍。

5.2.1 基本操作

使用管理的 API 需要首先实例化 HBaseAdmin 类，构造函数如下：

```
HBaseAdmin(Configuration conf)
    throws MasterNotRunningException, ZooKeeperConnectionException
```



本节论述中忽略了一点，管理功能 API 中大多数方法都抛出 IOException (或继承自它的异常)，或者是 InterruptedException。前者是客户端与服务器端的通信异常，后者是执行过程中的干扰异常，例如，region 服务器的执行命令在完成前被停止所引起的问题。

已有的配置实例提供了足够的配置信息，所以当前的 API 可以通过使用 ZooKeeper 的相关配置信息查找集群，类似于普通客户端 API 的使用方法。具有管理功能的 API 实例应该在使用后进行销毁，换言之，这个实例不应该长期保留。



HBaseAdmin 实例的生命周期不宜太长，例如，在 master 故障恢复的过程中，它是短暂有效的。

这个类实现了 Abortable 接口，并实现了以下方法：

```
void abort(String why, Throwable e)
```

以上方法被框架隐式调用，例如，当发生致命连接错误或关闭集群时。用户不能直接调用这个方法，但是在紧急的情况下，例如需要完整的关机或重启时，系统会调用该方法。

以下方法可以获得 master 的远程对象：

```
HMasterInterface getMaster()
    throws MasterNotRunningException, ZooKeeperConnectionException
```

上面的方法会返回 HMasterInterface 接口的 RPC 代理实例，允许用户直接通过这个实例访问 master 服务器。不过这个方法不是必须的，HBaseAdmin 内置了 master 所有 RPC 接口代理的封装。



除非用户确定自身的调用是安全的，否则不要直接调用 getMaster() 来获取 HMasterInterface 远程对象。HBaseAdmin 自身已经封装了 HMasterInterface 远程对象的调用功能，例如，检查输入是否合法，并转化成远程异常返回，或优化同步处理为异步处理以提升执行能力。

除了上述接口，HBaseAdmin 类还提供了以下基本接口。


```
boolean isMasterRunning()
```

通过这个接口检查 **master** 是否正在运行。用户也可以通过客户端程序在实例化 **HBaseAdmin** 类之前直接调用该接口确认其可以与 **master** 通信。

```
HConnection getConnection()
```

返回连接实例。详情见 4.6 节。

```
Configuration getConfiguration()
```

访问创建 **HBaseAdmin** 实例时使用到的配置实例，用户可以通过修改这个实例中的变量达到改变 **HBaseAdmin** API 调用时依赖的配置的目的。

```
close()
```

关闭 **HBaseAdmin** 实例的所有资源，包括与远程服务器的连接。

5.2.2 表操作

在介绍了基本操作后，下面我们介绍有关表的一系列操作。这些操作可以达到帮助表工作的目的，而非实际的内部模式。5.2.3 节讲解了命令行的表模式操作信息。

在开始工作前，首先要做的是建表，以下 API 就是建表的相关方法。

```
void createTable(HTableDescriptor desc)
void createTable(HTableDescriptor desc,byte[] startKey,
    byte[] endKey,int numRegions)
void createTable(HTableDescriptor desc,byte[][] splitKeys)

void createTableAsync(HTableDescriptor desc,byte[][] splitKeys)
```

上面描述的方法都使用到了 **HTableDescriptor** 实例，详情见 5.2.2 节，其中提到了建表的要点，其中包括列族的创建要点。例 5.1 是一个简单的有关建表的例子。

例 5.1 使用客户端 API 建表

```
Configuration conf = HBaseConfiguration.create();

HBaseAdmin admin = new HBaseAdmin(conf);❶

HTableDescriptor desc = new HTableDescriptor(❷
    Bytes.toBytes("testtable"));

HColumnDescriptor coldef = new HColumnDescriptor(❸
    Bytes.toBytes("colfam1"));
desc.addFamily(coldef);
```

```
admin.createTable(desc); ❶

boolean avail = admin.isTableAvailable(Bytes.toBytes("testtable"));❷
System.out.println("Table available: " + avail);
```

- ❶ 创建 HBaseAdmin 实例。
- ❷ 创建表描述符。
- ❸ 添加列族描述符到表描述符中。
- ❹ 调用建表方法 createTable()。
- ❺ 检查表是否可用。

另一种高级建表功能是，伴随建表操作进行预分区，将表在创建的时候划分出若干特定的 region。例 5.2 使用了两种预定义 region 的建表方法。

例 5.2 通过预分区的方式建表

```
private static void printTableRegions(String tableName)throws IOException { ❶
    System.out.println("Printing regions of table: " + tableName);
    HTable table = new HTable(Bytes.toBytes(tableName));
    Pair<byte[][]> pair = table.getStartEndKeys();❷
    for(int n = 0;n < pair.getFirst().length;n++){
        byte[] sk = pair.getFirst()[n];
        byte[] ek = pair.getSecond()[n];
        System.out.println("[ " + (n + 1) + " ]" +
            " start key: " +
            (sk.length == 8 ? Bytes.toLong(sk) : Bytes.toStringBinary(sk)) + ❸
            ", end key: " +
            (ek.length == 8 ? Bytes.toLong(ek) : Bytes.toStringBinary(ek)));
    }
}

public static void main(String[] args)throws IOException, InterruptedException {
    Configuration conf = HBaseConfiguration.create();
    HBaseAdmin admin = new HBaseAdmin(conf);

    HTableDescriptor desc = new HTableDescriptor(
        Bytes.toBytes("testtable1"));
    HColumnDescriptor coldef = new HColumnDescriptor(
        Bytes.toBytes("colfam1"));
    desc.addFamily(coldef);

    admin.createTable(desc,Bytes.toBytes(1L),Bytes.toBytes(100L),10); ❹
    printTableRegions("testtable1");

    byte[][] regions = new byte[][] { ❺
        Bytes.toBytes("A"),
        Bytes.toBytes("D"),
        Bytes.toBytes("G"),
```

```

        Bytes.toBytes("K"),
        Bytes.toBytes("O"),
        Bytes.toBytes("T")
    };
    desc.setName(Bytes.toBytes("testtable2"));
    admin.createTable(desc, regions);❹
    printTableRegions("testtable2");
}

```

- ❶ 打印表中 **region** 信息的帮助方法。
- ❷ 返回表中所有 **region** 的起始行键与终止行键列表。
- ❸ 打印这些行键，但不包括空的行键（起始与终止）。
- ❹ 执行建表命令，同时设置 **region** 边界。
- ❺ 创建表中 **region** 的拆分行键。
- ❻ 使用新表名和 **region** 的已拆分键值列表作为参数调用建表命令。

以上代码例子可以得到如下的输出：

```

Printing regions of table: testtable1
[1] start key:,end key: 1
[2] start key: 1,end key: 13
[3] start key: 13,end key: 25
[4] start key: 25,end key: 37
[5] start key: 37,end key: 49
[6] start key: 49,end key: 61
[7] start key: 61,end key: 73
[8] start key: 73,end key: 85
[9] start key: 85,end key: 100
[10] start key: 100,end key:
Printing regions of table: testtable2
[1] start key:,end key: A
[2] start key: A,end key: D
[3] start key: D,end key: G
[4] start key: G,end key: K
[5] start key: K,end key: O
[6] start key: O,end key: T
[7] start key: T,end key:

```

以上例子使用了 `HTable` 类中的方法 `getStartEndKeys()` 来获取所有 **region** 的边界。第一个 **region** 的起始行键与最后一个 **region** 的终止行键都是空字节，这是 **HBase** 中默认的规则。起始和终止行键都是已经计算好的，或是提供给用户的拆分键。需要注意的是，前一个 **region** 的终止行键与后一个 **region** 的起始行键是串联起来的——终止行键不包含在前一个 **region** 中，而是作为起始行键包含在后一个

region 中。

`createTable(HTableDescriptor desc,byte[] startKey,byte[] endKey,int numRegions)` 方法能够以特定数量拆分特定起始行键和特定终止行键,并创建表。参数中的 `startKey` 必须小于 `endKey`, 并且 `numRegions` 需要大于等于 3, 否则会抛出异常, 这样才能够确保 region 有最小的集合。

在这个方法中, region 的边界是通过终止行键减去起始行键然后除以给定的 region 数量计算得到的。在上面的例子中, 用户能够学习到怎样计算合适的 region 数量, 以保证计算出的行键可以填充表。

`createTable(HTableDescriptor desc,byte[][] splitKeys)` 方法在例子的第二部分中被使用到, 换句话说, 它使用已拆分行键的集合: 使用了已经拆分好的 region 边界列表, 因此结果都是与预期相符的。



上面提到的两种 `createTable()` 方法实际上是有联系的, `createTable(HTableDescriptor desc, byte[] startKey,byte[] endKey,int numRegions)` 方法使用 `Bytes.split()` 方法计算 region 分界, 然后将计算得到的边界作为已拆分边界列表, 并调用 `createTable(HTableDescriptor desc, byte[][] splitKeys)` 方法建表。

最后, 还有 `createTableAsync(HTableDescriptor desc,byte[][] splitKeys)` 方法, 这个方法使用表描述符和预拆分的 region 边界作为参数, 并进行异步建表, 但执行过程与 `createTable()` 方法殊途同归。



表的大多数管理功能都是异步的, 这对发送命令而不需要等待执行结果的场景是非常有用的。但是, 也有一些操作必须等待并获知操作执行成功后, 才可以继续执行其他操作, 因此, 表提供了异步与同步两种操作模式。

实际上, 同步模式仅仅是异步模式的简单封装, 增加了不断检查这个任务是否已经完成的循环操作。例如, `createTable()` 方法包装了 `createTableAsync()` 方法, 循环检查远程服务器的建表操作是否已经执行完成。

建表后用户通过下面的帮助方法可以获取所有表的列表和已创建表的表描述符, 或检查该表是否存在:

```
boolean tableExists(String tableName)
boolean tableExists(byte[] tableName)
HTableDescriptor[] listTables()
HTableDescriptor getTableDescriptor(byte[] tableName)
```

例 5.1 使用 `tableExists()` 方法检查建表是否成功, 并使用 `listTables()` 方法获取所有

已建表的表描述符对象，使用 `getTableDescriptor()` 方法获取某一个已建表的表描述符，例 5.3 使用了 `listTables()` 和 `getTableDescriptor()`，并展示了具有管理功能的 API 的返回值。

例 5.3 获取所有已建表的表结构

```
HBaseAdmin admin = new HBaseAdmin(conf);

HTableDescriptor[] htlds = admin.listTables();
for(HTableDescriptor htd : htlds){
    System.out.println(htd);
}

HTableDescriptor htd1 = admin.getTableDescriptor(
    Bytes.toBytes("testtable1"));
System.out.println(htd1);

HTableDescriptor htd2 = admin.getTableDescriptor(
    Bytes.toBytes("testtable10"));
System.out.println(htd2);
```

由于打印了所有表描述符以及表描述符中的信息，因此控制台的输出非常长，以下我们节选的部分关键信息。

```
Printing all tables...
{NAME => 'testtable1',FAMILIES => [{NAME => 'colfam1',BLOOMFILTER =>
'NONE',REPLICATION_SCOPE => '0',COMPRESSION => 'NONE',VERSIONS => '3',
TTL => '2147483647',BLOCKSIZE => '65536',IN_MEMORY => 'false',BLOCKCACHE
=> 'true'},{NAME => 'colfam2',BLOOMFILTER => 'NONE',REPLICATION_SCOPE
=> '0',COMPRESSION => 'NONE',VERSIONS => '3',TTL => '2147483647',
BLOCKSIZE => '65536',IN_MEMORY => 'false',BLOCKCACHE => 'true'},{NAME =>
'colfam3',BLOOMFILTER => 'NONE',REPLICATION_SCOPE => '0',COMPRESSION =>
'NONE',VERSIONS => '3',TTL => '2147483647',BLOCKSIZE => '65536',
IN_MEMORY => 'false',BLOCKCACHE => 'true'}}]
...
Exception org.apache.hadoop.hbase.TableNotFoundException: testtable10
...
at ListTablesExample.main(ListTablesExample.java)
```

上述的异常信息值得一提，如果用户试图访问一个不存在的表描述符，就会得到上述对应的异常，因此用户在访问这个 API 的时候，必须使用已存在的表的表名，或者对上述的代码使用 try/catch 封装来处理这些异常。

描述过建表的过程后，我们要提一下删除表的操作，HBaseAdmin 中的对应的 API 如下：

```
void deleteTable(String tableName)
void deleteTable(byte[] tableName)
```

这部分 API 提供了 String 与 byte 数组的参数，另外一点需要注意：一旦表被删除，所有的数据都会被删除。

在删除表之前，用户需要确认这张表是否已经被禁用（disabled），可通过如下方法设置该参数：

```
void disableTable(String tableName)
void disableTable(byte[] tableName)
void disableTableAsync(String tableName)
void disableTableAsync(byte[] tableName)
```

用户将表设置为禁用时，region 服务器会先将内存中近期内还未提交的已修改数据刷写到磁盘中，然后关闭所有的 region，并更新这张表的元数据，将所有 region 标记为下线状态。

用户可选择使用异步与同步两种模式，且这两种模式均支持不同格式的表名。



用户将表设置为禁用时可能会花费非常长的时间，甚至长达几分钟。这取决于在服务器内存中有多少近期更新的数据还没有写入磁盘。将一个 region 下线会先将内存中的数据写入到磁盘中，如果用户设置了较大的堆，这将导致 region 服务器需要向磁盘写入数 MB 甚至 GB 的数据。在负载很高的系统中进行数据写入时，多个进程间的竞争会使这个操作的执行时间变长。

一旦表被设置为禁用，但用户并不想删除它，可以通过以下方式重新启用该表。

```
void enableTable(String tableName)
void enableTable(byte[] tableName)
void enableTableAsync(String tableName)
void enableTableAsync(byte[] tableName)
```

这个操作在转移这张表的 region 到其他可用服务器时比较有用。此外，以下是检查表的状态的一组方法：

```
boolean isTableEnabled(String tableName)
boolean isTableEnabled(byte[] tableName)
boolean isTableDisabled(String tableName)
boolean isTableDisabled(byte[] tableName)
boolean isTableAvailable(byte[] tableName)
boolean isTableAvailable(String tableName)
```

例 5.4 集合了建表、删表、禁用表以及检查表状态等操作。

例 5.4 禁用、启用和检查表的状态

```
HBaseAdmin admin = new HBaseAdmin(conf);
```

```

HTableDescriptor desc = new HTableDescriptor(
    Bytes.toBytes("testtable"));
HColumnDescriptor coldef = new HColumnDescriptor(
    Bytes.toBytes("colfam1"));
desc.addFamily(coldef);
admin.createTable(desc);

try {
    admin.deleteTable(Bytes.toBytes("testtable"));
} catch(IOException e){
    System.err.println("Error deleting table: " + e.getMessage());
}

admin.disableTable(Bytes.toBytes("testtable"));
boolean isDisabled = admin.isTableDisabled(Bytes.toBytes("testtable"));
System.out.println("Table is disabled: " + isDisabled);

boolean avail1 = admin.isTableAvailable(Bytes.toBytes("testtable"));
System.out.println("Table available: " + avail1);

admin.deleteTable(Bytes.toBytes("testtable"));

boolean avail2 = admin.isTableAvailable(Bytes.toBytes("testtable"));
System.out.println("Table available: " + avail2);

admin.createTable(desc);
boolean isEnabled = admin.isTableEnabled(Bytes.toBytes("testtable"));
System.out.println("Table is enabled: " + isEnabled);

```

控制台输出如下（异常打印缩写）：

```

Creating table...
Deleting enabled table...
Error deleting table:
    org.apache.hadoop.hbase.TableNotDisabledException: testtable
...
Disabling table...
Table is disabled: true
Table available: true
Deleting disabled table...
Table available: false
Creating table again...
Table is enabled: true

```

用户试图删除一张已经被启用的表时会抛出类似上述的异常，告诉用户首先要将该表设置为禁用状态，或者客户端根据应用的需要对异常进行处理，用户可以明确地禁用表并重试删除表操作。

`isTableAvailable()` 方法返回了 `true`，即使表被设置为禁用状态时，该方法同样也

会返回 true。换句话说,这个方法只对表进行物理检查,而不关心表本身的逻辑状态,因此要获取这张表的可用状态需要使用 `isTableEnabled()` 和 `isTableDisabled()` 方法。

用户建表后,如果需要修改表结构必须删除表结构然后重建表,或采用如下方法改变表结构:

```
void modifyTable(byte[] tableName,HTableDescriptor htd)
```

与前面提到的 `deleteTable()` 操作类似,用户要先将表设置为禁用状态才能修改表结构。例 5.5 展示了如何先建表后修改表结构。

例 5.5 修改表结构

```
byte[] name = Bytes.toBytes("testtable");
HBaseAdmin admin = new HBaseAdmin(conf);
HTableDescriptor desc = new HTableDescriptor(name);
HColumnDescriptor coldef1 = new HColumnDescriptor(
    Bytes.toBytes("colfam1"));
desc.addFamily(coldef1);

admin.createTable(desc);❶

HTableDescriptor htd1 = admin.getTableDescriptor(name);❷
HColumnDescriptor coldef2 = new HColumnDescriptor(
    Bytes.toBytes("colfam2"));
htd1.addFamily(coldef2);
htd1.setMaxFileSize(1024 * 1024 * 1024L);

admin.disableTable(name);
admin.modifyTable(name,htd1);❸
admin.enableTable(name);

HTableDescriptor htd2 = admin.getTableDescriptor(name);
System.out.println("Equals: " + htd1.equals(htd2));❹
System.out.println("New schema: " + htd2);
```

- ❶ 使用旧结构建表。
- ❷ 获取表结构,增加列族,并修改最大文件限制属性。
- ❸ 禁用表,修改,然后启用表。
- ❹ 检查表结构是否已经被修改成功。

以下输出说明用户通过客户端代码修改表结构和通过服务器端查询表结构,经匹配查询到的表结构与客户端保留的表结构是一致的。

```
Equals: true
New schema: {NAME => 'testtable',MAX_FILESIZE => '1073741824',FAMILIES =>
```



```
{(NAME => 'colfam1',BLOOMFILTER => 'NONE',REPLICATION_SCOPE => '0',
COMPRESSION => 'NONE',VERSIONS => '3',TTL => '2147483647',BLOCKSIZE =>
'65536',IN_MEMORY => 'false',BLOCKCACHE => 'true')},{(NAME => 'colfam2',
BLOOMFILTER => 'NONE',REPLICATION_SCOPE => '0',COMPRESSION => 'NONE',
VERSIONS => '3',TTL => '2147483647',BLOCKSIZE => '65536',IN_MEMORY =>
'false',BLOCKCACHE => 'true')}}}
```

调用 `HTableDescriptor` 对象的 `equals()` 方法比较客户端本地的实例与从元数据获取的实例是否一致（包括所有列族以及与它们相关的设置）。



`modifyTable()` 方法只提供了异步的操作模式，没有提供同步的操作模式。如果用户需要确认修改是否已经成功，需要在客户端代码中显式循环地调用 `getTableDescriptor()` 方法获取元数据，直到结果与本地实例匹配。

5.2.3 模式操作

除了 `modifyTable()` 方法，`HBaseAdmin` 类另外提供了几种表结构修改方法。当然，首先要确保表已经被禁用。

与列相关操作的方法集合如下：

```
void addColumn(String tableName,HColumnDescriptor column)
void addColumn(byte[] tableName,HColumnDescriptor column)
void deleteColumn(String tableName,String columnName)
void deleteColumn(byte[] tableName,byte[] columnName)
void modifyColumn(String tableName,HColumnDescriptor descriptor)
void modifyColumn(byte[] tableName,HColumnDescriptor descriptor)
```

通过以上方法，用户可以创建、删除、修改列族。增加或修改一个列族需要准备一个 `HColumnDescriptor` 实例，详情见 5.1.3 节。使用 `getTableDescriptor()` 方法可以查询当前表结构，然后调用 `getColumnFamilies()` 方法可以查询到所有已存在列族的信息。

如果用户不采用上述方法，其只能通过粗粒度地提供常用格式的表名以及列族名来调用删除操作，但是这些调用都是异步的，结果不可控，用户要自己承担这种风险。

使用场景：Hush

值得一提的是，表创建、修改都可以基于一个外部的配置文件。Hush 模式恰好利用了这个想法，其把表和列的描述定义在 XML 文件中，这个文件可读并包含了与当前表结构的对比。下面的示例提供了相应的核心代码。

```
private void createOrChangeTable(final TableSchema schema)
throws IOException {
```

```

HTableDescriptor desc = null;
if (tableExists(schema.getName(), false)) {
    desc = getTable(schema.getName(), false);
    LOG.info("Checking table " + desc.getNameAsString() + "...");
    final HTableDescriptor d = convertSchemaToDescriptor(schema);

    final List<HColumnDescriptor> modCols =
        new ArrayList<HColumnDescriptor>();
    for (final HColumnDescriptor cd : desc.getFamilies()) {
        final HColumnDescriptor cd2 = d.getFamily(cd.getName());
        if (cd2 != null && !cd.equals(cd2)) { ❶
            modCols.add(cd2);
        }
    }
    final List<HColumnDescriptor> delCols =
        new ArrayList<HColumnDescriptor>(desc.getFamilies());
    delCols.removeAll(d.getFamilies());
    final List<HColumnDescriptor> addCols =
        new ArrayList<HColumnDescriptor>(d.getFamilies());
    addCols.removeAll(desc.getFamilies());

    if (modCols.size() > 0 || addCols.size() > 0 || delCols.size() > 0 || ❷
        !hasSameProperties(desc, d)) {
        LOG.info("Disabling table...");
        hbaseAdmin.disableTable(schema.getName());
        if (modCols.size() > 0 || addCols.size() > 0 || delCols.size() > 0) {
            for (final HColumnDescriptor col : modCols) {
                LOG.info("Found different column -> " + col);
                hbaseAdmin.modifyColumn(schema.getName(), col, col.GetNameAsString(), ❸
                    col);
            }
            for (final HColumnDescriptor col : addCols) {
                LOG.info("Found new column -> " + col);
                hbaseAdmin.addColumn(schema.getName(), col); ❹
            }
            for (final HColumnDescriptor col : delCols) {
                LOG.info("Found removed column -> " + col);
                hbaseAdmin.deleteColumn(schema.getName(), col, col.GetNameAsString()); ❺
            }
        } else if (!hasSameProperties(desc, d)) {
            LOG.info("Found different table properties...");
            hbaseAdmin.modifyTable(Bytes.toBytes(schema.getName()), d); ❻
        }
        LOG.info("Enabling table...");
        hbaseAdmin.enableTable(schema.getName());
        LOG.info("Table enabled");
        desc = getTable(schema.getName(), false);
        LOG.info("Table changed");
    } else {
        LOG.info("No changes detected!");
    }
} else {

```

```

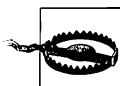
desc = convertSchemaToDescriptor(schema);
LOG.info("Creating table " + desc.getNameAsString() + "...");
hbaseAdmin.createTable(desc);❷
LOG.info("Table created");
}
}

```

- ❶ 用 HBase 表结构的元数据与 XML 定义文件进行对比，用户可以看到两者之间的差异。
- ❷ 检查列族和表的定义是否有区别。
- ❸ 修改列族信息前一定要确保表已经被禁用。
- ❹ 增加列。
- ❺ 删除列。
- ❻ 如果发现表结构不同就修改表结构。
- ❼ 如果表不存在就创建表。

5.2.4 集群管理

HBaseAdmin 类提供的最后一组操作是集群管理操作。允许用户查看集群当前的状态，执行表级任务和管理 region。8.6 节描述了有关 region 的生命周期。



以下操作都是面向高级用户的，请谨慎使用。

```

static void checkHBaseAvailable(Configuration conf)
ClusterStatus getClusterStatus()

```

checkHBaseAvailable() 方法可以验证客户端应用是否能与给定文件配置中的远程 HBase 集群进行通信。如果失败，该方法会抛出异常，换句话说，该方法并返回布尔型变量，即成功了无返回值且失败了抛出异常。

getClusterStatus() 方法可以通过查询 ClusterStatus 类的实例返回集群信息，这个对象包含了集群状态的详细信息。详情见 5.2.5 节。

```

void closeRegion(String regionname,String hostAndPort)
void closeRegion(byte[] regionname,String hostAndPort)

```

以上方法可以让已经在 region 服务器中上线的特定 region 下线。任何可用表的所有 region 都应该是在线状态，所以用户不能随意地下线某个 region。

使用这个方法时，用户需要提供准确的 regionname，这个参数需要与元数据表 .META. 中存储的一样，此外还需要 hostAndPort 参数，如果这个参数不为空，程序逻辑就

不会再去 .META. 元数据表中查找 hostAndPort 的信息了。

这个命令不经过 master 节点，即客户端直接与 region 服务器通信并发送 region 下线命令，该命令对 master 节点是透明的。

```
void flush(String tableNameOrRegionName)
void flush(byte[] tableNameOrRegionName)
```

表中所有的更新在未刷写到磁盘之前都会先写入 region 的 MemStore 实例中。客户端程序可以在达到 memstore 刷写上限（memstore flush size）（见 5.1.2 节）之前显式地以同步模式调用当前方法，以达到将数据刷写到磁盘的目的。

该方法需要 region 名或表名，且在该方法执行时会自行判断用户提供的名称与已存在的表是否匹配。如果输入的是已存在的表名，就按照表级别处理；如果不存在该表名，就按照 region 名处理；如果既不是表名也不是 region 名，就抛出 UnknownRegionException 异常。

```
void compact(String tableNameOrRegionName)
void compact(byte[] tableNameOrRegionName)
```

当前方法与前面的方法类似，都需要将 region 名和表名作为参数，这个方法是个异步方法，因为合并是个花费时间比较长的操作，客户端没有必要一直等待这个操作完成。调用这个方法后，这个表或者 region 会加入到这个 region 所在的 region 服务器的一个执行队列，并会在后台完成操作，或者是在上线该表 region 的所有 region 服务器中执行（具体过程见 1.4.3 节）。

```
void majorCompact(String tableNameOrRegionName)
void majorCompact(byte[] tableNameOrRegionName)
```

以上方法类似于 compact() 方法，也依赖于后台队列操作，不过是执行的 major 合并。提供了表名后，API 内部会迭代这张表所有的 region，并顺序调用 region 的合并操作。

```
void split(String tableNameOrRegionName)
void split(byte[] tableNameOrRegionName)
void split(String tableNameOrRegionName,String splitPoint)
void split(byte[] tableNameOrRegionName,byte[] splitPoint)
```

这个方法用于拆分一个 region 或拆分整张表，如果提供了表名，API 内部会迭代这张表的所有 region 并调用拆分命令。

值得注意的一个参数是 splitPoint，如果这个参数不为空，并指定了特定的 region，那么这个 region 会按照这个指定的行键来拆分。如果指定的是表名，整张表的 region 在执行拆分前会进行检查，且包含这个特定的行键的 region 会按照这个特定的行键进行拆分。

如果使用 `splitPoint`，用户首先要确保行键是合法的，即它必须在给定的 `region` 中，并且必须大于 `region` 的起始行键，因为在一个 `region` 的起始行键处进行 `region` 拆分是无效的。如果提供的行键不正确，这个行键会被忽略，并且客户端没有任何反馈，但部署这个 `region` 的 `region` 服务器会在日志中打印如下的信息：

```
Split row is not inside region key range or is equal to startkey:
<split row>
```

```
void assign(byte[] regionName,boolean force)
void unassign(byte[] regionName,boolean force)
```

客户端可以通过上面的方法来控制 `region` 的上线和下线。第一个方法可以提供上线操作，上线操作会基于 `master` 生成的上线计划自动执行，第二个方法提供了 `region` 的下线操作。

`force` 参数设置为 `true` 对上面两个方法的意义是不一样的：第一个方法 `assign()` 会强制在 `ZooKeeper` 中标记这个 `region` 为下线状态，并将这个 `region` 转移到一个新的 `region` 服务器中。用户使用时需要注意这个 `region` 是否已经上线。

第二个方法 `unassign()` 意味着无论当前 `region` 是否在线都会强制再做一次下线操作。如果 `force` 设置为 `false` 就不会带来任何影响。

```
void move(byte[] encodedRegionName,byte[] destServerName)
```

使用 `move()` 方法可以通过客户端控制某个 `region` 在哪台服务器上线。用户可以使用此方法将一个 `region` 从当前 `region` 服务器移动到一个新的 `region` 服务器。`destServerName` 参数可以设置为 `null`，这样会获得一个随机的服务器地址，否则必须获取一个合法的服务器地址，即一个正在运行的 `region` 服务器进程。如果这个参数有误或者这个服务器当前没有回应，这个 `region` 会在其他服务器上线。最糟糕的情况下，这次移动 `region` 的操作会失败，并让这个 `region` 下线。

```
boolean balanceSwitch(boolean b)
boolean balancer()
```

第一个方法可以控制 `region` 的负载均衡算法是否开启。如果负载均衡算法已经打开，`balancer()` 能主动运行负载均衡算法将每台 `region` 服务器上线的 `region` 进行均匀再分配。11.5 节解释了相应的细节。

```
void shutdown()
void stopMaster() {
void stopRegionServer(String hostnamePort)
```

这些方法会分别进行关闭集群、关闭 `master` 进程和关闭某台 `region` 服务器操作。一旦调用这个方法，被执行的服务器会马上进入关闭状态，即不存在延时，且

是个不可逆的过程。

第 8 章和第 11 章描述了更多更高级、更强大的功能。使用时需要特别注意！

5.2.5 集群状态信息

调用 `HBaseAdmin.getClusterStatus()` 可以查询 `ClusterStatus` 实例，这个实例包含了 `master` 搜集到的整个集群信息。注意，这个类也有 `setter` 方法，此方法允许用户修改里面的信息，但通过 `set` 方法修改的仅仅是本地副本的变量，除非用户需要修改本地副本的变量值。

表 5-4 列出了 `ClusterStatus` 类的所有方法。

表 5-4 `ClusterStatus` 提供信息概览

方法	描述
<code>int getServersSize()</code>	当前活着的 <code>region</code> 服务器的数量，此数量不包括不可用状态的 <code>region</code> 服务器
<code>Collection<ServerName> getServers()</code>	当前存活的 <code>region</code> 服务器的列表，包括 <code>region</code> 服务器的服务、IP、RPC 端口、启动时间戳等
<code>int getDeadServers()</code>	当前处于不可用状态的 <code>region</code> 服务器的数量，此数量不包括可用状态的 <code>region</code> 服务器
<code>Collection<ServerName>getDeadServerNames()</code>	当前处于不可用状态的 <code>region</code> 服务器的列表，包括 <code>region</code> 服务器的服务 ip、RPC 端口等
<code>double getAverageLoad()</code>	平均每台 <code>region</code> 服务器上上线了多少 <code>region</code> ，该方法类似于 <code>getRegionsCount()</code>
<code>int getRegionsCount()</code>	集群中 <code>region</code> 的总数量
<code>int getRequestsCount()</code>	集群的请求 TPS
<code>String getHBaseVersion()</code>	返回当前集群的软件编译版本
<code>byte getVersion()</code>	返回 <code>ClusterStatus</code> 实例的版本号，通过序列化的方式在 RPC 阶段传输
<code>String getClusterId()</code>	返回集群的唯一标识。这个值是集群第一次启动时通过 UUID 生成的，存在根目录下的 <code>hbase.id</code> 中
<code>Map<String,RegionState> getRegionsInTransition()</code>	返回当前集群正在进行处理的 <code>region</code> 的事务列表，即移动操作、上线操作和下线操作。键是编码后的 <code>region</code> 名（由 <code>HRegToInfo.getEncodeName()</code> 返回），值是 <code>Region State</code> ^a 的实例
<code>HServerLoad getLoad(ServerName sn)</code>	返回给定 <code>region</code> 服务器的当前负载状况

^a 详情见 8.6 节。

用户通过查看集群状态可以在较高层次上看到集群的整体情况。用户查看使用 getServers() 方法返回的 ServerName 实例能够获取所有当前处于可用状态的服务器的实际信息。表 5-5 罗列了所有的可用方法。

表 5-5 ServerName 提供信息概览

方法	描述
String getHostname()	返回服务器的域名，如果域名不可用会直接返回 IP 地址
String getHostAndPort()	返回域名与 RPC 端口的合并字符串，用“:”分隔，例如，<hostname>:<rpc-port>
long getStartcode()	服务器启动的时间，单位是毫秒，使用 System.currentTimeMillis() 进行获取
String getServerName()	获取服务器名，服务器名是<hostname>、<rpc-port>和<start-code>的组合
int getPort()	返回服务器端的 RPC 端口

通过提供 HServerLoad 实例，每台 region 服务器可以提供它们的负载信息。用户调用 ClusterStatus 实例的 getLoad() 方法可以获取 HServerLoad 实例。使用上面提到的 ServerName，通过 getServers() 方法可以获取并迭代访问服务器的信息，而当前提到的 HServerLoad 不仅提供了服务器本身的信息，还提供了每台服务器管理的 region 的信息。表 5-6 列出了所有可用方法。

表 5-6 HServerLoad 提供信息概览

方法	描述
byte getVersion()	返回 HServerLoad 的版本号，RPC 序列化进程时会用到这个参数
int getLoad()	等同于 getNumberOfRegions() 的返回值
int getNumberOfRegions()	当前 region 服务器上线的 region 数量
int getNumberOfRequests()	返回当前 region 服务器这个周期内的 TPS，周期可以通过参数 hbase.regionserver.msginterval 来设定。请求数会在一个周期结束后清零，它会统计所有的 API 请求，如 get、put、increment、delete 等
int getUsedHeapMB()	JVM 已使用的内存，单位为 MB
int getMaxHeapMB()	JVM 最大可使用内存，单位为 MB
int getStorefiles()	当前 region 服务器的存储文件数量，即包括这个服务器管理的所有 region
int getStorefileSizeInMB()	当前 region 服务器的存储文件的总存储量，单位为 MB
int getStorefileIndexSizeInMB()	当前 region 服务器的存储文件的索引大小，单位是 MB
int getMemStoreSizeInMB()	当前 region 服务器的已用写缓存的大小，包括这个 region 服务器上所有的 region
Map<byte[],RegionLoad> getRegionsLoad()	返回当前 region 服务器中每个 region 的负载情况，以 Map 的形式返回，键是 region 名，值是之前讨论过的 RegionLoad 实例

最后，我们来讨论这个专用类 RegionLoad。表 5-7 列出了所有可用信息。

表 5-7 RegionLoad 提供信息概览

方法	描述
byte[] getName()	返回 region 名，以原始的二进制 byte 数组格式返回
String getNameAsString()	将二进制 region 名转换为字符串并返回
int getStores()	当前 region 的列族数量
int getStorefiles()	当前 region 的存储文件数量
int getStorefileSizeMB()	当前 region 的存储文件占用空间，MB 为单位
int getStorefileIndexSizeMB()	当前 region 的存储文件的索引信息的大小，MB 为单位
int getMemStoreSizeMB()	当前 region 使用的 MemStore 的大小，单位是 MB
long getRequestsCount()	当前 region 的本次统计周期内的 TPS
long getReadRequestsCount()	当前 region 的本次统计周期内的 QPS
long getWriteRequestsCount()	当前 region 的本次统计周期内的 WPS

例 5.6 展示了所有可用的 getter 方法。

例 5.6 获取集群状态

```
HBaseAdmin admin = new HBaseAdmin(conf);

ClusterStatus status = admin.getClusterStatus();❶

System.out.println("Cluster Status:\n-----");
System.out.println("HBase Version: " + status.getHBaseVersion());
System.out.println("Version: " + status.getVersion());
System.out.println("No. Live Servers: " + status.getServersSize());
System.out.println("Cluster ID: " + status.getClusterId());
System.out.println("Servers: " + status.getServers());
System.out.println("No. Dead Servers: " + status.getDeadServers());
System.out.println("Dead Servers: " + status.getDeadServerNames());
System.out.println("No. Regions: " + status.getRegionsCount());
System.out.println("Regions in Transition: " +
    status.getRegionsInTransition());
System.out.println("No. Requests: " + status.getRequestsCount());
System.out.println("Avg Load: " + status.getAverageLoad());

System.out.println("\nServer Info:\n-----");
for(ServerName server : status.getServers()){❷
    System.out.println("Hostname: " + server.getHostname());
    System.out.println("Host and Port: " + server.getHostAndPort());
    System.out.println("Server Name: " + server.getServerName());
    System.out.println("RPC Port: " + server.getPort());
    System.out.println("Start Code: " + server.getStartcode());

HServerLoad load = status.getLoad(server);❸
```



```

System.out.println("\nServer Load:\n-----");
System.out.println("Load: " + load.getLoad());
System.out.println("Max Heap(MB): " + load.getMaxHeapMB());
System.out.println("Memstore Size(MB): " + load.getMemStoreSizeInMB());
System.out.println("No. Regions: " + load.getNumberOfRegions());
System.out.println("No. Requests: " + load.getNumberOfRequests());
System.out.println("Storefile Index Size(MB): " +
    load.getStorefileIndexSizeInMB());
System.out.println("No. Storefiles: " + load.getStorefiles());
System.out.println("Storefile Size(MB): " + load.getStorefileSizeInMB());
System.out.println("Used Heap(MB): " + load.getUsedHeapMB());

System.out.println("\nRegion Load:\n-----");
for(Map.Entry<byte[],HServerLoad.RegionLoad> entry : ❶
    load.getRegionsLoad().entrySet()){
    System.out.println("Region: " + Bytes.toStringBinary(entry.getKey()));

    HServerLoad.RegionLoad regionLoad = entry.getValue();❷

    System.out.println("Name: " + Bytes.toStringBinary(
        regionLoad.getName()));
    System.out.println("No. Stores: " + regionLoad.getStores());
    System.out.println("No. Storefiles: " + regionLoad.getStorefiles());
    System.out.println("Storefile Size(MB): " +
        regionLoad.getStorefileSizeMB());
    System.out.println("Storefile Index Size(MB): " +
        regionLoad.getStorefileIndexSizeMB());
    System.out.println("Memstore Size(MB): " +
        regionLoad.getMemStoreSizeMB());
    System.out.println("No. Requests: " + regionLoad.getRequestsCount());
    System.out.println("No. Read Requests: " +
        regionLoad.getReadRequestsCount());
    System.out.println("No. Write Requests: " +
        regionLoad.getWriteRequestsCount());
    System.out.println();
}
}

```

- ❶ 获取 ClusterStatus 实例以获得集群状态。
- ❷ 迭代所有服务器的信息。
- ❸ 获取当前服务器负载信息。
- ❹ 迭代当前服务器所有 region 的信息。
- ❺ 获得当前 region 的负载信息。

如果以单机模式启动，并用这本书早期的例子，可以看到如下信息：

```
Cluster Status:
```

Avg Load: 12.0
HBase Version: 0.91.0-SNAPSHOT
Version: 2
No. Servers: [10.0.0.64,60020,1304929650573]
No. Dead Servers: 0
Dead Servers: []
No. Regions: 12
No. Requests: 0

Server Info:

Hostname: 10.0.0.64
Host and Port: 10.0.0.64:60020
Server Name: 10.0.0.64,60020,1304929650573
RPC Port: 60020
Start Code: 1304929650573

Server Load:

Load: 12
Max Heap(MB): 987
Memstore Size(MB): 0
No. Regions: 12
No. Requests: 0
Storefile Index Size(MB): 0
No. Storefiles: 3
Storefile Size(MB): 0
Used Heap(MB): 62

Region Load:

Region: -ROOT-,0
Name: -ROOT-,0
No. Stores: 1
No. Storefiles: 1
Storefile Size(MB): 0
Storefile Index Size(MB): 0
Memstore Size(MB): 0
No. Requests: 52
No. Read Requests: 51
No. Write Requests: 1

Region: .META.,,1
Name: .META.,,1
No. Stores: 1
No. Storefiles: 0
Storefile Size(MB): 0
Storefile Index Size(MB): 0
Memstore Size(MB): 0
No. Requests: 4764
No. Read Requests: 4734

No. Write Requests: 30

Region: hush,,1304930393059.1ae3ea168c42fa9c855051c888ed36d4.

Name: hush,,1304930393059.1ae3ea168c42fa9c855051c888ed36d4.

No. Stores: 1

No. Storefiles: 0

Storefile Size(MB): 0

Storefile Index Size(MB): 0

Memstore Size(MB): 0

No. Requests: 20

No. Read Requests: 14

No. Write Requests: 6

Region: ldom,,1304930390882.520fc727a3ce79749bcbbad51e138fff.

Name: ldom,,1304930390882.520fc727a3ce79749bcbbad51e138fff.

No. Stores: 1

No. Storefiles: 0

Storefile Size(MB): 0

Storefile Index Size(MB): 0

Memstore Size(MB): 0

No. Requests: 14

No. Read Requests: 6

No. Write Requests: 8

Region: sdom,,1304930389795.4a49f5ba47e4466d284cea27629c26cc.

Name: sdom,,1304930389795.4a49f5ba47e4466d284cea27629c26cc.

No. Stores: 1

No. Storefiles: 0

Storefile Size(MB): 0

Storefile Index Size(MB): 0

Memstore Size(MB): 0

No. Requests: 8

No. Read Requests: 0

No. Write Requests: 8

Region: surl,,1304930386482.c965c89368951cf97d2339a05bc4bad5.

Name: surl,,1304930386482.c965c89368951cf97d2339a05bc4bad5.

No. Stores: 4

No. Storefiles: 0

Storefile Size(MB): 0

Storefile Index Size(MB): 0

Memstore Size(MB): 0

No. Requests: 1329

No. Read Requests: 1226

No. Write Requests: 103

Region: testtable,,1304930621191.962abda0515c910ed91f7520e71ba101.

Name: testtable,,1304930621191.962abda0515c910ed91f7520e71ba101.

No. Stores: 2

No. Storefiles: 0

Storefile Size(MB): 0

Storefile Index Size(MB): 0

Memstore Size(MB): 0
No. Requests: 29
No. Read Requests: 0
No. Write Requests: 29

Region: testtable,row-030,1304930621191.0535bb40b407321d499d65bab9d3b2d7.
Name: testtable,row-030,1304930621191.0535bb40b407321d499d65bab9d3b2d7.
No. Stores: 2
No. Storefiles: 2
Storefile Size(MB): 0
Storefile Index Size(MB): 0
Memstore Size(MB): 0
No. Requests: 6
No. Read Requests: 6
No. Write Requests: 0

Region: testtable,row-060,1304930621191.81b04004d72bd28cc877cb1514dbab35.
Name: testtable,row-060,1304930621191.81b04004d72bd28cc877cb1514dbab35.
No. Stores: 2
No. Storefiles: 0
Storefile Size(MB): 0
Storefile Index Size(MB): 0
Memstore Size(MB): 0
No. Requests: 41
No. Read Requests: 0
No. Write Requests: 41

Region: url,,1304930387617.a39d16967d51b020bb4dad13a80a1a02.
Name: url,,1304930387617.a39d16967d51b020bb4dad13a80a1a02.
No. Stores: 1
No. Storefiles: 0
Storefile Size(MB): 0
Storefile Index Size(MB): 0
Memstore Size(MB): 0
No. Requests: 11
No. Read Requests: 8
No. Write Requests: 3

Region: user,,1304930388702.60bae27e577a620ae4b59bc830486233.
Name: user,,1304930388702.60bae27e577a620ae4b59bc830486233.
No. Stores: 1
No. Storefiles: 0
Storefile Size(MB): 0
Storefile Index Size(MB): 0
Memstore Size(MB): 0
No. Requests: 11
No. Read Requests: 9
No. Write Requests: 2

Region: user-surl,,1304930391974.71b9cecc9c111a5217bd1a81bde60418.
Name: user-surl,,1304930391974.71b9cecc9c111a5217bd1a81bde60418.
No. Stores: 1

No. Storefiles: 0
Storefile Size(MB): 0
Storefile Index Size(MB): 0
Memstore Size(MB): 0
No. Requests: 24
No. Read Requests: 21
No. Write Requests: 3

第 6 章

可用客户端

HBase 面对不同的编程语言拥有不同的客户端。本章会简单地介绍可用的客户端。

6.1 REST、Thrift 和 Avro 的介绍

访问 HBase 的均是目前比较流行的编程语言和环境。用户可以直接使用 HBase 客户端 API，或者使用一些能够将请求转换成 API 调用的代理。这些代理将原生 Java API 包装成其他协议，这样客户端可以使用 API 提供的任意外部语言来编写程序。通常来说，外部 API 实现了专门基于 Java 的服务，而这种服务能够在内部使用由 HTable 客户端提供的 API。这种方式简化了网关服务器的实现和维护。

客户端与网关之间的协议是由当前可用选择以及远程客户端的需求决定的。最常用的就是 REST（Representational State Transfer，表述性状态转移）^①，其基于现有网络技术。实际的传输是典型的 HTTP 协议——它是 Web 应用的标准传输协议。由于协议层负责传输可互操作格式的数据，这使得 REST 成为异构系统之间传输数据的理想选择。

REST 自定义了语义，这样协议就可以被用来以普通的方式定位远程服务的资源。不改变 REST 的协议，REST 可以直接兼容现有的技术，例如，Web 服务和代理。资源作为请求 URI 的一部分是唯一确定的，这定义了一个新协议的标准，但这跟基于 SOAP^②的服务完全相反。

REST 与 SOAP 协议都面临冗长的协议等级。客户端与服务器端间的通信协议是可读

① 见 Roy T. Fielding 在 2000 年发表的文章“Architectural Styles and the Design of Network-based Software Architectures”（<http://www.ics.uci.edu/~fielding/pubs/dissertation/top.html>）。

② 见 SOAP 的官方文档（<http://www.w3.org/TR/soap/>）。SOAP 全称是 Simple Object Access Protocol，SOAP 使用 HTTP 作为传输协议，但每个服务都拓展了一个不同的 API。

的简单文本或者基于 XML 的文本，数据在网络传输前进行透明压缩可以一定程度上缓解这个问题。

尤其在拥有大规模集群的公司中，巨额的带宽消费和许多相互隔离的服务，让人觉得需要减少开销并实现自己的 RPC 层。Google 就是其中之一，他们开发了 Protocol Buffer 框架^①。由于最初的实现没有发布，Facebook 就开发了一套类似的版本，叫做 Thrift^②。随后，Hadoop 项目创建者启动了第 3 个项目，叫做 Apache Avro^③，提供了另一种可选的实现。

这些框架都有类似的特性集，它们能够支持大量的语言，而且或多或少能够提升编码的效率。Protocol Buffer 与 Thrift 和 Avro 之间最大的不同是，它没有自己的 RPC 堆，而它生成的 RPC 定义需要被后来其他的 RPC 库使用。

HBase 提供了 REST、Thrift、Avro 的辅助服务。它们可以被实现成专门的网关服务，这些服务可以运行在专有的或共享的机器中。因为 Thrift 与 Avro 都有各自的 RPC 实现，所以网关服务仅仅是在它们的基础上进行了封装。至于 REST，HBase 则采用了自己的实现，并提供了访问存储数据的途径。



事实上，REST 服务器支持 Protocol Buffer，Protocol Buffer 能够利用 HTTP 协议的 Accept 首部来发送和接受被 Protocol Buffer 编码过的数据，而不用另外实现 RPC 服务。详情见 6.2.2 节。

图 6-1 展示了远程客户端如何使用已提供的端点访问专门的网关服务。

这些服务内部直接使用基于 HTable 的客户端 API 来访问表。用户能看见它们如何从 region 服务器的顶端开始处理，以及共享相同的物理机。没有人明确地建议网关服务器如何部署，用户可以把它们部署到专门的机器上。

另一种方法是直接在客户端节点运行网关服务，例如，用户的网页服务使用 PHP 构建 HTML 页面，这更有利于在同一台物理机中运行网关服务与 Web 服务。这种方式下，客户端与服务器端的通信是本地通信，因此网关与 HBase 之间的 RPC 协议是原生协议。



从客户端连接 HBase 时要格外小心地检查，网关服务要部署在恰当的物理机上。网关服务的性能会被当前机器的负载和网络传输数据量所影响：因此要确认那些不等待资源的进程，例如，CPU 时钟周期，以及网络带宽的使用。

① 见官方 Protocol Buffer 项目网站 <http://code.google.com/p/protobuf/>。

② 见 Thrift 项目网站 <http://thrift.apache.org/>。

③ 见 Apache Avro 项目网站 <http://avro.apache.org/>。

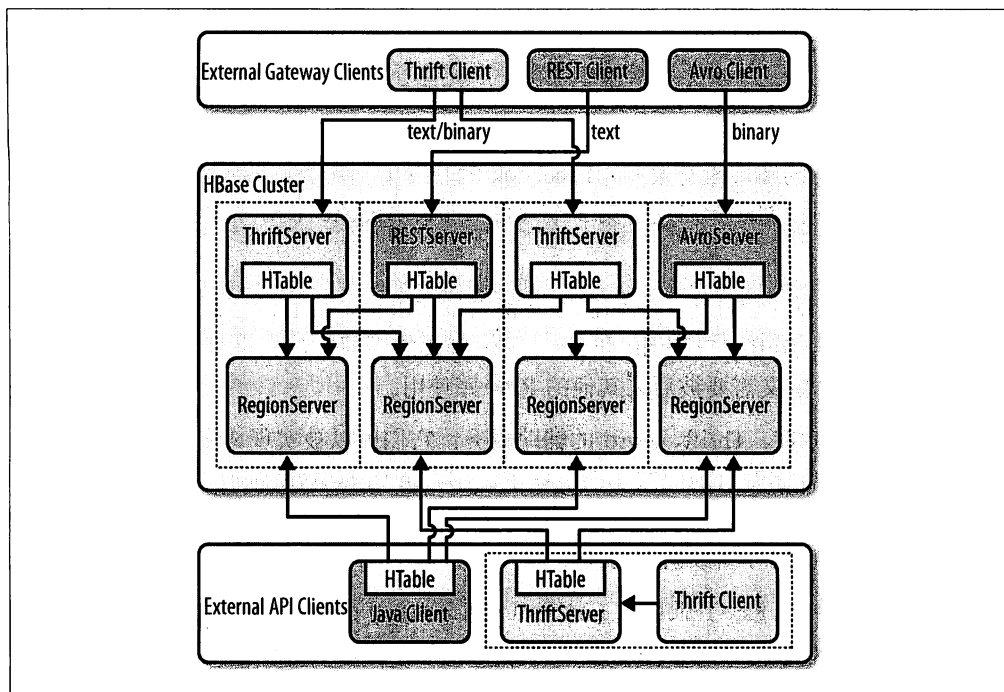


图 6-1 客户端通过网关服务器进行连接

每个请求使用一个服务而非建立一个新连接的优势要追溯到 4.5 节——用户需要复用连接来获得最优性能。生存期短的进程将在建立连接以及准备元数据上花费更多的时间，而非在其实际操作上花费时间。尤其是服务器端缓存着 region 的位置信息，这使得复用非常重要，否则每个客户端都将进行一次完整的从行到 region 的逐字节的查找来获取它们想要的信息。

选择一种网关类型并不是一件简单的任务，这取决于用户自身的实际场景。如果只有一种选择，自然没有必要纠结使用 3 个不同的网关机来服务。在 REST 以及更有效率的 Thrift 中，初始化参数或者相似的序列化格式都表明在高吞吐量场景中，纯二进制格式具有优势。从另一方面来说，如果用户仅有少许的请求，但是数据量比较大，那么 REST 是比较合适的。大致的场景分类如下所示。

REST 场景

REST 支持现有的基于 Web 的体系，它能够完美地融合反向代理和其他缓存技术。并行运行许多 REST 服务可以分摊它们之间的负载。例如，用户可以在每台拥有的应用服务器中运行 REST，创建一个单点应用到服务（single-app-to-server）的关系。

当用户从吞吐量的角度考虑需要的最佳性能时，可以使用严谨的二进制协议。用户可以运行较少的服务，例如，在多应用服务（*many-app-to-server*）的基数下，每个 region 服务器运行一个服务。

6.2 交互客户端

第一组客户端是具有交互性（interactive）的一组，它们按需发送客户端 API 调用到服务器端，例如，get、put 和 delete 操作。基于用户可选择的协议，用户可以使用提供的网关服务来获得从应用程序访问服务的途径。

6.2.1 原生 Java

第 3 章和第 4 章有关于原生 Java API 的详细描述。在很多场景中，用户可以直接使用 Htable 并通过原生的 RPC 调用与 HBase 服务器进行交互，而不需要开启网关服务。参见提到的章节来实现一个原生 Java 客户端。

6.2.2 REST

HBase 提供了强大的 REST 服务器，它能够支持完整的客户端以及管理 API。此外，它还提供了不同的消息格式，客户端可以灵活选择用于应用程序与服务器端之间交互的消息格式。

1. 操作

基于 REST 服务的客户端在能够与 HBase 通信之前需要先启动 REST 网关服务。这个工作可以由已提供的脚本来完成。下面的命令展示了如何获得命令行帮助，然后启动一个非守护进程模式的 REST 服务器：

```
$ bin/hbase rest
usage: bin/hbase rest start [-p <arg>] [-ro]
-p,--port <arg>    Port to bind to [default: 8080]
-ro,--readonly      Respond only to GET HTTP method requests [default:
                    false]

To run the REST server as a daemon,execute bin/hbase-daemon.sh start|stop
rest [-p <port>] [-ro]

$ bin/hbase rest start
^C
```

用户需要按 Ctrl+C 组合键退出进程。帮助信息展示了用户需要通过不同的脚本启动后台进程：

```
$ bin/hbase-daemon.sh start rest
starting rest, logging to /var/lib/hbase/logs/hbase-larsgeorge-rest-<servername>.out
```

一旦网关服务启动后，用户可以在命令行中运行 `curl`^① 来检查它是否正常运行：

```
$ curl http://<servername>:8080/
testtable

$ curl http://<servername>:8080/version
rest 0.0.2 [JVM: Apple Inc. 1.6.0_24-19.1-b02-334] [OS: Mac OS X 10.6.7 \
x86_64] [Server: jetty/6.1.26] [Jersey: 1.4]
```

输入根路径的 URL，例如，`/` 可以返回当前可用表的表名列表，在这里是 `testtable`。使用 `/version` 可以检索当前 REST 服务器的版本，以及正在运行它的机器的详细信息。

如果需要停止 REST 服务器，可以执行与启动类似的命令，但需要将 `start` 替换为 `stop`：

```
$ bin/hbase-daemon.sh stop rest
stopping rest..
```

REST 服务器提供了 HBase 表提供的所有操作。



当前 REST 的在线文档地址是 <http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/rest/package-summary.html>。请参阅文档中提供的操作，并仔细阅读那个页面的 XML 模式，它解释了用户请求信息时需要使用的模式，以及服务返回的信息。

用户可以根据自己的喜好启动任意数量的 REST 服务，并且可以使用负载均衡器来路由它们。因为每台 REST 服务都是无状态的——任何所需状态都是请求的一部分，用户可以采用轮询或类似的算法来做负载均衡。

最后，使用 `-p` 或 `--port` 参数可以指定 REST 服务的监听端口，默认为 8080。

2. 支持的格式

通过使用 HTTP Content-Type 和 Accept 头，调用者可以自由选择发送和接收信息的数据格式。例如，用户可以使用类似如下的 Shell 脚本命令在 HBase 中创建表且插入列：

```
hbase(main):001:0> create 'testtable','colfam1'
0 row(s) in 1.1790 seconds

hbase(main):002:0> put 'testtable','\x01\x02\x03','colfam1:col1','value1'
```

① `curl` 是一个使用 URL 语法的命令行传输数据的工具，支持的协议繁多。详情见该项目网站 <http://curl.haxx.se/>。

```
0 row(s) in 0.0990 seconds
```

```
hbase(main):003:0> scan 'testtable'
```

```
ROW COLUMN+CELL
```

```
\x01\x02\x03 column=colfam1:col1,timestamp=1306140523371,value=value1
```

```
1 row(s) in 0.0730 seconds
```

插入一行使用了二进制的行键 0x01 0x02 0x03（十六进制数），其中指定一个列族中的一个列，值为 value1。

Plain (text/plain)。用户可以在一些操作中指定以文本格式返回数据。例如，前面提到的 /version 操作：

```
$ curl -H "Accept: text/plain" http://<servername>:8080/version
rest 0.0.2 [JVM: Apple Inc. 1.6.0_24-19.1-b02-334] [OS: Mac OS X 10.6.7 \
x86_64] [Server: jetty/6.1.26] [Jersey: 1.4]
```

另一方面，对于比较复杂的返回值，纯文本格式很难达到预期的结果。

```
$ curl -H "Accept: text/plain" \
  http://<servername>:8080/testtable/%01%02%03/colfam1:col1
<html> http://<servername>:8080/testtable/%01%02%03/colfam1:col1
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859- 1"/>
<title>Error 406 Not Acceptable</title>
</head>
<body><h2>HTTP ERROR 406</h2>
<p>Problem accessing /testtable/%01%02%03/colfam1:col1. Reason:
<pre>    Not Acceptable</pre></p><hr /><i><small>Powered by Jetty: //</small> </i><br/>
<br/>
...
</body>
</html>
```

这是因为实际上服务器本身并不能假设复杂的文本将转换成简单的文本，用户需要使用一种能够表达自身嵌套的格式。



使用前面提到的已创建的表，行键由长度为 3 的二进制字节数组组成，用户通过 REST 方式访问这行数据就需要依赖 *URL 编码*^①，解析后的数据为 %01%02%03。单元格查询的完整 URL 如下：

```
http://<servername>:8080/testtable/%01%02%03/colfam1:col1
```

详细语法可在线文档中查阅。

XML (text/xml)。默认的存储与查询格式是 XML。例如，如果没有指定 Accept 头，用户收到的信息如下：

① 基本思路是任何非安全的或不可打印的字符都用 %ASCII 来表示。因为它使用百分号为前缀，它也被称为百分比编码，有关详情用户可以查阅维基百科 <http://en.wikipedia.org/wiki/Percent-encoding>。

```
$ curl http://<servername>:8080/testtable/%01%02%03/colfam1:col1
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<CellSet>
  <Row key="AQID">
    <Cell timestamp="1306140523371" \
      column="Y29sZmFtMTpjb2wx">dmFsdWUx</Cell>
  </Row>
</CellSet>
```

默认的数据返回格式是 XML，列名与值都经过了 Base64^① 编码，详细解释见在线模式文档。以下是解释片段：

```
<complexType name="Row">
  <sequence>
    <element name="key" type="base64Binary"></element>
    <element name="cell" type="tns:Cell" maxOccurs="unbounded" \
      minOccurs="1"></element>
  </sequence>
</complexType>

<element name="Cell" type="tns:Cell"></element>

<complexType name="Cell">
  <sequence>
    <element name="value" maxOccurs="1" minOccurs="1">
      <simpleType><restriction base="base64Binary">
        </simpleType>
      </element>
    </sequence>
    <attribute name="column" type="base64Binary" />
    <attribute name="timestamp" type="int" />
  </complexType>
```

REST 服务器返回的值都经过了 base64Binary 编码，这些返回值都能包含在键或者值中，以二进制数据形式进行安全传输。



客户端向 REST 服务发送的数据也经过了安全编码。请确保阅读了在线帮助文档，并且正确地编码，包括网络传输内容，即实际的值、列名、行键等。

我们可以使用 base64 命令在控制台做一个快速测试：

```
$ echo AQID | base64 -d | hexdump
```

① 详情见维基百科关于 Base64 的页面 (<http://en.wikipedia.org/wiki/Base64>)。

```
0000000 01 02 03
```

```
$ echo Y29sZmFtMTpj2wx | base64 -d  
colfam1:coll
```

```
$ echo dmFsdWUx | base64 -d  
value11
```

这明显仅在命令行的验证中 useful，对于用户的代码来说，用户可以使用任意一个可用的 Base64 编码实现来解码返回值。

JSON(application/json)。JSON 是类似于 XML 的格式，发送请求时在头部设置 JSON 格式即可：

```
$ curl -H "Accept: application/json" \  
http://<servername>:8080/testtable/%01%02%03/colfam1:coll  
  
{  
  "Row": [{  
    "key": "AQID",  
    "Cell": [{  
      "timestamp": 1306140523371,  
      "column": "Y29sZmFtMTpj2wx",  
      "$": "dmFsdWUx"  
    }]  
  }]  
}
```



上面提到的 JSON 格式的结果经过重新格式化后更容易阅读，通常返回结果在控制台中独立显示的一行，例如：

```
{ "Row": [{ "key": "AQID", "Cell": [{ "timestamp": 1306140523371,  
  "column": "Y29sZmFtMTpj2wx", "$": "dmFsdWUx" } ] } ] }
```

编码之后的值类似于 XML，例如，Base64 可以编码任何包含二进制数据的值。不同之处在于，与 XML 格式相比，JSON 格式不包含无名字的数据字段。在 XML 格式中，一个单元格的标签是 Cell，JSON 格式指定了键/值对，因此没有可用的标签。由于这个原因，JSON 格式中有特殊的字段 “\$”（美元符号），美元符号对应的值就是 XML 格式中单元格标签对应的值。从上面的例子中，用户可以看到正在使用以下代码：

```
...  
"$": "dmFsdWUx"  
...
```

用户可以获取美元符号对应的字段值，从而得到经过 Base64 编码的值。

Protocol Buffer (application/x-protobuf)。一个非常有趣的 REST 应用是关于切换编码的。由于 Protocol Buffer 并不依赖本地 RPC 栈，因此 HBase REST 服务器提供了对这种编码格式的支持。用户可以详细查阅在线文档来了解具体模式。

获得以 Protocol Buffer 编码的返回结果需要匹配 Accept 头：

```
$ curl -H "Accept: application/x-protobuf" \
  http://<servername>:8080/testtable/%01%02%03/colfam1:col1 | hexdump -C
00000000 0a 24 0a 03 01 02 03 12 1d 12 0c 63 6f 6c 66 61 |. $. . . . . . . . colfa|
00000010 6d 31 3a 63 6f 6c 31 18 eb f6 aa e0 81 26 22 06 |ml:col1.....&".|
00000020 76 61 6c 75 65 31                                |value1|
```

使用 hexdump 可以打印编码后的二进制格式信息，不过用户需要使用 Protocol Buffer 解码器才能将其解析成结构化数据。在上面的例子中，ASCII 在右边打印出示例行的列名和单元格的值。

Raw Binary (application/octet-stream)。最后，用户可以按照原始形式转存数据，同时忽略结构化数据。通过下面的命令行，只有存储在单元格内的数据被返回。

```
$ curl -H "Accept: application/octet-stream" \
  http://<servername>:8080/testtable/%01%02%03/colfam1:col1 | hexdump -C
00000000 76 61 6c 75 65 31                                |value1|
```



根据格式要求，REST 服务器可以将结构化的数据插入自定义的头部。例如，按上述设置之后，头部结构如下（增加-D-到 curl 命令中）：

```
HTTP/1.1 200 OK
Content-Length: 6
X-Timestamp: 1306140523371
Content-Type: application/octet-stream
```

单元格中的时间戳已经移动到头部，例如，X-Timestamp。因为行键和列键是请求的 URI 中一部分，但它们在响应的时候不再返回，因此节省了不必要的网络传输。

3. REST 的 Java 客户端

REST 服务器同样有全面的 Java 客户端 API，位于 org.apache.hadoop.hbase.rest.client 包中。其中的核心类是 RemoteHTable 和 RemoteAdmin。例 6.1 展示了 RemoteHTable 类的使用实例。

例 6.1 REST 客户端类的使用实例

```
Cluster cluster = new Cluster();
cluster.add("localhost", 8080); ❶
```

```

Client client = new Client(cluster);❷

RemoteHTable table = new RemoteHTable(client,"testtable");❸

Get get = new Get(Bytes.toBytes("row-30"));❹
get.addColumn(Bytes.toBytes("colfam1"),Bytes.toBytes("col-3"));
Result result1 = table.get(get);

System.out.println("Get result1: " + result1);

Scan scan = new Scan();
scan.setStartRow(Bytes.toBytes("row-10"));
scan.setStopRow(Bytes.toBytes("row-15"));
scan.addColumn(Bytes.toBytes("colfam1"),Bytes.toBytes("col-5"));
ResultScanner scanner = table.getScanner(scan);❺

for(Result result2 : scanner){
    System.out.println("Scan row[" + Bytes.toString(result2.getRow())+
        "]: " + result2);
}

```

- ❶ 设置已知的 REST 服务器集群地址列表。
- ❷ 创建处理 HTTP 交互的客户端。
- ❸ 创建 RemoteHTable 实例，将 REST 访问封装到一个熟悉的接口中。
- ❹ 执行一个 get() 操作，如同直接连接 HBase 的操作。
- ❺ 扫描表，然后就像使用原生 Java API 一样进行调用。

上述的例子需要在本机中已经运行了 REST 服务器，并监听了特定端口。如果在其他机器或其他端口中已经运行了 REST 服务器，用户需要首先将服务地址新增到 Cluster 实例中。

以下是上述例子在运行过程中打印到控制台的信息：

```

Adding rows to table...
Get result1: keyvalues={row-30/colfam1:col-3/1306157569144/Put/vlen=8}
Scan row[row-10]: keyvalues={row-10/colfam1:col-5/1306157568822/Put/ vlen=8}
Scan row[row-100]: keyvalues={row-100/colfam1:col-5/1306157570225/Put/ vlen=9}
Scan row[row-11]: keyvalues={row-11/colfam1:col-5/1306157568841/Put/ vlen=8}
Scan row[row-12]: keyvalues={row-12/colfam1:col-5/1306157568857/Put/ vlen=8}
Scan row[row-13]: keyvalues={row-13/colfam1:col-5/1306157568875/Put/ vlen=8}
Scan row[row-14]: keyvalues={row-14/colfam1:col-5/1306157568890/Put/ vlen=8}

```

由于 HBase 是按照行键的字典序排序的，因此用户收到的行数据中包含了预期的列。

使用 RemoteHTable 与一定数量的 REST 服务器进行通信是一个非常便捷的方式，并且能够使用正常的 Java 客户端 API 类，如 Get 或 Scan。



当前 REST 的 Java 客户端实现在内部使用了用于和 REST 服务器进行通信的 Protocol Buffer 编码，这是最合适的网络传输协议，因此提供了较高的带宽利用率。

6.2.3 Thrift

Apache Thrift 是由 C++ 编写的框架，但是提供了跨语言的模式定义文件，包括 Java、C++、Perl、PHP、Python 和 Ruby 等。一旦用户编译了一种模式，用户就可以在一种或多种语言实现的系统之间传输消息。

1. 安装

用户应首先安装 Thrift，安装时最好使用适合当前操作系统的二进制安装包，如果没有就需要从源码重新编译。

从网站下载源码 tar 包，并解压到常用目录：

```
$ wget http://www.apache.org/dist/thrift/0.6.0/thrift-0.6.0.tar.gz
$ tar -xzvf thrift-0.6.0.tar.gz -C /opt
$ rm thrift-0.6.0.tar.gz
```

首先，用户需要安装依赖的库，如 Automake、LibTool、Flex、Bison 以及 Boost 库：

```
$ sudo apt-get install build-essential automake libtool flex bison libboost
```

之后就可以编译并安装 Thrift：

```
$ cd /opt/thrift-0.6.0
$ ./configure
$ make
$ sudo make install
```

通过调用 thrift 命令可以验证安装是否已经成功，例如：

```
$ thrift -version
Thrift version 0.6.0
```

一旦安装好 Thrift，你就可以编译模式文件，并生成用户指定语言的 RPC 代码。HBase 提供了客户端 API 与管理 API 依赖的模式文件。用户可以使用 Thrift 二进制文件为自己的开发环境创建包。



已提供的模式文件公开了大量的 API 功能，但是在某些领域还有欠缺。当 HBase 拥有不同的 API 时，其模式文件就已经创建好了，当用户需要使用其时，可以很容易地找到。

例如，API 中的不同之处在于，正在使用的是 `mutateRow()` 方法，而新 API 使用的是 `get()` 方法。

这个工作在 HBASE-1744 (<http://issues.apache.org/jira/browse/HBASE-1744>) 中已完成，并且将 Thrift 模式文件转移到了现在的 API 中。一旦完成，它会被增加到 `thrift2` 包中，这样用户就能在迁移到新模式时维护并使用现有的旧模式代码。

在使用 Thrift 访问 HBase 之前，用户也需要启动提供的 ThriftServer。

2. 操作

启动 Thrift 服务器是通过已提供的脚本完成的。用户在使用命令后添加 `-h` 选项可以得到帮助信息或者省略所有的选项：

```
$ bin/hbase thrift
usage: Thrift [-b <arg>] [-c] [-f] [-h] [-hsha | -nonblocking |
             -threadpool] [-p <arg>]
-b,--bind <arg>  Address to bind the Thrift server to. Not supported by
                  the Nonblocking and HsHa server [default: 0.0.0.0]
-c,--compact      Use the compact protocol
-f,--framed       Use framed transport
-h,--help         Print help information
-hsha            Use the THsHaServer. This implies the framed transport.
-nonblocking      Use the TNonblockingServer. This implies the framed
                  transport.
-p,--port <arg>  Port to bind to [default: 9090]
-threadpool       Use the TThreadPoolServer. This is the default.
To start the Thrift server run 'bin/hbase-daemon.sh start thrift'
To shutdown the thrift server run 'bin/hbase-daemon.sh stop thrift' or
send a kill signal to the thrift server pid
```

脚本提供了非常多的选项。`server`、`protocol` 和 `transport` 类型是被客户端强制使用的，不过并非所有的语言都支持这些选项。通过命令行的帮助信息可以看到以上选项的用例，例如，使用非阻塞（non-blocking）服务器来响应框架传输（framed transport）。

用户可以使用默认参数提供的非守护模式启动 Thrift 服务：

```
$ bin/hbase thrift start
^C
```

用户可以使用 `Ctrl+C` 组合键退出进程。帮助信息提供了如何通过后台进程启动 Thrift 服务：

```
$ bin/hbase-daemon.sh start thrift
```

```
starting thrift,logging to /var/lib/hbase/logs/hbase-larsgeorge-thrift-
<servername>.out.
```

使用如下命令可以停止 Thrift 服务，作为守护进程运行，包含相同的脚本，仅仅只需把 start 换成 stop：

```
$ bin/hbase-daemon.sh stop thrift
stopping thrift..
```

Thrift 服务提供了所有 HTable 可以提供的操作。



有关 Thrift 服务的文档在 <http://wiki.apache.org/hadoop/Hbase/ThriftApi> 中。用户可以在文档中查阅所有相关操作，也可以通过阅读模式定义文件 `$HBASE_HOME/src/main/resources/org/apache/hadoop/hbase/thrift/Hbase.thrift` 来确认提供的所有可用操作。

用户可以启动多台 Thrift 服务器，并可以通过负载均衡算法来均摊客户端请求，每台 Thrift 服务器都是无状态的，例如，用户可以采用轮询或类似的算法来分摊负载。

最后 `-p` 或 `--port` 参数可以绑定服务器监听端口，默认端口为 9090。

3. 示例：PHP

HBase 不仅需要 Thrift 模式文件，而且还需要多编程语言的客户端代码。下面我们使用 PHP 实现来演示所需的步骤。



用户首先需要按照服务器文档的步骤启动 Web 服务的 PHP 支持。

第一步，复制模式文件并编译成必备的 PHP 源代码。

```
$ cp -r $HBASE_HOME/src/main/resources/org/apache/hadoop/hbase/thrift ~/thrift_src
$ cd thrift_src/
$ thrift -gen php Hbase.thrift
```

以上 thrift 命令执行不能有任何错误，最后在 `thrift_src` 目录中能够找到名字为 `gen-php` 的目录，其中包含了两个自动生成的并能够访问 HBase 的 PHP 文件。

```
$ ls -l gen-php/Hbase/
total 616
-rw-r--r-- 1 larsgeorge staff 285433 May 24 10:08 Hbase.php
-rw-r--r-- 1 larsgeorge staff 27426 May 24 10:08 Hbase_types.php
```

这些自动生成文件依赖于 Thrift 提供的 PHP 支持库，用户需要把这些库以及自动生成文件复制到 Web 服务器的文档根目录（document root）中：

```
$ cd /opt/thrift-0.6.0
```

```
$ sudo cp lib/php/src $DOCUMENT_ROOT/thrift
$ sudo mkdir $DOCUMENT_ROOT/thrift/packages
$ sudo cp -r ~/thrift_src/gen-php/Hbase $DOCUMENT_ROOT/thrift/packages/
```

自动生成的 PHP 文件复制到了子目录 *packages* 中，类似于前面提到的 Thrift 库，如果该目录不存在就必须创建该目录。



上述 `$DOCUMENT_ROOT` 目录可能会在 `/var/www` 下，例如，在 Linux 系统部署 Apache 服务的目录中，或 Apple Mac OS 10.6 的 `/Library/WebServer/Documents/` 目录中。具体可检查 Web 服务的配置。

HBase 提供了一个 *DemoClient.php* 文件，这个文件可以通过自动生成的文件与服务器端通信。同样，这个文件也会被复制到与 Web 服务相同的根目录中：

```
$ sudo cp $HBASE_HOME/src/examples/thrift/DemoClient.php $DOCUMENT_ROOT/
```

用户在使用前需要编辑 *DemoClient.php* 文件，并调整文件的开头部分：

```
# Change this to match your thrift root
$GLOBALS['THRIFT_ROOT'] = 'thrift';
...
# According to the thrift documentation, compiled PHP thrift libraries should
# reside under the THRIFT_ROOT/packages directory. If these compiled libraries
# are not present in this directory, move them there from gen-php/.
require_once($GLOBALS['THRIFT_ROOT'].'/packages/Hbase/Hbase.php');
...
$socket = new TSocket('localhost', 9090);
...
```

通常编辑第一行即可设置 `THRIFT_ROOT` 路径，由于 *DemoClient.php* 文件仍在默认的根本目录中，用户此时可以为 Thrift 设置变量，即之前已经将 Thrift 源码复制到了该目录中。

上述例子的最后一行硬编码了服务器地址和端口，如果用户想在一个分布式环境中调试这个例子，其需要调整这一行的参数。

当一切都准备好了，用户就可以在浏览器中输入以下地址进行访问：

```
http://<webserver-address>/DemoClient.php
```

浏览器页面中显示的信息如下：

```
scanning tables...
  found: testtable
creating table: demo_table
column families in demo_table:
  column: entry:,maxVer: 10
  column: unused:,maxVer: 3
Starting scanner...
...
```

类似的客户端，如 C++、Java、Perl、Python 和 Ruby，都需要按照 **PHP** 例子中描述的步骤进行设置：启动 Thrift 服务器，编译模式文件，生成所需语言的客户端 RPC 代码，最后启动客户端。用户必须将根据目标语言生成的 RPC 代码部署到客户端找到的位置。

HBase 目前已经提供了用于与 Thrift 服务通信的 Java 的 RPC 生成代码，用户也可以根据模式文件重新自行生成这部分代码，但为了便于使用它们已经被集成了。

6.2.4 Avro

Apache Avro，类似于 Thrift，提供了针对多种编程语言的模式定义文件，例如 Java、C++、PHP、Python 和 Ruby 等。一旦用户编译了预定义模式文件，就可以在异构系统中进行跨语言通信。

1. 安装

用户在使用 Avro 之前需要先安装它，安装时最好使用适合当前操作系统的二进制安装包，如果没有就需要从源码中重新编译它。

一旦安装好 Avro，用户就需要根据选择的编程语言编译模式预定义文件以生成指定语言的 RPC 代码。HBase 提供了客户端 API 以及管理 API 的模式文件，用户需要使用 Avro 工具来创建支持当前开发环境的封装。

当然，用户在使用 Avro 服务器前，必须要先启动已提供的 AvroServer。

2. 操作

用户可以通过提供的脚本来启动 Avro 服务器。用户可以添加 -h 选项或者省略全部选项以获得命令行的帮助信息：

```
$ bin/hbase avro
Usage: java org.apache.hadoop.hbase.avro.AvroServer --help | [--port=PORT] start
Arguments:
  start Start Avro server
  stop Stop Avro server
Options:
  port Port to listen on. Default: 9090
  help Print this message and exit
```

用户可以使用默认参数提供的非守护模式启动 Avro 服务器：

```
$ bin/hbase avro start
^C
```

用户按下 Ctrl+C 组合键可以退出进程。HBase 也提供了通过后台进程的形式启动 Avro 服务器的脚本：

```
$ bin/hbase-daemon.sh start avro
starting avro, logging to /var/lib/hbase/logs/hbase-larsgeorge-avro-<servername>.out
```

用户可以使用如下命令停止 Avro 服务器，使用的是相同的脚本，只是把 start 换成了 stop:

```
$ bin/hbase-daemon.sh stop avro
stopping avro..
```

Avro 服务器提供了所有 HTable 可以提供的操作。



有关 Avro 服务的文档请访问 <http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/avro/package-summary.html>。用户可以从文档内查阅所有相关操作，也可以通过阅读模式定义文件 `$HBASE_HOME/src/main/java/org/apache/hadoop/hbase/avro/hbase.avpr` 来确认提供的所有可用操作。

用户可以启动多台 Avro 服务器，并可以通过负载均衡算法来均摊客户端请求，每台 Avro 服务器都是无状态的，例如，可以采用轮询或类似的算法。

最后，`-p` 或 `--port` 参数可以绑定服务器监听端口，默认为 9090。

6.2.5 其他客户端

HBase 还提供了其他若干客户端来访问 HBase 集群。大致上可以分为使用 JVM 直接访问和使用网关服务与 HBase 集群通信两种情况。以下是一些例子。

JRuby

HBase Shell 是个使用基于 JVM 语言访问 Java API 的最佳例子，它的源码非常全面，因此用户可以使用它添加相同的功能到用户自己的 JRuby 代码中。

HBql

HBql 在 HBase 基础上提供了 SQL 语法访问，如果要拓展新功能则需要 HBase 添加对应功能，详情见 HBql 项目官方网址 <http://www.hbql.com/>。

HBase-DSL

这个项目提供了特定的类，该类可帮助格式化查询 HBase 集群。项目风格是类似于 builder 模式的代码，用户可以快速组合需要的参数和选项，详情见项目官方 wiki (<https://github.com/nearinfinity/hbase-dsl/wiki>)。

JPA/JPO

例如，用户可以使用 DataNucleus (<http://www.datanucleus.org>) 在 HBase 之上封装一层 JPA/JPO。

PyHBase

PyHBase 项目 (<https://github.com/hammer/pyhbase/>) 提供了与 Avro 网关服务通信的 HBase 客户端。

AsyncHBase 提供了完全异步、非阻塞、线程安全的客户端来访问 HBase 集群。它使用本地 RPC 协议直接和许多服务器直接通信,详情见这个工程的官方主页 <https://github.com/stumbleupon/asynchbase>。



需要注意的是,在以上提到的工程中,有的已经很久没有更新。很多作者初衷只是满足自己的个人需要才将其开源。用户最好以这些工程为基础进行二次研发后使用。

6.3 批处理客户端

另外一种客户端的交互使用场景是批量访问数据。不同之处是这些批量处理通常是异步运行在后台,需要扫描大量的数据,例如,扫描索引、基于数学模型的机器学习或报表统计需求。

这些处理案例大多数不是用户直接驱动,而且所需要的运行时间非常漫长,因此用户不会特别关注单次访问的延时。大多数处理批量读写 HBase 的框架是基于 MapReduce 的模式。

6.3.1 MapReduce

Hadoop MapReduce 框架的目标是处理 PB 级的数据,具有高可用、目标明确、编程模型简单易用等特点。MapReduce 提供了多种以 HBase 为数据源或目标数据库执行的方法。

1. 原生 Java

关于基于 MapReduce Java API 的访问的详细内容在第 7 章中讨论。

2. Clojure

HBase-Runner 项目 (<https://github.com/mudphone/hbase-runner/>) 为功能性编程语言 Clojure 访问 Hbase 提供了支持。用户可以直接使用 Clojure 编写 MapReduce 程序来获取 HBase 表。

6.3.2 Hive

Apache Hive^①项目提供了基于 Hadoop 的数据仓库。Hive 最早由 Facebook 开发,现在是 Hadoop 生态圈的开源项目。

Hive 提供了类似于 SQL 的处理语言,叫 HiveQL,允许用户查询存储在 Hadoop 中的半结

① <http://hive.apache.org/>。

构化数据。最终查询会转化成 MapReduce 作业，在本地执行或在分布式的 MapReduce 集群中执行。数据在作业执行的时候被解析，并且 Hive 提供了一个不仅可以访问 HDFS 的数据还可以访问其他数据源的存储处理（storage handler）^①层。存储层的数据获取对用户查询来说是透明的。

Hive 0.6.0 及之后的版本提供了对 HBase^②的支持。用户可以直接定义将 Hive 表存储为 HBase 表，并按需要映射列值，在需要的时候行键可以作为独立的一列。

HBase 版本支持

此外，Hive 0.7.0 仅仅支持 HBase 0.89.0-SNAPSHOT 版本，但很快就可以支持 HBase 的更高版本。言外之意是，两者之间的版本需要匹配，细微的 RPC 变化都可能影响到通信交互。

当前唯一的办法是用新的 HBase JAR 包覆盖旧的 JAR 包，并重新编译 Hive 代码。用户既可以更新 Ivy 设置中的 HBase 版本（包括 Hadoop），然后复制新的 HBase JAR 包到 \$HIVE_HOME/src/build/dist/lib 目录中，并重新编译（YMMV）。

更好的方式就是设置 Ivy 来加载远程资源，然后正常编译 Hive。首先要先从网站中下载 Hive 安装包，并解压到指定路径：

```
$ wget http://www.apache.org/dist/hive/hive-0.7.0/hive-0.7.0.tar.gz
$ tar -xvzf hive-0.7.0.tar.gz -C /opt
```

然后编辑 Ivy 配置文件：

```
$ cd /opt/hive-0.7.0/src
$ vim ivy/libraries.properties
...

#hbase.version=0.89.0-SNAPSHOT
#hbase-test.version=0.89.0-SNAPSHOT
hbase.version=0.91.0-SNAPSHOT
hbase-test.version=0.91.0-SNAPSHOT
...
```

用户可以到工程中执行 ant，在此之前必须为即将编译的 Hadoop 版本设置环境变量：

```
$ export HADOOP_HOME="/<your-path>/hadoop-0.20.2"
$ ant package
Buildfile: /opt/hive-0.7.0/src/build.xml
```

① 见 Hive wiki 页面 <http://wiki.apache.org/hadoop/Hive/StorageHandlers>，查看有关存储处理器的更多细节。

② Hive wiki 页面（<http://wiki.apache.org/hadoop/Hive/HBaseIntegration>）完整地解释了 HBase 如何与 Hive 集成。

```

jar:

create-dirs:

compile-ant-tasks:

...

package:
    [echo] Deploying Hive jars to /opt/hive-0.7.0/src/build/dist

BUILD SUCCESSFUL

```

编译过程需要一段时间，因为 Ivy 需要下载所有依赖的库，下载速度取决于用户的网速。一旦编译完成，用户就可以开始使用新版本的 HBase 处理数据。

在某些情况下，用户需要编辑目录 *src/hbase-handler/src/java/org/apache/hadoop/hive/hbase/* 下的所有代码文件，并使用如下的方式修改：

```
HBaseConfiguration hbaseConf = new HBaseConfiguration(hiveConf);
```

新方式使用了静态工厂方法：

```
Configuration hbaseConf = HBaseConfiguration.create(hiveConf);
```

安装完 Hive 之后，用户需要编辑配置文件使 Hive 能够访问 HBase 的 JAR 文件，以及修改附带的配置。修改 *\$HIVE_HOME/conf/hive-env.sh* 文件，要修改的行如下所示：

```

# Set HADOOP_HOME to point to a specific hadoop install directory
HADOOP_HOME=/usr/local/hadoop
HBASE_HOME=/usr/local/hbase

# Hive Configuration Directory can be controlled by:
# export HIVE_CONF_DIR=
export HIVE_CLASSPATH=/etc/hbase/conf

# Folder containing extra libraries required for hive compilation/execution
# can be controlled by:
export HIVE_AUX_JARS_PATH=/usr/local/hbase/hbase-0.91.0-SNAPSHOT.jar

```



用户需要复制系统提供的 *\$HIVE_HOME/conf/hive-env.sh.template* 文件，并保存到相同的目录中，但是需要去掉后缀 *.template*，复制完成后用户可以按照前面描述的情况编辑该文件。

Hive 安装好后就可以使用新的存储处理器。首先，启动 Hive 命令行，创建一张 Hive 本地表，并使用已提供的示例文件插入数据：

```
$ build/dist/bin/hive
Hive history file=/tmp/larsgeorge/hive_job_log_larsgeorge_201105251455_2009910117.txt
hive> CREATE TABLE pokes(foo INT,bar STRING);
OK
Time taken: 3.381 seconds

hive> LOAD DATA LOCAL INPATH '/opt/hive-0.7.0/examples/files/kv1.txt'
OVERWRITE INTO TABLE pokes;
Copying data from file:/opt/hive-0.7.0/examples/files/kv1.txt
Copying file: file:/opt/hive-0.7.0/examples/files/kv1.txt
Loading data to table default.pokes
Deleted file:/user/hive/warehouse/pokes
OK
Time taken: 0.266 seconds
```

这里使用了表 pokes，表结构见向导 <http://wiki.apache.org/hadoop/Hive/GettingStarted> 里的描述。然后用户可以创建一张如下的表：

```
hive> CREATE TABLE hbase_table_1(key int,value string)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES("hbase.columns.mapping" = ":key,cf1:val")
TBLPROPERTIES("hbase.table.name" = "hbase_table_1");
OK
Time taken: 0.144 seconds
```

上述的 DDL 语句描述了使用 TBLPROPERTIES、SERDEPROPERTIES 和 HBase 处理器来处理 Hive 的表 hbase_table_1。hbase.columns.mapping 参数提供了特殊的功能，使用 “:key” 映射行键，用户可以把映射行键的特殊列放置在任意位置。这里我们将其放置在了第一列，恰好对应到了 Hive 表中的键这一列以及 HBase 表中行键这一列。

hbase.table.name 这个属性是可选的，仅当用户想在 Hive 和 HBase 中使用不同名字的书表名时才需要填写。在这里，它被设置为相同的名字，因此可以被省略。

接下来要加载先前填充过的 Hive 表 pokes。根据映射关系，这里会将 pokes.foo 的值保存在行键中，将 pokes.bar 的值保存在 cf1:val1 列中：

```
hive> INSERT OVERWRITE TABLE hbase_table_1 SELECT * FROM pokes;
Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks is set to 0 since there's no reduce operator
Execution log at: /tmp/larsgeorge/larsgeorge_20110525152020_de5f67d1-9411-446f-99bb-35621e1b259d.log
```

```

Job running in-process(local Hadoop)
2011-05-25 15:20:31,031 null map = 100%,reduce = 0%
Ended Job = job_local_0001
OK
Time taken: 3.925 seconds

```

这里启动了示例中第一个 MapReduce 作业。用户通过上面的例子可以看到 Hive 在命令行中打印了哪些使用值。这个作业复制内部 Hive 表中的数据，并导入到 HBase 表中。



在某些特殊设置下，尤其是在本地模式、伪分布式模式下，可能会发生 Hive 作业失败的情况，并且异常信息很隐蔽。在分析详细信息前需要使用本地 MapReduce 模式运行 Hive。在 Hive CLI 中输入：

```
hive> SET mapred.job.tracker=local;
```

然后再执行一遍 Hive 作业。这种工作模式已经增加到了 Hive 0.7.0 中，但对于用户来说可能并不能直接使用。如果用户想尝试使用它，这个模式避免了使用 Hadoop MapReduce 框架——当调试 Hive 作业时可以减少我们的部分担心。

下面的使用场景统计了表 pokes 和表 hbase_table_1 的行数（一些 CLI 输出细节已经被省略）：

```

hive> select count(*) from pokes;
Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer(in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapred.reduce.tasks=<number>
Execution log at: /tmp/larsgeorge/larsgeorge_20110525152323_418769e6-1716-\48ee-a0ab-dacd59e55da8.log
Job running in-process(local Hadoop)
2011-05-25 15:23:07,058 null map = 100%,reduce = 100%
Ended Job = job_local_0001
OK
500
Time taken: 3.627 seconds

hive> select count(*) from hbase_table_1;
...
OK
309

```

```
Time taken: 4.542 seconds
```

有趣的是，两者之间的统计结果有差异，差异超过 100 行，这意味着以 HBase 为存储表的数据较少。原因究竟是什么？由于 HBase 中行键是唯一的，因此重复的行键已经相互覆盖，而 pokes.foo 这一列存在着相同的重复值。这和在原表中使用 SELECT DISTINCT 是一样的：

```
hive> select count(distinct foo) from pokes;
...
OK
309
Time taken: 3.525 seconds
```

上述执行结果最终比对正确，这证明了我们的判断是正确的。

最后我们删除这两张表，同时也移除了 Hive 表对应的实体 HBase 表：

```
hive> drop table pokes;
OK
Time taken: 0.741 seconds

hive> drop table hbase_table_1;
OK
Time taken: 3.132 seconds

hive> exit;
```

用户也可以将一张 Hive 表与已有 HBase 表关联，或者将多张 Hive 表与已有 HBase 表关联。当 HBase 表中有不同列族时这种方式比较有用，可以起到分离查询的作用。由于进行 Scan 操作时可以只扫描需要的列族，因此能够显著提升查询性能。用户设置列族可以尽可能扫描最小的磁盘文件，而不用扫描全部数据然后过滤出相应数据。

使用 Hive EXTERNAL 关键字可以映射已有的表，其同时也被其他地方用来获取未经 Hive 管理的 Hive 表中的数据，这种模式可以帮助实际存储不受 Hive 的控制：

```
hive> CREATE EXTERNAL TABLE hbase_table_2(key int,value string)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES("hbase.columns.mapping" = ":key,cf1:val")
TBLPROPERTIES("hbase.table.name" = "<existing-table-name>");
```

在 Hive 中删除外部表时是不会删除实际数据的，这个过程仅仅删除了该表的元数据。此外，用户还有其他选项可以使 HBase 的列直接映射到 Hive 的列，或者可以将整个列族映射到 Hive 的 MAP 类型。当用户事先并不知道具体的表结构时，这种方式非常有用，在 Hive 查询中映射列族，并且在查询时自动遍历所有列。



没有映射到 Hive 中的 HBase 列在 Hive 中是访问不到的。

由于存储层对 Hive 的上层来说是透明的, 用户还可以使用 Hive 提供的自定义函数(user defined function, *UDF*) —— 用户自己定义的函数。

当前版本中存在如下缺点。

无自定义序列化

HBase 当前存储格式为 `byte[]` 数组, 因此 Hive 需要将每一列都转化成为 `String`, 并从中进行序列化。例如, 在 Hive 中, `INT` 列被设置为 12, 即需要使用 `Bytes.toBytes("12")` 这种方式存储。

无版本控制

目前在处理 HBase 的表时, 没有任何有关版本细节的控制。Hive 总是返回最新版本数据。

最后, 用户在使用 Hive 前, 需要检查上述功能是否已经添加。

6.3.3 Pig

Apache Pig^① 项目提供了一个分析海量数据的平台。它提供了自己的查询语言, 叫做 Pig Latin, Pig 语法使用了命令式编程风格, 迭代式执行, 并最终将输入转化为输出。Pig 语言的编程风格与 Hive 模拟 SQL 的实现方式的风格截然相反。

但 Pig Latin 与 HiveQL 相比更适合有编程经验的人使用, 但本身的结构使得 Pig 更适合并行处理。用户将 Hadoop 与 MapReduce 框架结合起来使用时, 可以在一个可接受的时间范围内处理海量数据。

Pig 0.7.0 版本介绍了 `LoadFunc/StoreFunc` 类和相关功能, 这个功能使得 Pig 可以读取不同数据源的数据, 而不仅仅是 HDFS 中的数据。例如, 在 `HBaseStorage` 类中实现的 HBase 数据源。

Pig 支持 HBase 表的读写, 用户可以映射 HBase 表中的列到 Pig 元组 (tuple), 并且可以选择行键作为第一列读取。写入的时候第一个字段通常当做行键写入。

存储层也支持基本的过滤, 它在行级别上工作并且提供了比较操作, 详情见 4.1.1 节的“比较运算符”。^②

① <http://pig.apache.org/>。

② 内部使用 `RowFilter` 类, 详情见 4.1.2 节的 `RowFilter`。

Pig 安装

用户可以根据选择的操作系统直接安装已经编译好的二进制程序,如果没有合适的安装程序也可以下载工程源码并重新编译到本地。例如,在 Linux 系统中需要按照如下步骤编译。^①

从官方网站下载源码,然后解压到指定目录:

```
$ wget http://www.apache.org/dist/pig/pig-0.8.1/pig-0.8.1.tar.gz
$ tar -xzf pig-0.8.1.tar.gz -C /opt
$ rm pig-0.8.1.tar.gz
```

将 *pig* 脚本添加到环境变量中,将 `PIG_HOME` 环境变量设置如下:

```
$ export PATH=/opt/pig-0.8.1/bin:$PATH
$ export PIG_HOME=/opt/pig-0.8.1
```

最后,用户可以通过以下命令检查安装是否成功:

```
$ pig -version
Apache Pig version 0.8.1
compiled May 27 2011,14:58:51
```

用户可以按照教程中提供的代码和数据去实践通过 Pig 访问 HBase。在使用 Pig 之前,用户要通过 HBase Shell 创建表:

```
hbase(main):001:0> create 'excite','colfam1'
```

启动 Pig Shell 脚本,巧妙地调用 *pig* 脚本 *Grunt*,如果用户想使用本地测试,则需要使用 `-x local` 命令:

```
$ pig -x local
grunt>
```

本地模式意味着 Pig 并不运行在 MapReduce 模式中,而是使用 Hadoop 的一个组件 LocalJobRunner 运行,并模拟 MapReduce 程序将作业都运行在同一个进程中。这种模式有利于测试和原型设计阶段,但如果数据量较大时,这种模式则不适合。

用户可以通过编辑器提前编写脚本,然后通过 *pig* 脚本执行。用户也可以在 Pig Shell 中敲入 Pig Latin 语句。最终, Pig 语句会转化成一个或多个 MapReduce 作业,但并非所有的语句都能触发执行,因此用户最好逐行定义这些语句,然后调用 DUMP 或 STORE,使执行能够按照预定逻辑过程执行。



Pig Latin 函数区分大小写,虽然通常它们都是大写。用户定义的名字与字段就像 Pig Latin 函数一样对大小写敏感。

① 在 Pig 安装页面 <http://pig.apache.org/docs/r0.8.0/setup.html> 中可查看完整信息。

Pig 教程使用了由 Excite 提供的少量的数据集用于测试, 包含了匿名用户 ID、时间戳和用于用户搜索的索引标题。第一步, 用户需要将这些数据加载到 HBase 中, 并经过简单的组合转化, 以强制确保每个条目行键的唯一性:

```
grunt> raw = LOAD 'tutorial/data/excite-small.log' \
USING PigStorage('\t')AS(user,time,query);
T = FOREACH raw GENERATE CONCAT(CONCAT(user,'\u0000'),time),query;
grunt> STORE T INTO 'excite' USING \
org.apache.pig.backend.hadoop.hbase.HBaseStorage('colfam1:query');
...
2011-05-27 22:55:29,717 [main] INFO org.apache.pig.backend.hadoop. \
executionengine.mapReduceLayer.MapReduceLauncher - 100% complete
2011-05-27 22:55:29,717 [main] INFO org.apache.pig.tools.pigstats. PigStats \
- Detected Local mode. Stats reported below may be incomplete
2011-05-27 22:55:29,718 [main] INFO org.apache.pig.tools.pigstats. PigStats \
- Script Statistics:

HadoopVersion PigVersion UserId StartedAt FinishedAt Features
0.20.20.8.1 larsgeorge 2011-05-27 22:55:22 2011-05-27 22:55:29 UNKNOWN

Success!

Job Stats(time in seconds):
JobId Alias Feature Outputs
job_local_0002 T,raw MAP_ONLY excite,

Input(s):
Successfully read records from: "file:///opt/pig-0.8.1/tutorial/data/ excite-small.log"

Output(s):
Successfully stored records in: "excite"

Job DAG:
job_local_0002
```



用户可以使用 `DEFINE` 语句创建 `HbaseStorage` 类的 Java 包名的短引用。
例如:

```
grunt> DEFINE LoadHBaseUser org.apache.pig.backend.hadoop.hbase.HBaseStorage(\
'data:roles', '-loadKey');
grunt> U = LOAD 'user' USING LoadHBaseUser;
grunt> DUMP U;
...
```

这对复用特定的功能函数比较有用。

上述的 `STORE` 语句开启了一个 `MapReduce` 作业, 该作业从预先给出的日志文件中读取

数据并加载到 HBase 表中。上述例子改变了组合行键之间的关系——在前面的 STORE 语句中指定了第一列用于存储行键——行键由用户和时间戳两列的值组合而成，之间由字节 0 分隔开。

处理数据需要另外的 LOAD 语句，这次需要使用 HBaseStorage 类：

```
grunt> R = LOAD 'excite' USING \
  org.apache.pig.backend.hadoop.hbase.HBaseStorage('colfam1:query','-loadKey')\
  AS(key: chararray,query: chararray);
```

括号里的参数定义了字段到列的映射信息，以及其他相关的额外参数，这些参数定义了把行键作为首列载入。AS 部分显式地定义了行键和 colfam1:query 列在访问时会被转化为 Pig 的字符串类型 chararray。默认它们以 bytearray 类型返回，这个和它们存储在 HBase 表中的方式相同。转换数据类型是被允许的，例如，随后拆分的行键。

用户可以通过转储内容 R 来测试先前语句的结果。

```
grunt> DUMP R;
...
Success!
...
(002BB5A52580A8ED970916150445,margaret laurence the stone angel)
(002BB5A52580A8ED970916150505,margaret laurence the stone angel)
...
```

该元组的第一列存放着行键，行键在首次从文件复制到 HBase 的过程中被创建。现在行键可以被拆分成两列，这样原始的文本文件将被重新创建：

```
grunt> S = foreach R generate FLATTEN(STRSPLIT(key,'\u0000',2))AS \
  (user: chararray,time: long),query;
grunt> DESCRIBE S;
S: {user: chararray,time: long,query: chararray}
```

再次使用 DUMP 显示最终结果：

```
grunt> DUMP S;
(002BB5A52580A8ED,970916150445,margaret laurence the stone angel)
(002BB5A52580A8ED,970916150505,margaret laurence the stone angel)
...
```

用户可以使用之前的代码替换 LOAD 和 STORE 语句，然后按照剩余的 Pig 教程尝试处理。

最后，输入 QUIT 命令可以退出 Grunt Shell：

```
grunt> QUIT;
$
```

当前版本的 Pig 对 HBase 支持尚有一些缺点，如下所示。

无版本支持

目前在处理 HBase 的表时，没有任何版本细节控制。Pig 总是返回最近一个版本的数据。

固定的列映射

行键只能是第一列，且不能被其他列取代。但是，这个缺陷可以通过 FOREACH...GENERATE 语句克服，进行重新布局。

最后，用户在使用 Pig 前，需要检查是否已经添加上述功能。

6.3.4 Cascading

Cascading 是 MapReduce 的替代 API，实际上它使用 MapReduce 执行作业，但用户在开发时不必以 MapReduce 的模式来考虑它的执行。

该模型类似现实世界的管道装置，数据来源是水龙头（tap），输出是汇总（sink）。这些管道一起形成了处理流程，数据传输通过管道并在这个过程中进行转换。管道可以连接到更大的管道组件，形成更复杂的处理流程。

数据流（stream）管道，可以拆分、合并、分组或汇总，数据被表示为元组（tuple）并形成了一个元组流（tuple stream）。这种面向可视化的模型使得开发 MapReduce 的任务更像是构造工作，并且降低了实际工作的复杂度。

Cascading（1.0.1 版本）支持从 HBase 集群读写数据。更多细节和源码可在模块页面 <http://www.cascading.org/modules.html> 中进行查阅。

例 6.2 展示了汇总数据到 HBase 集群的流程。从模块页面可以链接到 GitHub 仓库，点击可查看更详细的 API。

例 6.2 使用 Cascading 向 HBase 插入数据

```
// read data from the default filesystem
// emits two fields: "offset" and "line"
Tap source = new Hfs(new TextLine(),inputFileLhs);

// store data in a HBase cluster, accepts fields "num","lower",and "upper"
// will automatically scope incoming fields to their proper familyname,
// "left" or "right"
Fields keyFields = new Fields("num");
String[] familyNames = {"left","right"};
Fields[] valueFields = new Fields[] {new Fields("lower"),
    new Fields("upper")};
Tap hBaseTap = new HBaseTap("multitable",new HBaseScheme(keyFields,
```



```

    familyNames,valueFields),SinkMode.REPLACE);

// a simple pipe assembly to parse the input into fields
// a real app would likely chain multiple Pipes together for more complex
// processing
Pipe parsePipe = new Each("insert",new Fields("line"),
    new RegexSplitter(new Fields("num","lower","upper")," "));

// "plan" a cluster executable Flow
// this connects the source Tap and hBaseTap(the sink Tap)to the parsePipe
Flow parseFlow = new FlowConnector(properties).connect(source,hBaseTap,
    parsePipe);

// start the flow,and block until complete
parseFlow.complete();

// open an iterator on the HBase table we stuffed data into
TupleEntryIterator iterator = parseFlow.openSink();

while(iterator.hasNext()){
    // print out each tuple from HBase
    System.out.println("iterator.next() = " + iterator.next());
}

iterator.close();

```

与 Hive 和 Pig 不同的是, Cascading 提供了 Java API, 而不提供特定领域语言 (domain specific languages, DSL) 的封装和访问。在 Cascading 这个项目之上还有一些其他的开源项目提供了 DSL。

6.4 Shell

HBase Shell 是 HBase 集群的命令行接口。用户可以使用 Shell 访问本地或远程服务器并与其进行交互, Shell 同时提供了客户端和管理功能的操作, 这些细节我们在本书的前面几章中已经介绍过了。

6.4.1 基础

Shell 试验的第一步是启动 Shell:

```

$ $HBASE_HOME/bin/hbase shell
HBase Shell;enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.91.0-SNAPSHOT,r1127782,Thu May 26 10:28:47 CEST 2011

```

```
hbase(main):001:0>
```

HBase Shell 是基于 Jruby 的, JRuby 是基于 Ruby 实现的 Java 虚拟机。^①更确切地说, 它使用的是交互式 Ruby Shell (Interactive Ruby Shell, IRB), 输入命令并快速得到响应。HBase 使用 Java 基本的 API 拓展了 Ruby 脚本, 并且继承了对历史记录和实现的内置支持, 以及所有的 Ruby 命令。



Ruby 并不需要刻意安装, HBase 提供了 JRuby Shell 运行所需要的 JAR 文件。用户可以直接使用已经提供的、基于 Java 运行的 Hbase Shell 脚本。

启动后, 用户输入 `help` 命令然后回车, 会返回文本的帮助信息 (以下代码示例省略了部分内容):

```
hbase(main):001:0> help
HBase Shell, version 0.91.0-SNAPSHOT, r1127782, Thu May 26 10:28:47 CEST 2011
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
  .
  Group name: ddl
  Commands: alter, create, describe, disable, drop, enable, exists,
            is_disabled, is_enabled, list
  ...

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:
...
```

如上所述, 用户可以通过在调用时添加命令来请求特定的帮助, 还可以对一组命令请求帮助, 命令或组名都需要用引号括起来。

要离开命令行可以输入 `exit` 或 `quit`:

```
hbase(main):002:0> exit
```

^① 详情见 Ruby 网站 <http://www.ruby-lang.org/>。

\$

Shell 命令在启动时也有特定的命令行选项，添加 -h 或 --help，切换到命令行时会看到这些命令行选项：

```
$ $HBASE_HOME/bin/hbase shell -h
HBase Shell command-line options:
  format      Formatter for outputting results: console | html. Default: console -d | --debug
  Set DEBUG log levels.
```

调试模式

增加的 -d 或 --debug 可以使 Shell 启动时进入调试 (debug) 模式，主要是将日志级别设置为 DEBUG 级别，并让 Shell 打印出与 Java *stacktraces* 信息相似的 *backtrace* 信息。

在命令行内部时，可以通过 debug 命令来切换调试模式：

```
hbase(main):001:0> debug
Debug mode is ON
```

```
hbase(main):002:0> debug
Debug mode is OFF
```

输入 debug? 命令可以查看当前的调试模式是否已经打开：

```
hbase(main):003:0> debug?
Debug mode is OFF
```

非调试模式的 Shell 日志级别是 ERROR，并且根本不会在控制台上打印 *backtrace* 信息。

此外，还有一个选项用于切换输出数据的显示格式，但是这个选项只在控制台中可用。

Shell 命令启动时默认选择 \$HBASE_HOME 中的配置目录。用户可以覆盖默认配置目录的配置，最重要的是可以连接到不同的集群。新建包含 *hbase-site.xml* 文件的单独目录，配置 `hbase.zookeeper.quorum` 属性并指定另外一个集群，然后像这样启动 Shell：

```
$ HBASE_CONF_DIR="/<your-other-config-dir>/" bin/hbase shell
```

注意，你必须指定一个完整目录，而不仅仅是 *hbase-site.xml* 文件。

6.4.2 命令

所有的命令被分为 5 类，分别代表了它们之间的语义关系。在输入命令时，必须要遵循一定的规则。

引用名

命令行要求在使用表名和列名时必须通过单引号或双引号对其进行引用。

引用值

命令行支持二进制、八进制、十六进制的输入和输出。用户在引用时必须使用双引号，否则 Shell 将把它们解释成文本。

```
hbase> get 't1','key\x00\x6c\x65\x6f\x6e'
hbase> get 't1','key\000\154\141\165\162\141'
hbase> put 't1','test\xef\xff','f1:',"\x01\x33\x70"
```

注意上述的混合引用，用户必须确保这个引用值是正确的，否则无法获得预期的结果。文本在单引号内将被当做文本对待，在双引号内将被替换，比如会将八进制或者十六进制值转换成字节。

使用逗号分隔参数

参数之间需要使用逗号进行分隔。例如：

```
hbase(main):001:0> get 'testtable','row-1',
'colfam1:qual1'
```

Ruby 散列属性

一些命令中需要设置键/值对属性。使用 Ruby 散列并按以下方式来完成：

```
{'key1' => 'value1','key2' => 'value2',...}
```

键/值对需要被包括在花括号中，键/值之间使用“=>”分隔。使用键/值模式赋值的属性通常是 NAME、VERSIONS 或 COMPRESSION，并且不需要使用引号。例如：

```
hbase(main):001:0> create 'testtable',{NAME =>
'colfam1',VERSIONS => 1,\
TTL => 2592000,BLOCKCACHE => true}
```

限制输出

get 命令有一个用于限制输出结果长度的可选项，这对输出列非常多或输出值比较长的情况非常有用。限制长度可以得到快速预览值，并阻止控制台打印超出长度的数据，否则控制台会很快变得难以控制。

下面的例子中，插入了一个非常长的值，检索时使用 MAXLENGTH 限制了返回长度：

```
hbase(main):001:0> put
'testtable','rowlong','colfam1:qual1','abcdefghijklmnopqrstuvwxyabcdefghijklmnopghi \
jklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyabcdefghijklmnopqrstuvwxyabcde \
...
```

```
xyzabcdefghijklmnopqrstuvwxyabcdefghijklmnopqrstuvwxy'
```

```
hbase(main):018:0> get 'testtable','rowlong',MAXLENGTH => 60
```

```
COLUMN          CELL
```

```
colfam1:qual1   timestamp=1306424577316,value=abcdefghijklmnopqrstuvwxyabc
```

MAXLENGTH 这个值是从一行的开头开始统计，即该包括列名。如果将其设置为控制台的长度，它可以更好地显示这一行。

每一个命令的使用方法都可以通过 `help '<command>'` 来获取详细信息，例如：

```
hbase(main):001:0> help 'status'
```

```
Show cluster status. Can be 'summary','simple',or 'detailed'. The
default is 'summary'. Examples:
```

```
hbase> status
```

```
hbase> status 'simple'
```

```
hbase> status 'summary'
```

```
hbase> status 'detailed'
```

大多数命令都能对应到客户端 API 或具有管理功能的 API 提供的方法。后面几节将简要介绍每个命令以及这些 API 的匹配关系。

1. 普通命令

表 6-1 列出了常用命令。这些命令为用户提供了获取集群详细状态的功能，以及 HBase 运行时版本信息。详情见 5.2.5 节描述的 `ClusterStatus` 类。

表 6-1 普通 Shell 命令

命令	描述
<code>status</code>	返回 <code>ClusterStatus</code> 类中各种级别的信息。通过帮助可以查看简单(<code>simple</code>)、总结(<code>summary</code>)和详细(<code>detailed</code>)状态
<code>Version</code>	返回当前版本信息、仓库版本和编译信息。见表 5.4 中的 <code>ClusterStatus.getHBaseVersion()</code> 方法

2. 数据定义

表 6-2 列出了所有 DDL 命令，其中大多数来自具有管理功能的 API，详情见第 5 章。

表 6-2 DDL 命令

命令	描述
<code>alter</code>	使用 <code>modifyTable()</code> 修改现有表结构，详情见 5.2.3 节

命令	描述
create	创建新表, 详情见 5.2.2 节中的 createTable() 调用方法
describe	打印 HTableDescriptor 对象, 详情见 5.2.2 节
disable	禁用表, 详情见 5.2.2 节中的 disableTable() 方法
drop	删除表, 详情见 5.2.2 节中的 deleteTable() 方法
enable	启用表, 详情见 5.2.2 节中的 enableTable() 方法
exists	检查表是否存在, 详情见 5.2.2 节中的 tableExists() 方法
is_disabled	检查表是否已经禁用, 详情见 5.2.2 节中的 isTableDisabled() 方法
is_enabled	检查表是否已经启用, 详情见 5.2.2 节中的 isTableEnabled() 方法
List	返回所有表, 详情见 5.2.2 节中的 listTables() 方法

3. 数据操作

表 6-3 列出了 DML 操作, 其中大多数操作来自于客户端 API, 详情见第 3 章和第 4 章。

表 6-3 DML 命令

命令	描述
count	统计一张表的行数, 内部使用了 Scan, 详情见 3.5 节
delete	删除一个单元格, 详情见 3.2.3 节中的 Delete 类
deleteall	类似于 delete 但不仅仅删除一行, 主要会删除一个列族或列, 详情见 3.2.3 节中的 Delete 类
get	获取一个单元格, 详情见 3.2.2 节中的 Get 类
get_counter	返回一个计数器数值。这个和 get 命令类似, 但是它将计数器值转换成了可以阅读的数字, 详情见 3.2.2 节中的 Get 类
incr	给计数器加一, 详情见 4.2 节中的 Increment 类
put	存储一个单元格, 详情见 3.2.1 节中的 Put 类
scan	扫描一个范围的数据, 依赖于 Scan 类, 详情见 3.5 节中的 Scan 类
truncate	清理一张表中的数据, 相当于 disable、drop、create 在使用同一个模式下顺序执行

4. 工具

表 6-4 列出了工具类命令, 这些命令都来自于管理 API, 详情见 5.2.4 节。

表 6-4 工具命令

命令	描述
assign	分配一个 region 到一台 region 服务器中, 详情见 5.2.4 节中的 assign() 方法

命令	描述
balance_switch	切换负载均衡状态，详情见 5.2.4 节中的 balanceSwitch() 方法
balancer	启动负载均衡，详情见 5.2.4 节中的 balancer() 方法
close_region	关闭一个 region，详情见 5.2.4 节中的 closeRegion() 方法
Compact	开启某个 region 或一张表的异步合并操作，详情见 5.2.4 节中的 compact() 方法
Flush	开启某个 region 或一张表的异步刷写操作，详情见 5.2.4 节中的 flush() 方法
major_compact	开启某个 region 或一张表的异步强制合并操作，详情见 5.2.4 节中的 majorCompact() 方法。
Move	移动一个 region 到不同的服务器中，详情见 5.2.4 节中的 move() 方法
split	拆分一个 region 或一张表，详情见 5.2.4 节中的 split() 方法
unassign	下线一个 region，详情见 5.2.4 节中的 unassign() 方法
zk_dump	转存 ZooKeeper 固有信息到 HBase 中，这是内部类提供的特殊功能，HBase Master 的 Web UI 也提供了类似的信息

5. 复制

表 6-5 列出了复制（replication）的命令。

表 6-5 复制命令

命令	描述
add_peer	增加复制单元
disable_peer	禁用一个复制单元
enable_peer	启用一个复制单元
remove_peer	移除一个复制单元
start_replication	开启复制进程
stop_replication	关闭复制进程

6.4.3 脚本

用户有时希望脚本是交互式地执行，并且可以立即得到返回值，有时则希望通过调度系统（如 cron 或 at）定时执行一个命令。或者用户可以使用 Nagios 或其他监控工具发送脚本并作出反应。用户还可以通过管道（piping）的形式运行命令：

```
$ echo "status" | bin/hbase shell
HBase Shell;enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.91.0-SNAPSHOT,r1127782,Thu May 26 10:28:47 CEST 2011

status
```

```
1 servers,0 dead,44.0000 average load
```

一旦这个命令完成，Shell 会自动退出，并且程序控制权会返回给调用者。最终，用户也可以在一开始就提交整个 Shell 执行脚本：

```
$ cat ~/hbase-shell-status.rb
status
$ bin/hbase shell ~/hbase-shell-status.rb
1 servers,0 dead,44.0000 average load

HBase Shell;enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.91.0-SNAPSHOT,r1130916,Sat Jul 23 12:44:34 CEST 2011

hbase(main):001:0> exit
```

一旦脚本执行完成，用户就可以继续在 Shell 中执行脚本或者和平时一样退出 Shell。用户可以通过一个可选的开关直接使用原生 JRuby 解释器，并以 Java 应用的模式启动脚本，用户在 hbase 脚本中通过设置 classpath 可以访问任意 Java 类。以下的例子获得了远程集群上表的列表信息：

```
$ cat ~/hbase-shell-status-2.rb
include Java
import org.apache.hadoop.hbase.HBaseConfiguration
import org.apache.hadoop.hbase.client.HBaseAdmin

conf = HBaseConfiguration.new
admin = HBaseAdmin.new(conf)
tables = admin.listTables
tables.each { |table| puts table.getNameAsString() }

$ bin/hbase org.jruby.Main ~/hbase-shell-status-2.rb
testtable
```

因为 Shell 基于 Jruby 的 IRB，所以我们可以使用它的内置功能，例如，命令补全和命令历史记录。启用这个功能需要在 home 目录中创建 *irbrc* 文件，Shell 启动时会自动读取：

```
$ cat ~/.irbrc
require 'irb/ext/save-history'
IRB.conf[:SAVE_HISTORY] = 100
IRB.conf[:HISTORY_FILE] = "#{ENV['HOME']}/.irb-save-history"
```

启动命令行历史记录能够保存执行过的 Shell 命令。命令补全已经被 HBase 脚本启用了。

交互式解释器拥有可以直接调用 HBase 类和功能函数的优点，例如，一些应用要求必须写 Java 程序。下面的例子将二进制字节数组通过 Bytes.toBytes() 调用转化为了整型：

```
hbase(main):001:0>
org.apache.hadoop.hbase.util.Bytes.toInt(\
  "\x00\x01\x06[".to_java_bytes)
=> 67163
```




注意 Shell 如何将前 3 个不可见字符编码为十六进制值,而第 4 个字符“[”则作为一个字符打印。

另一个例子就是将日期类型转化为 Linux 时间数,然后再将其转化为人类可读的日期:

```
hbase(main):002:0> java.text.SimpleDateFormat.new("yyyy/MM/dd HH:mm:ss").parse(\
    "2011/05/30 20:56:29").getTime()
=> 1306781789000

hbase(main):002:0> java.util.Date.new(1306781789000).toString()
=> "Mon May 30 20:56:29 CEST 2011"
```

最后, 还有循环添加数据的例子, 例如, 填充测试数据到测试表:

```
hbase(main):003:0> for i in 'a'..'z' do for j in
    'a'..'z' do put 'testtable', \
    "row-#{i}#{j}", "colfam1:#{j}", "#{j}" end end
```

一个更复杂的填充计数可能如下:

```
hbase(main):004:0> require 'date';
import java.lang.Long
import org.apache.hadoop.hbase.util.Bytes
(Date.new(2011,01,01)..Date.today).each { |x| put "testtable", "daily", \
    "colfam1:" + x.strftime("%Y%m%d"), Bytes.toBytes(Long.new(rand * \
    4000).longValue).to_a.pack("CCCCCCCC") }
```

很明显地, 这些看起来就像是 Ruby 本身的功能。如果用户拥有一些其他语言的编程技巧, 就能够很容易地使用基于 Shell 提供的 IRB 功能。这是一个由简单到复杂的过程。

6.5 基于 Web 的 UI

HBase 提供了基于 Web 的用户接口 (简称 UI), 通过 Web UI 可以查看集群状态以及数据表的服务状态。其中大多数功能是只读的, 但也有少量功能可以通过 Web UI 触发。

6.5.1 master 的 UI

HBase 启动了 Web UI 并列出了其所有重要属性。master 的 Web 默认端口是 60010, region 服务器的 Web 默认端口是 60030。如果 master 运行在默认端口上, 并且服务器的名称为 master.foo.com, 那么 HBase Web 的访问地址就是 <http://master.foo.com:60010>。



Web 服务器端口可以通过配置文件 *hbase-site.xml* 来修改, 其属性包括:

```
hbase.master.info.port
hbase.regionserver.info.port
```

1. 主页

图 6-2 展示了集群 Web UI 的主页。在这个页面中,用户可以查看 HBase 集群的当前状态、提供服务的表和 region 服务器等。

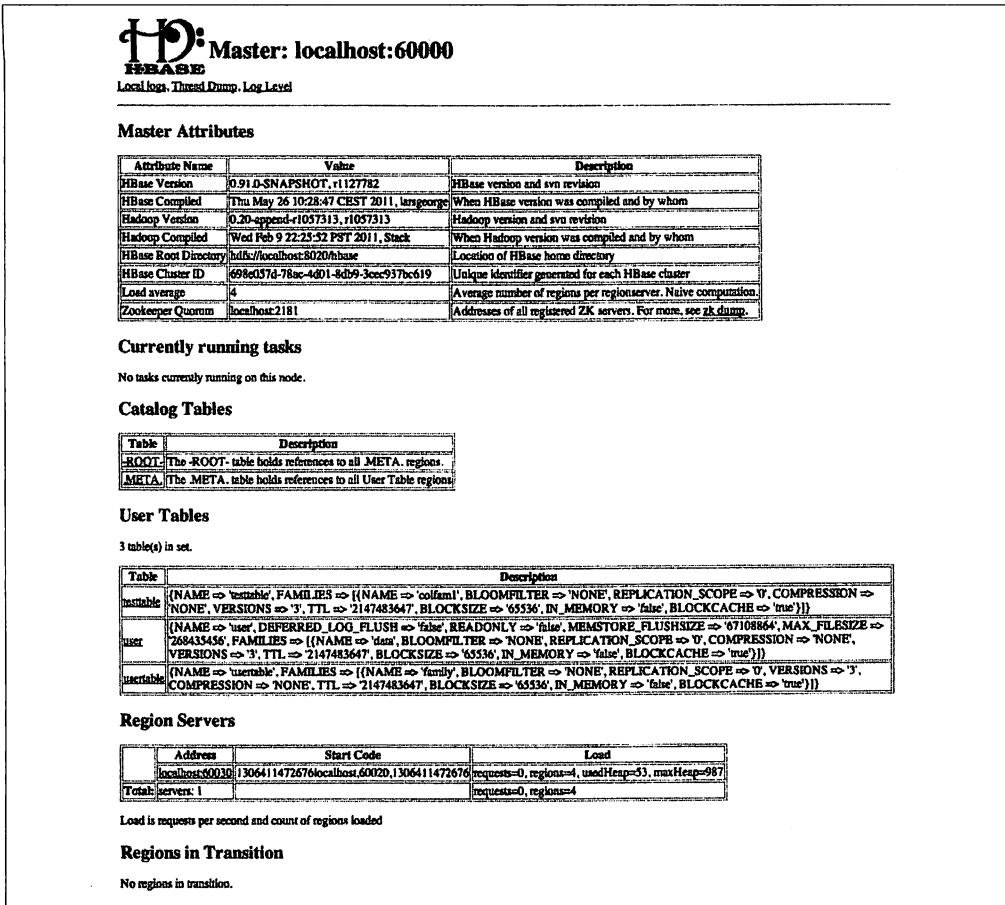


图 6-2 HBase Master UI

主页的信息划分为以下几类。

Master Attributes

表格的顶部显示了集群的粗粒度信息,包括 HBase 的版本、Hadoop 的版本、当前 HBase 集群在 HDFS 中的根目录^①、平均负载、ZooKeeper 的连接地址。

此外,还有一个 ZooKeeper 的链接可以查看 HBase 当前存储在 ZooKeeper 中的信息,详情见 8.7 节。

^① 不能使用/tmp, 否则一旦机器重启会丢失数据, 详情见 2.1 节。

Running Tasks

主页中的下一组信息描述了当前正在执行的任务（currently running tasks）。master 执行的每个正在运行的内部操作都被列在这里，并等待其执行完成。白色背景表示正在执行的任务，绿色背景表示已成功完成的任務，黄色背景表示已经被撤销的任务——当因为状态不一致而操作失败时会发生此类情况。图 6-3 展示了一个已完成的任务、一个正在执行的任务和一个失败的任务。

Currently running tasks		
Description	Status	Age
Doing distributed log split in hdfs://localhost:8020/hbase/.logs/42.1.3.74.60020,1306849761977	finished splitting (more than or equal to) 120291 bytes in 2 log files in hdfs://localhost:8020/hbase/.logs/42.1.3.74.60020,1306849761977 in 2370ms	45s (Completed 43s ago)
Doing distributed log split in hdfs://localhost:8020/hbase/.logs/10.132.225.53.60020,1306687635971	Checking directory contents...	45s (Completed 2s ago)
Master startup	Assigning META region	52s

图 6-3 当前集群正在执行的任务

Catalog Tables

本节有两张表，.META. 和 -ROOT-，用户点击表名可以查看表的详细信息，例如，哪些服务器加载了这张表。

User Tables

用户可以通过此页面查看当前 HBase 集群的数据表，这些表都是用户通过 API 或 HBase Shell 所创建的表。表格属性列展示了表描述，其中包含了所有的列族描述，5.1 节解释了如何阅读它们。

点击表名可以链接到另一个页面，具体内容见本书“用户表信息页面”中的介绍。

Region Servers

这个表格显示了 master 知晓的所有 region 服务器。这张表列出了地址，用户点击该地址可以获取到更详细的信息。这些信息包括 region 服务器的启动码（启动时间戳表示的 ID）和服务器负载等信息。详情见 5.2.5 节，尤其是 HServerLoad 类。

Regions in Transition

处于打开中、关闭中和拆分中的 region 都会出现在这一队列中，执行操作前，将 region 加入到这个列表中，操作完成后，将 region 从这个列表中移除。8.6 节描述了 region 的所有可能状态。图 6-4 展示了正在拆分中的 region。

Regions in Transition	
Region	State
9978c89b9b3c6cb0d9fe41203a322fcb	usertable,user1300651358,1306663063767,9978c89b9b3c6cb0d9fe41203a322fcb, state=SPLITTING, ts=1306669151165, server=localhost,60020,1306663024434

图 6-4 master Web UI 中展示的处于事务中的 region

2. 用户表信息页面

用户点击表链接会进入已选择的表的信息页面。图 6-5 展示了用户表信息页面的精简版（只显示了部分 region）。

用户表信息页面展示了如下几类信息。

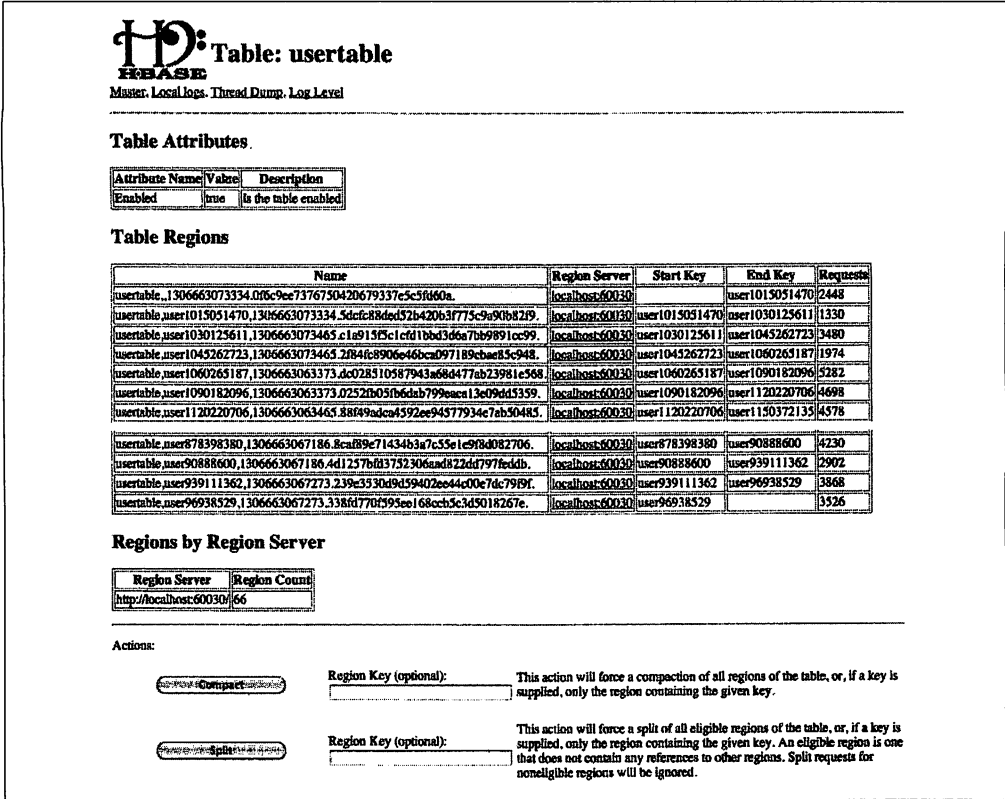


图 6-5 所选的表的相关信息页面

Table Attributes

显示了表自身的信息，仅包括表状态（table status），即表是否可用。显示结果为 true 则表示表可用，显示结果为 false 则表示表当前被禁用。详情见 5.2.2 节的 disableTable() 调用。

布尔值描述了表是否被启用，所以当你这一列中看见 true 时则表明表被启用了。相反地，false 就表示表目前被禁用。

Table Regions

这个表格列出了当前表中所有的 region。其中包括了 region 名、所在服务器地址（该地址链接到部署该 region 的服务器 UI，详情见 6.5.2 节）。

有时表格会显示未部署 (not deployed) 的信息, 这一字段本来应该显示这个 region 在哪台服务器中服务。这是因为该 region 尚未在任何一台服务器中提供服务。图 6-6 展示了这个情况的例子。

user1722720758,1305724102196.dbbcc63057028b09ef747d7e3d079c58.	localhost:60030	user1722720758	user1782933999	16
user1782933999,1305724102196.d47e20439a72381465940c7e42add3d8.		user1782933999	user184325569	0
user184325569,1305724060793.8728710ab629d5266bce7d56643f3469.	localhost:60030	user184325569	user1903293836	16

图 6-6 region 没有被任何服务器加载则显示未部署

Start Key 与 End Key 列展示了 region 的起始行键和终止行键。最后, Requests 列显示了从 region 部署到现在 region 的每秒请求数, 包括了所有读 (get 和 scan) 和写 (put 和 delete) 操作。

Regions by Region Server

最后一个属性格显示了每个 region 服务器加载的当前表的 region 数量, 通常这些数值比较平均。如果这些数值不够平均, 用户可以手动使用 HBase Shell 或者具有管理功能的 API 初始化负载均衡器或执行 move 命令重新均衡表的负载 (详情见 5.2.4 节)。

当前页面还提供了针对特定 region 和整表的管理功能。详情见 5.2.4 节以及 11.4 节。以下是页面中的可用操作。

Compact

这里可以触发合并 (compact) 操作, 该操作在后台异步执行。执行该命令时设置 region 名更具有选择性, region 名可以从上面的表中找到, 即 Table Regions 表中的 Name 列的值。



在复制 region 名时, 一定要将尾部的 “.” 包括在内!

如果没有设置 region 名, 合并操作会将全表的所有 region 作为目标。

Split

类似于合并操作, 拆分以单个 region 或全表为目标。但是, 并非所有的 region 都能执行拆分, 例如, 一些不包含数据或包含非常少数据的 region, 或已经执行拆分操作但并未执行合并操作的 region 均不能执行拆分操作。

一旦触发其中一个操作, 用户会进入到一个确认页面, 例如, 执行拆分命令后会看到:

Split request accepted.

Reload.


点击浏览器后退键可以退回到前面访问的用户表页面。

3. ZooKeeper 页面

在描述列中有一个链接可以让用户转存 HBase 所有在 ZooKeeper 中存储的信息。这个页面会显示所有 HBase 存储在 ZooKeeper 中的节点，这对于查看集群状态并解决问题很有用（详情见 12.5 节）。

这个页面展示了与使用 HBase Shell 调用 `zk dump` 操作相同的信息。它展示了 HBase 在配置文件系统根目录，用户可以看到当前的 master 地址、-ROOT-表地址和所有提供服务的 region 服务器。图 6-7 展示了 ZooKeeper 页面的示例输出。

```


ZooKeeper Dump
Master, Local logs, Thread Dump, Log Level

HBase is rooted at /hbase
Master address: localhost,60000,1306663023371
Region server holding ROOT: localhost,60020,1306663024434
Region servers:
  localhost,60020,1306663024434
Quorum Server Statistics:
  localhost:2181
  zookeeper version: 3.3.2-1031432, built on 11/05/2010 05:32 GMT
Clients:
  /fe80:0:0:0:0:0:1%1:54451[1]{queued=0,recved=298,sent=300}
  /fe80:0:0:0:0:0:1%1:55217[1]{queued=0,recved=283,sent=283}
  /fe80:0:0:0:0:0:1%1:52307[1]{queued=0,recved=634,sent=638}
  /0:0:0:0:0:0:1%0:59324[1]{queued=0,recved=6897,sent=6901}
  /0:0:0:0:0:0:1%0:54449[1]{queued=0,recved=870,sent=877}
  /127.0.0.1:60703[0]{queued=0,recved=1,sent=0}
  /0:0:0:0:0:0:1%0:64611[1]{queued=0,recved=12815,sent=12824}
  /127.0.0.1:54442[1]{queued=0,recved=316,sent=318}
  /fe80:0:0:0:0:0:1%1:54433[1]{queued=0,recved=1141,sent=1549}

Latency min/avg/max: 0/0/780
Received: 274270
Sent: 276689
Outstanding: 0
Exid: 0x90a4a
Mode: standalone
Node count: 19
```

图 6-7 ZooKeeper 页面，列出了 HBase 和 ZooKeeper 的一些细节，调试 HBase 的安装过程中很有用


6.5.2 region 服务器的 UI

region 服务器提供了独立的基于 Web 的 UI，用户通常可以通过在 master 的 UI 中点击提供的服务器名字链接进行访问。用户也可以直接输入以下地址进行询问。

`http://<region-server-address>:60030`

主页

region 服务器主页提供了当前进程的细节信息、任务以及加载的 region。图 6-8 展示了这



Region Server: localhost:60020

Local Logs, Thread Dump, Log Viewer

[Home](#)

Region Server Attributes

Attribute Name	Value	Description
HBase Version	0.91.0-SNAPSHOT, r1127782	HBase version and svn revision
HBase Compiled	Thu May 26 10:28:47 CEST 2011, lan.george	When HBase version was compiled and by whom
Metrics	requests=70.333336, regions=70, stores=70, storefiles=73, storefileIndexSize=5, memstoreSize=0, readRequestsCount=357385, writeRequestsCount=239, compactionQueueSize=30, flushQueueSize=0, usedHeap=139, maxHeap=987, blockCacheSize=100321464, blockCacheFree=106772296, blockCacheCount=1510, blockCacheHitCount=189520, blockCacheMissCount=104532, blockCacheEvictedCount=533, blockCacheHitRatio=64, blockCacheHitCachingRatio=98	RegionServer Metrics; file and heap sizes are in megabytes
Zookeeper Quorum	localhost:2181	Addresses of all registered ZK servers

Currently running tasks

Description	Status	Age
Compacting family in usertable.user1994048399.1306663065728.f886662627e8ba1e2a9a3e935d900e8d.	Compacting store family	2s

Online Regions

Region Name	Start Key	End Key	Metrics
.ROOT-.0.70236052			stores=1, storefiles=2, storefileSizeMB=0, memstoreSizeMB=0, storefileIndexSizeMB=0, readRequestsCount=132, writeRequestsCount=1
.META.,,1.1028785192			stores=1, storefiles=2, storefileSizeMB=0, memstoreSizeMB=0, storefileIndexSizeMB=0, readRequestsCount=8099, writeRequestsCount=238
metastable.,1306157568418.ce2b0fc91f9dd40ed498d1c8edb75bc.			stores=1, storefiles=2, storefileSizeMB=0, memstoreSizeMB=0, storefileIndexSizeMB=0, readRequestsCount=0, writeRequestsCount=0
user.,1305633633075.fc7c2af59dd458ee4d7d20edc5efa8905.			stores=1, storefiles=1, storefileSizeMB=0, memstoreSizeMB=0, storefileIndexSizeMB=0, readRequestsCount=0, writeRequestsCount=0
usertable.,1306663073334.0f6c9ee7376750420679337c5c5fd60a.		user1015051470	stores=1, storefiles=1, storefileSizeMB=32, memstoreSizeMB=0, storefileIndexSizeMB=0, readRequestsCount=2504, writeRequestsCount=0
			stores=1, storefiles=1, storefileSizeMB=0, memstoreSizeMB=0, storefileIndexSizeMB=0, readRequestsCount=0, writeRequestsCount=0
usertable.user96938529.1306663067273.338fd770f595ee168oct5c3d5018267e.	user96938529		stores=1, storefiles=1, storefileSizeMB=65, memstoreSizeMB=0, storefileIndexSizeMB=360, readRequestsCount=3600, writeRequestsCount=0

Region names are made of the containing table's name, a comma, the start key, a comma, and a randomly generated region id. To illustrate, the region named `domains.apache.org.5464829424211263407` is partly in the table `domains`, has an id of `5464829424211263407` and the first key in the region is `apache.org`. The `.ROOT-` and `.META.` tables are internal system tables (or 'catalog' tables in db-speak). The `.ROOT-` keeps a list of all regions in the `.META.` table. The `.META.` table keeps a list of all regions in the system. The empty key is used to denote table start and table end. A region with an empty start key is the first region in a table. If region has both an empty start and an empty end key, it's the only region in the table. See [HBase Home](#) for further explanation.

显示信息分以下几个部分。

可用客户端 271

Region Server Attributes

这组信息包含了正在运行的 HBase 版本信息、编译时间、服务进程的监控指标和正在使用的 ZooKeeper 连接地址。关于监控指标在 10.2.3 节中有详细解释。

Running Tasks

这个表格列出了当前正在执行的任务，白色背景表示正在执行的任务，绿色背景表示已经成功完成的任务，黄色背景表示已经被撤销的任务。失败任务或已完成的任务会在一分钟之内被移除。

Online Regions

用户在这里可以看到当前机器加载的所有 region 信息，包括 region 名、起始与终止行键和 region 监控信息。

6.5.3 共享页面

master、region 服务器和表的主页都有少量通用的页面链接到子页面，以显示或者控制额外的细节。

本地日志

这是一个不需要登录服务器只需点击链接便可以访问日志的便捷方式，它会列出 log 目录下所有的日志文件，用户可以任意选择并访问，详情见 12.5.2 节。图 6-9 展示了当前的例子。

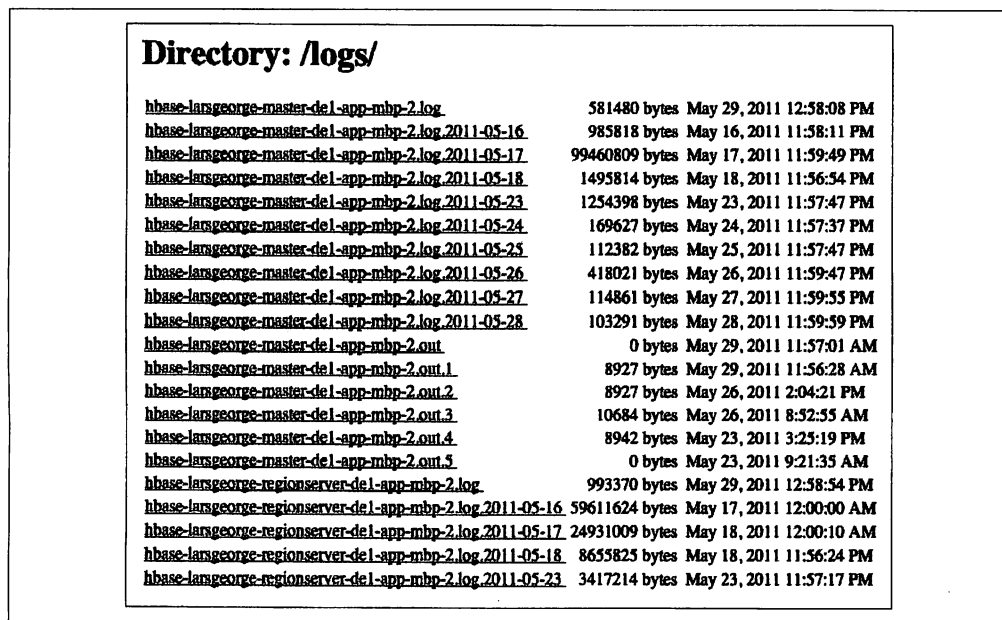


图 6-9 本地日志页面

线程转储

基于调试的目的，用户通过此页面可以查看正在运行的 HBase 进程的 Java stack traces 信息，详情见 12.5 节。图 6-10 展现了示例输出。

```
Process Thread Dump:
55 active threads
Thread 265 (409754308@qtp-873128399-22):
  State: RUNNABLE
  Blocked count: 11
  Waited count: 11
  Stack:
    sun.management.ThreadImpl.getThreadInfo0(Native Method)
    sun.management.ThreadImpl.getThreadInfo(ThreadImpl.java:147)
    sun.management.ThreadImpl.getThreadInfo(ThreadImpl.java:123)
    org.apache.hadoop.util.ReflectionUtils.printThreadInfo(ReflectionUtils.java:149)
    org.apache.hadoop.http.HttpServer$StackServlet.doGet(HttpServer.java:513)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:707)
    javax.servlet.http.HttpServlet.service(HttpServlet.java:820)
    org.mortbay.jetty.servlet.ServletHolder.handle(ServletHolder.java:511)
    org.mortbay.jetty.servlet.ServletHandler.handle(ServletHandler.java:401)
    org.mortbay.jetty.security.SecurityHandler.handle(SecurityHandler.java:216)
    org.mortbay.jetty.servlet.SessionHandler.handle(SessionHandler.java:182)
    org.mortbay.jetty.handler.ContextHandler.handle(ContextHandler.java:766)
    org.mortbay.jetty.webapp.WebAppContext.handle(WebAppContext.java:450)
    org.mortbay.jetty.handler.ContextHandlerCollection.handle(ContextHandlerCollection.java:230)
    org.mortbay.jetty.handler.HandlerWrapper.handle(HandlerWrapper.java:152)
    org.mortbay.jetty.Server.handle(Server.java:324)
    org.mortbay.jetty.HttpConnection.handleRequest(HttpConnection.java:542)
    org.mortbay.jetty.HttpConnection$RequestHandler.headerComplete(HttpConnection.java:928)
    org.mortbay.jetty.HttpParser.parseNext(HttpParser.java:549)
    org.mortbay.jetty.HttpParser.parseAvailable(HttpParser.java:212)
Thread 254 (210958960@qtp-873128399-20):
  State: TIMED_WAITING
  Blocked count: 159
  Waited count: 157
  Stack:
    java.lang.Object.wait(Native Method)
    org.mortbay.thread.QueuedThreadPool$PoolThread.run(QueuedThreadPool.java:626)
Thread 76 (MASTER_SERVER_OPERATIONS-localhost,60000,1306663023371-2):
  State: WAITING
  Blocked count: 108
  Waited count: 99
  Waiting on java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject@6063f5af
  Stack:
    sun.misc.Unsafe.park(Native Method)
    java.util.concurrent.locks.LockSupport.park(LockSupport.java:158)
    java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:1987)
    java.util.concurrent.LinkedBlockingQueue.take(LinkedBlockingQueue.java:399)
    java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:947)
    java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:907)
    java.lang.Thread.run(Thread.java:680)
```

图 6-10 线程转储页面

日志级别

这个链接对应的页面允许用户获取并重设 HBase 进程的日志级别，详情见 12.4 节。图 6-11 展示了页面布局。

Log Level

Get / Set

Log: Get Log Level

Log: Level: Set Log Level

[Hadoop. 2011.](#)

图 6-11 日志级别页面

例如，我们在第一个格中输入 `org.apache.hadoop.hbase`，然后点击 **Get Log Level** 按钮，会得到图 6-12 所展示的结果。

Log Level

Results

Submitted Log Name: **org.apache.hadoop.hbase**
Log Class: **org.apache.commons.logging.impl.Log4JLogger**
Effective level: **DEBUG**

Get / Set

Log:

Log: Level:

[Hadoop, 2011.](#)

图 6-12 日志级别结果页面

HBase 服务提供的 Web UI 是快速获取集群状态、查看表信息等操作的好途径。用户也可以使用 HBase Shell 来达到这个目的，但是这依赖于控制台。

用户可以使用 UI 触发选定的管理操作，但并不是每个用户都有权如此操作：类似于 Shell，这些管理操作最好由集群管理员负责。

如果用户需要建表、删表、操作表等操作，最好通过 HBase 上额外的一层，例如，使用 Thrift 或 REST 作为网关服务器提供此类服务给最终用户。

与 MapReduce 集成

HBase 最大的特点之一就是可以紧密地与 Hadoop 的 MapReduce 框架集成。下面我们将介绍如何利用这些特点，同时 HBase 的某些特性如何发挥其优势。

7.1 框架

在介绍如何在 MapReduce 中使用 HBase 之前，我们先看一下 MapReduce 中的一些基本概念。

7.1.1 MapReduce 介绍

MapReduce 被设计为在可扩展的方式下解决超过 TB 级数据处理过程中的问题。应当有一种方法可以建立一个性能随机器数增加而线性提升的系统，这就是 MapReduce 努力要做到的。它遵守分治原则，通过将数据拆分到分布式文件系统中的不同机器上，让服务器（CPU 或新式的内核）能尽快直接访问和处理数据。这种方法的问题是用户最终需要合并全局结果。同样，MapReduce 为解决这个问题将其内置了。图 7-1 提供了这个过程的高层概览。

下面这张图非常简明地展示了 MapReduce 处理数据的过程。首先它会可靠地将输入的数据拆分成大小合理的块，然后服务器每次处理一个块。一般来说，拆分工作需要尽可能地利用可用的服务器和基础设施。图示中的数据可以是非常大的日志文件，处理时被划分为大小相等的分片，这样做非常适合处理如 Apache 日志类型的文件。输入的数据也可以是二进制文件，但是此时用户需要实现自己的 `getSplits()` 方法，同时也有可能需要涉及以下内容。

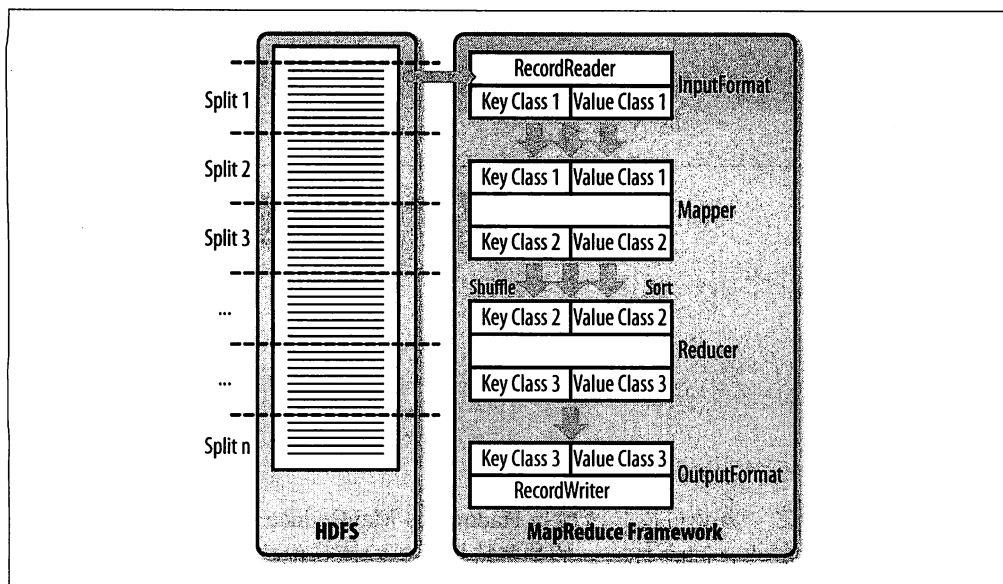


图 7-1 MapReduce 过程

7.1.2 类

上图也展示了与 Hadoop MapReduce 实现相关的一些类。下面介绍它们的功能以及 HBase 中提供的基于它们的具体实现。



Hadoop 0.20.0 版本提供了一套新的 MapReduce API。这些类在 `mapreduce` 包中，之前的类在 `mapred` 包中。旧的 API 已经不推荐使用了，并且原计划在 0.21.0 版本中被废弃，不过最终因为新 API 的功能不完整而搁置，所以旧的 API 并没有被废弃。

HBase 中也有两个稍有不同的包。新的 API 更被社区支持，所以基于它的写作业不受 Hadoop 改动的影响。本章只介绍新的 API。

1. InputFormat

第一个接触到的类是 `InputFormat`（见图 7-2），它负责两件事情：第一件是拆分输入数据，同时返回一个 `RecordReader` 实例，这个实例定义了键值对象的类，并提供了 `next()` 方法来遍历输入的数据。

就 HBase 而言，它提供了一组专用的实现，叫做 `TableInputFormatBase`，该实现的子类是 `TableInputFormat`。`TableInputFormatBase` 实现了其中的大部分功能，但仍旧是抽象类，它的子类 `TableInputFormat` 是一个轻量级的实体类版本，同时也被很多我们提供的例子和真实的 MapReduce 类使用。

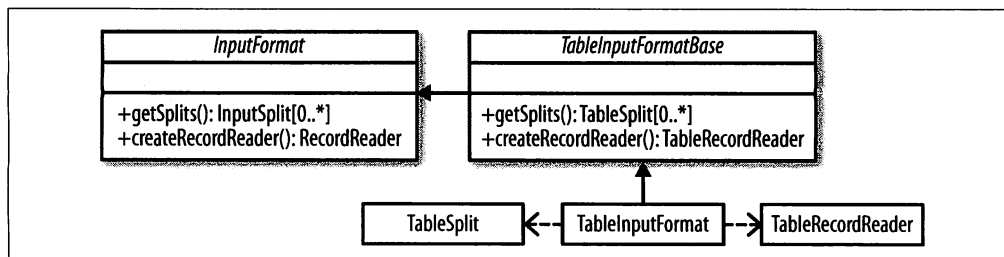


图 7-2 InputFormat 类的层次结构

这些类提供了一个扫描 HBase 全表的实现。用户需要提供一个 Scan 实例：设置起止行键、添加过滤器和指定版本数目等。TableInputFormat 将表拆分成大小合适的块，同时交给后面的 MapReduce 过程处理。如何拆分表的细节参见 7.1.5 节。

2. Mapper

Mapper 类构成了 MapReduce 过程的下一个阶段，同时也是其命名的原因之一（见图 7-3）。在这一阶段中，从 RecordReader 读到的每一个数据都由一个 map() 方法来处理。这些在图 7-1 中也有介绍。Mapper 读取一组特定的键/值对，但是可能会输出其他的类型。这个过程非常便于将原始数据转化为更有用的数据类型，以做进一步处理。

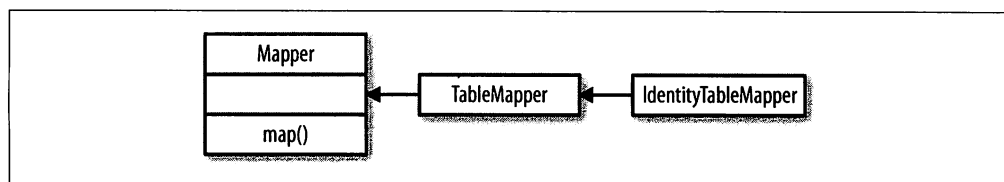


图 7-3 Mapper 类的层次结构

HBase 提供了一个 TableMapper 类，将键的类型强制转换为一个 ImmutableBytesWritable，同时将值的类型强制转换为 Result 类型，这些构成了 TableRecordReader 类返回的结果。

有一个 TableMapper 的特殊实现是 IdentityTableMapper，它也是一个展示如何将自己想要的功能添加到 HBase 提供的类中的比较好的例子。TableMapper 类没有实现任何实际的功能，它只是添加了键/值对的签名。IdentityTableMapper 只是简单地把键/值对传到下一个处理阶段而已。

3. Reducer

Reducer 阶段和类的层次结构（见图 7-4）与之前介绍的 Mapper 阶段十分相似。这次我们得到了 Mapper 类的输出，同时这些输出会经过 shuffle 和 sort 处理。

在 Mapper 和 Reducer 阶段间的内部 shuffle 阶段中，每个不同的 Map 输出的中间结果都会被复制到 Reduce 服务器，同时 sort 阶段会将所有经过 suffled (copied) 阶段处理的数据进行联合排序，这样 Reducer 得到的中间结果就是一个已排序的数据集，在这个数据集中，每一个特定的键与它所有可能的值关联在一起。

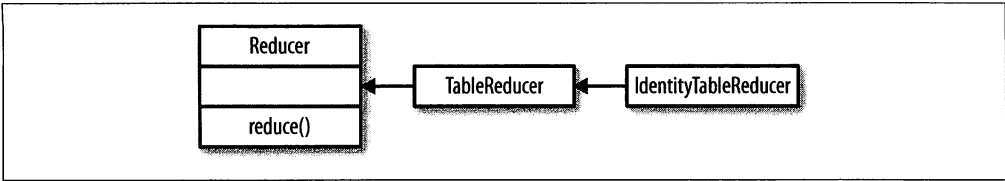


图 7-4 Reducer 类的层次结构

4. OutputFormat

最后阶段由 OutputFormat 类处理（见图 7-5）。它的工作是将数据持久化到不同的位置。这里提供了一些具体实现来允许结果能被输出到文件或者 HBase 表中。输出到 HBase 表中时，用户可以使用 TableOutputFormat 类。它使用 TableRecordWriter 类将数据写到特定的 HBase 表中。

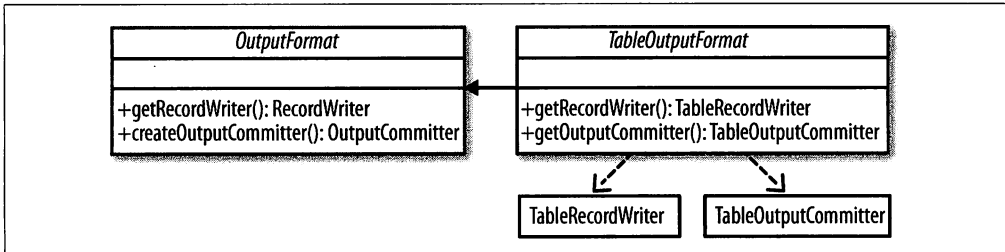


图 7-5 OutputFormat 类的层次结构

同时也需要注意一下基数。一般情况下，有许多 Mapper 将记录传输给 Reducer，但是只有一个 OutputFormat 处理它对应的 Reducer 输出。这是最后一个用于处理键/值对并将它们写到最终存储位置的类，这个存储位置可以是文件或者表。

Hadoop 类需要 TableOutputCommitter 类来实现它们的功能。如果不需要 HBase 来存储数据，则它是一个空类，并没有实现实际的功能。不过其他 OutputFormat 的实现需要这个类。

当创建作业时，输出表的名字就已经被指定了。除此之外，TableOutputFormat 不会增加其他复杂性。其他值得注意的是，表的缓冲区自动刷写被关闭，同时缓冲区由系统内部自动管理。这对提高导入大量数据时的速度十分有帮助。请参阅 11.7 节来了解如何提高扫描的性能。

7.1.3 支撑类

MapReduce 的支撑类与 TableMapReduceUtil 类一同协作在 HBase 上执行 MapReduce 作业。它有一个静态方法能配置作业，并使作业可以使用 HBase 作为数据源或目标。

7.1.4 MapReduce 的执行地点

Hadoop 的块备份策略对上层用户十分不透明：这个过程是自动完成的，用户不用为其操心。HBase 依靠这个副本策略对上层提供持久化存储。虽然这些对于上层完全透明，但是更进一步的问题是，这些会如何影响系统性能？

不论是基于 HBase 还是 Hadoop，这个问题都是用户开始使用 MapReduce 作业时会产生的疑问。特别是当有大量数据存储于 HBase 中时，系统如何将数据存放在需要它的地方？这些都要参考数据的位置以及 HBase 如何使用 Hadoop 文件系统（HDFS）。

首先，让我们看一下 Hadoop 如何处理这些事情：MapReduce 文档中指出任务在靠近其数据的地方执行。为了达到这一点，HDFS 中的大文件被切分成一些小的块，默认大小是 64 MB（实际一般设为 128 MB 或更大）。

每块的数据被指定到一个 map 任务处理。这也说明较大的块会减少所需 map 的数目，因为 map 的数目由需要处理的块的数目决定。Hadoop 知道块的位置，所以会直接在其对应位置上运行 map 任务（实际上由于每个块都有若干副本，副本数量默认为 3，框架可以在这 3 个中任意选择一个执行，用于负载均衡）。这是 MapReduce 保证数据本地计算的方法。

回到 HBase，当用户明白这一点后，可能会对这种机制如何在 HBase 上起作用产生疑问。如 8.2 节介绍的，HBase 将数据存储于 HDFS 上，存储过程对于用户来说是透明的。HBase 通过数据文件 HFile 和它的日志文件（WAL）来实现。如果用户查看代码，会发现它使用 Hadoop API 方法 `FileSystem.create(Path path)` 来创建这些文件。



如果用户的 Hadoop 和 HBase 没有共用集群，而是彼此分开且相互独立，此时相当于运行一个 MapReduce 集群，作业不能在相应的 datanode 上执行。如果需要任务在数据所在的位置执行则必须提供相应的数据位置信息，所以这种情况下 HDFS、MapReduce 和 HBase 必须使用同一套集群。

Hadoop 如何知道 HBase 中数据的分布呢？最重要的就是 HBase 没有被频繁重启，同时日常维护操作都在正常按时运行。当有新数据加入时，合并会重写数据文件。所有 HDFS 上的文件被写入之后都是不可修改的。因此，数据将被写入新文件，而新文件数目也会增加，HBase 会将多个文件合并为一个新的文件。

此外，HDFS 可以根据数据的使用位置智能地放置数据的副本。它自带一个副本放置策略，该策略可以在写入数据时先让数据写入到辅助定位服务器。收到数据的节点先比

较自己的服务器名称和写数据的程序所在的机器是不是一样，如果一样就将数据写入本地磁盘。然后数据的一个副本被送到同一个机架上的另一台服务器，另一个送到远程的机架——假设用户在 HDFS 中启用了机架感知。如果没有启用，额外的副本会被放置在集群中负载最轻的节点上。

如果用户配置了一个更大的副本数目，更多的副本会存在不同的机器上。但最重要的一点是，用户可以取到本地的副本。这就意味着，当 HBase 的 region 服务器长时间没有重启时，表经过 major 合并之后（major 合并可以由用户手动触发或由系统配置定时运行），表的数据就都会有一份本地副本，本地的 Data Node 也会有 region 服务器上所需数据的本地副本。如果用户运行 scan 和 get 或者其他操作，用户会得到更好的性能。

region 在负载均衡或服务器故障时可能会出现移动的情况，当这种情况发生时，数据可能不在本地了，但随着时间的推移，其可能会再次移动到数据所在地。master 在集群重启时会考虑到这种情况：它会把 region 分配到它的原属 region 服务器上。只有当原属服务器不存在时才进行随机分配。

7.1.5 表拆分

当用户使用 MapReduce 作业从表中读取数据时，实质上是使用 TableInputFormat 取得数据。它符合 MapReduce 框架，并重载了公有方法 getSplits() 和 create RecordReader()。在一个作业被执行前，框架调用 getSplit() 来决定如何划分块（chunk），从而由块数目决定映射任务的数目。

对于 HBase 来说，TableInputFormat 基于用户提供的 Scan 实例取得所需要的表信息，并且按 region 来划分边界。由于它不能直接地预测到可选过滤器的执行效果，所以它简单地使用起止键来确定 region。拆分块的数目与起止键之间的 region 数目相等。如果用户没有设置这两个键，则所有 region 都被包含在其中。

作业启动时，框架会按拆分的数目调用 createRecordReader()，并返回与当前块对应的 TableRecordReader 实例。换句话说，每个 TableRecordReader 实例处理一个对应的 region，读取并遍历一个 region 的所有行。

每一个拆分也包含对应 region 所在服务器的信息。正是这个信息能使 HBase 上的 MapReduce 作业本地化执行：框架会比较 region 对应的服务器名，如果有任务 tracker 在对应服务器上运行，它会挑选这个服务器来执行作业。因为 region 服务器与 Data Node 运行在一个节点，所以扫描会从本地磁盘读取文件。



当在 HBase 上运行 MapReduce 时，推荐用户关闭预测执行模式。这样会增加 region 和服务器的负载，同时与本地化运行相违背：预测作业在一个不同的机器上运行，因此没有本地 region 服务器，这会增加网络带宽开销。

7.2 在 HBase 之上的 MapReduce

以下将为读者介绍如何结合MapReduce来使用HBase。用户在使用HBase作为数据源、目标库，或者同时作为两者来处理数据之前，用户还需要先做一些使用Hadoop的准备。

7.2.1 准备

当运行MapReduce作业所需库中的类不是绑定在Hadoop或MapReduce框架中时，用户就必须确保这些库在作业执行之前已经可用。用户一般有两个选择：在所有的任务节点上准备静态的库，或直接提供作业所需的所有库。

1. 静态配置

对于经常使用的库来说，最好将这些JAR文件安装在MapReduce任务tracker的本地，可以通过以下步骤来完成。

1. 把 JAR 文件复制到所有节点的常用路径中。
2. 将这些 JAR 文件的完整路径写入 `hadoop-env.sh` 配置文件中，按照如下方式编辑 `HADOOP_CLASSPATH` 变量：

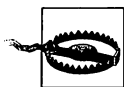
```
# Extra Java CLASSPATH elements. Optional.  
# export HADOOP_CLASSPATH="<extra_entries>:$HADOOP_CLASSPATH"
```

3. 重启所有的任务 tracker 使变更生效。

显然这个技术是纯静态的，并且每次变更（即增加新库）都需要重启任务tracker。如果需要添加HBase的支持，至少需要增加HBase和ZooKeeper的JAR文件。按照如下方式编辑`hadoop-env.sh`：

```
export HADOOP_CLASSPATH="$HBASE_HOME/hbase-0.91.0-SNAPSHOT.jar: \  
$ZK_HOME/zookeeper-3.3.2.jar:$HADOOP_CLASSPATH"
```

这里的前提是假设用户已经定义了两个`$XYZ_HOME`环境变量，并在环境变量中指出软件的安装路径。



请注意，这将可以修复全球范围内任何指定的服务器和服务器上的配置文件所需的库。

所需的特定版本库被锁定的问题可以通过动态配置来解决，稍后做解释。

2. 动态配置

在一些情况下，用户希望每个他们运行的作业可以使用不同的库，或者在作业运行时

能够更新库，这种情况下动态配置就非常有用。

针对这种情况，Hadoop有一个特殊的功能：它可以读取操作目录中`/lib`目录下包含的所有库的JAR文件。用户可以使用此功能生成所谓的“胖”JAR文件，因为它们传输的不仅仅是实际的作业代码，还有所有需要的库。这意味着作业JAR文件更大，但从另一方面来说，其中已经自己包含了处理作业的完整的代码。

使用 Maven

本书中的示例代码使用 Maven 来编译 JAR 文件（详情见前言的“编辑示例程序”一节）。Maven 不仅可以帮助用户编译示例程序，还可以编译加强的“胖”JAR 文件并将其部署到 MapReduce 框架中。这避免了编辑服务器端的配置文件。

Maven 已经支持了 profiles，这可以帮助用户自定义编译过程。本章的 `pom.xml` 利用这个功能添加了一个 `fatjar` 配置，并将 `/lib` 目录下的所有库文件打包到了一个最终的作业 JAR 中。在这个工作属性中，一些依赖可以将 `scope` 属性定义为 `provided`，以使这些文件不被包含在上述复制过程中。当一些库在服务器中已经可用时，可以适当增加这样的标记，例如，Hadoop 的 JAR：

```
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-core</artifactId>
  <version>0.20-append-r1044525</version>
  <scope>provided</scope>
  ...
</dependency>
```

本书资料库的根目录下的父 POM 文件定义了上述信息，此外，本章 POM 中定义依赖的地方也包含此信息。其中一个例子是 Apache Commons CLI 库，该库已经是 Hadoop 的一部分。

胖 JAR 配置使用了 Maven Assembly 插件，在 `src/main/assembly/job.xml` 文件中配置了哪些文件应该被包含，哪些文件不应该被包含在目标 JAR 文件中（即跳过已提供的库）。为取代现有的配置，用户可以编译一个“瘦”JAR，即一个只包含作业类，且需要更新服务器端配置来包含 HBase 和 ZooKeeper 的 JAR，例如：

```
<ch07>$ mvn package
```

上述命令可以编译一个能够被 MapReduce 框架执行的 JAR，运行时可以使用如下命令：

```
<ch07>$ hadoop jar target/hbase-book-ch07-1.0.jar
An example program must be given as the first argument.
Valid program names are:
  AnalyzeData: Analyze imported JSON
  ImportFromFile: Import from file
  ParseJson: Parse JSON into columns
```

```
ParseJson2: Parse JSON into columns (map only)
```

```
...
```

上述命令列出了所有可能的作业名称。它使用了 Hadoop Program Driver 类，该类由所有已知的作业类和它们的名称构成。Maven 构建了 Driver 类，该类封装了 Program Driver 实例，并作为 JAR 文件的主要类；因此它可以自动被 Hadoop 的 *jar* 命令执行。构建一个“胖”JAR 需要使用以下命令：

```
<ch07>$ mvn package -Dfatjar
```

生成的 JAR 文件附加了一个后缀来彼此区分，但这仅仅是一种尝试（用户可以根据喜好覆盖“瘦”JAR）：

```
<ch07>$ hadoop jar target/hbase-book-ch07-1.0-job.jar
```

它的行为与“瘦”JAR 相似，并且用户可以使用相同的参数执行同一个作业。不同之处在于，它包含了所有需要的库，并避免了配置在服务器中发生改变：

```
$ unzip -l target/hbase-book-ch07-1.0-job.jar
```

```
Archive: target/hbase-book-ch07-1.0-job.jar
```

Length	Date	Time	Name
0	07-14-11	12:01	META-INF/
159	07-14-11	12:01	META-INF/MANIFEST.MF
0	07-13-11	15:01	mapreduce/
0	07-13-11	10:06	util/
740	07-13-11	10:06	mapreduce/Driver.class
3547	07-14-11	12:01	mapreduce/ImportFromFile\$ImportMapper.class
5326	07-14-11	12:01	mapreduce/ImportFromFile.class
...			
8739	07-13-11	10:06	util/HBaseHelper.class
0	07-14-11	12:01	lib/
16046	05-06-10	16:08	lib/json-simple-1.1.jar
58160	05-06-10	16:06	lib/commons-codec-1.4.jar
598364	11-22-10	21:43	lib/zookeeper-3.3.2.jar
2731371	07-02-11	15:20	lib/hbase-0.91.0-SNAPSHOT.jar
14837	07-14-11	12:01	lib/hbase-book-ch07-1.0.jar
-----			-----
3445231			16 files

Maven 并不是生成不同作业的 JAR 唯一的方法，用户也可以使用 Apache Ant。无论用户使用哪种方式编译 JAR，它们都需要包含必要的信息（只是代码，或代码及其必备的库）。

另一种动态提供必备库的方式是 Hadoop MapReduce 框架的 *libjars* 功能。当用户使用 `GenericOptionsParser` 创建一个 MapReduce 作业时，可以得到 *libjar* 参数提供的支持。以下是该参数类的文档：

```
GenericOptionsParser is a utility to parse command line arguments generic  
to the Hadoop framework GenericOptions Parser recognizes several standarad
```

command line arguments, enabling applications to easily specify a namenode, a jobtracker, additional configuration resources etc.

Generic Options

The supported generic options are:

- conf <configuration file> specify a configuration file
- D <property=value> use value for given property
- fs <local|namenode:port> specify a namenode
- jt <local|jobtracker:port> specify a job tracker
- files <comma separated list of files> specify comma separated files to be copied to themap reduce cluster
- libjars <comma separated list of jars> specify comma separated jar files to include in the classpath.
- archives <comma separated list of archives> specify comma separated archives to be unarchived on the compute machines.

The general command line syntax is:

bin/hadoop command [genericOptions] [commandOptions]

请仔细阅读文档，该文档详细地解释了libjars参数，并指出了怎样在命令行中使用该参数。添加libjars参数失败会导致MapReduce作业失败，最终可在任务的执行单元日志中查看相关信息。启动作业时，错误也会同时反馈到命令行中，例如：

```
$ HADOOP_CLASSPATH=$HBASE_HOME/target/hbase-0.91.0-SNAPSHOT.jar: \
$ZK_HOME/zookeeper-3.3.2.jar hadoop jar target/hbase-book-ch07-1.0.jar \
ImportFromFile -t testtable -i test-data.txt -c data:json
...
11/08/08 11:13:17 INFO mapred.JobClient: Running job: job_201108081021_0003
11/08/08 11:13:18 INFO mapred.JobClient: map 0% reduce 0%
11/08/08 11:13:29 INFO mapred.JobClient: Task Id : \
attempt_201108081021_0003_m_000002_0, Status : FAILED
java.lang.RuntimeException: java.lang.ClassNotFoundException: \
org.apache.hadoop.hbase.mapreduce.TableOutputFormat
at org.apache.hadoop.conf.Configuration.getClass(Configuration.java:809)
at org.apache.hadoop.mapreduce.JobContext.getOutputFormatClass(JobContext.java:197)
at org.apache.hadoop.mapred.Task.initialize(Task.java:413)
at org.apache.hadoop.mapred.MapTask.run(MapTask.java:288)
at org.apache.hadoop.mapred.Child.main(Child.java:170)
```

HADOOP_CLASSPATH需要使用命令行。Driver类在启动时需要访问HBase和ZooKeeper类，解决上述问题需要按照如下方式添加libjars参数：

```
$ HADOOP_CLASSPATH=$HBASE_HOME/target/hbase-0.91.0-SNAPSHOT.jar: \
$ZK_HOME/zookeeper-3.3.2.jar hadoop jar target/hbase-bk-ch07-1.0.jar \
ImportFromFile -libjars $HBASE_HOME/target/hbase-0.91.0-SNAPSHOT.jar, \
$ZK_HOME/zookeeper-3.3.2.jar -t testtable -i test-data.txt -c data:json
...
11/08/08 11:19:38 INFO mapred.JobClient: Running job: job_201108081021_0006
11/08/08 11:19:39 INFO mapred.JobClient: map 0% reduce 0%
```

```
11/08/08 11:19:48 INFO mapred.JobClient: map 100% reduce 0%
11/08/08 11:19:50 INFO mapred.JobClient: Job complete: job_201108081021_0006
```

最终，HBase辅助类TableMapReduceUtil提供了可以帮助用户在作业中通过代码动态添加依赖的JAR和配置文件的方法：

```
static void addDependencyJars(Job job) throws IOException;
static void addDependencyJars(Configuration conf, Class... classes)
    throws IOException;
```

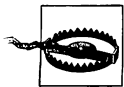
用户过去都使用后一种功能来添加所需的HBase、ZooKeeper和作业类：

```
addDependencyJars(job.getConfiguration(),
    org.apache.zookeeper.ZooKeeper.class,
    job.getMapOutputKeyClass(),
    job.getMapOutputValueClass(),
    job.getInputFormatClass(),
    job.getOutputKeyClass(),
    job.getOutputValueClass(),
    job.getOutputFormatClass(),
    job.getPartitionerClass(),
    job.getCombinerClass());
```

读者可以在ImportTsv类的源代码中查看使用说明：

```
public static Job createSubmittableJob(Configuration conf, String[] args)
    throws IOException, ClassNotFoundException {
    ...
    Job job = new Job(conf, NAME + "_" + tableName);
    ...
    TableMapReduceUtil.addDependencyJars(job);
    TableMapReduceUtil.addDependencyJars(job.getConfiguration(),
        com.google.common.base.Function.class /* Guava used by TsvParser */);
    return job;
}
```

这里首先调用addDependencyJars()方法添加作业和必要的类，包括输入输出格式和键值的类型等。第二个调用添加了Google Guava JAR，这个JAR需求程度在其他库之上。需要注意的是，这个方法并不需要用户指定实际的JAR文件，它直接使用Java的类加载器进行加载但有可能会找到相同的JAR文件，不过在这里这是无关紧要的，重要的是你可以在Java CLASSPATH中访问该类；否则，这些调用最终会失败，并返回一个ClassNotFoundException错误，类似于之前你看到过的信息。用户在一个没有准备好的Hadoop启动中，至少需要填加HADOOP_CLASSPATH到命令行中。

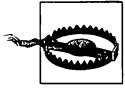


你将选择哪种方法？胖JAR的优势在于，其包含HBase启动过程所有作业所需的库，而另一种方式则至少需要准备classpath。

就本书而言，我们将使用胖JAR的方式编译和运行MapReduce作业。

7.2.2 数据流向

随后，我们将使用MapReduce作业作为过程的一部分，该作业可以使用HBase进行读取或写入。第一个例子是使用HBase作为数据流向，这个例子是在TableOutputFormat类的帮助下实现的，详情见例7.1。



这个例子中使用的数据基于 Delicious (<http://delicious.com>) 提供的公开订阅源。Arvind Narayanan 使用这个订阅源收集了一个简单的数据集，已经发布在他的博客中。

不是必须使用这个数据集，或捕捉订阅源信息 (<http://feeds.delicious.com/v2/rss/recent>)；你可以任意使用其他开源的资源，包括 JSON 格式的数据记录。另一方面，Delicious 的数据提供了完整的 Hush 模式：每个实体都有链接、用户名、时间和类别等。

本书资料库中的 *test-data.txt* 是一个小型公开数据集的子集。对于测试来说，这个数据集已经足够，但显然使用完整的数据集执行作业更好。

以下是其完整的代码，包含标准的模板排序。随后的例子并不会包含这些样板代码，例如，命令行参数解析。

例 7.1 MapReduce 作业从一个文件中读取数据并写入 HBase 表

```
public class ImportFromFile {
    public static final String NAME = "ImportFromFile";❶
    public enum Counters { LINES }

    static class ImportMapper
        extends Mapper<LongWritable, Text, ImmutableBytesWritable, Writable> {❷

        private byte[] family = null;
        private byte[] qualifier = null;

        @Override
        protected void setup(Context context)
            throws IOException, InterruptedException {
            String column = context.getConfiguration().get("conf.column");
            byte[][] colkey = KeyValue.parseColumn(Bytes.toBytes(column));
            family = colkey[0];
            if (colkey.length > 1) {
                qualifier = colkey[1];
            }
        }

        @Override
        public void map(LongWritable offset, Text line, Context context)❸
```

```

throws IOException {
    try {
        String lineString = line.toString();
        byte[] rowkey = DigestUtils.md5(lineString);❶
        Put put = new Put(rowkey);
        put.add(family, qualifier, Bytes.toBytes(lineString));❷
        context.write(new ImmutableBytesWritable(rowkey), put);
        context.getCounter(Counters.LINES).increment(1);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

private static CommandLine parseArgs(String[] args) throws ParseException {❸
    Options options = new Options();
    Option o = new Option("t", "table", true,
        "table to import into (must exist)");
    o.setArgName("table-name");
    o.setRequired(true);
    options.addOption(o);
    o = new Option("c", "column", true,
        "column to store row data into (must exist)");
    o.setArgName("family:qualifier");
    o.setRequired(true);
    options.addOption(o);
    o = new Option("i", "input", true,
        "the directory or file to read from");
    o.setArgName("path-in-HDFS");
    o.setRequired(true);
    options.addOption(o);
    options.addOption("d", "debug", false, "switch on DEBUG log level");
    CommandLineParser parser = new PosixParser();
    CommandLine cmd = null;
    try {
        cmd = parser.parse(options, args);
    } catch (Exception e) {
        System.err.println("ERROR: " + e.getMessage() + "\n");
        HelpFormatter formatter = new HelpFormatter();
        formatter.printHelp(NAME + " ", options, true);
        System.exit(-1);
    }
    return cmd;
}

public static void main(String[] args) throws Exception {
    Configuration conf = HBaseConfiguration.create();
    String[] otherArgs =
        new GenericOptionsParser(conf, args).getRemainingArgs();❹
    CommandLine cmd = parseArgs(otherArgs);
}

```

```

String table = cmd.getOptionValue("t");
String input = cmd.getOptionValue("i");
String column = cmd.getOptionValue("c");
conf.set("conf.column", column);
Job job = new Job(conf, "Import from file " + input + " into table " + table); ❸
job.setJarByClass(ImportFromFile.class);
job.setMapperClass(ImportMapper.class);
job.setOutputFormatClass(TableOutputFormat.class);
job.getConfiguration().set(TableOutputFormat.OUTPUT_TABLE, table);
job.setOutputKeyClass(ImmutableBytesWritable.class);
job.setOutputValueClass(Writable.class);
job.setNumReduceTasks(0); ❹
FileInputFormat.addInputPath(job, new Path(input));

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

```

- ❶ 为后续的使用定义一个作业名。
- ❷ 定义 mapper 类，继承自 Hadoop 已有的类。
- ❸ map() 函数将 InputFormat 提供的键值对转化为了 OutputFormat 需要的类型。
- ❹ 行键是经过 MD5 散列之后随机生成的键值。
- ❺ 存储原始数据到给定的表中的一列。
- ❻ 使用 Apache Commons CLI 类解析命令行参数。这些已经是 HBase 的一部分，因此用户可以很方便地处理作业的参数。
- ❼ 传递命令行参数到解析器中，并优先解析“-Dxyz”属性。
- ❽ 使用特定的类定义作业。
- ❾ 这是一个只包含 map 阶段的作业，框架会直接跳过 reduce 阶段。

用户在代码的 main() 函数中进行了设置，在运行 MapReduce 作业之前，首先解析命令行以获取目标表名、目标列和输入文件的名称。虽然这里可以硬编码，但是最好还是写一段支持可配置的代码。

下一步是构建作业实例，通过命令行和已确定的参数（如类名）指定变量的细节。其中的一个细节是指定 mapper 类，将其设置为 ImportMapper。这个类与 main 类在同一个源代码文件中，并定义了作业在 map 阶段要完成的逻辑。

Main() 函数的代码指定了输出格式的类，即前面描述过的 TableOutputFormat 类。这个类由 HBase 提供，并能够方便地向一张表中写入数据，键的类型被该类隐式地设置为 ImmutableBytesWritable 类，值被隐式地设置为 Writable 的类。

用户在执行作业前，最好先创建一个目标表，例如，使用HBase Shell来创建目标表：

```
hbase(main):001:0> create 'testtable', 'data'
0 row(s) in 0.5330 seconds
```

一旦准备好了表，用户就可以运行该作业了：

```
$ hadoop dfs -put /projects/private/hbase-book-code/ch07/test-data.txt.
$ hadoop jar target/hbase-book-ch07-1.0-job.jar ImportFromFile \
-t testtable -i test-data.txt -c data:json
...
11/08/08 12:35:01 INFO mapreduce.TableOutputFormat: \
Created table instance for testtable
11/08/08 12:35:01 INFO input.FileInputFormat: Total input paths to process : 1
11/08/08 12:35:02 INFO mapred.JobClient: Running job: job_201108081021_0007
11/08/08 12:35:03 INFO mapred.JobClient: map 0% reduce 0%
11/08/08 12:35:10 INFO mapred.JobClient: map 100% reduce 0%
11/08/08 12:35:12 INFO mapred.JobClient: Job complete: job_201108081021_0007
```

第一个命令是`hadoop dfs -put`，该命令将样本数据集存储到HDFS的用户目录中。第二个命令则可以运行这个作业，作业的完成需要一段时间。使用默认的`TextInputFormat`类读取数据，这个类是由Hadoop及其MapReduce框架提供的。这个输入格式能够读取用换行符换行的文本文件。每读取一行都会调用一次指定的mapper类的`map()`方法，这将触发`ImportMapper.map()`函数。

如例7.1所示，`ImportMapper`类定义了两个方法，这两个方法重载了父类`Mapper`，方法名均相同。

重载中存在的问题

强烈推荐在重载的方法中添加`@Override`的注释，这样错误的标签在编译时会被检查出来。否则，隐式的`map()`与`reduce()`方法会被调用，并实现恒等功能。例如，以下`reduce()`方法：

```
public void reduce(Writable key, Iterator<Writable> values,
    Context context) throws IOException, InterruptedException {
    ...
}
```

上述方法看起来正确，实则不然，其实并没有重载`Reducer`类的`reduce()`方法，而只是定义了`reduce()`方法的一个多态。`MapReduce`框架会忽略这个方法，并执行由`Reducer`类提供的默认实现。

究其原因，实际应该标记的方法应该是：

```
protected void reduce(KEYIN key, Iterable<VALUEIN> values, \
    Context context) throws IOException, InterruptedException
```

这是常见错误，在这里 `Iterable` 被错误地替换成了 `Iterator` 类。在任务执行发生奇怪的行为之前，在你的代码中为重载的方法添加 `@Override` 注释能够帮助用户在编译的时候抛出错误（用户最好在后台完成 IDE 检查）。将注释添加到方法的前面，例如：

```
@Override
public void reduce(Writable key, Iterator<Writable> values,
    Context context) throws IOException, InterruptedException {
    ...
}
```

用户正在使用的 IDE 可能已经出现了错误，但编译器仍会抛出如下错误：

```
...
[INFO]-----
[ERROR] BUILD FAILURE
[INFO]-----
[INFO] Compilation failure
ch07/src/main/java/mapreduce/InvalidReducerOverride.java:[18,4] method does not override or
implement a method from a supertype
```

`ImportMapper` 类的重载的 `setup()` 方法只会在框架初始化该类时调用一次。在这个例子中，它主要被用于将指定的列信息解析为列族和列限定符。

这个类中的 `map()` 才是执行实际工作的实体，它会被输入的文本文件中的每一行调用，每一行都包含了一个 JSON 格式的记录。这段代码会使用一行数据的 MD5 散列值来创建一个 HBase 行键，然后使用 HBase 提供的名为 `data:json` 的列存储一行的内容。

这个例子通过 `TableOutputFormat` 类使用了隐式的写缓冲区。调用 `context.write()` 方法时，该方法内部会传入给定的 `Put` 实例并调用 `table.put()`。在作业结束前，`TableOutputFormat` 会主动调用 `flushCommits()` 以保存仍就驻留在写缓冲区的数据。



`map()` 方法可以通过写 `Put` 实例来存储输入数据，用户也可以通过写 `Delete` 实例来删除目标表中的数据。这就是设置作业输出键的类型为 `Writable` 接口，而并非显式的 `Put` 类的原因。

`TableOutputFormat` 当前仅能处理 `Put` 与 `Delete` 实例，将消息设置为除 `Put` 或 `Delete` 之外的实例都会导致抛出一个 `IOException` 异常。

最后，请注意作业在没有 `reduce` 阶段的情况下，`map` 阶段是怎样工作的。这是相当典型的 HBase 与 MapReduce 作业结合：由于数据是存储在排序表中的，并且每行数据都拥有唯一的行键，用户可以在流程中避免更消耗的 `sort`、`shuffle` 和 `reduce` 阶段。

7.2.3 数据源

将数据导入到表之后，我们可以使用其中包含的数据来解析JSON记录，并从中提取信息。这里完全使用了TableInputFormat类，恰好对应了TableOutputFormat。在MapReduce处理过程中，它以一张表为输入。例7.2使用了已提供的InputFormat类。

例 7.2 MapReduce 作业读取已导入的数据并解析

```
static class AnalyzeMapper extends TableMapper<Text, IntWritable> {❶

    private JSONParser parser = new JSONParser();
    private IntWritable ONE = new IntWritable(1);

    @Override
    public void map(ImmutableBytesWritable row, Result columns, Context context)
        throws IOException {
        context.getCounter(Counters.ROWS).increment(1);
        String value = null;
        try {
            for (KeyValue kv : columns.list()) {
                context.getCounter(Counters.COLS).increment(1);
                value = Bytes.toStringBinary(kv.getValue());
                JSONObject json = (JSONObject) parser.parse(value);
                String author = (String) json.get("author");❷
                context.write(new Text(author), ONE);
                context.getCounter(Counters.VALID).increment(1);
            }
        } catch (Exception e) {
            e.printStackTrace();
            System.err.println("Row: " + Bytes.toStringBinary(row.get()) +
                ", JSON: " + value);
            context.getCounter(Counters.ERROR).increment(1);
        }
    }
}

static class AnalyzeReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {❸

    @Override
    protected void reduce(Text key, Iterable<IntWritable> values,
        Context context) throws IOException, InterruptedException {
        int count = 0;
        for (IntWritable one : values) count++;❹
        context.write(key, new IntWritable(count));
    }
}

public static void main(String[] args) throws Exception {
    ...
    Scan scan = new Scan();❺
    if (column != null) {
        byte[][] colkey = KeyValue.parseColumn(Bytes.toBytes(column));
```

```

        if (colkey.length > 1) {
            scan.addColumn(colkey[0], colkey[1]);
        } else {
            scan.addFamily(colkey[0]);
        }
    }
}

Job job = new Job(conf, "Analyze data in " + table);
job.setJarByClass(AnalyzeData.class);
TableMapReduceUtil.initTableMapperJob(table, scan, AnalyzeMapper.class,
    Text.class, IntWritable.class, job); ❸
job.setReducerClass(AnalyzeReducer.class);
job.setOutputKeyClass(Text.class); ❹
job.setOutputValueClass(IntWritable.class);
job.setNumReduceTasks(1);
FileOutputFormat.setOutputPath(job, new Path(output));

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

```

- ❶ 继承已提供的TableMapper类，设置用户自己的输出键值类型。
- ❷ 解析JSON数据，提取编辑用户，并统计发生次数。
- ❸ 拓展一个Hadoop Reducer类，并指定适当的类型。
- ❹ 统计发生次数，并计算其总和。
- ❺ 创建并配置一个Scan实例。
- ❻ 使用提供的库来设置表的map阶段。
- ❼ 使用常规的Hadoop语法来配置reduce阶段。

这个作业运行了一个完整的MapReduce过程，map阶段主要负责从源表中读取JSON数据，reduce阶段主要负责对每个用户做聚合统计。这个例子与Hadoop提供的WordCount例子非常相似，mapper负责记录统计值ONE，reducer负责对每个键做聚合操作，例7.2操作的键是编辑用户（Author）。在命令行上按照如下方式执行作业：

```

$ hadoop jar target/hbase-book-ch07-1.0-job.jar AnalyzeData \
-t testtable -c data:json -o analyzeel
11/08/08 15:36:37 INFO mapred.JobClient: Running job: job_201108081021_0021
11/08/08 15:36:38 INFO mapred.JobClient: map 0% reduce 0%
11/08/08 15:36:45 INFO mapred.JobClient: map 100% reduce 0%
11/08/08 15:36:57 INFO mapred.JobClient: map 100% reduce 100%
11/08/08 15:36:59 INFO mapred.JobClient: Job complete: job_201108081021_0021
11/08/08 15:36:59 INFO mapred.JobClient: Counters: 19
...
11/08/08 15:36:59 INFO mapred.JobClient: mapreduce.AnalyzeData$Counters
11/08/08 15:36:59 INFO mapred.JobClient:   ROWS=993
11/08/08 15:36:59 INFO mapred.JobClient:   COLS=993

```

```
11/08/08 15:36:59 INFO mapred.JobClient: VALID=993
```

```
...
```

这个例子的结果是包含每个编辑用户的统计列表，并且可以通过命令行方式进行访问，例如，`hadoop dfs -text analyze1/part-r-00000`：

```
$ hadoop dfs -text analyze1/part-r-00000
10sr      1
13tohl    1
14bcps    1
21721725  1
2centime  1
33rpm     1
...
```

这个例子展示了怎样使用TableMapReduceUtil类，通过使用TableMapReduceUtil的静态方法可以快速配置一个作业，并附带必要的类。由于作业需要reduce阶段，main()函数代码添加了必要的Reducer类，并在没有其他特殊指定（本例中是TextOutputFormat类）的情况下隐式地使用默认值。

显然，这是一个非常简单的例子，实际上用户不得不执行更多的分析处理。但即便如此，例子所展现的模式与之前仍类似：用户从表中读取数据，提取必要信息，最终将结果输出到一个特定目标。

7.2.4 数据源与数据流向

如上节所描述的，一个MapReduce作业中的源和目标都可以是HBase表，但也有可能在一个作业中同时使用HBase作为输入和输出。换句话说，第三类的MapReduce模板同时使用HBase表作为输入与输出。这里需要将TableInputFormat和TableOutputFormat类设置到各自的作业配置项中。如前所述，这意味着不同的键值类型，如例7.3所示。

例 7.3 MapReduce 作业解析每行数据为若干列

```
static class ParseMapper
extends TableMapper<ImmutableBytesWritable, Writable> {

    private JSONParser parser = new JSONParser();
    private byte[] columnFamily = null;

    @Override
    protected void setup(Context context)
    throws IOException, InterruptedException {
        columnFamily = Bytes.toBytes(
            context.getConfiguration().get("conf.columnfamily"));
    }

    @Override
    public void map(ImmutableBytesWritable row, Result columns, Context context)
    throws IOException {
        context.getCounter(Counters.ROWS).increment(1);
    }
}
```

```

String value = null;
try {
    Put put = new Put(row.get());
    for (KeyValue kv : columns.list()) {
        context.getCounter(Counters.COLS).increment(1);
        value = Bytes.toStringBinary(kv.getValue());
        JSONObject json = (JSONObject) parser.parse(value);
        for (Object key : json.keySet()) {
            Object val = json.get(key);
            put.add(columnFamily, Bytes.toBytes(key.toString()), ❶
                Bytes.toBytes(val.toString()));
        }
    }
    context.write(row, put);
    context.getCounter(Counters.VALID).increment(1);
} catch (Exception e) {
    e.printStackTrace();
    System.err.println("Error: " + e.getMessage() + ", Row: " +
        Bytes.toStringBinary(row.get()) + ", JSON: " + value);
    context.getCounter(Counters.ERROR).increment(1);
}
}
}

public static void main(String[] args) throws Exception {
    ...
    Scan scan = new Scan();
    if (column != null) {
        byte[][] colkey = KeyValue.parseColumn(Bytes.toBytes(column));
        if (colkey.length > 1) {
            scan.addColumn(colkey[0], colkey[1]);
            conf.set("conf.columnfamily", Bytes.toStringBinary(colkey[0])); ❷
            conf.set("conf.columnqualifier", Bytes.toStringBinary(colkey[1]));
        } else {
            scan.addFamily(colkey[0]);
            conf.set("conf.columnfamily", Bytes.toStringBinary(colkey[0]));
        }
    }

    Job job = new Job(conf, "Parse data in " + input + ", write to " + output);
    job.setJarByClass(ParseJson.class);
    TableMapReduceUtil.initTableMapperJob(input, scan, ParseMapper.class, ❸
        ImmutableBytesWritable.class, Put.class, job);
    TableMapReduceUtil.initTableReducerJob(output, ❹
        IdentityTableReducer.class, job);

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

```

- ❶ 将顶层的JSON记录的键存储为列，其值的集合存储为列值。
- ❷ 在mapper的配置实例中存储列族以备后用。
- ❸ 使用通用方法设置map阶段的细节。

④ 配置一个reducer实例用于存储解析后的数据。

这个例子使用通用方法配置map阶段和reduce阶段，并用到了ParseMapper类，这个类用于从一行原始JSON数据中提取信息，还使用了IdentityTableReducer类将数据存储到目标表。请注意，源表与目标表可以是相同的。用户可以通过以下命令行提交作业：

```
$ hadoop jar target/hbase-book-ch07-1.0-job.jar ParseJson \  
-i testtable -c data:json -o testtable  
11/08/08 17:44:33 INFO mapreduce.TableOutputFormat: \  
Created table instance for testtable  
11/08/08 17:44:33 INFO mapred.JobClient: Running job: job_201108081021_0026  
11/08/08 17:44:34 INFO mapred.JobClient: map 0% reduce 0%  
11/08/08 17:44:41 INFO mapred.JobClient: map 100% reduce 0%  
11/08/08 17:44:50 INFO mapred.JobClient: map 100% reduce 100%  
11/08/08 17:44:52 INFO mapred.JobClient: Job complete: job_201108081021_0026  
...
```

这个百分比表示map阶段与reduce阶段已经完成，并且整个作业也已经完成。使用IdentityTableReducer类存储数据并不是必需的，实际上，只需要在代码中增加一行就可以让作业只拥有map阶段。例7.4展示了需要增加的一行代码。

例 7.4 MapReduce 作业解析每行数据为若干列（仅限 map 阶段）

```
...  
Job job = new Job(conf, "Parse data in " + input + ", write to " + output + "(maponly)");  
job.setJarByClass(ParseJson2.class);  
TableMapReduceUtil.initTableMapperJob(input, scan, ParseMapper.class,  
    ImmutableBytesWritable.class, Put.class, job);  
TableMapReduceUtil.initTableReducerJob(output,  
    IdentityTableReducer.class, job);  
job.setNumReduceTasks(0);  
...
```

用户可以从命令行展示的正在运行的作业中看出reduce阶段已经被跳过了：

```
$ hadoop jar target/hbase-book-ch07-1.0-job.jar ParseJson2 \  
-i testtable -c data:json -o testtable  
11/08/08 18:38:10 INFO mapreduce.TableOutputFormat: \  
Created table instance for testtable  
11/08/08 18:38:11 INFO mapred.JobClient: Running job: job_201108081021_0029  
11/08/08 18:38:12 INFO mapred.JobClient: map 0% reduce 0%  
11/08/08 18:38:20 INFO mapred.JobClient: map 100% reduce 0%  
11/08/08 18:38:22 INFO mapred.JobClient: Job complete: job_201108081021_0029  
...
```

reduce阶段一直显示为0%，直到作业完成。用户可以使用Hadoop MapReduce的UI来确认已执行的作业确实没有reduce阶段。跳过reduce阶段的优点是可以更快地完成作业，这是由于框架不需要增加额外的数据处理工作。

这两个不同的ParseJson作业的作用相同，用户可以通过HBase Shell查看其结果（由于展示空间的缘故，省略重复的行键）：

```

hbase(main):001:0> scan 'testtable'
...
\xFB!\Nn\x8F\x89}\xD8\x91+\xB9o9\xB3E\xD0
column=data:author, timestamp=1312821497945, value=bookrdr3
column=data:comments, timestamp=1312821497945,
  value=http://delicious.com/url/409839abddbce807e4db07bf7d9cd7ad
column=data:guidislink, timestamp=1312821497945, value=false
column=data:id, timestamp=1312821497945,
  value=http://delicious.com/url/409839abddbce807e4db07bf7d9cd7ad#bookrdr3
column=data:link, timestamp=1312821497945,
  value=http://sweetsassafras.org/2008/01/27/how-to-alter-a-wool-sweater
...
column=data:updated, timestamp=1312821497945,
  value=Mon, 07 Sep 2009 18:22:21 +0000
...
993 row(s) in 1.7070 seconds

```

这个导入过程利用了HBase支持任意列名的特点：JSON的键被转化为列限定符，并形成了新的一列。

7.2.5 自定义处理

用户并非必须使用HBase提供的类来读写表数据，实际上这些类非常轻量级并且仅起到了辅助类的作用。例7.5改变了前面的代码，以将已解析的JSON数据拆分到两张不同的目标表中，链接和它对应的值存储到了一张名为linktable的独立的表中，而其他列则存储到了名为infotable的表中。

例 7.5 MapReduce 作业解析每行数据为若干列

```

static class ParseMapper
extends TableMapper<ImmutableBytesWritable, Writable> {

    private HTable infoTable = null;
    private HTable linkTable = null;
    private JSONParser parser = new JSONParser();
    private byte[] columnFamily = null;

    @Override
    protected void setup(Context context)
    throws IOException, InterruptedException {
        infoTable = new HTable(context.getConfiguration(),
            context.getConfiguration().get("conf.infotable")); ❶
        infoTable.setAutoFlush(false);
        linkTable = new HTable(context.getConfiguration(),
            context.getConfiguration().get("conf.linktable"));
        linkTable.setAutoFlush(false);
        columnFamily = Bytes.toBytes(
            context.getConfiguration().get("conf.columnfamily"));
    }

    @Override
    protected void cleanup(Context context)

```



```

throws IOException, InterruptedException {
    infoTable.flushCommits();
    linkTable.flushCommits();❶
}

@Override
public void map(ImmutableBytesWritable row, Result columns, Context context)
throws IOException {
    context.getCounter(Counters.ROWS).increment(1);
    String value = null;
    try {
        Put infoPut = new Put(row.get());
        Put linkPut = new Put(row.get());
        for (KeyValue kv : columns.list()) {
            context.getCounter(Counters.COLS).increment(1);
            value = Bytes.toStringBinary(kv.getValue());
            JSONObject json = (JSONObject) parser.parse(value);
            for (Object key : json.keySet()) {
                Object val = json.get(key);
                if ("link".equals(key)) {
                    linkPut.add(columnFamily, Bytes.toBytes(key.toString()),
                        Bytes.toBytes(val.toString()));
                } else {
                    infoPut.add(columnFamily, Bytes.toBytes(key.toString()),
                        Bytes.toBytes(val.toString()));
                }
            }
        }
        infoTable.put(infoPut);❷
        linkTable.put(linkPut);
        context.getCounter(Counters.VALID).increment(1);
    } catch (Exception e) {
        e.printStackTrace();
        System.err.println("Error: " + e.getMessage() + ", Row: " +
            Bytes.toStringBinary(row.get()) + ", JSON: " + value);
        context.getCounter(Counters.ERROR).increment(1);
    }
}

public static void main(String[] args) throws Exception {
    ...
    conf.set("conf.infotable", cmd.getOptionValue("o"));❸
    conf.set("conf.linktable", cmd.getOptionValue("l"));
    ...
    Job job = new Job(conf, "Parse data in " + input + ", into two tables");
    job.setJarByClass(ParseJsonMulti.class);
    TableMapReduceUtil.initTableMapperJob(input, scan, ParseMapper.class,
        ImmutableBytesWritable.class, Put.class, job);
    job.setOutputFormatClass(NullOutputFormat.class);❹
    job.setNumReduceTasks(0);
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}

```

- ❶ 在`setup()`方法中创建并配置两张目标表。
- ❷ 当任务完成时，刷新所有挂起的提交。
- ❸ 保存已解析的数据到两张不同的表中。
- ❹ 在mapper的配置实例中存储表名以备后用。
- ❺ 设置框架会忽略的输出格式。



用户需要创建两张表，请使用 HBase Shell 进行以下操作：

```
hbase(main):001:0> create 'infotable', 'data'
hbase(main):002:0> create 'linktable', 'data'
```

在当前例子中，这两张表将会被用作目标存储表。

通过如下命令行执行作业，并忽略输出内容：

```
$ hadoop jar target/hbase-book-ch07-1.0-job.jar ParseJsonMulti \
-i testtable -c data:json -o infotable -l linktable
11/08/08 21:13:57 INFO mapred.JobClient: Running job: job_201108081021_0033
11/08/08 21:13:58 INFO mapred.JobClient: map 0% reduce 0%
11/08/08 21:14:06 INFO mapred.JobClient: map 100% reduce 0%
11/08/08 21:14:08 INFO mapred.JobClient: Job complete: job_201108081021_0033
...
```

到目前为止，这类似于之前解析JSON的示例，所不同的是结果表及其内容。用户可以在作业完成后使用HBase Shell和扫描命令罗列内容。用户可以从看出链接（link）表只包含了链接，而信息（info）表则包含了原JSON数据的剩余字段。

用户可以通过自行编写的MapReduce代码来控制作业的执行内容。例如，用户可以在存储组合结果的不同表中查询数据。用户从哪里读数据和向哪里写数据是不受限制的。HBase提供的类是辅助类，或多或少地服务了大量的用例。如果用户发现了它们功能上的限制，可以简单地拓展它们，或直接通过MapReduce的API实现可以以任何形式访问HBase的代码。

对于高级用户或乐于尝试的初级开发者来说，充分理解他们所选择系统的内部运行机制是一件很有益处的事情。本章将深入介绍 HBase 各个组成部分以及它们之间如何协同工作。

8.1 数据查找和传输

我们详细分析架构前，会首先介绍典型的 RDBMS 和其他非关系型数据库底层存储结构之间的不同。我们会重点介绍 B 树和 B+树^①，这两种数据结构被关系型存储引擎广泛采用，同时还有 LSM 树（Log-Structured Merge Tree）^②，它在某种程度上构成了 BigTable 的底层存储架构，这些曾在 1.4 节中讨论过。



请注意，RDBMS 所使用的存储架构并不局限于 B 树，也不是所有 NoSQL 解决方案采用的架构都与 RDBMS 不同。你将会看到各种各样的技术被组合到一起，同时它们又服务于一个共同的目标：为当前的问题提供最好的存储策略。

8.1.1 B+树

B+树的一些特性使其能够通过主键对记录进行高效插入、查找以及删除。它表示为一个动态、多层并有上下界的索引。同时要注意维护每一段（也被称作页表）所包含的主键数目。分段 B+树的效果远好于二叉树的数据划分，其大大减少了查询特定主键所需的 I/O 操作。

除此以外，B+树能够提供高效的范围扫描功能，这得益于它的叶节点相互连接并且按主键

^① 见维基百科中的“B+ trees”页面（http://en.wikipedia.org/wiki/B%2B_tree）。

^② 见 O'Neil 在 1996 年发表的论文“LSM 树”（<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.2782>）。

有序，扫描时避免了耗时的遍历树操作。这也是 B+树被关系型数据库用作索引的原因之一。

在一棵 B+树索引中，用户可以得到页表（在一些系统中被称作块）级别的位置信息，例如，叶节点页表如下：

```
[link to previous page]
[link to next page]
key1  →rowid
key2  →rowid
key3  →rowid
```

为了添加一个新的索引项 key1.5，需要更新叶节点页表并添加一个新的索引项 key1.5 并指向对应的 rowid。对于一个有固定大小的页表来说，页表大小超过容量限制时就会产生问题。这时需要将该页表拆分成两个新的页表，同时更新原页表的父节点，并使其指向刚创建的两个新节点。可以把图 8-1 当做一个例子，向图中的一个满页表添加新的键时会拆分页表。

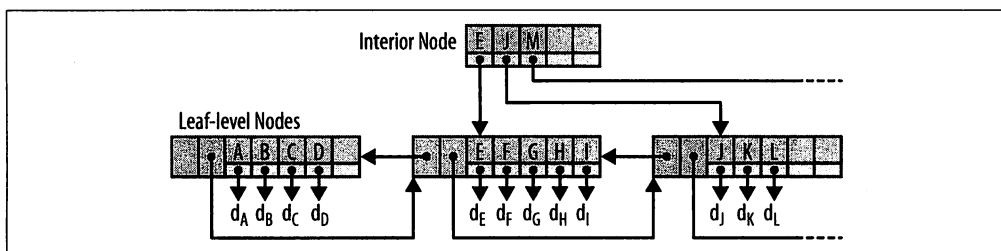


图 8-1 一个满页表的 B+树

现在的问题是，新建的两个页表在硬盘中并不一定是彼此相邻的。如果要进行一次从 key1 到 key3 的范围查询，则需要读取两个在磁盘上不连续甚至可能相隔很远的叶节点页表。这也是为什么我们在大部分基于 B+树的设计中都能找到一组被称为 OPTIMIZE TABLE（优化表）命令的原因，B+树这种数据组织方式只是简单地按顺序把表重写，从而使表的范围查询变成了磁盘的多段连续读取。

8.1.2 LSM 树

LSM 树（log-structured merge-tree）则按另一种方式组织数据。输入数据首先被存储在日志文件，这些文件内的数据完全有序。当有日志文件被修改时，对应的更新会先保存在内存中来加速查询。

当系统经历过许多次数据修改，且内存空间被逐渐被占满后，LSM 树会把有序的“键-记录”对写到磁盘中，同时创建一个新的数据存储文件。此时，因为最近的修改都被持久化了，内存中保存的最近更新就可以被丢弃了。

存储文件的组织与 B 树相似，不过其为磁盘顺序读取做了优化，所有节点都是满的并按页存储。修改数据文件的操作通过滚动合并完成，也就是说，系统将现有的页与内

存刷写数据混合在一起进行管理，直到数据块达到它的容量。

图 8-2 展示了在内存中多个块存储归并到磁盘的过程，合并写入会产生一个新的结果块，最终多个块被合并为更大块。

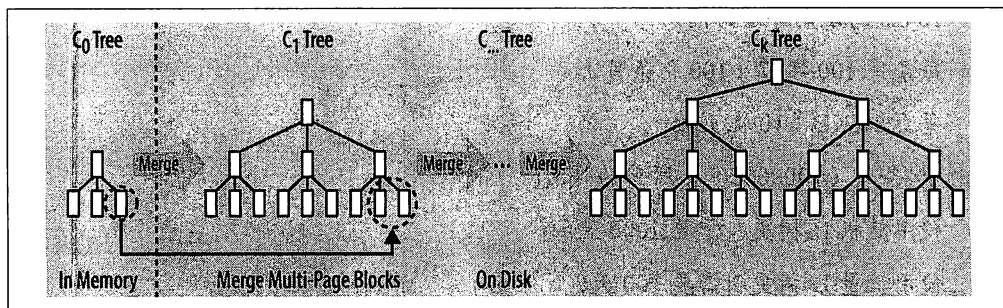


图 8-2 LSM 树中的多页数据块迭代合并的过程

多次数据刷写之后会创建许多数据存储文件，后台线程就会自动将小文件聚合成大文件，这样磁盘查找就会被限制在少数几个数据存储文件中。磁盘上的树结构也可以拆分成独立的小单元，这样更新就可以被分散到多个数据存储文件中。所有的数据存储文件都按键排序，所以没有必要在存储文件中为新的键预留位置。

查询时先查找内存中的存储，然后再查找磁盘上的文件。这样在客户端看来数据存储文件的位置是透明的。

删除是一种特殊的更改，当删除标记被存储之后，查找会跳过这些删除过的键。当页被重写时，有删除标记的键会被丢弃。

此外，后台运维过程可以处理预先设定的删除请求。这些请求由 TTL (time-to-live) 触发，例如，当 TTL 设为 20 天后，合并进程会检查这些预设的时间戳，同时在重写数据块时丢弃过期的记录。

B 树和 LSM 树最主要的区别在于它们的结构如何利用硬件，特别是磁盘。

查找与排序和合并的性能瓶颈^①

我们的大规模计算策略受制于磁盘传输，尽管 CPU、RAM 和磁盘大小每 18~24 个月翻一番，但数据查找的速度每年只能提高 5%。

如上面讨论的，在数据库中有两种范式，一种是利用存储的随机查找能力，另一种是利用存储的连续传输能力。随机查找在 RDBMS 中是由 B-树和 B+树数据结构组织。它工作的速度受制于磁盘的寻道速度，每次查找需要访问磁盘 $\log(N)$ 次。

^① 来自 Doug Cutting 在 2005 年 12 月 5 日发表的一篇论文 “Open Source Search” (<http://www.hajfa.ibm.com/Workshops/ir2005/papers/DougCutting-Hajfa05.pdf>)。

另一方面，存储的连续传输能力被 LSM 树使用，并以一定的传输速率排序和合并文件，需要执行 $\log(\text{updates})$ 次操作。在以下给定数值的情况下，其性能对比如下：

- 10 MB/s 的传输带宽；
- 10 ms 的磁盘寻道时间；
- 每条目 100 字节（100 亿条目）；
- 每页 10 KB（10 亿页）。

更新 1% 条目（100000000）所需的时间：

- 使用随机 B-树更新需要 1000 天；
- 使用批量 B-树更新需要 100 天；
- 使用排序和合并需要 1 天。

由此我们能断定，与利用存储的连续传输能力相比，大规模数据查找非常低效。

比较 B+树和 LSM 树的意义在于理解它们的相对优势和不足。在没有太多的修改时，B+树表现得很好，因为这些修改要求执行高代价的优化操作以保证查询能在有限时间内完成。在任意位置添加数据的规模越大、速度越快，这些页成为碎片的速度就越快。最后，用户写入的速度可能比优化后重写文件的处理速度更快。由于更新和删除以磁盘寻道的速率完成，这就强制用户就范于磁盘提供的较差的性能指标。

LSM 树以磁盘传输速率工作并能较好地扩展以处理大量的数据。它们使用日志文件和内存存储来将随机写转换成顺序写，因此也能保证稳定的数据插入速率。由于读和写独立，因此在这两种操作之间没有冲突。

由于存储数据的布局较优，查询一个键需要的磁盘寻道次数在一个可预测的范围内，并且读取与该键连续的任意数量的记录都不会引发任何额外的磁盘寻道。一般来说，基于 LSM 树的系统强调的是成本透明：假如有 5 个存储文件，一个访问需要最多 5 次磁盘寻道。反观关系型数据库，即使在存在索引的情况下，它也没有办法确定一次查询需要的磁盘寻道次数。

最终，HBase 与 BigTable 一样，都是基于 LSM 树的系统。下一节将解释存储架构，并将涉及本书之前章节中的内容。

8.2 存储

HBase 一个很少为人知的内容就是数据存储，因为大部分用户都不会接触到它的底层存

储结构，不过如果用户想使用一些进阶的配置选项就有必要了解这些内容了。第 11 章列举了一些常用的进阶配置，同时附录 A 中有一份完整的列表。

用户需要了解底层存储结构的另一个原因是，当某些原因导致系统出现重大问题时，用户可能需要恢复 HBase 中的数据。此时用户就需要知道数据存在何处，以及如何从 HDFS 中访问这些内容。当然，这种情况不应当发生，但在实际的系统中，任何异常情况都是可能的。

8.2.1 概览

了解 HBase 存储层中大量移动块的第一步是画一个顶层结构图。图 8-3 展示了 HBase 是如何与 Hadoop 文件系统协作完成数据存储的。

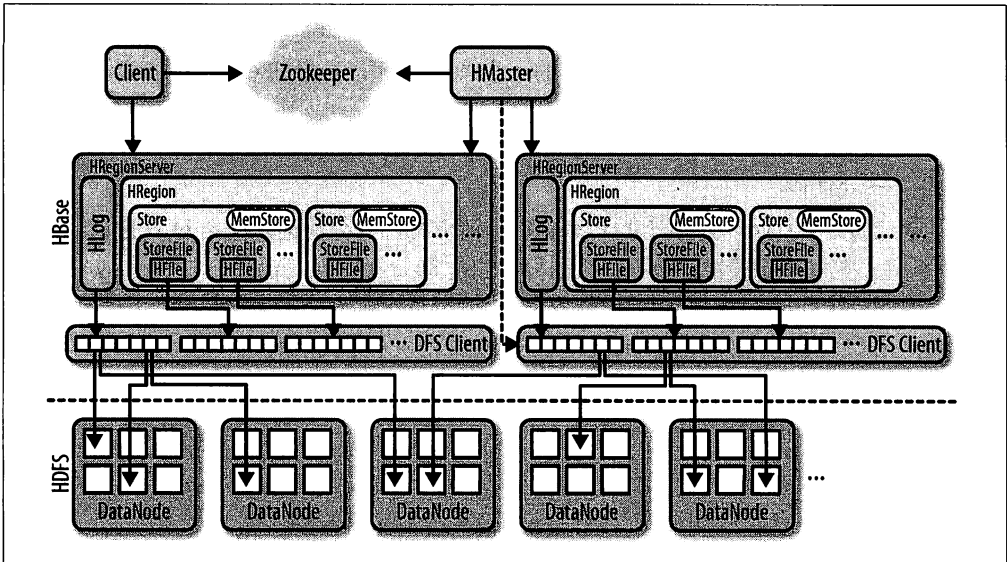


图 8-3 HBase 如何透明地操作存储在 HDFS 上文件的概览

从这张图可以看出 HBase 主要处理两种文件：一种是预写日志（Write-Ahead Log, WAL），另一种是实际的数据文件。这两种文件主要由 HRegionServer 管理。在某些情况下，HMaster 也可以进行一些底层的文件操作（在 0.92.0 中与 0.90.x 中稍有不同）。当存储数据到 HDFS 中时，用户可能注意到实际的数据文件会被切分成更小的块。也正是这一点，用户可以配置系统来更好地处理较大或较小的文件。更多的信息请参阅 8.2.4 节。

一个基本的流程是客户端首先联系 ZooKeeper 子集群（quorum）（一个由 ZooKeeper 节点组成的单独集群）查找行键。上述过程是通过 ZooKeeper 获取含有 -ROOT- 的 region 服务器名（主机名）来完成的。通过含有 -ROOT- 的 region 服务器可以查询到含有 .META. 表中对应的 region 服务器名，其中包含请求的行键信息。这两处的主要内容都被缓存

下来了，并且都只查询一次。最终，通过查询.META.服务器来获取客户端查询的行键数据所在 region 的服务器名。

一旦知道了数据的实际位置，即 region 的位置，HBase 会缓存这次查询的信息，同时直接联系管理实际数据的 HRegionServer。所以，之后客户端可以通过缓存信息很好地定位所需的数据位置，而不用再次查找.META.表，参见 8.5 节。



启动 HBase 时，HMaster 负责将所有 region 分配到每个 HRegion Server 上，其中也包括特别的 -ROOT- 和 .META. 表，参见 8.6 节。

HRegionServer 负责打开 region，并创建对应的 HRegion 实例。当 HRegion 被打开后，它会为每个表的 HColumnFamily 创建一个 Store 实例，这些列族是用户之前创建表时定义的。每个 Store 实例包含一个或多个 StoreFile 实例，它们是实际数据存储文件 HFile 的轻量级封装。每个 Store 还有其对应的一个 MemStore，一个 HRegionServer 分享了一个 HLog 实例（参见 8.3 节）。

8.2.2 写路径

当用户向 HRegionServer 发起 HTable.put (Put) 请求时，其会将请求交给对应的 HRegion 实例来处理。第一步是要决定数据是否需要写到由 HLog 类实现的预写日志中。^①WAL 是标准的 Hadoop SequenceFile，并且存储了 HLogKey 实例。这些键包括序列号和实际数据，所以在服务器崩溃时可以回滚还没有持久化的数据。

一旦数据被写入到 WAL 中，数据就会被放到 MemStore 中。同时还会检查 MemStore 是否已经满了，如果满了，就会被请求刷写到磁盘中去。刷写请求由另外一个 HRegionServer 的线程处理，它会把数据写成 HDFS 中的一个新 HFile。同时也会保存最后写入的序号，系统就知道哪些数据现在被持久化了。

关闭前预刷写

memstore 被刷写到磁盘的第二个理由是：预刷写 (prefushing)。当 region 服务器被要求关闭时，会首先检查 memstore，任何大于配置值 hbase.hregion.preclose.flush.size（默认值为 5 MB）的 memstore 会刷写到磁盘，然后在最后一轮阻塞正常访问的刷写后关闭 region。

另一方面，关闭 region 服务器会强制所有的 memstore 被刷写到磁盘，而不会关心 memstore 是否达到了配置的最大值，可以使用配置项 hbase.hregion.memstore.flush.size（默认值为 64 MB）或者通过创建表（查看 5.1.2 节的“文件大小限制”）

^① 在特殊情况下，用户可以通过 Put.setwriteToWAL (boolean) 方法关闭该步骤，但并不推荐禁用持久。

来进行设置。一旦所有的 memstore 都被刷写到了磁盘, region 会被关闭, 且在转移到其他 region 服务器时不会重做 WAL。

使用额外的一轮预刷写会提高 region 的可用性: 在预刷写时, 服务器与 region 仍旧可用, 这类似于通过 API 或 Shell 命令调用刷写 (flush)。当剩下的比较小的 memstore 完成了第二轮刷写时, 此时会停止所有请求。这一轮刷写会保存预刷写过程中的所有修改, 以保证服务器可以干净地退出。

8.2.3 文件

HBase 使用一个 HDFS 中可配置的根目录, 默认设为 "/hbase"。12.3.1 节展示了如何使用不同根目录来让多个 HBase 集群共享 HDFS。用户可以使用 `hadoop dfs -lsr` 命令来查看 HBase 的目录结构。在这之前, 我们先建立一个包含多个 region 的表。

```
hbase(main):001:0> create 'testtable','colfam1',\
  { SPLITS => ['row-300','row-500','row-700','row-900'] }
0 row(s) in 0.1910 seconds

hbase(main):002:0> for i in '0'..'9' do for j in '0'..'9' do \
  for k in '0'..'9' do put 'testtable','row-#{i}#{j}#{k}',\
    "colfam1:#{j}#{k}", "#{j}#{k}" end end end
0 row(s) in 1.0710 seconds
0 row(s) in 0.0280 seconds
0 row(s) in 0.0260 seconds
...

hbase(main):003:0> flush 'testtable'
0 row(s) in 0.3310 seconds

hbase(main):004:0> for i in '0'..'9' do for j in '0'..'9' do \
  for k in '0'..'9' do put 'testtable','row-#{i}#{j}#{k}',\
    "colfam1:#{j}#{k}", "#{j}#{k}" end end end
0 row(s) in 1.0710 seconds
0 row(s) in 0.0280 seconds
0 row(s) in 0.0260 seconds
...
```

`flush` 命令可以将内存中的数据写到存储文件中, 否则就必须等插入的数据达到配置的刷写大小。最后的循环使用 `put` 命令来再次填充 WAL。经过这些操作后, HBase 的根目录结构如下:

```
$ $HADOOP_HOME/bin/hadoop dfs -lsr /hbase
...
0 /hbase/.logs
0 /hbase/.logs/foo.internal,60020,1309812147645
```

```

0 /hbase/.logs/foo.internal,60020,1309812147645/ \
foo.internal%2C60020%2C1309812147645.1309812151180
0 /hbase/.oldlogs
38 /hbase/hbase.id
3 /hbase/hbase.version
0 /hbase/testtable
487 /hbase/testtable/.tableinfo
0 /hbase/testtable/.tmp
0 /hbase/testtable/1d562c9c4d3b8810b3dbeb21f5746855
0 /hbase/testtable/1d562c9c4d3b8810b3dbeb21f5746855/.oldlogs
124 /hbase/testtable/1d562c9c4d3b8810b3dbeb21f5746855/.oldlogs/ \
hlog.1309812163957
282 /hbase/testtable/1d562c9c4d3b8810b3dbeb21f5746855/.regioninfo
0 /hbase/testtable/1d562c9c4d3b8810b3dbeb21f5746855/.tmp
0 /hbase/testtable/1d562c9c4d3b8810b3dbeb21f5746855/colfam1
11773 /hbase/testtable/1d562c9c4d3b8810b3dbeb21f5746855/colfam1/ \
646297264540129145
0 /hbase/testtable/66b4d2adcc25f1643da5e6260c7f7b26
311 /hbase/testtable/66b4d2adcc25f1643da5e6260c7f7b26/.regioninfo
0 /hbase/testtable/66b4d2adcc25f1643da5e6260c7f7b26/.tmp
0 /hbase/testtable/66b4d2adcc25f1643da5e6260c7f7b26/colfam1
7973 /hbase/testtable/66b4d2adcc25f1643da5e6260c7f7b26/colfam1/ \
3673316899703710654
0 /hbase/testtable/99c0716d66e536d927b479af4502bc91
297 /hbase/testtable/99c0716d66e536d927b479af4502bc91/.regioninfo
0 /hbase/testtable/99c0716d66e536d927b479af4502bc91/.tmp
0 /hbase/testtable/99c0716d66e536d927b479af4502bc91/colfam1
4173 /hbase/testtable/99c0716d66e536d927b479af4502bc91/colfam1/ \
1337830525545548148
0 /hbase/testtable/d240e0e57dcf4a7e11f4c0b106a33827
311 /hbase/testtable/d240e0e57dcf4a7e11f4c0b106a33827/.regioninfo
0 /hbase/testtable/d240e0e57dcf4a7e11f4c0b106a33827/.tmp
0 /hbase/testtable/d240e0e57dcf4a7e11f4c0b106a33827/colfam1
7973 /hbase/testtable/d240e0e57dcf4a7e11f4c0b106a33827/colfam1/ \
316417188262456922
0 /hbase/testtable/d9ffc3a5cd016ae58e23d7a6cb937949
311 /hbase/testtable/d9ffc3a5cd016ae58e23d7a6cb937949/.regioninfo
0 /hbase/testtable/d9ffc3a5cd016ae58e23d7a6cb937949/.tmp
0 /hbase/testtable/d9ffc3a5cd016ae58e23d7a6cb937949/colfam1
7973 /hbase/testtable/d9ffc3a5cd016ae58e23d7a6cb937949/colfam1/ \
4238940159225512178

```



由于篇幅原因，输出结果被精简到只有目录及文件的大小。用户自己执行这些命令时可以看到更多的细节。

文件可以被分为两类，一类位于 HBase 根目录下，另一类位于根目录中的表目录下。

1. 根级文件

第一组文件是被 HLog 实例管理的 WAL 文件，这些日志文件被创建在 HBase 的根目录下一个名为 *logs* 的目录中。对于每个 HRegionServer，日志目录中都包含一个对应的子目录。在每个子目录中有多个 HLog 文件（因为日志滚动）。一个 region 服务器的所有 region 共享同一组 HLog 文件。

一个有趣的现象是，由于文件最近才被创建，所以日志文件大小被报告为 0。这个现象很典型，由于 HDFS 使用内置的 *append* 机制来追加写入此文件，所以只有等到文件大小达到一个完整的块时，文件对用户才是可见的——包括 *hadoop dfs-lsr* 命令。虽然 put 操作的数据被安全持久化了，但当前写入日志文件的数据大小是稍微偏离的。

在等待一个小时后，日志文件被滚动（参见 8.3.6 节），这个时间是由 `hbase.regionserver.logroll.period` 配置属性（默认设置为 60 分钟）控制的。此时，由于日志文件被关闭，HDFS 能列出其正确的大小，紧接着它的下一个新日志文件的大小又从 0 开始了：

```
249962 /hbase/.logs/foo.internal,60020,1309812147645/ \
foo.internal%2C60020%2C1309812147645.1309812151180
0 /hbase/.logs/foo.internal,60020,1309812147645/ \
foo.internal%2C60020%2C1309812147645.1309815751223
```

当所有包含的修改都被持久化到存储文件中，从而不再需要日志文件时，它们会被放到 HBase 根目录下的 *oldlogs* 目录中。当条件满足配置上的阈值会触发日志的滚动。在 10 分钟（默认情况下）后，旧的日志文件将被 master 删除，这是通过 `hbase.master.logcleaner.ttl` 属性设置的。master 每分钟（默认情况下）检查一次这些文件，这是通过 `hbase.master.cleaner.interval` 属性设置的。



过期的日志文件可以及时被移除。复制特性（查看 8.8 节）可以利用这种行为来访问仍然存在的修改。

hbase.id 和 *hbase.version* 文件包含集群的唯一 ID 和文件格式版本信息。

```
$ hadoop dfs -cat /hbase/hbase.id
$e627e130-0ae2-448d-8bb5-117a8af06e97
$ hadoop dfs -cat /hbase/hbase.version
7
```

它们在内部使用，且携带的信息不多。此外，随着时间的推移还会有更多根级别的目录出现。*splitlog* 和 *corrupt* 文件夹分别被用来存储日志拆分过程中产生的中间拆分文件和损坏的日志。例如：

```

0 /hbase/.corrupt
0 /hbase/splitlog/foo.internal,60020,1309851880898_hdfs%3A%2F%2F \
localhost%2Fhbase%2F.logs%2Ffoo.internal%2C60020%2C1309850971208%2F \
foo.internal%252C60020%252C1309850971208.1309851641956/testtable/ \
d9ffc3a5cd016ae58e23d7a6cb937949/recovered.edits/00000000000000002352

```

这个例子中没有损坏的日志文件，所以仅显示了一个中间拆分文件。日志拆分过程在 8.3.7 节中有解释。

2. 表级文件

在 HBase 中，每张表都有自己的目录，其位于文件系统中 HBase 根目录下。每张表目录包括一个名为 *tableinfo* 的顶层文件，该文件存储表对应序列化后的 *HTableDescriptor*（详情请查看 5.1.1 节）。其中包括表和列族的定义，同时其内容也可以被读取，例如，用户可以使用一些工具查看表的大致结构。*.tmp* 目录中包含一些临时数据，例如，当更新 *tableinfo* 文件时生成的临时数据就会被存放到该目录中。

3. region 级文件

在每张表的目录里面，表模式中每个列族都有一个单独的目录。目录的名字是一部分 *region* 名字的 MD5 散列值。例如，下面的内容是在点击 *master* 网页界面中用户表区域的 *testtable* 链接时得到的。

```
testtable,row-500,1309812163930.d9ffc3a5cd016ae58e23d7a6cb937949.
```

MD5 散列值是 *d9ffc3a5cd016ae58e23d7a6cb937949*，它是通过编码 *region* 名字得到的，例如，*testtable,row-500,1309812163930.*，末尾的点是这个 *region* 名字的一部分：它表示这是一个包含散列值的新样式名字，在以前的 HBase 版本中，*region* 名字不包含散列值。



注意，*-ROOT-*和 *.META.*这些目录表用的依然是旧命名格式，例如，它们的 *region* 名字不包括散列值，因此末尾不是以点结束的：

```
.META.,,1.1028785192
```

它们在磁盘目录的 *region* 名字的编码也不同：它们用 *Jenkins* 散列值来编码 *region* 的名字。

散列能保证目录名称不违背文件系统的命名规则：它们不能包含任何特殊字符，例如，用来划分路径的斜线（“/”）。*region* 文件的总体结构是：

```
/<hbase-root-dir>/<tablename>/<encoded-regionname>/<column-family>/<filename>
```

用户可以在每个列族目录中看到实际的数据文件，在 8.2.4 节有更详细的介绍。这些文

件的名字仅仅是一个由 Java 内置的随机数生成器生成的随机数。代码能够智能地检查出发生的碰撞，即新生成号码对应的文件已经存在。程序将一直循环，直到找到一个未使用的数字。

`region` 目录中也有一个 `.regioninfo` 文件，这个文件包含了对应 `region` 的 `HRegionInfo` 实例序列化后的信息。与 `tableinfo` 文件类似，它能被外部工具用来查看 `region` 的相关信息。例如，HBase 的 `hbck` 工具就用它来检查并生成元数据表中丢失的条目。

可选的 `.tmp` 目录是按需求创建的，它被用来存放临时文件，例如，一次合并的重写文件。一旦这个过程完成，这些临时生成的文件通常会被移到 `region` 目录中。在极少的情况下，用户可能发现遗留的文件，这些文件将在 `region` 重新打开时被清理掉。

在 WAL 回放时，任何未提交的修改都会被写入到每个 `region` 的一个单独的文件中。以上是第一步（查看本节的“Root 级文件”的 `splitlog` 目录），然后如果日志拆分过程已经成功完成，这些文件将被自动移动到临时的 `recovered.edits` 目录中。当 `region` 被打开时，`region` 服务器将会看到需要恢复的文件，并且回放其中相应的条目。



WAL 的拆分（参见 8.3.7 节）和 `region` 的拆分（参见本节的“`region` 拆分”）有明显的区别，有时候很难区分文件系统中的文件名和目录名，因为它们的名称中都有拆分字样。用户需要仔细辨别它们的目的来避免混淆或者错误。

一旦 `region` 超过了配置中 `region` 大小的最大值，`region` 就需要拆分，它会创建一个对应的 `splits` 目录，它被用来临时存放两个子 `region` 相关的数据。如果拆分过程成功（通常这个过程持续几秒或更短时间），之后它们会被移动到表目录中，并形成两个新的 `region`，每个 `region` 代表原始 `region` 的一半。

换言之，当用户看见一个 `region` 的目录中没有 `.tmp` 目录，这就意味着还没有进行过压缩操作。如果没有 `recovered.edits` 目录，这就意味着 WAL 还没有进行过回放操作。



在 HBase0.90.x 之前的版本中还有些额外的文件，这些文件现在已经不再使用了。一个是 `oldlogfile.log` 文件，它包含给定 `region` 已经回放过的 WAL。`oldlogfile.log.old` 文件（注意额外的 `.old` 后缀名）表示当一个新的 `oldlogfile.log` 被放进来时已经存在一个旧的文件。

在旧版本的 HBase 中，另外一个值得注意的文件是 `compaction.dir`，它现在已经被 `.tmp` 目录代替。

以上总结了在 HBase 根文件夹中经常出现的各种目录的列表。`region` 拆分的过程还会

产生一些中间文件，这些文件将在下一节单独讨论。

4. region 拆分

当一个 region 里的存储文件增长到大于配置的 `hbase.hregion.max.filesize` 大小或者在列族层面配置的大小时，region 会被一分为二。这个过程通常非常迅速，因为系统只是为新 region（也称作孩子）创建了两个对应的文件，每个 region 是原始 region（称作父亲）的一半。

region 服务器通过在父 region 中创建 *splits* 目录来完成这个过程。接下来关闭该 region，此后这个 region 不再接受任何请求。

然后 region 服务器通过在 *splits* 目录中设立必需的文件结构来准备新的子 region（使用多线程），包括新 region 目录和参考文件。如果这个过程成功完成，它将把两个新 region 目录移到表目录中。`.META.`表中父 region 的状态会被更新，以表示其现在拆分的节点和子节点是什么。以上过程可以避免父 region 被意外重新打开。`.META.`表中的内容大致如下：

```
row: testtable,row-500,1309812163930.d9ffc3a5cd016ae58e23d7a6cb937949.

column=info:regioninfo,timestamp=1309872211559,value=REGION => {NAME => \
'testtable,row-500,1309812163930.d9ffc3a5cd016ae58e23d7a6cb937949. \
TableName => 'testtable',STARTKEY => 'row-500',ENDKEY => 'row-700',\
ENCODED => d9ffc3a5cd016ae58e23d7a6cb937949,OFFLINE => true,
SPLIT => true,}

column=info:splitA,timestamp=1309872211559,value=REGION => {NAME => \
'testtable,row-500,1309872211320.d5a127167c6e2dc5106f066cc84506f8. \
TableName => 'testtable',STARTKEY => 'row-500',ENDKEY => 'row-550',\
ENCODED => d5a127167c6e2dc5106f066cc84506f8,}

column=info:splitB,timestamp=1309872211559,value=REGION => {NAME => \
'testtable,row-550,1309872211320.de27e14ffcf1f3fff65ce424fcf14ae42. \
TableName => [B@62892cc5',STARTKEY => 'row-550',ENDKEY => 'row-700',\
ENCODED => de27e14ffcf1f3fff65ce424fcf14ae42,}
```

由此可见，原始 region 在 row-550 如何被拆分成两个 region。在 `info:regioninfo` 列值中的 `SPLIT => true` 也表示 region 现在正在拆分成叫做 `info:splitA` 和 `info:splitB` 的 region。

引用文件的名字是另外一个随机数字，不过其后带有被引用的 region 的散列值作为后缀，例如：

```
/hbase/testtable/d5a127167c6e2dc5106f066cc84506f8/colfam1/ \
6630747383202842155.d9ffc3a5cd016ae58e23d7a6cb937949
```

上面所示的引用文件代表散列值为 d9ffc3a5cd016ae58e23d7a6cb937949 的原始 region 的一半，这个 region 也出现在上面的例子中。参考文件仅包含很少的信息：原始 region 被拆分处的键以及它是否为表顶部或者底部的引用文件。值得注意的是，这些引用文件后面会被 HalfHFileReader 类（这个在前面的概述中被省略了，因为它只是短暂地被使用）用来读原始 region 的数据文件。

两个子 region 都准备好后，将会被同一个服务器并行打开。打开的过程包括更新 .META. 表，这样可以把两个 region 像其他 region 一样作为可用 region 列出来。之后这两个 region 会上线并开始服务请求。

同时也会初始化为两个 region 并对 region 中的内容执行合并，合并过程在替换引用文件之前会把父 region 的存储文件异步重写到两个子 region 中。以上过程会在子 region 的 .tmp 目录中执行。一旦生成了重写之后的文件，它们将自动取代引用文件。

最终父 region 会被清理掉，这意味着它在 .META. 表中的表项会被移除，并且它在磁盘上所有的文件都会被删除。最后，master 被告知关于拆分的情况，并且可以由于负载均衡而把新 region 移动到其他 region 服务器上。



在拆分过程中，所有的步骤都在 ZooKeeper 中进行跟踪。这使得在一个服务器失效时，其他进程可以知道这个 region 的状态。

5. 合并

存储文件会被后台的管理进程仔细地监控起来以确保它们处于控制之下。随着 memstore 的刷写会生成很多磁盘文件。如果文件的数目达到阈值，合并（compaction）过程将把它们合并成数量更少的体积更大的文件。这个过程持续到这些文件中最大的文件超过配置的最大存储文件大小，此时会触发一个 region 拆分。

压缩合并有两种，即 minor 和 major。minor 合并负责重写最后生成的几个文件到一个更大的文件中。文件数量是由 hbase.hstore.compaction.min（之前被称做 hbase.hstore.compactionThreshold，尽管已经废弃了，但是这个属性依然可用）属性设置的。它的默认值为 3，并且最小值需要大于或等于 2。过大的数字将会延迟 minor 合并的执行，同时也会增加执行时消耗的资源及执行的时间。

minor 合并可以处理的最大文件数量默认为 10，用户可以通过 hbase.hstore.compaction.max 来配置。hbase.hstore.compaction.min.size（默认设置为 region 的 memstore 刷写大小）和 hbase.hstore.compaction.max.size（默认设置为 Long.MAX_VALUE）配置项属性进一步减少了需要合并的文件列表。任何比最大合并大小大的文件都会被排除在外。最小合并大小的功能稍有不同：它是一个阈值，而不是每个文件的限制。它包括所有小于

限制的文件，直到达到每次压缩允许的总文件数量。

图 8-4 显示了所有小于最小压缩阈值的文件都被包括在了压缩过程中。

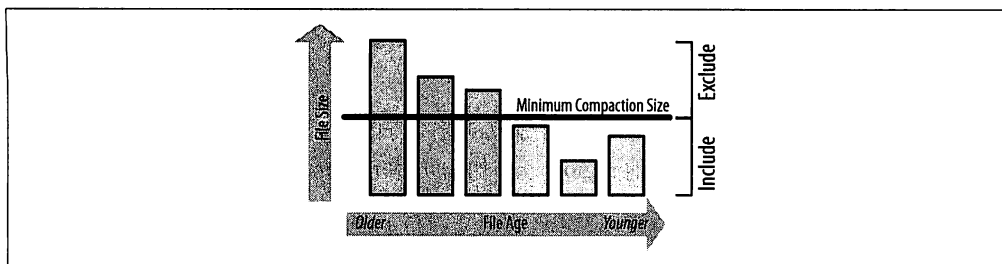


图 8-4 最小压缩阈值限制下的文件集

算法使用 `hbase.hstore.compaction.ratio`（默认为 1.2，或者 120%）来确保在选择过程中包括足够的文件。经过跟新文件总的存储文件比较之后，这个比例仍将选择达到那个值的文件。评估总是按照从老文件到新文件的顺序来进行的，这样可以确保更老的文件首先被合并。这些属性的组合允许用户微调一个 `minor` 合并包括文件的数量。

HBase 支持的另外一种合并是 `major` 合并：它们把所有文件压缩成一个单独的文件。在执行压缩检查时，系统自动决定运行哪种合并。在 `memstore` 被刷写到磁盘后会触发检查，或在 Shell 命令 `compact`、`major_compact` 之后触发检查，或是相应 API 在被调用后触发检查，抑或是被一个异步的后台进程触发后。`region` 服务器运行这个线程，而其功能由 `CompactionChecker` 类实现，它以一个固定的周期触发检查，这个周期由 `hbase.server.thread.wakefrequency` 参数控制（乘以 `hbase.server.thread.wakefrequency.multiplier`，设为 1000，这样它的执行频率不会像其他基于线程的任务那么频繁）。

除非用户使用 Shell 命令 `major_compact` 或者使用 `majorCompact()` 这个 API，将强制运行 `major` 合并，否则服务器会首先检查上次运行到现在是否达到 `hbase.hregion.majorcompaction`（设为 24 小时）指定的时限。`hbase.hregion.majorcompaction.jitter`（设为 0.2，即 20%）参数会将所有存储的时间周期分开。如果没有这个抖动，所有的存储文件都将在每 24 小时的同一时间运行一个 `major` 合并。参见 11.4.1 节，用户可了解以上方法的问题以及如何更好地进行管理。

如果还没有达到 `major` 合并的执行周期，系统会选择 `minor` 合并执行。基于以上配置属性，服务器将检查是否有足够的文件供 `minor` 合并执行，如果有就继续。

当 `minor` 合并包括所有的存储文件，且所有文件均未达到设置的每次压缩的最大文件数时，`minor` 合并可能被提升成 `major` 合并。

8.2.4 HFile 格式

实际的存储文件功能是由 HFile 类实现的，它被专门创建以达到一个目的：有效地存储 HBase 的数据。它们基于 Hadoop 的 TFile 类^①，并模仿-Google 的 BigTable 架构使用的 SSTable 格式。曾在 HBase 中使用过的 Hadoop 的 MapFile 类被证明性能不够好。图 8-5 显示了文件格式的详细信息。

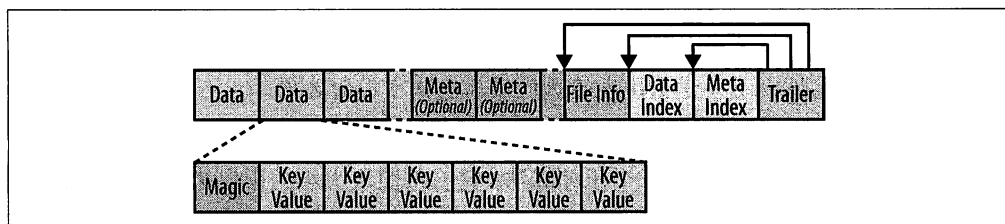


图 8-5 HFile 结构

这些文件是可变长度的，唯一固定的块是 *File Info* 块和 *Trailer* 块。如图 8-5 所示，*Trailer* 有指向其他块的指针。它是在持久化数据到文件结束时写入的，写入后即确定其成为不可变的数据存储文件。*Index* 块记录 *Data* 和 *Meta* 块的偏移量。*Data* 和 *Meta* 块实际上都是可选的，但是考虑到 HBase 如何使用数据文件，在存储文件中用户几乎总能找到 *Data* 块。

块大小是由 HColumnDescriptor 配置的，而该配置可以在创建表时由用户指定或者使用比较合理的默认值。下面是在 master 的 Web 界面中显示的例子：

```
{NAME => 'testtable', FAMILIES => [{NAME => 'colfam1',  
  BLOOMFILTER => 'NONE', REPLICATION_SCOPE => '0', VERSIONS => '3',  
  COMPRESSION => 'NONE', TTL => '2147483647', BLOCKSIZE => '65536',  
  IN_MEMORY => 'false', BLOCKCACHE => 'true'}]}
```

这里的默认值是 64 KB（或 655 356 字节）。下面是 HFile 的 JavaDoc 解释：

“块大小的最小值。对于一般的应用，我们建议将最小的块大小设置为 8 KB ~ 1 MB。如果应用主要涉及顺序访问，较大的块大小将更加合适。不过这会降低随机读性能（因为需要解压缩更多的数据）。较小的块更有利于随机数据访问，不过同时也需要更多的内存来存储块索引，并且可能创建过程也会变得更慢（因为我们必须在每个数据块结束的时候刷写压缩流，这会导致了一个 FS I/O 刷写）。此外，由于压缩解码器存在内部缓存，导致可能的最小的块大小是 20 KB ~ 30 KB。”

每个块都包含一个 *magic* 头部和一定数量的序列化的 KeyValue 实例（详见 8.2.5 节，并查看它们的格式）。如果用户没有使用压缩算法，每个块大小和配置的块大小差不多。但

① 见社区官方问题跟踪记录 HADOOP-3315 的细节（<http://issues.apache.org/jira/browse/HADOOP-3315>）。

这并不是什么精密科学，写入程序必须适应用户写入的数据：如果用户存储了一个比块大小更大的 Key/Value 实例，则 HBase 也必须接受它。不过即使是较小的值，对于块大小的检查也是在最后一个值写入后才进行的，所以在实际情况中，大部分块会稍大。

当使用压缩算法时，用户对于块大小的控制力将更弱。压缩解码器在能够自己控制获取的数据量时才能达到最有效的压缩比率。例如，把块大小设置为 256 KB，并使用 LZO 压缩算法，系统将写更小的块来适应 LZO 的内部缓冲区大小。



很多压缩库有一系列参数，用户可以用来指定缓冲区大小和其他更具体的属性。请参考 JNI 库的源代码来查找可用的选项。

HBase 不知道用户是否选择了一个压缩算法：它将按照块大小的限制来写原始数据，并尽量让原始数据的大小与这个限制接近。如果用户启用了压缩，则保存到磁盘上的数据将更少。这意味着最终的存储文件由相同数量的块组成，但是由于每一个块都更小，所以总大小也更小。

在 HDFS 中，文件的默认块大小是 64 MB，这个是 HFile 默认块大小的 1024 倍。因此 HBase 存储文件的块与 Hadoop 的块之间没有匹配关系。事实上，这两种块类型之间根本没有相关性。HBase 把它的文件透明地存储到文件系统中，而 HDFS 也使用块来切分文件仅仅是一个巧合，并且 HDFS 不知道 HBase 存储的是什么，它只能看到二进制文件。图 8-6 展示了 HFile 的内容怎样在整个 HDFS 块中进行分布。

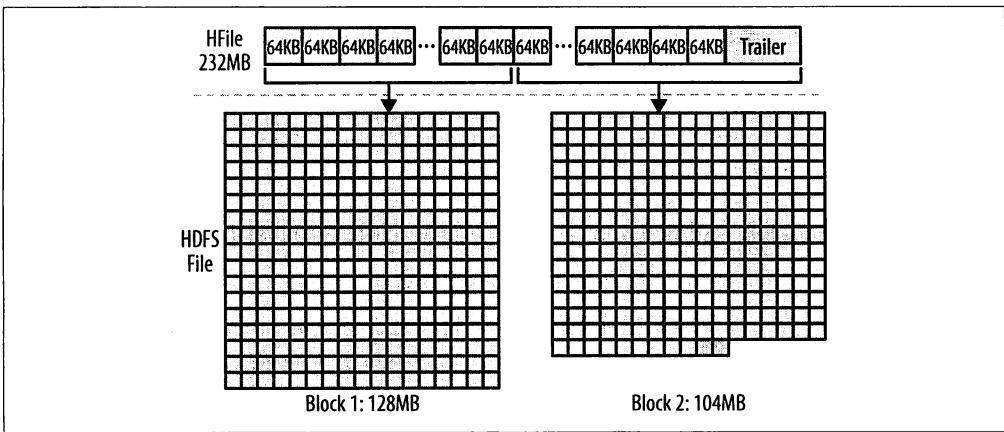


图 8-6 很多更小的 HFile 块透明地存储在两个更大的 HDFS 块中

有时候，用户有必要绕过 HBase 并直接访问一个 HFile，例如，检查它的健康程度，或者转存它的内容。HFile.main() 方法提供了这样的工具：

```
$ ./bin/hbase org.apache.hadoop.hbase.io.hfile.HFile
usage: HFile [-a] [-b] [-e] [-f <arg>] [-k] [-m] [-p] [-r <arg>] [-v]
```

```

-a,--checkfamily      Enable family check
-b,--printblocks      Print block index meta data
-e,--printkey         Print keys
-f,--file <arg>       File to scan. Pass full-path;e.g.
                        hdfs://a:9000/hbase/.META./12/34
-k,--checkrow         Enable row order check;looks for out-of-order keys
-m,--printmeta        Print meta data of file
-p,--printkv          Print key/value pairs
-r,--region <arg>    Region to scan. Pass region name;e.g. '.META.,,1'
-v,--verbose          Verbose output;emits file and meta data delimiters

```

上面的例子展示了输出的形式（精简过）：

```

$ ./bin/hbase org.apache.hadoop.hbase.io.hfile.HFile -f \
/hbase/testtable/de27e14ffc1f3fff65ce424fcf14ae42/colfam1/2518469459313898451 \
-v -m -p
Scanning -> /hbase/testtable/de27e14ffc1f3fff65ce424fcf14ae42/colfam1/2518469459313898451
K: row=550/colfam1:50/1309813948188/Put/vlen=2 V: 50
K: row=550/colfam1:50/1309812287166/Put/vlen=2 V: 50
K: row=551/colfam1:51/1309813948222/Put/vlen=2 V: 51
K: row=551/colfam1:51/1309812287200/Put/vlen=2 V: 51
K: row=552/colfam1:52/1309813948256/Put/vlen=2 V: 52
...
K: row=698/colfam1:98/1309813953680/Put/vlen=2 V: 98
K: row=698/colfam1:98/1309812292594/Put/vlen=2 V: 98
K: row=699/colfam1:99/1309813953720/Put/vlen=2 V: 99
K: row=699/colfam1:99/1309812292635/Put/vlen=2 V: 99
Scanned kv count -> 300
Block index size as per heapsize: 208
reader=/hbase/testtable/de27e14ffc1f3fff65ce424fcf14ae42/colfam1/ \
2518469459313898451,compression=none,inMemory=false,\
firstKey=row=550/colfam1:50/1309813948188/Put,\
lastKey=row=699/colfam1:99/1309812292635/Put,avgKeyLen=28,avgValueLen=2,\
entries=300,length=11773
fileinfoOffset=11408,dataIndexOffset=11664,dataIndexCount=1,\
metaIndexOffset=0,metaIndexCount=0,totalBytes=11408,entryCount=300,\
version=1
Fileinfo:
MAJOR_COMPACTION_KEY = \xFF
MAX_SEQ_ID_KEY = 2020
TIMERANGE = 1309812287166....1309813953720
hfile.AVG_KEY_LEN = 28
hfile.AVG_VALUE_LEN = 2
hfile.COMPARATOR = org.apache.hadoop.hbase.KeyValue$KeyComparator
hfile.LASTKEY = \x00\x07row=699\x07colfam199\x00\x00\x010\xF6\xE5|\xB\x04
Could not get bloom data from meta block

```

输出的第一部分是序列化的 KeyValue 实例所存储的真实数据。第二部分转存内部的

HFile.Reader 属性和 trailer 块的详细信息。最后一个部分以 Fileinfo 开头，是 file info 块的值。

这里提供的信息是有价值的，例如，用来确认一个文件是否被压缩过以及所使用的压缩类型。它也会显示已经存储了多少个单元，它们的键和值的平均大小。在这个例子中，上面创建的键比值大得多。这是因为 KeyValue 类需要存储许多相关数据，下面再进一步解释。

8.2.5 KeyValue 格式

本质上，HFile 中的每个 KeyValue 都是一个低级的字节数组，它允许零复制访问数据。图 8-7 显示了所包含数据的布局。

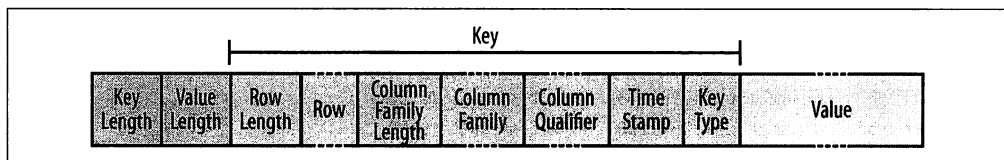


图 8-7 KeyValue 格式

该结构以两个分别表示键长度（Key Length）和值长度（Value Length）的定长数字开始。有了这个信息，用户就可以在数据中跳跃，例如，可以忽略键直接访问值。其他情况下，用户也可以从键中获取必要的信息。一旦其被转换成一个 KeyValue 的 Java 实例，用户就能通过对应的 getter 方法得到更多的细节信息，在 3.2.1 节的“KeyValue 类”部分有详细介绍。

上面的例子中，平均键比平均值大的原因可以归结为键中包含的数据项：它包含了指定单元的全维度内容。键包含了行键、列族名和列限定符等。相对于一个较小的有效负载，这将导致相当巨大的开销。如果用户处理的值较小，那么应当保持键尽量小。选择一个短的行和列键（列族名是一个单字节，同时列限定符也一样短）来保证键值比率合适。

另一方面，压缩有助于缓解这一问题，因为它着眼于有限的数据窗口，并且其中所有重复的数据都能够被有效地压缩。存储文件中所有的 KeyValue 都被有序地存储，这样有助于把类似的键（如果用户使用了版本，那么相似的值也会这样）放在一起。

8.3 WAL

region 服务器会将数据保存到内存中，直到积攒足够多的数据再将其刷写到硬盘上，这样可以避免创建很多小文件。存储在内存中的数据是不稳定的，例如，在服务器断电的情况下数据就可能会丢失。这是一个典型的问题，已经在 8.1 节中进行了解释。

一个比较常见的解决这个问题的方法是预写日志 (WAL)^①：每次更新（也叫做“编辑”）都会写入日志，只有写入成功才会通知客户端操作成功，然后服务器可以按需自由地批量处理或聚合内存中的数据。

8.3.1 概述

当灾难发生的时候，WAL 就是所需的生命线。类似于 MySQL 的 binary log，WAL 存储了对数据的所有更改。这在主存储器出现意外的情况下非常重要。如果服务器崩溃，它可以有效地回放日志，使得服务器恢复到服务器崩溃以前。这也就意味着如果将记录写入到 WAL 失败时，整个操作也会被认为是失败的。

8.2.1 节展示了 WAL 是怎样和 HBase 的架构结合在一起的。因为它被同一个 region 服务器的所有 region 共享，所以对于每一次修改它就像一个日志中心一样。图 8-8 展示了编辑流是怎样在 memstore 和 WAL 之间分流的。

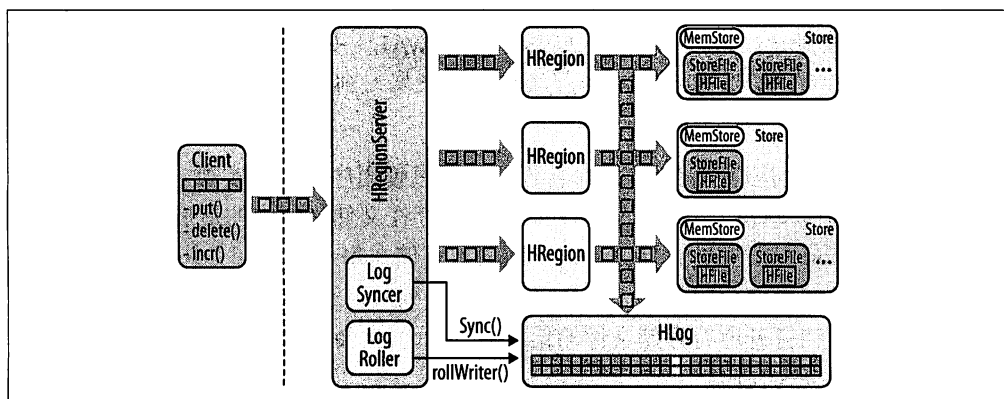


图 8-8 所有的修改都先保存到 WAL，再传递给 memstore

处理过程如下：首先客户端启动一个操作来修改数据。例如，可以对 `put()`、`delete()` 和 `increment()` 进行调用。每一个修改都封装到一个 `KeyValue` 对象实例中，并通过 `RPC` 调用发送出去。这些调用（理想情况下）成批地发送给含有匹配 `region` 的 `HRegionServer`。

一旦 `KeyValue` 实例到达，它们会被发送到管理相应行的 `HRegion` 实例。数据被写入到 `WAL`，然后被放入到实际拥有记录的存储文件的 `MemStore` 中。实质上，这就是 `HBase` 大体的写路径。

最后，当 `memstore` 达到一定的大小或是经历一个特定的时间之后，数据就会异步地连续写入到文件系统中。在写入的过程中，数据以一种不稳定的状态存放在内存中，即使在服务器完全崩溃的情况下，`WAL` 也能够保证数据不会丢失，因为实际的日志存储在 `HDFS` 上。其他服务器可以打开日志文件然后回放这些修改——恢复操作并不在这

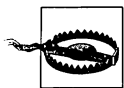
^① 见维基百科中“Write-ahead logging”一节 (http://en.wikipedia.org/wiki/write-ahead_logging)。

些崩溃的物理服务器上进行。

8.3.2 HLog 类

实现了 WAL 的类叫做 HLog。当 HRegion 被实例化时，HLog 实例会被当做一个参数传入到 HRegion 的构造器中。当一个 region 接收到一个更新操作时，它可以直接将数据保存到一个共享的 WAL 实例中去。

HLog 类的核心功能是 `append()` 方法。注意，为了提高性能，在 Put、Delete 和 Increment 中可以使用一个额外的参数集合：`setWriteToWAL(false)`。如果用户在设置时调用这个方法，例如，用户在设置一个 Put 实例时调用该方法会导致向 WAL 写入数据的过程停止。这也是为什么图 8-8 中使用虚线创建的向下的箭头来表示可选步骤。默认情况下，用户最好使用 WAL。但是，如果用户在运行一个离线的大批量导入数据的 MapReduce 作业时，其可以获得额外的性能，但是需要特别注意是否有数据在导入的时候丢失。



强烈建议用户不要毫无顾忌地关闭 WAL。如果这样做了，数据迟早会丢失并且 HBase 不能恢复丢失的且未写入到日志中的数据。

HLog 的另一个特性是追踪修改，这个特性可以通过使用序列号来实现。它在内部使用一个进程安全的 AtomicLong，且从 0 开始或从保存在文件系统中的最后一个所知的数字开始：当 region 打开它的存储文件时，它读取存储在每一个 HFile 中 meta 域中最大的序列号，并且如果这个序列号大于之前记录的序列号，它就会把 HLog 的序列号设定为这个值。所以在打开所有存储文件的结尾后，HLog 就会被初始化以反映存储在哪里结束以及从哪里继续存储。

图 8-9 展示了 3 个不同的 region，它们存储在同一个 region 服务器上，并且每一个都包含不同的行键范围。每一个 region 都共享同一个 HLog 实例。这意味着数据按照到达的顺序写入到 WAL 中，当日志需要回放（查看 8.3.7 节）时会产生额外工作。但是由于这很少发生，所以最优的做法就是按顺序存储，这样能够提供最好的 I/O 性能。

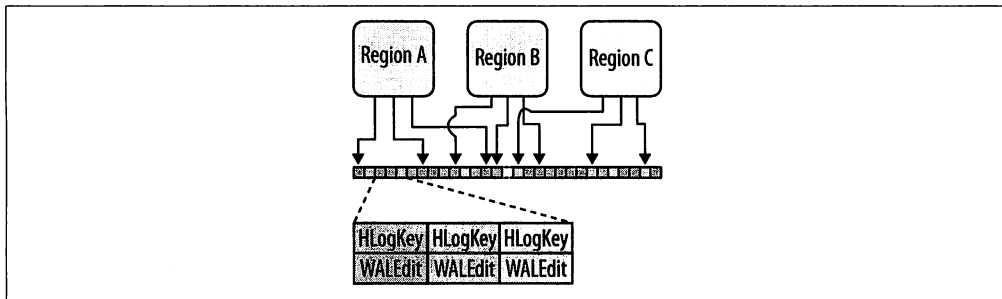


图 8-9 WAL 按照修改的时间顺序存储，包括在同一个服务器上的所有 region

8.3.3 HLogKey 类

WAL 当前使用的是 Hadoop 的 SequenceFile，这种文件格式按照键/值集合的方式存储记录。对 WAL 来说，值仅仅是客户端发送的修改请求。Key 被 HLogKey 实例代表：由于 KeyValue 仅代表行键、列族、列限定词、时间戳、类型以及值，所以要有一个地方来存储 KeyValue 的归属，即 region 和表名，这个信息存储在 HLogKey 中。HLogKey 还存储了上面所提到的序列号。每一条记录的数字是递增的，以保持一个连续的编辑序列。

它还记录了写入时间，这是一个表示修改是什么时候被写入到日志的时间戳。最后，这个类存储了多个集群之间进行复制所需要的集群 ID (cluster ID)。

8.3.4 WALEdit 类

客户端发送的每一个修改都会被封装到一个 WALEdit 实例。它通过日志级别来管理原子性。假设更新了一行中的 10 列，每一列或每一个单元格都是一个单独的 KeyValue 实例。如果服务器将它们中的 5 个写入到 WAL 后就失败了，用户就会得到一半修改内容被持久化了的行。

这可以通过将包含多个单元格的，且所有被认为是原子的更新都写入到一个 WALEdit 实例中来解决。这一组的修改都会在一次操作中被写入，以保证日志的一致性。



HBase 在 0.90.x 之前的版本中单独保存 KeyValue 实例。

8.3.5 LogSyncer 类

表的描述符允许用户设置一个叫做延迟日志刷写 (deferred log flush) 的标志，这个标志已经在 5.1.2 节中进行了解释。这个值默认为 false，这意味着每一次编辑被发送到服务器时，它都会调用写日志的 sync() 方法。这个调用强迫写入日志的更新都会被文件系统确认，所以用户获得了持久性保证。

不幸的是，调用这个方法会涉及一对 N 的服务器管道写 (其中 N 是 WAL 文件的复制因子)。由于这是一个高代价的操作，所以可以选择稍微延迟这个调用，并让它在后台执行。记住，如果不调用 sync() 方法，那么在服务器出现故障的时候，将有一定几率造成数据丢失。请小心使用这个设置。

管道与多路写的对比

当前 sync() 的实现是管道写，这意味着当修改被写入时，它会被发送到第一个 Data Node 进行存储。一旦成功，第一个 DataNode 就会把修改发送到另一个 Data Node 来进行相同的工作。只有 3 个 DataNode 都已经确认了写操作，客户端才被允许继

续进行。

另一种存储修改的方法是多路写，也就是写入被同时发送到 3 台主机上。当所有主机确认了写操作之后，客户端才可以继续。

这两种方法的区别是管道写需要时间去完成，所以它有很高的延迟，但是它能更好地利用网络带宽。多路写有着比较低的延迟，因为客户端只需要等待最慢的 `DataNode` 确认（假设其余已经成功确认）。但是写入需要共享发送服务器的网络带宽，这对于有着很高负载的系统来说会是一个瓶颈。

目前有正在进行的工作能让 HDFS 支持上面两种方式，这能让你选择一种方式来达到最佳性能。

用户将 `deferred log flush` 标志位设置为 `true` 会导致修改被缓存在 `region` 服务器中，然后在服务器上 `LogSyncer` 类会作为一个线程运行，负责在非常短的时间间隔内调用 `sync()` 方法。默认的时间间隔为 1 秒，可以通过 `hbase.regionserver.optionallogflushinterval` 属性来设置。

注意这只作用于用户表：所有的目录表会一直保持同步。

8.3.6 LogRoller 类

日志的写入是有大小限制的。`LogRoller` 类会作为一个后台线程运行，并且在特定的时间间隔内滚动日志。这可以通过 `hbase.regionserver.logroll.period` 属性来控制，默认值是 1 小时。

每 60 分钟旧的日志文件被关闭，然后开始使用新的日志文件。经过一段时间，系统会积攒一系列数量不断递增的日志文件，这些文件也需要维护。`LogRoller` 会调用 `HLog.rollWriter()` 方法来做上面所说的滚动当前日志文件的工作，接着 `HLog.rollWriter()` 会调用 `HLog.cleanOldLogs()`。

`HLog.cleanOldLogs()` 会检查写入到存储文件中的最大序列号是多少，这是因为到这个序列号为止（小于或等于这个序列号）的所有修改都已经被保存了。然后它会检查是不是有日志文件的序列号都小于这个数字。如果是的话，它就会将这些文件移动到 `oldlogs` 文件夹中，留下其余的日志。



也许用户会在日志文件中看见以下晦涩的信息。

```
2011-06-15 01:45:48,427 INFO org.apache.hadoop.hbase.region server.HLog: \
Too many hlogs: logs=130,maxlogs=96;forcing flush of 8 region(s):
testtable,row=500,1309872211320.d5a127167c6e2dc5106f066c c8 4506f8,...
```

这些信息被打印出来是因为需要保留的日志文件数超过了设置的最大日

志文件数，但是仍有一些数据更新没有被保存。造成这种情况的原因之一是文件系统负载较大以至于它不能以新数据被添加进来的速率来存储数据；否则 memstore 的刷写不会受此影响。

注意，当这条信息被打印的时候，服务器会进入到一个特殊的模式来强制刷写内容中的更新数据，以减少需要保存的日志量。

其他控制日志滚动的参数有 `hbase.regionserver.hlog.blocksize`（设置为文件系统默认的块大小或 `fs.local.block.size`，默认值是 32MB）和 `hbase.regionserver.logroll.multiplier`（设为 0.95），这个参数表示当日志达到块大小的 95% 时就会滚动日志。所以，不管日志文件被认为已经满了，还是经过一定的时间文件达到了预设的大小，日志文件就会被滚动。

8.3.7 回放

master 和 region 服务器需要配合起来精确地处理日志文件，特别是需要从服务器失效中恢复的时候。WAL 用来保持数据更新的安全，而回放则是一个使得系统恢复到一致性状态的更加复杂的过程。

1. 单日志

因为所有的数据更新都会被写入到 region 服务器中的一个基于 HLog 的日志文件中，为什么不分开将每个 region 的所有数据更改都写入到一个单独的日志文件中呢？下面是引自 BigTable 论文的相关内容：

如果我们将不同表的日志提交到不同日志文件中，就需要向 GFS 并发地写入大量文件。以上操作依赖于每个 GFS 服务器文件系统的底层实现，这些写入会导致大量的硬盘寻道来向不同的物理日志文件中写入数据。

由于相同的原因，HBase 也遵循这个原则：同时写入太多的文件，且需要保留滚动的日志会影响系统的扩展性。这种设计最终是由底层文件系统决定的。虽然在 HBase 中可以替换底层文件系统，但是通常情况下安装还是会选用 HDFS。

通常情况下，以上设计不会造成什么问题，但当系统遇到一些错误时，以上设计可能将带来一些麻烦。如果用户的数据被及时地、安全地持久化了，那么所有事情就非常正常。但是只要遇到服务器崩溃，系统就需要拆分日志，即把日志分成合适的片，这些在下一节有相应的描述。现在的问题是，所有的数据更改的日志都混在一个日志文件中，并且没有任何索引。正是由于这个原因，master 不可能立即把一个崩溃的服务器上的 region 部署到其他服务器上，它需要等待对应 region 的日志被拆分出来。如果服务器崩溃之前已经来不及将数据更新刷写到文件系统，对应的需要拆分的 WAL 数量也将非常庞大。

2. 日志拆分

有两种日志文件需要被回放的情况：集群启动时或服务失效时。当 master 启动的时候——这也包括备用的 master 接管系统的时候——它会检查文件系统中 HBase 根目录 `logs` 文件夹下是不是有日志文件，以及这些日志有没有分配的 region 服务器。日志的名字不仅包含服务器的名字，还包含服务器的启动码（start code）。这个数字会在每次 region 服务器重启的时候重置。master 可以通过这个数字来检查日志是否被遗弃了。

master 还需要负责监控服务器如何使用 ZooKeeper。当 master 检测到一个服务器失效时，它就会在重新分配 region 到新的服务器前立即启动一个进程来恢复日志文件。这项工作是由 `ServerShutdownHandler` 类完成的。

在日志中的数据改动被回放之前，日志需要被单独放在每个 region 对应的单独的日志文件中。这个过程叫做日志拆分（log splitting）：读取混在一起的日志，并且所有的条目都按照它所归属的 region 来分组。这些分组的修改记录被存放在一个紧挨着目标 region 的文件中以供接下来的数据恢复过程使用。

日志拆分的实质操作过程几乎在每一个 HBase 版本中都不太一样：早期版本会直接在 master 上通过一个线程来读取文件。这个后来又提升为至少不同 region 对应的修改是通过多线程的方式来重新执行。在 0.92.0 版本中，终于引入了分布式日志拆分（distributed log splitting）的概念，切分日志的实际工作从 master 转移到了 region 服务器上。

现在我们考虑一个更大的 region 服务器集群，该集群有很多的 region 服务器和很大的日志文件，过去 master 只能分别串行地恢复每个日志文件，这样它才不会在 I/O 和内存使用方面过载。这就意味着，每一个拥有被挂起的数据更改的 region 都会被阻塞，直到日志拆分以及恢复完成之后才能被打开。

最新的分布式模式使用 ZooKeeper 来将每一个被丢弃的日志文件分发给一个 region 服务器。它们通过监测 ZooKeeper 来发现需要执行的工作，一旦 master 指出某个日志是可以被处理的，那么它们会竞争这个任务。获胜的 region 服务器就会在一个线程（为了不使已经负载很重的 region 服务器过载）中读取并且拆分这个日志文件。



可以通过 `hbase.master.distributed.log.splitting` 设置属性来关闭新的分布式日志拆分。设置该项为 `false` 可以停用分布式拆分，则系统只能直接通过 master 来进行这项工作。

在非分布式模式下写入是多线程的，这是通过 `hbase.regionserver.hlog.splitlog.writer.threads` 属性来控制的，其默认值为 3。由于增加这个值后，读取日志的性能很可能会受限于单一的日志读者，所以用户需要权衡。

拆分过程会首先将数据改动写入到 HBase 根目录下的 *splitlog* 暂存文件夹。它们已经被放置在与所需的目标 *region* 相同的路径下。例如：

```
0 /hbase/.corrupt
0 /hbase/splitlog/foo.internal,60020,1309851880898_hdfs%3A%2F%2F \
localhost%2Fhbase%2F.logs%2Ffoo.internal%2C60020%2C1309850971208%2F \
foo.internal%252C60020%252C1309850971208.1309851641956/testtable/ \
d9ffc3a5cd016ae58e23d7a6cb937949/recovered.edits/00000000000000002352
```

为了与其他日志区分开能够执行并发操作，路径中包含了日志文件名。路径中还包含了表名、*region* 名（散列值）和 *recovered.edits* 文件夹。最后，拆分文件的名称是对应 *region* 的日志中第一次数据更改所对应的序列 ID。

.corrupt 文件夹包含所有不能被解析的日志文件。这种情况可以通过 *hbase.hlog.split.skip.errors* 属性来改变，通常将其默认设置为 *true*。这表示任何不可以从文件中读出来的数据更改都会使整个日志文件被移动到 *.corrupt* 文件夹中。如果将这一标志设置为 *false*，则会抛出 *IOException* 异常，并且整个日志拆分过程都会被停止。

一旦日志文件被成功拆分，则每个 *region* 对应的文件就会被移动到实际的 *region* 目录。然后 *region* 就可以使用对应的日志来恢复数据了。这也就是为什么拆分需要阻塞打开 *region*，因为它需要首先提供挂起的修改来回放。

3. 数据恢复

当集群启动，或 *region* 从一个 *region* 服务器移动到另一个 *region* 服务器时，*region* 都会被打开，且此时 *region* 会首先检查 *recovered.edits* 目录是否存在。如果该目录存在，它就会打开该目录中的文件，并开始读取文件所包含的数据更改记录。由于文件是按照含序列 ID 的文件名排序的，*region* 便可以按照序列 ID 的顺序来恢复数据。

任何序列 ID 小于或者等于保存在硬盘中存储文件序列 ID 的更改记录都会被忽略，因为这些记录之前的数据已经被刷写了。其他数据更新都会被添加到对应 *region* 的 *memstore* 中来恢复之前的数据状态。最后，一次强制的 *memstore* 刷写会将当前数据写入到硬盘中。

一旦 *recovered.edits* 文件夹中的文件都被处理完，且其中的数据更改也都被写入到硬盘后，该文件夹就会被删除。当出现文件不可读的情况时，*hbase.skip.errors* 属性决定了接下来系统的行为：该属性为默认值 *false* 时，整个 *region* 恢复过程失败；该属性为 *true* 时，对应文件就会被重命名为原始文件名加上 *.<currentTimeMillis>*。不管哪种情况，用户都需要小心地检查日志文件来判断为什么恢复会遇到问题，然后修复问题来让恢复过程继续执行。

8.3.8 持久性

用户都想依靠系统来保存自己写入的所有数据，不论系统在内部使用了什么新奇的算法。在 HBase 系统中，用户可以尽量降低日志的刷写次数，也可以在每次数据更改时都同步日志。不管怎样做，用户都需要依赖上文提到的文件系统来做最后的持久化工作。用来存储数据的输出流已经被刷写了，但是对应的数据是否已经写入到硬盘中了呢？我们在谈论的是 fsync 相关的问题。对 HBase 来说，大多数情况下还是会选用 Hadoop 的 HDFS 作为存储数据的文件系统。

到现在为止，大家应当已经清楚日志是用来保证数据安全的。正是由于这个原因，日志有可能保持打开状态一小时（或者更多，如果用户配置过的话）。随着数据被不断写入，新的键/值对被写入到 SequenceFile 中，同时也间断地刷写到硬盘中。

但是，这种使用文件的方式并不是 Hadoop 设计的使用模式通常情况下，Hadoop 会提供一套 API 并通过它向文件中写入数据（最好是大量的数据），此后立即关闭这个文件，并使之成为一个不可更改的文件，随后大家都可以多次读取这个文件。

同时文件只有在被关闭之后才对其他人可见和可读。如果在写入数据的过程中进程崩溃了，那么这个文件很大可能性会被认为丢失了。但对日志文件来说，能够读到上一次服务器崩溃时写操作的位置。这种属性被添加到了新版本的 HDFS 中，通常称为追加（append）属性。

插曲：HDFS 追加、hflush、hsync 和 sync……

追加（append）功能被 HBase 用来保证持久性，但过去的 Hadoop 不支持此功能。很长一段时间后，HBase 才支持了此功能，并需要打若干补丁。所有这些内容都源自 HADOOP-1700 问题（<http://issues.apache.org/jira/browse/HADOOP-1700>）。问题是在 Hadoop 0.19.0 中被提交的，其本意是解决文件追加的问题。但是实际并不是这样，Hadoop 0.19.0 的追加兼容性很差，以至于一个简单的 `hadoop fsck /` 都会因为 HBase 的日志文件是打开状态而报告 HDFS 已损坏。

所以这个问题又在 HADOOP-4379 或 HDFS-200（<http://issues.apache.org/jira/browse/HDFS-200>）中被重新处理，并实现了 `syncFs()` 来帮助文件的同步修改更可靠。曾有一段时间我们使用一种特定的代码（见 HBASE-1470，<http://issues.apache.org/jira/browse/HBASE-1470>）来检测拥有这个 API 的打过补丁的 Hadoop。

然后是 HDFS-265（<http://issues.apache.org/jira/browse/HDFS-265>），它从整体上重新审视了追加的想法，并引入了 Syncable 接口，这个接口拥有 `hsync()` 和 `hflush()` 方法。

要注意的是 `SequenceFile.Writer.sync()` 方法和上面所提到的同步方法并不相同：它向文件写入一个可以帮助读取或恢复数据的同步标记。

HBase 现在会检测当前的 Hadoop 库是否支持 `syncFs()` 或 `hflush()`。如果在写入日志的时候触发 `sync()`，则 HBase 会在内部调用以上两种方法之一。但是，如果 HBase 运行在一个不需要持久的设置下，它什么也不会调用。`sync()` 使用的是管道写来保证日志文件中数据更改记录的持久性，这部分内容在 8.3.5 节中进行了介绍。在服务器崩溃的情况下，系统可以安全地从废弃的日志文件中读取到最新的数据更改。

总之，如果没有使用 Hadoop 0.21.0 及以后的版本，或移植了追加支持的特殊的 0.20.x 版本的话，用户就可能面对数据丢失。更多信息详见 2.2.3 节中的“Hadoop”部分。

8.4 读路径

HBase 的每个列族使用多个存储文件来进行数据存储，这些文件包含着实际的数据单元或 Key-Value 实例。当 memstore 中存储的对 HBase 的修改信息最后作为存储文件被刷写到硬盘中时，这些文件就被创建了。后台合并进程通过将小文件写入到大文件中来减少文件的数目，从而保证文件的总数在可控范围之内。major 合并最后将文件集中的文件合并成一个文件，此后刷写又会不断创建小文件。

由于所有的存储文件都是不可变的，从这些文件中删除一个特定的值是做不到的，通过重写存储文件将已经被删除的单元格移除也是毫无意义的。墓碑标记就是用于此类情况的，它标记着“已删除”信息，这个标记可以是单独一个单元格、多个单元格或一整行。

假设用户今天在给定的一行中写入了一列数据，然后在未来的几天里不停地在其他几行中添加数据，然后再在之前给定行的其他列中写入数据。需要考虑的问题是，假设之前添加的列数据已经作为 Key-Value 在硬盘存储了一段时间，由于新写入的列数据会存储在 memstore 里或者也被刷写到硬盘中，那么逻辑上的一整行数据到底存储在了哪里呢？

换句话说，当使用 Shell 对那一行执行一个 `get` 命令时，系统怎么知道该返回什么？作为客户端，我们希望两列都被返回——就好像它们被存储在同一个实体中一样。但是实际上数据存储是分离的 Key-Value 实例中，横跨任意数目个存储文件。

如果删除了最初的列值，然后再次执行 `get` 命令时，我们期望这个值已经被删除，然而实际上它仍然存在于某处，只是墓碑标记指出用户已经删除了它。但是标记可能存储在距离所需删除数据很远的地方。8.1 节详细解释了这个方式背后的架构。

HBase 通过使用 QueryMatcher 和 ColumnTracker 来解决这个问题：其中一个需要精确地匹配用户要取出的列，另一个则需要包含所有的列。它们都可以设定最大匹配的版本

数。它们都会跟踪要被包含到最终结果中的内容。

为什么 Get 都是 Scan

在 HBase 以前的版本中, Get 方法是作为一个单独的代码路径实现的。在最近的版本中, 这个方式作出了改变并且在内部完全被 Scan API 所使用的代码替换。

既然单独一个 Get 应该会比 Scan 快, 用户可能会怀疑为什么要这样做。一个单独的代码路径可以使用一类特殊的知识来快速访问用户所要求的数据。

这时该介绍一下 HBase 的架构了。HBase 中没有可以使我们直接访问特定行或列的索引文件。HFile 中最小的单元是块, 并且为了找到所要求的数据, RegionServer 代码和它底层实现的 Store 实例必须载入整个可能存储着所需数据的块并且扫描这个块。以上就是 Scan 做的事情。

也就是说, 一次 Get 就仅仅对单独一行进行扫描。用户可以创建一次 Scan, 并且将开始行设定为你寻找的那行, 且将结束行设定为 `start row + 1`。

在读取所有存储文件来查找匹配的条目之前, 需要有一个快速的排除检查阶段: 使用时间戳以及可选的布隆过滤器来跳过那些绝对不包含所需 Key-Value 的文件。然后, 扫描剩下的存储文件以及 memstore 来寻找匹配的键。

扫描是通过 RegionScanner 类实现的, 该类为每一个 Store 实例获取一个 StoreScanner, 每个 Store 实例代表着一个列族。如果读操作排除了某些列族, 那么它们的存储也同样会被省略。

StoreScanner 类合并了 Store 实例包含的存储文件和 memstore。这也是基于布隆过滤器或时间戳的筛选发生的地方。如果用户想要最近一个小时内的版本的数据, 那么用户可以跳过所有存储时间超过最近一小时的文件: 它们不会包含任何有用的信息。9.1 节详细介绍了如何排除无用信息以及如何利用 StoreScanner。

StoreScanner 类也同样包含了 QueryMatcher (这里是 ScanQueryMatcher 类), 它会记录到底哪个 Key-Value 需要被包含在最终的结果中。

RegionScanner 在内部是使用 KeyValueHeap 类, 并按时间戳排序来处理存储扫描器。StoreScanner 也使用这个类按照同样的方法对存储进行排序。这保证了用户可以按照正确的顺序来读取 Key-Value (例如, 按时间戳降序)。

当存储扫描器打开的时候, 它们会直接定位到所要求的行键上, 或者在调用 `get()` 的情况下定位到下一个不匹配的行键上 (大于起始键的键)。然后扫描器就准备读取数据了, 图 8-10 展示了其运行机制。

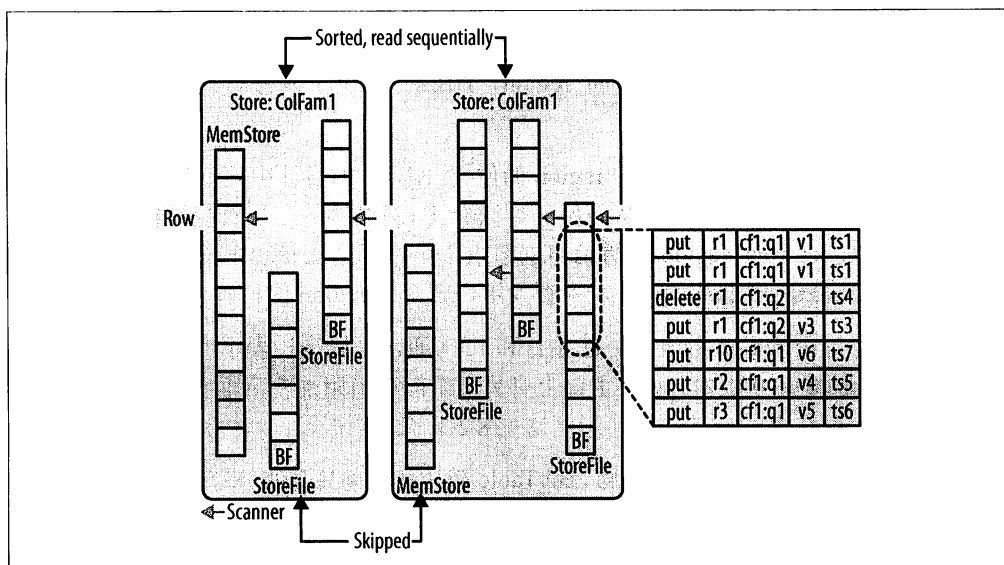


图 8-10 横跨存储文件、硬盘以及内存来存储或扫描行

对于一次 `get()` 调用，所有的服务器需要做的是在 `RegionScanner` 上调用 `next()`。这个调用在内部读取所有可能会包含在最后结果中的东西。这将包含所有需要的版本。假设一列有 3 个版本，用户要求取得所有的版本，而这 3 个 `KeyValue` 实例有可能分散在任意存储文件、硬盘以及内存中。`next()` 一直读取所有的存储文件直到到达下一行，或已经找到足够多的版本来返回数据。

与此同时，它也会持续跟踪删除标记。由于它扫描过当前行所有的 `KeyValue`，因此会遇到这些删除标记，并会意识到所有时间戳小于或等于此标记的数据都是已经被删除了的数据。

图 8-10 将一个逻辑行表示为一组 `KeyValue`，其中一些在相同的存储文件中，一些在另外一些文件中，并能够横跨多个列族。一个存储文件和 `memstore` 在基于时间戳和布隆过滤器的排除过程中被跳过。最后一个存储文件中的删除标记被用来屏蔽条目，但是它们仍然属于同一行数据。扫描器——被表示为存储文件附近的箭头——处于文件中的第一个符合的条目或是紧挨着所要求的行，后者是因为这个存储文件中没有直接匹配的条目。

在调用 `next()` 的期间，只有那些在合适的行上的扫描器才会被考虑。内部循环会按照时间降序一个接一个地从存储文件中读取 `KeyValue`，直到它们超过所需要的行键。

对于扫描操作，`ResultScanner` 会重复调用 `next()` 直到到达结束行或表的结尾，或者对于当前的一批（扫描器缓存设定）已经读了足够多的行。

最终的结果是符合给定 `get` 或 `scan` 操作要求的一组 `KeyValue` 实例。它们会返回给客户端，然后客户端可以通过 `API` 方法来访问其中所包含的列。

8.5 region 查找

为了让客户端找到包含特定主键的 **region**，HBase 提供了两张特殊的目录表 **-ROOT-** 和 **.META.**。^①

-ROOT-表用来查询所有 **.META.**表中 **region** 的位置。HBase 的设计中只有一个 **root region**，即 **root region** 从不进行拆分，从而保证类似于 B+树结构的三层查找结构：第一层是 ZooKeeper 中包含 **root region** 位置信息的节点，第二层是从 **-ROOT-**表中查找对应 **meta region** 的位置，第三层是从 **.META.**表中查找用户表对应 **region** 的位置。

目录表中的行键由 **region** 的表名、起始行和 ID（通常是以毫秒表示的当前时间）连接而成。从 HBase 0.90.0 版本开始，主键上有另一个散列值附加在后面。不过目前这个附加部分只用在用户表的 **region** 中。示例请见 8.2.3 节。



不用担心三层定位策略，BigTable 的论文中阐述了当 **.META.**的 **region** 大小为 128 MB 时，它可以定位 2^{34} 个 **region**，或 2^{61} 字节的数据（当 **region** 为 128 MB 时）。由于 **region** 的大小可以在对存储模式没有任何影响的情况下进行扩展，所以这只是一个保守估计，在数据量增加时，**region** 的大小可以适当调高。

虽然客户端缓存了 **region** 的地址，但是初始化需求时需要重新查找 **region**，例如，缓存过期了，并发生了 **region** 的拆分、合并或移动。客户端库函数使用递归查找的方式从目录表中重新定位当前 **region** 的位置。它会从对应的 **.META.** **region** 查找对应行键的地址。如果对应的 **.META.** **region** 地址无效，它就向 **root** 表询问当前对应的 **.META.** 表的位置。最后，如果连 **root** 表的地址也失效了，它会向 ZooKeeper 节点查询 **root** 表的新地址。

在最坏的情况下，客户端需要 6 次网络往返请求来定位一个用户 **region**，由于系统假设了 **region** 的分配情况，特别是 **meta region** 的分配情况不会经常变化，所以只有当查找失败时，客户端才会认为缓存的 **region** 地址失效。当缓存为空时，客户端需要 3 次网络往返请求来更新缓存。如果用户想减少未来请求 **region** 地址的次数，可以在请求之前预刷写缓存地址。参考 3.6 节详细了解主动刷写 **region** 地址缓存的方法。

图 8-11 展示了先通过 **meta** 表，最终通过 **root** 表来确定一个用户 **region** 位置的过程。一旦获取到用户 **region** 的位置，其数据就可以被直接访问。图中的查询被编号了，并假定开始时缓存是空的。如果缓存不为空，但缓存的用户 **region**、**meta region** 和 **root region** 的位置都失效，则需要额外的 3 次查询，即总共需要 6 次查询来完成这次定位。

^① 后面它们分别被引用为 **root** 表和 **meta** 表，例如，**-ROOT-**反映了它在 HBase 中实际的表名，而称它为 **root** 表则强调了它的作用。

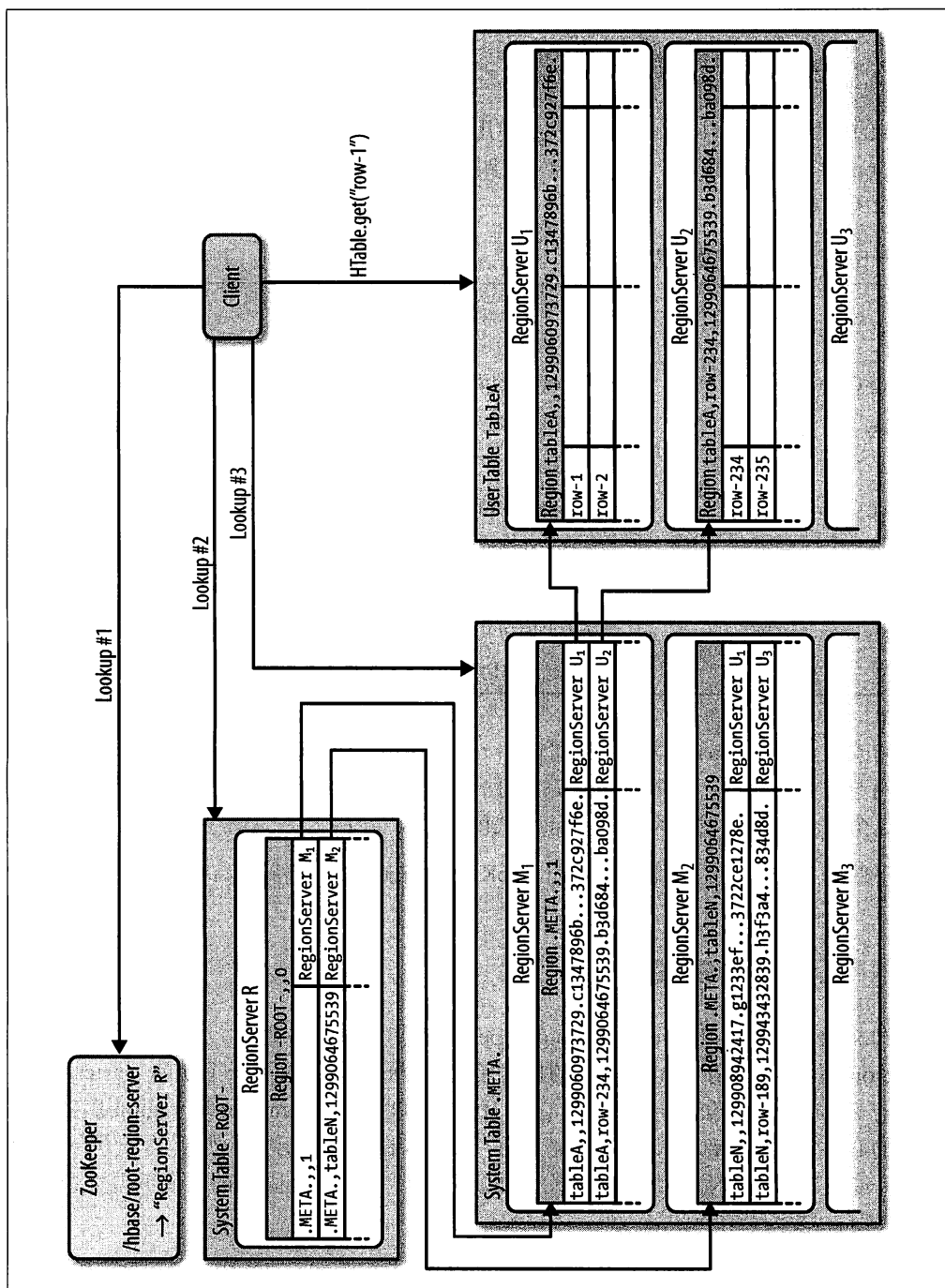


图 8-11 用户 region 的映射从空缓存开始，客户端需要进行 3 次查询

8.6 region 生命周期

region 的各种状态均由 master 触发，并使用 AssignmentManager 类进行管理。这个类会从 region 的下线（offline）状态开始一直跟踪，并管理它的状态。表 8-1 列举了 region 可能的所有状态。

表 8-1 region 的可能状态

状态	描述
Offline	region 下线
Pending Open	打开 region 的请求已经发送到了服务器
Opening	服务器开始打开 region
Open	region 已经打开，并且完全可以使用
Pending Close	关闭 region 的请求被送到服务器
Closing	服务器正在处理要关闭的 region
Closed	region 已经被关闭了
Splitting	服务器开始切分 region
Split	region 已经被切分了

状态的改变可能由 master 发起，也可能由 region 服务器发起。例如，当 master 把 region 分配到一个服务器后，由服务器来完成打开过程。此外，拆分过程由 region 服务器发起，这个过程可能引发一系列的 region 关闭和打开事件。

由于事件都是分布式的，服务器使用 ZooKeeper 来跟踪一个特定 znode 的状态。

8.7 ZooKeeper

从 0.20.x 版本开始，HBase 使用 ZooKeeper 作为其协同服务组件。其主要功能包括跟踪 region 服务器、保存 root region 的地址等。在 0.90.x 中引入了一个新的 master 实现，使其与 ZooKeeper 集成得更紧密。它使 HBase 可以移除在 master 和 region 服务器间传递的心跳信息，这部分功能现在被放在了 ZooKeeper 中。现在，如果某一方有变动，HBase 能及时通知到集群的其他部分，而之前需要等待一个固定的时间间隔才能发出通知。

这里有 HBase 建立的 znode 列表。默认为 /hbase，这个 znode 的名称由 zookeeper.znode.parent 属性决定。以下是 znode 的列表以及它们的作用。



例子使用了 ZooKeeper 命令行接口（简称为 CLI）来执行命令。用户可以按照以下方法进行启动：

```
$ $ZK_HOME/bin/zkCli.sh -server <quorum-server>
```

输出的某些内容被简化了。

/hbase/hbaseid

包含 cluster ID，与存储在 HDFS 上的 *hbase.id* 文件内容相同，例如：

```
[zk: localhost(CONNECTED) 1] get /hbase/hbaseid
e627e130-0ae2-448d-8bb5-117a8af06e97
```

/hbase/master

包含服务器名（参见 5.2.5 节），例如：

```
[zk: localhost(CONNECTED) 2] get /hbase/master
foo.internal,60000,1309859972983
```

/hbase/replication

包含副本信息，参见 8.8.2 节。

/hbase/root-region-server

包含 -ROOT- region 所在 region 服务器的机器名，这个经常在 region 定位中使用（参考 8.5 节）。例如：

```
[zk: localhost(CONNECTED) 3] get /hbase/root-region-server
rs1.internal,60000,1309859972983
```

/hbase/rs

这个 **znode** 是作为所有 region 服务器的根节点，集群用它来跟踪服务器异常。每个 **znode** 都是临时节点，并且 node 名是 region 服务器的名称，例如：

```
[zk: localhost(CONNECTED) 4] ls /hbase/rs
[rs1.internal,60000,1309859972983,rs2.internal,60000,1309859345233]
```

/hbase/shutdown

这个节点用来跟踪集群状态信息，其包括集群启动的时间，以及当集群被关闭时的空状态，例如：

```
[zk: localhost(CONNECTED) 5] get /hbase/shutdown
Tue Jul 05 11:59:33 CEST 2011
```

/hbase/splitlog

协调日志拆分相关的父节点，参见 8.3.8 节的“日志拆分”部分：

```
[zk: localhost(CONNECTED) 6] ls /hbase/splitlog
[hdfs%3A%2F%2Flocalhost%2Fhbase%2F.logs%2Ffoo.internal%2C60020%2C \
1309850971208%2Ffoo.internal%252C60020%252C1309850971208.1309851636647,
hdfs%3A%2F%2Flocalhost%2Fhbase%2F.logs%2Ffoo.internal%2C60020%2C \
1309850971208%2Ffoo.internal%252C60020%252C1309850971208.1309851641956,
...
hdfs%3A%2F%2Flocalhost%2Fhbase%2F.logs%2Ffoo.internal%2C60020%2C \
1309850971208%2Ffoo.internal%252C60020%252C1309850971208.1309851784396]

[zk: localhost(CONNECTED) 7] get /hbase/splitlog/ \
\hdfs%3A%2F%2Flocalhost%2Fhbase%2F.logs%2Fmemcache1.internal%2C \
60020%2C1309850971208%2Fmemcache1.internal%252C60020%252C1309850971208. \
1309851784396
unassigned foo.internal,60000,1309851879862

[zk: localhost(CONNECTED) 8] get /hbase/splitlog/ \
\hdfs%3A%2F%2Flocalhost%2Fhbase%2F.logs%2Fmemcache1.internal%2C \
60020%2C1309850971208%2Fmemcache1.internal%252C60020%252C1309850971208. \
1309851784396
owned foo.internal,60000,1309851879862

[zk: localhost(CONNECTED) 9] ls /hbase/splitlog
[RESCAN0000293834,hdfs%3A%2F%2Flocalhost%2Fhbase%2F.logs%2Fmemcache1. \
internal%2C60020%2C1309850971208%2Fmemcache1.internal%252C \
60020%252C1309850971208.1309851681118,RESCAN0000293827,RESCAN0000293828, \
RESCAN0000293829,RESCAN0000293838,RESCAN0000293837]
```

这个例子展示了许多东西：用户可以看到被拆分的日志先是被 `unassign`，之后被一个 `region` 服务获取。`RESCAN` 节点代表 `workers` (`region` 服务器) 要进行很多检查工作来防止拆分工作在其他机器上失败。

/hbase/table

当表被禁用，信息会被添加到这个 `znode` 之下。表名是新建的 `znode` 名，内容是“`DISABLED`”，例如：

```
[zk: localhost(CONNECTED) 10] ls /hbase/table
[testtable]
[zk: localhost(CONNECTED) 11] get /hbase/table/testtable
DISABLED
```

/hbase/unassigned

这个 `znode` 被 `AssignmentManager` 用来跟踪集群的 `region` 状态。它包含未打开 (`open`) 的 `region` 的 `znode`，不过 `znode` 的状态是变化的，`znode` 名是 `region` 的散列值。例如：

```
[zk: localhost(CONNECTED) 11] ls /hbase/unassigned
[8438203023b8cbb347eb6fc118312a7]
```

8.8 复制

HBase 复制是一种在不同 HBase 部署中复制数据的方法。它可以作为一种灾难恢复的方法，并且可以提供 HBase 层的高可用性。同时在实际应用中，例如，将数据从一个面向页面的集群复制到一个 MapReduce 集群，后者可以同时处理新数据和历史数据。然后再自动将数据传回面向页面请求的集群。

HBase 复制中最基本的架构模式是“主推送”（master-push），因为每个 region 服务器都有自己的 WAL（或 HLog），所以及很容易保存现在正在复制的位置，其类似于其他众所周知的解决方案，例如，MySQL 的主/从复制只使用二进制日志来跟踪修改。一个主集群可以将数据复制到任意数目的从集群，每个 region 服务器都会参与复制自己的修改。

复制是异步进行的，意味着集群可以是地理上彼此远离的，它们之间的连接可以在某些时间断开，在主集群上的修改不能马上在从集群上进行同步（最终一致性）。图 8-12 展示了复制的工作流程和架构。

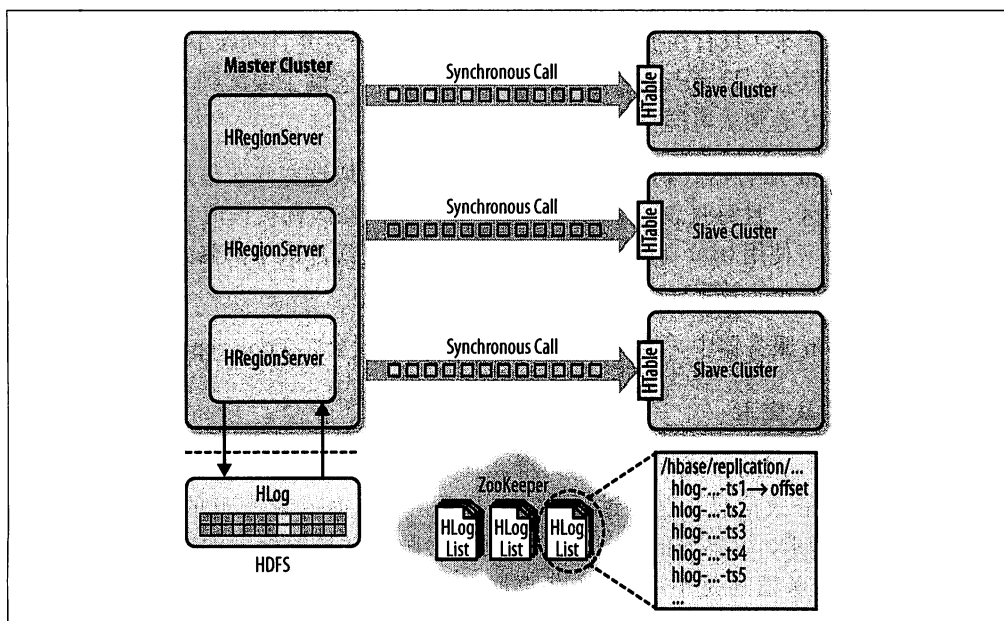


图 8-12 集群复制架构图

这里使用的复制格式与 MySQL 基于语句的复制概念相同。^①与 SQL 语句不同，所有的 WALEdits（包括来自客户端的 Put 和 Delete 产生的多单元格操作）都会被复制以保证原子性。

① 查看在线文档 <http://dev.mysql.com/doc/refman/5.1/en/replication-formats.html> 以获得更多细节信息。

来自每个 **region** 服务器的 **HLog** 是 **HBase** 复制的基础，并且只要它们需要将数据复制到从集群，它们就必须被保存在 **HDFS** 上。每个 **region** 服务器从它需要的最老的日志开始复制，同时在 **ZooKeeper** 中保存当前恢复的位置来简化错误恢复。每个从集群恢复的位置可能不同，但它们处理的 **HLog** 队列内容是相同的。

参与复制的集群的规模可以不对等，主集群会通过随机分配尽量均衡从集群的负载。

8.8.1 Log Edit 的生命周期

接下来将介绍了一条数据修改如何从客户端发起之后同主集群交互，并同时复制到从集群的过程。

1. 常规处理

客户端利用 **HBase API** 发送一个 **Put**、**Delete** 或 **Increment** 到 **region** 服务器。这些请求包含的键/值对被 **region** 服务器转化为 **WALEdit**，同时 **WALEdit** 会被复制程序检查，并以列族为单元复制数据。修改被添加到 **WAL** 中，并把实际数据添加到 **MemStore**。

另外一个线程从日志中读取数据（作为批量处理的一部分），并且只有可以复制的 **KeyValue** 被保存下来（只有在拥有用户数据日志的情况下，并且列族中有 **GLOBAL** 域定义的复制才会被保存，目录表的修改不被保存）。当缓冲区满了或读到了文件末尾，缓冲区中的修改被发送到从集群一个随机的 **region** 服务器上。

同步地，当 **region** 服务器收到这些修改后，它会顺序读取这些修改信息，并且把它们分成不同的缓冲区，且每张表一个缓冲区。一旦所有的修改都被读取后，每个缓冲使用 **HBase** 客户端（使用 **HTablePool** 管理的 **HTable**）来把缓冲区中的修改写入表中。这样可以达到并行写入的效果（**MultiPut**）。

在主集群的 **region** 服务器中，当前 **WAL** 中被复制的位置会在 **ZooKeeper** 中注册。

2. 没有反馈的从集群

数据修改信息以相同的方式插入到主集群。在另外的线程中，**region** 服务器读出日志、过滤并缓冲修改信息，这些过程与正常过程一样。如果从集群的 **region** 服务器没有响应 **RPC** 请求，主集群的 **region** 服务器将会睡眠并按照配置的次数重试。如果从集群 **region** 服务器还是不可用，主集群服务器会重新选择一台从集群服务器来提交修改。

与此同时，**WAL** 会回滚并存储在一个 **ZooKeeper** 的队列中。日志被 **region** 服务器归档（归档只是简单地把日志从 **region** 服务器的一个目录移动到一个集中的归档目录中），并更新它们在复制线程内存队列中的路径。

当从集群最终可用之后，缓冲区中的修改会被按照正常情况执行。主集群的 **region** 服务器将会把积压的日志复制到从集群。

8.8.2 内部机制

本节将深入介绍复制的各种内部机制和操作。

1. 挑选要复制的目标服务器

当主集群 region 服务器初始化复制源到从集群时，它会首先用提供给它的集群键连接从集群的 ZooKeeper 群组（使用到的集群键包括 `hbase.zookeeper.quorum`、`zookeeper.znode.parent` 和 `hbase.zookeeper.property.clientPort` 的值）。然后它会扫描 `/hbase/rs` 目录来发现所有可用的汇聚（可以用来接收复制流的服务器）并随机挑选一部分服务器来复制数据（默认是 10%）。例如，当从集群有 150 台服务器时，15 台服务器会被挑选为接收复制流的服务器，由于所有的主集群 region 服务器都会向这 10% 的服务器发送复制流，所以很可能从集群 region 服务器的负载非常高。例如，当主集群中 10 台服务器复制数据到一个 5 台服务器的从集群时，主集群每次复制时都会随机挑选从集群中一台 region 服务器，这样从集群中服务器被重复选中的概率会变高。

2. 跟踪日志中被复制到的位置

每个主集群的 region 服务器在 `znode` 目录中都有它自己对应的 `znode`，并且每个从集群都有其对应的 `znode`（5 个从集群，就会有 5 个 `znode` 被创建），并且每个 `znode` 都包含一个需要处理的 HLog 队列。每个队列都会跟踪 region 服务器创建的 HLog，不过它们队列的大小可能不同。例如，如果一个从集群一段时间不可用，那么它的 HLog 不应被删除，而需要保存在队列中（而其他从集群的日志都被处理完了）。请参考本节的“region 服务器失效”部分。

当源初始化好后，它便包括当前 region 服务器要写入的 HLog。当日志滚动时，新的文件在可用之前被添加到每个从集群的 `znode` 队列中。这样可以保证所有资源都知道有新的 HLog 可以复制到自己的集群中，但这种操作现在还是十分消耗系统资源的。当复制线程从日志文件中读不到新的日志条目（因为它到达了文件的最后一个块），并且队列中还有其他文件的路径时，队列中的旧路径就会被丢弃。这意味着，当一个源是最新的，且 region 服务器正在写入时，读到当前文件末尾的文件不会被删除。

当日志被归档（由于日表、不再被使用，或日志的数目超过了 `hbase.regionserver.maxlogs` 配置的值，也有可能是由于修改的写入速度比 region 刷写速度快），它会通知源线程日志地址发生的变化。如果一个源已经同步完了这个日志，便会忽略这条信息。如果这个文件在队列中，那么路径就会被更新。如果日志正在被同步，且修改是原子性的，因此读进程在文件移动完成之前不会尝试读取文件。同时，移动文件是由 NameNode 操作完成的，所以当日志正在被读取时，不会产生什么异常。

3. 读取、过滤以及发送数据修改

默认情况下，一个源会尝试读取日志文件，并尽可能快地将它们传送到从集群的接收服务器上。这件事首先受到日志过滤的限制，只有被分为 GLOBAL 类的且不属于目录表日志项的 Key-Value 会被保留。第二个限制是每个从集群同步总大小，默认为 64 MB。这意味着 3 个从集群会占用 192 MB 来存储要同步的数据，且不包括被过滤的数据。

一旦达到缓冲修改信息的最大值，或读到了日志文件的末尾，源线程将会停止读取并随机挑选一个可接收数据的从集群服务器来同步数据（从一个生成的从集群服务器子集列表中挑选）。它会直接把 RPC 请求发到挑选的服务器上，如果返回成功，源会判断当前文件是否读完。如果读完，则源会从队列中删掉这个 `znode`。如果没有读完，则在日志的 `znode` 中注册一个新的位移。如果 RPC 抛出异常，源会在重试 10 次之后挑选一个新的服务器。

4. 清理日志

如果没有启用同步复制，`master` 的日志清理线程会按照配置的生存期（Time-To-Live, TTL）删除旧的日志。这种机制在有复制时表现不太好，因为归档日志超出 TTL 后有可能还在队列中。所以，默认行为被增强了一些，即如果日志超过了 TTL，清理线程会查看每个队列直到找到日志（同时缓存其找到的日志）。如果在队列中没有找到日志，则日志会被删除。下次还需要查找日志时，它会先检查缓存。

5. region 服务器异常

只要 `region` 服务器没有失效，在 `ZooKeeper` 中跟踪日志就不会添加新值。不幸的是，服务器失效是一种常见的情况。因为 `ZooKeeper` 是高度可靠的，所以我们可以依靠它和它的语义来帮助我们管理队列的传输。

主集群的 `region` 服务器都会为其他服务器保留一个监听器（`watcher`），以便当其他服务器崩溃时收到通知（`master` 也是这样做的）。当有其他服务器崩溃时，它们会竞争着为宕机服务器的 `znode` 创建一个叫做 `lock` 的 `znode`，且该 `znode` 中包含其队列。创建成功的服务器会把队列添加到自己的 `znode` 中去（由于 `ZooKeeper` 不支持改名操作，所以必须逐个进行添加），并且在这个过程完成之后，它将删除旧的队列。它恢复队列的 `znode` 会用当前服务器的 ID 附加宕机服务器的名称来命名。

一旦这些完成，主集群 `region` 服务器会为每个复制后的队列创建一个新的源线程，并且每个线程都会按照读取/过滤/传输的模式工作。最主要的不同是，队列不会有新的数据，因为它们不属于新的 `region` 服务器，也就是说，当读进程读到日志的结尾时，队列的 `znode` 会被删除，并且主集群的 `region` 服务器会关闭这个复制源。

例如，假设一个主集群有 3 个 `region` 服务器同时将数据同步到 id 为 2 的从集群。下面的目录结构代表了 `znode` 的布局。可以发现 `region` 服务器的 `znode` 都包含一个 `peers` `znode`，其中包括一个队列。队列中 `znode` 的名字代表 HDFS 中实际的文件名，格式为

“address, port.timestamp”。

```
/hbase/replication/rs/
  1.1.1.1,60020,123456780/
    peers/
      2/
        1.1.1.1,60020.1234 (Contains a position)
        1.1.1.1,60020.1265
  1.1.1.2,60020,123456790/
    peers/
      2/
        1.1.1.2,60020.1214 (Contains a position)
        1.1.1.2,60020.1248
        1.1.1.2,60020.1312
  1.1.1.3,60020, 123456630/
    peers/
      2/
        1.1.1.3,60020.1280 (Contains a position)
```

现在，我们可以认为 1.1.1.2 的 ZooKeeper 会话丢失。其他 region 服务器会通过竞争来创建锁，此时 1.1.1.3 创建成功。然后，它将队列加上宕机服务器的名字后，将所有队列都转移到自己的对等 znode 下。在 1.1.1.3 清理老 znode 前，ZooKeeper 中的结构如下：

```
/hbase/replication/rs/
  1.1.1.1,60020,123456780/
    peers/
      2/
        1.1.1.1,60020.1234 (Contains a position)
        1.1.1.1,60020.1265
  1.1.1.2,60020,123456790/
    lock
    peers/
      2/
        1.1.1.2,60020.1214 (Contains a position)
        1.1.1.2,60020.1248
        1.1.1.2,60020.1312
  1.1.1.3,60020,123456630/
    peers/
      2/
        1.1.1.3,60020.1280 (Contains a position)

  2-1.1.1.2,60020,123456790/
    1.1.1.2,60020.1214 (Contains a position)
    1.1.1.2,60020.1248
    1.1.1.2,60020.1312
```

之后，在 1.1.1.3 完成从 1.1.1.2 复制最后一条 HLog 前，我们可以认为它也宕机了（也会

有一些新的日志在正常队列中)。最后剩下的 region 服务器会锁定 1.1.1.3 的 znode, 同时开始转移队列到它的 peer znode 下。目录结构如下:

```
/hbase/replication/rs/  
  1.1.1.1,60020,123456780/  
    peers/  
      2/  
        1.1.1.1,60020.1378 (Contains a position)  
  
        2-1.1.1.3,60020,123456630/  
        1.1.1.3,60020.1325 (Contains a position)  
        1.1.1.3,60020.1401  
  
        2-1.1.1.2,60020,123456790-1.1.1.3,60020,123456630/  
        1.1.1.2,60020.1312 (Contains a position)  
1.1.1.3,60020,123456630/  
  lock  
  peers/  
    2/  
      1.1.1.3,60020.1325 (Contains a position)  
      1.1.1.3,60020.1401  
  
      2-1.1.1.2,60020,123456790/  
      1.1.1.2,60020.1312 (Contains a position)
```

复制仍被认为是一个实验性功能, 使用前请仔细考虑它是否满足你的需求。

这一章会更深入地探讨 HBase 存储设计架构中的各种问题。只有设计合适的表结构、行键、列名等才能充份利用 HBase 体系结构的优势。

9.1 行键设计

HBase 有两种基本的键结构：行键（row key）和列键（column key）。两者都可以存储有意义的信息，这些信息有两类，一种是键本身存储的内容，另一种是键的排列顺序。下面会利用这些键来解决一些用户在设计存储方案时常遇到的问题。

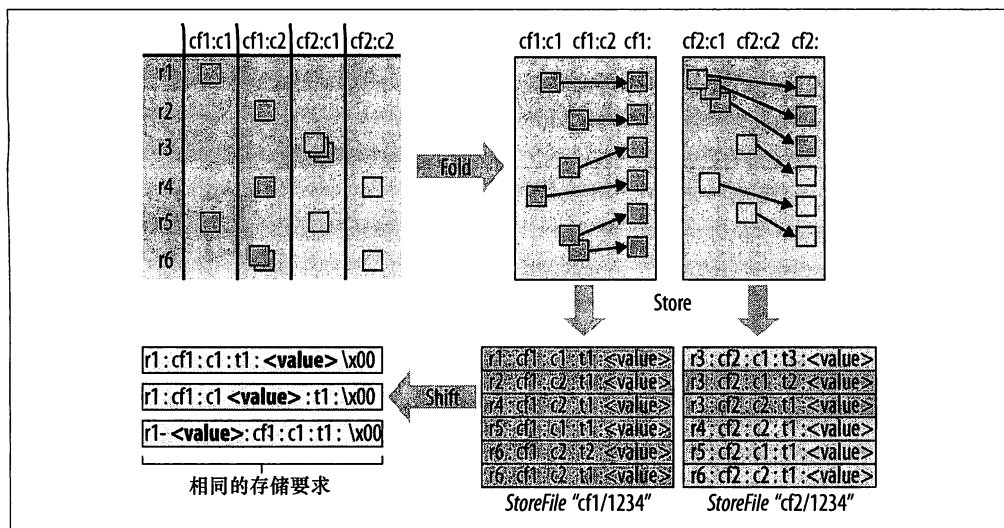
9.1.1 概念

首先我们要分析一下，与磁盘文件相比，HBase 中表的数据分布的各种细节信息。HBase 的表中的数据分割主要使用列族而不是列，这与一般传统的列式存储数据库的概念有所不同。图 9-1 表明了，虽然用户逻辑上把一个单元格的数据存到了一张表中，但实际上底层存储是按列族线性地存储了单元格，同时单元格包含了所有它必要的信息。

左上角的图片展示了数据的逻辑布局，用户设定了行和列。列包括了 HBase 特有的列族和列限定符，从而组成列键。同时，每一行还有行键，所以用户可以通过行键得到逻辑布局中一行的所有列。

右上角的图片展示了逻辑布局如何转换为实际的物理存储布局。每一行的单元格被有序存储，同时不同列族的数据存储在不同文件中。换句话说，磁盘上一个列族下所有的单元格都存储在一个存储文件（store file）中，不同列族的单元格不会出现在同一个存储文件中。

因为 HBase 不存储任何在表中没有值的单元格（在 RDBMS 中，NULL 可作为空值存储），磁盘文件中也只有这些已经有值的单元格。同时每个单元格在实际存储时也保存了行键和列键，所以每个单元格都单独存储了它在表中所处位置的相关信息。



此外，同一个单元格的多个版本被单独存储为连续的单元格，当单元格被存储时还需要添加必要的时间戳。单元格按照时间戳降序排列，所以在 HFile 的 Reader 读取数据时，最新的值先被读到，这也是 HBase 设计模式中典型的读取数据的方式。

含有结构信息的整个单元格在 HBase 中被叫做 KeyValue。其中不仅包含用户生成时设定的列 (column) 和对应的值, 也包含行键和时间戳。KeyValue 存储时先按行键排序, 当一行有多个单元格时内部再按列键排序。

右下角图片展示了一张逻辑表在物理存储文件中的数据布局。**HBase API** 包含多种访问存储文件的方法，由于键从左到右降序排列：用户可以按行键检索一行数据，这样可以有效地减少查询特定行和行范围的时间。设定列族可以有效地减少查询的存储文件，建议用户在查询时指定所需的特定列族。

虽然时间戳或者版本在整个键的最右边，但是它是很重要的筛选内容。存储文件中为每个单元格 (cell) 都保存了时间戳，所以当用户查询一个两小时前修改过的单元格时，就可以跳过只包含如 4 小时前数据的存储文件。详细内容请参见 8.4 节。

另一个层次的查询粒度是列限定符 (column qualifier)。用户可以在查询数据时指定特定的列，或定义过滤器时包含或排除用户需要访问的列。不过在这一粒度上筛选数据时，系统不得不检查每个送到过滤器的 Key/Value，所以通过限定符筛选数据只会有小幅度的性能提升。

值 (value) 是查询筛选时最后一个筛选条件, 也是应用最为广泛的筛选条件, 使用值筛选数据引起的性能提升与使用限定符时类似: 系统需要检查每个单元格来确定是否满足用户的筛选条件。用户只能使用过滤器在服务器端筛选值。图 9-2 展示了 KeyValue 不同字段的作用。

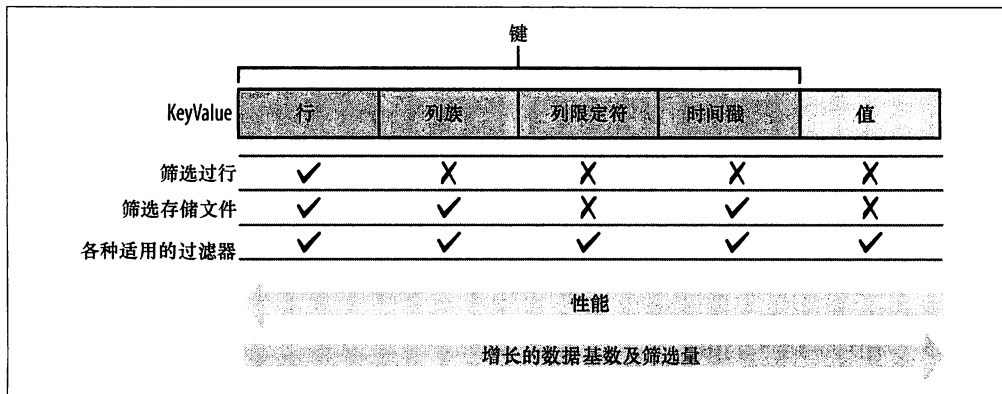


图 9-2 从左到右查询数据的性能变差

图 9-1 左下角的图片展示了“数据位移”。对于一个 KeyValue，由于筛选的效率从左至右明显下降，所以在 KeyValue 设计时用户可以考虑把一些重要的筛选信息左移到合适的位置，从而在不改变数据量的情况下，用户可以改变数据的排序并提高查询性能。

9.1.2 高表与宽表

到目前为止，用户可能会问该如何在 HBase 中存储自己的数据。通常情况下，HBase 中的表可以设计为高表（tall-narrow table）和宽表（flat-wide table）两类。前者指表中列少而行多，后者则正好相反。根据之前我们介绍过的 KeyValue 信息的筛选粒度信息，用户应当尽量将需要查询的维度或信息存储在行键中，因为用它筛选数据的效率最高。

此外，HBase 只能按行分片，因此高表更有优势。设想用户将一个其所有电子邮件都存在一行中。这在大部分情况下都是合适的，但也有些人的收件箱中有大量的邮件。大到一行数据就超过了最大 HFile 的限制，此时这个 HFile 无法拆分，同时也导致 region 无法在合适的位置进行拆分。

解决此问题的更好的方法是把一个用户的每个电子邮件都存在单独的一行中，而行键可以是用户 ID (userId) 和消息 ID (messageId) 的组合，如图 9-1 所示，用户会发现：在磁盘上，不管消息 ID (messageId) 在列限定符还是在行键中，每个单元格都仍然包含单个电子邮件的所有消息。以下是宽表在磁盘上的数据分布，并包括一些示例：

```
<userId> : <colfam> : <messageId> : <timestamp> : <email-message>

12345 : data : 5fc38314-e290-ae5da5fc375d : 1307097848 : "Hi Lars,..."
12345 : data : 725aae5f-d72e-f90f3f070419 : 1307099848 : "Welcome,and ..."
12345 : data : cc6775b3-f249-c6dd2b1a7467 : 1307101848 : "To Whom It ..."
12345 : data : dcbee495-6d5e-6ed48124632c : 1307103848 : "Hi,how are ..."
```

以下是高表在磁盘上的数据布局：

```

<userId>-<messageId> : <colfam> : <qualifier> : <timestamp> : <email- message>

12345-5fc38314-e290-ae5da5fc375d : data : : 1307097848 : "Hi Lars,..."
12345-725aae5f-d72e-f90f3f070419 : data : : 1307099848 : "Welcome, and ..."
12345-cc6775b3-f249-c6dd2b1a7467 : data : : 1307101848 : "To Whom It ..."
12345-dcbee495-6d5e-6ed48124632c : data : : 1307103848 : "Hi, how are ..."

```

以上布局使用了空限定符（参见 5.1.3 节）。消息 ID 被移动到左边，查询数据时它的筛选作用也更显著，这样每个电子邮件占用一行。最后这个表更容易被拆分，查询时用户筛选数据也更有效率（更优的筛选粒度）。

9.1.3 部分键扫描

HBase 的扫描功能和基于 HTable 的 API 更适合在高表上筛选数据，用户可以通过只包含部分键的扫描检索数据，且同时不丢失查询的粒度。

在上面的例子中，用户每一封邮件都是单独的一行。而之前的例子是把一个用户所有的邮件都放在了一行数据中，这样一个用户的收件箱就是一行数据，同时取一行数据时就能得到所有邮件的内容。每一列都是用户收件箱中的电子邮件消息。取数据时需要使用行键来匹配用户的 ID。

使用高表的例子中，消息 ID 在行键中作为用户 ID 的后缀。如果用户没有这两个 ID 确切的值，用户就无法得到一个特定的电子邮件。为避免出现这个问题，用户可以使用包含部分键的扫描：用户可以将扫描操作中键的开始和结束键设置为一个用户的 ID，当然结束键要设置为 `userId + 1`。

扫描的范围包括起始键，但不包括终止键。将起始键设为用户 ID 之后，HBase 内部会按字典序找到第一个行键的位置（这个行键要么是起始键，要么是刚好大于起始键的第一个行键）。由于表中没有等于起始键的行键，它会定位扫描下行。

```

<userId>-<lowest-messageId>

```

也就是说，它是排序时最小的用户 ID 和消息 ID 的组合。此后扫描会遍历这个用户的所有邮件，同时用户可以从行键中抽取消息 ID。

用户可以使用包含部分键的扫描机制设计出非常有效的左对齐索引（字典序从左到右排序），当一个字段被加到键中就多了一个可以检索的维度：

```

<userId>-<date>-<messageId>-<attachmentId>

```



用户需要保证行键中每个字段的值都被补齐到这个字段所设的长度，这样字典序才会按预期排列（按二进制内容比较，并升序排列）。用户需要为每个字段设定一个固定的长度来保证每个字段比较时只会与同字段内容从左向右比较，否则可能出现溢出的情况。^①

① 例如，用户可以使用 Orderly (<http://github.com/mwdalton/orderly>) 来生成组合行键。

用户可以根据查询精度的要求，构造特定的起始键和终止键来查询所需的数据。通常情况下，用户只需创建起始键，同时将终止键设置成相同的键，并在第一个字段最后的字节上附加一点数据。就以上例子而言，用户可以把起始键设为 12345（可以假设这是一个用户的 ID），终止键设为 123456。

表 9-1 展示了可能的起始键及其含义


命令	描述
<userId>	扫描一个给定用户 ID 下的所有消息
<userId>-<date>	扫描一个给定用户 ID 下特定日期的消息
<userId>-<date>-<messageId>	扫描一个给定用户 ID 和日期下的全部消息
<userId>-<date>-<messageId>-<attachmentId>	扫描一个给定用户 ID 和日期下的一个消息附件

这种组合行键与关系型数据库系统提供的功能相似，用户可以控制每个字段的内容以达到控制段内排序的目的。例如，用户可以把 long 值类型的数据（如 Linux 时间）转化为位形式，这样可以使数据按日期降序排列。此外还有一种转换方式：

```
Long.MAX_VALUE - <date-as-long>
```

这样可以反转日期的排序，并保证时间字段值是降序排列的。

在以上例子中，用户可以把日期字段放在行键后。这只是一种组合方式，如果用户不需要按日期排序来查询数据，则可以将这个字段设为其他合适的内容。



之前我们采用的组合键看上去是一种通用的优秀方法，但其原子性是一个很突出的问题。由于一个收件箱中的数据现在分布在多行中，所以不可能在一个简单操作中修改一个收件箱的全局属性。如果用户不需要一次修改整个收件箱中所有邮件的消息，之前我们提到的高表设计就非常适合。但如果用户有这种修改需求，宽表可能更为适合，因为 HBase 能保证数据操作的行级原子性。

9.1.4 分页

使用以上方法，用户可以很方便地遍历查询数据子集的行。原理与之前介绍的类似，用户可以设定起始键和终止键来限制要扫描的范围。同时用户可以在客户端添加 offset 和 limit 参数来筛选数据。



用户可以使用 4.1.3 节中“分页过滤器 (PageFilter)”或“列分页过滤器 (ColumnPaginationFilter)”提到的功能来实现分页。本节提到的方法重点阐述如何使用键设计来达到分页的效果。

但如果纯粹只是简单分页，因为列分页过滤器可以避免传输多余数据，所以其仍是被推荐使用的方式。

具体步骤如下。

1. 在起始键的位置处打开一个扫描器。
2. 跳过 offset 数目的行。
3. 读取 limit 数目的行，并返回给上层应用。
4. 关闭扫描器。

在收件箱应用中，可以对一个用户的所有邮件进行分页。假设通常情况下一个用户的收件箱中有几百封邮件，一般他只会查阅前面几封，如前 50 封，那么余下的邮件需要通过单击 Next 按钮加载下一页。

客户端可以把起始行设为用户 ID，同时将结束行设为用户 ID+1。然后使用上面提到的方法，当 offset 为 0 时，用户可以读取 50 封邮件。当用户单击 Next 按钮时，offset 设为 50 并跳过前 50 行，并返回第 51~100 行。

当行数很少时，这种方法可行。但是，当需要对几千行数据进行分页时，就需要采用其他方法了。用户可以添加一个序列 ID 到行键中来帮助起始键定位到对应偏移量的位置。用户也可以在行键中使用日期字段。如果用户使用 Unix 时间，则可以计算上一次扫描午夜的最后一条记录。这样用户可以重新扫描前一天的数据来决定具体向用户返回哪些内容。

通过不同的行键设计，用户可以有很多方式来实现子集扫描和分页，例如，我们在前面收件箱应用的例子。使用之前由用户 ID 和日期组合的行键，可以让邮件按时间逆序排列，从而使最新的邮件最先被读取到。但是如果用户要完全按照其他字段排序，并在不同的排序方式中转换查询方式，则用户可以采用 9.3 节中介绍的方法。

9.1.5 时间序列

当处理流式事件时，最常见的数据就是按时间序列组织的数据。这些数据可能来自电网的一个传感器、一个股票交易软件或一个信息化的监控系统。这些数据的突出特点是它们的行键都代表了事件发生的时间。由于 HBase 的数据组织方式，这样的数据在存储时会出现一个问题：这些数据会被有序存储到一个特定的范围内，也就是一个有特定起始键和停止键的 region 中。

由于一个 **region** 只能由一个服务器管理，所以所有的更新都会集中在一台服务器上。这会导致系统产生读写热点，并由于写入数据过分集中而导致整个系统性能下降。

要解决这个问题，用户需要想办法将数据分散到所有的 **region** 服务器上去。有很多方法可以达到这个目的，例如，在行键前添加一个不连续的前缀。通常情况下有如下选择。

salting 方式

用户可以使用 **salting** 前缀来保证数据分散到所有 **region** 服务器。例如：

```
byte prefix = (byte) (Long.hashCode(timestamp) % <number of region  
servers>);  
byte[] row key = Bytes.add(Bytes.toBytes(prefix), Bytes.toBytes(timestamp));
```

这个公式将产生足够的前缀数以确保将数据分散到所有 **region** 服务器中去。当然，这个公式假定服务器数目固定，如果用户的集群规模可能扩大就应当将前缀数翻倍。生成的行键可能如下：

```
0myrowkey-1, 1myrowkey-2, 2myrowkey-3, 0myrowkey-4, 1myrowkey-5, \  
2myrowkey-6, ...
```

当键被排序之后并发送到对应的 **region** 时，它们的顺序如下：

```
0myrowkey-1  
0myrowkey-4  
1myrowkey-2  
1myrowkey-5  
...
```

换句话说，0myrowkey-1 和 0myrowkey-4 这两条更新操作会被送到同一个 **region**（假设以 0 开头的数据还没有跨多个 **region**，不然数据会更分散），同时 1myrowkey-2 和 1myrowkey-5 会被送到另一个 **region**。

这样做的缺点是当用户要扫描一个连续的范围时，可能需要对每个 **region** 服务器都发起请求（因为之前连续的数据，现在已经分散到不同的服务器中）。这样也会带来好处，用户可以多线程并行地读取数据。这有些类似于一个小规模的 MapReduce 作业，这样查询的吞吐量会有所提高。

使用场景：Mozilla Socorro

Mozilla 为 Firefox 和 Thunderbird 建立了一个名为 Socorro^① 的崩溃报告系统，这个系统用来存储用户报告的程序异常及相关信息。这些报告随后会被 Mozilla 开发小组用来优化他们的软件，从而让他们的软件产品可以在不同的平台和配置下运行得更稳定。

① 详情请查看关于 Socorro 的 Mozilla 维基百科页面（<https://wiki.mozilla.org/socorro>）。

这个项目是开源的，代码可以在网上下载，它使用 Python 客户端通过 Thrift 与 HBase 集群进行通信。下面的例子展示了（在写这本书时）客户端如何在扫描时处理包装过之后的键。

```
def merge_scan_with_prefix(self, table, prefix, columns):
    """
    A generator based iterator that yields totally ordered rows starting with a
    given prefix. The implementation opens up 16 scanners(one for each leading
    hex character of the salt)simultaneously and then yields the next row in
    order from the pool on each iteration.
    """

    iterators = []
    next_items_queue = []
    for salt in '0123456789abcdef':
        salted_prefix = "%s%s" %(salt, prefix)
        scanner = self.client.scannerOpenWithPrefix(table, salted_prefix, columns)
        iterators.append(salted_scanner_iterable(self.logger, self.client,
            self._make_row_nice, salted_prefix, scanner))
    # The i below is so we can advance whichever scanner delivers us the polled
    # item.
    for i, it in enumerate(iterators):
        try:
            next = it.next
            next_items_queue.append([next(), i, next])
        except StopIteration:
            pass
    heapq.heapify(next_items_queue)

    while 1:
        try:
            while 1:
                row_tuple, iter_index, next = s = next_items_queue[0]
                #tuple[1] is the actual nice row.
                yield row_tuple[1]
                s[0] = next()
                heapq.heapreplace(next_items_queue, s)
            except StopIteration:
                heapq.heappop(next_items_queue)
        except IndexError:
            return
```

在 Python 代码按照需求数量创建了扫描器，它们与相应的包装前缀对应，在这这是总共 16 个不同的字符中的一个。注意这里使用 heapq 类来合并实际的数据和结果集的全局排序。

字段交换/提升权重

使用 9.1.3 节介绍的方法，用户可以将时间戳字段移开或添加其他字段作为前缀。这样做其实是想利用组合行键的思想来让连续递增的时间戳在行键中的位置从第一位变到第二位。

如果用户设计的行键已经包含多个字段了，则可以调整它们的位置。如果行键只包含了时间戳，则用户应当将其他字段从列键或值中提取出来，然后放到行键的前端。

将时间戳从组合键的左边向右移动也有缺点：用户能访问数据，尤其是时间范围内的数据，并使用一个给定的字段。

使用场景：OpenTSDB

OpenTSDB^① 项目提供了一个时间序列数据库，该数据库用来存储由外部代理程序收集的服务器和服务的各项监控指标 (metric)。所有数据都存储在 HBase 中，用户可以使用 UI 来查询各项指标，并进行实时组合抽样。

表模式 (schema) 就把监控指标 ID 放在了行键中，并形成了以下结构：

```
<metric-id><base-timestamp>...
```

由于生产系统有很多种监控指标，它们的 ID 取值的字段很分散，所以所有的更新操作也就被分散开了，用户最后得到了与 salt 前缀类似的效果：读写操作都分散到了各个不同的监控指标 ID 下。

这种方式很适合主要按行键前面部分查询的系统。如上例 OpenTSDB 中的行键结构，可以很方便地为用户在 UI 中提供按时间顺序显示一个或多个用户选择的监控指标的最近结果。

随机化

另一种完全不同的方式是将行键随机化，例如：

```
byte[] row key = MD5(timestamp)
```

采用 MD5 之类的散列函数能将行键分散到所有的 region 服务器上。对于时间连续的数据，这种方法明显不是个好方法。因为随机化之后，用户将不能再按时间范围扫描数据。

另一方面，由于用户可以用散列的方式重新生成行键，随机化的方式很适合每次只读取一行数据的应用。如果用户的数据不需要连续扫描而只需随机读取，用户就可以使用这种策略。

① 详情请查阅 OpenTSDB 项目的网站 (<http://opentsdb.net>)，尤其推荐阅读关于它的模式的页面 (<http://opentsdb.net/schema.html>)，因为它为需要高性能存储数据查询的应用提供了先进的键设计概念。

简单总结以上方法后，用户会发现在优化读写性能的同时找到正确的平衡点并不是一件简单的事情。它关系到用户的数据访问模式，该模式最终决定了用户如何设计行键的结构。图 9-3 展示了不同解决方案对读写性能的影响。

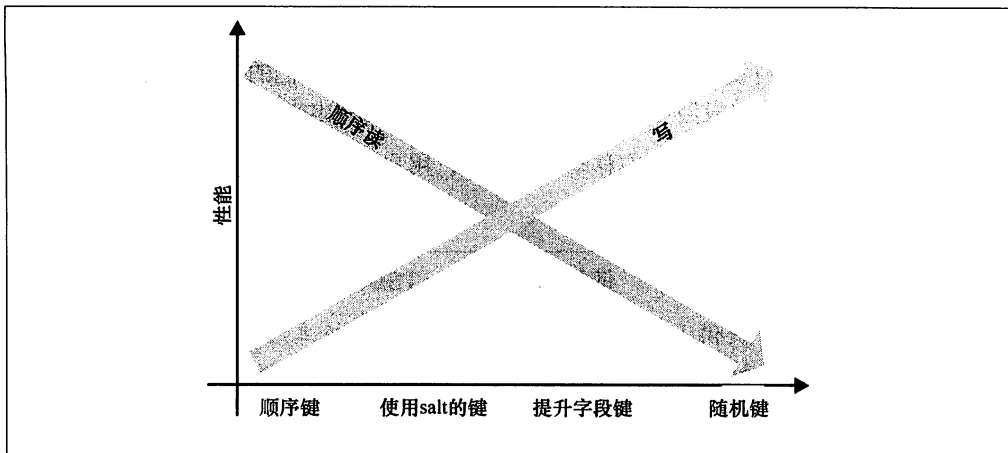


图 9-3 寻找顺序读写性能的平衡点

使用 salt 前缀或将某些不是连续取值的主键字段提前，可以使分散写压力并提高写入性能，同时扫描连续的键子集也可以提高读性能。但是，如果用户只需要随机读取数据，那么随机行键就更有用，因为它能完全避免某个 region 成为读写热点。

9.1.6 时间顺序关系

以上数据都会按照产生的时间顺序以独立行插入到 HBase 中，但是也可以使用另一种方式，即将新的事件以发生时间为列进行插入。因为列在 HBase 中是按列族组织的，所以每个列族下的列可以作为一个辅助索引单独进行排序，如同 RDBMS。虽然这不是推荐的设计模式，但少量的索引可能正是用户所需要的。

以之前的收件箱应用为例，它将用户的所有邮件存在一行中。由于用户需要按照收件顺序显示邮件，同时也有可能需要按照题目等顺序进行排序。设计时可以利用基于列的排序来显示收件箱的不同视图。



记住，不要为一张表设置过多的列族，特别是当数据量大的列族和数据量小的列族混用（大小指的是存储的数据量），用户可以把收件箱的邮件存储在一张表中，同时把辅助索引存在另一张表中。缺点是这样用户不能保证修改两张表的原子性（HBase 只保证行级原子性）。同时请参 9.3 节来克服这个限制。

用户首先要决定存储数据的主要顺序是什么，也就是说，用户大多数情况下会按什么顺序检索收件箱数据。假设按照收件顺序则可以按之前提到过的时间降序排列，这样

用户就需要将邮件的时间戳反序排列来达到按时间逆序排列的目的。

```
Long.MAX_VALUE - <date-as-long>
```

邮件内容存储在主要的列族下，索引被单独存储在另一个列族下。用户可以把邮件题目提取出来并添加到列键的前面来建立辅助索引。如果用户需要对题目进行降序排列，则需要再额外添加一个列族。

为了避免太多的列族，用户可以把所有辅助索引存储在一个单独的列族下，同时列键（列名）的最左段使用索引 ID 这个前缀来表示不同的排序，例如，idx-subject-desc 和 idx-to-asc 等。接下来用户要填入实际的排序值。单元格的值是主索引的键，同时主索引中存储着消息的信息。用户需要从以下 3 种存储方式中选择一种：从主表（主索引）中读取消息的内容，只展示辅助索引中存储的信息，或把消息中的信息冗余地存储到辅助索引中从而避免在主信息源中进行随机读取。反范式化在 HBase 中十分常见，它可用于减少读取时间，从而大大提高用户响应速度。

将以上的表设计模式付诸实施可以得到如下表：

```
12345 : data : 5fc38314-e290-ae5da5fc375d : 1307097848 : "Hi Lars,..."
12345 : data : 725aae5f-d72e-f90f3f070419 : 1307099848 : "Welcome,and ..."
12345 : data : cc6775b3-f249-c6dd2b1a7467 : 1307101848 : "To Whom It ..."
12345 : data : dcbee495-6d5e-6ed48124632c : 1307103848 : "Hi,how are ..."
...
12345 : index : idx-from-asc-mary@foobar.com : 1307099848 : 725aae5f- d72e...
12345 : index : idx-from-asc-paul@foobar.com : 1307103848 : dcbee495- 6d5e...
12345 : index : idx-from-asc-pete@foobar.com : 1307097848 : 5fc38314- e290...
12345 : index : idx-from-asc-sales@ignore.me : 1307101848 : cc6775b3- f249...
...
12345 : index : idx-subject-desc-\xa8\x90\x8d\x93\x9b\xde : \
1307103848 : dcbee495-6d5e-6ed48124632c
12345 : index : idx-subject-desc-\xb7\x9a\x93\x93\x90\xd3 : \
1307099848 : 725aae5f-d72e-f90f3f070419
...
```

在前面这段代码中，一个索引（idx-from-asc）按照电子邮件地址升序排列，另一个索引（idx subject-desc）按照题目降序排列。同时题目按照位反序（bit-inversed）排列以达到降序排列的目的，所以其内容已经不可读了。例如：

```
% String s = "Hello,";
% for(int i = 0;i < s.length();i++){
    print(Integer.toString(s.charAt(i)^ 0xFF,16));
}
b7 9a 93 93 90 d3
```

所有的索引值都存储在 index 列族下，并使用之前的前缀。客户端可以读取整个列族中并缓存其内容，从而可以快速转换邮件排序。在数据量巨大的情况下，用户可以读出索引（idx-subject-desc）的前 10 列来展示按标题排序的前 10 封邮件。使用行内扫描

(参见 3.5.3 节) 可以实现辅助索引分页功能。另一个选择是使用列分页过滤器 (ColumnPaginationFilter), 并与列前缀过滤器 (ColumnPrefixFilter) 组合来按页遍历索引。

9.2 高级模式

到目前为止, 我们已经讨论了怎样利用所提供的表模式把数据映射到 HBase 支持的面向列的存储结构中。用户需要设计行键和列键, 以优化应用的数据访问性能。

每列的值都是可以存储为任意字节数组的实际数据存储结点。这种可以随意添加列的存储模式让用户设计修改客户端程序时拥有更多自由发挥的余地, 当然也有些使用场景需要更正式、有更多功能且可以更新的序列化 API, 此时每列的值都可以表示更复杂的、嵌套的结构。

可能的解决办法包括已经讨论过的序列化包——详情请查看 6.1 节, 以下是一些例子。

Avro

HAvroBase^① 是一个使用 Avro 在每列中存储复杂记录的典型项目。它使用 Avro 的接口定义语言 (缩写为 IDL) 来定义实际的模式, 这个模式被用于在表的任意列中存储按 Avro 形式序列化的记录。

Protocol Buffer

与 Avro 类似, 用户可以用 Protocol Buffer 的 IDL 来定义一个外部模式, 这个模式被用来序列化复杂的数据结构到 HBase 的列中。

这种方法本质上是为用户提供一种可以定义初始模式的语言, 然后用户可以通过增加或者删除字段来更新这个模式。序列化 API 能够使用新模式来读取旧模式, 缺失的字段被忽略或者用默认值来填充。

9.3 辅助索引

尽管 HBase 没有为辅助索引提供原生支持, 但是有些应用场景仍需要使用辅助索引。通常的需求是用户能够通过主坐标 (行键、列族和列限定符) 来查找一个单元格, 也可以通过一个其他类型的坐标来进行查找。此外, 用户可以按照辅助索引的顺序从主表扫描数据。

与关系型数据库系统中的索引类似, 辅助索引存储了一个新坐标和现有的坐标之间的映射关系。以下列出了一些可行的解决方案。

① 查看 HAvroBase (<https://github.com/spu/ara/hourobases>) 在 GitHub 上的项目页面。

把责任完全转移到应用层的典型做法是把一个数据表和一个（或者多个）查找/映射表结合起来。每当程序写数据表时，它也同时更新映射表（也被称为辅助索引表）。读数据时可以直接在主表中进行查询，从辅助索引表中先查找原表行键，再在原表中读取实际数据。

这种做法既有优点也有缺点。首先，优点是整个逻辑都由客户端代码处理，用户可以按照需求设计映射关系。不过这样做的缺点更多，由于 HBase 中不能保证跨行操作的原子性，例如，以事务的角度来看，用户不能保证主表和依赖表的一致性。用户可以使用定期修剪工作来部分解决这个问题，例如，使用 MapReduce 程序来扫描表，删除过时的条目，或增加缺失的条目。

缺少事务的支持可能导致数据被存储在数据表中，但是在辅助索引表中没有相应的映射。这种情况可能发生在主表被更新后且索引表被成功写入前，如果这段时间出现了任何导致操作失败的问题都会使辅助索引和主表不一致。这个问题可以通过先写辅助索引表，在操作的最后再写数据表来缓解。如果在这个过程中有任何操作失败，就会出现孤立的映射，不过它们很容易被异步的定期修剪工作删除掉。

用户可以自由设计主索引和辅助索引之间的映射关系时，必须接受的缺点是用户需要实现所有存储和查找数据必需的方法。这种情况下，需要定义额外的规则来让应用访问正确的表，例如：

```
myrowkey-1
@myrowkey-2
```

第一个键表示直接在数据表中查找，但是第二个使用了“@”前缀的键则表示需要通过一个辅助索引表来完成一次映射。表的名字也可能需要通过数字编码或添加前缀来区分，另一种做法是，在应用程序中硬编码，并随模式改变而变更需求。

带索引的事务型 HBase

开源的带索引的事务型 HBase (Indexed-Transactional HBase, 简称为 ITHBase) 项目^①提供了一个不同的解决方案。它扩展了 HBase，并增加了特殊的客户端和服务端类的实现。

最核心的扩展是增加了用来保证所有的辅助索引更新操作一致性的事务功能。在此基础上，提供了一个可以客户端类 IndexedTableDescriptor，这个类定义了一个

^① ITHBase 项目最开始是 HBase 的一个 contrib 模块。它随后被转移到一个外部库，使其能够定义 HBase 的不同版本，并且根据自己的进度来开发。详情参见 GitHub 的项目页面，见 <https://github.com/hbase-trx/hbase-transactional-tableindexed>。

数据表的辅助索引支持。

大多数客户端和服务端端的类都被添加了索引支持功能的类替换掉了。例如，在客户端上，`HTable` 被 `IndexedTable` 替换掉。它有一个新方法叫做 `getIndexedScanner()`，该方法能让用户按辅助索引的顺序在表中取得数据。

以上所述的客户端管理索引在单独的表中存储主键和辅助键之间的映射关系，与之不同的是，这些操作在 `ITHBase` 中被自动处理，同时主表与辅助索引表之间的关系由相应的描述符（`descriptor`）定义。结合事务支持的索引更新，这个方案提供了一个 `HBase` 辅助索引的完全实现。

它的缺点是可能不支持最新可用的 `HBase` 版本，因为它不与 `HBase` 绑定发行。同时它也增加了同步的开销，这将导致性能的下降，所以需要相应的基准测试验证其可用性。

带索引的 HBase

在 `HBase` 中增加辅助索引的另外一种解决方案是带索引的 `HBase`（简称为 `IHBase`）^①。这种解决方案放弃了为每个索引使用单独的表，而是完全在内存中维护索引。当一个 `region` 第一次打开，或者一个 `memstore` 被刷写到磁盘上时，用户可以通过扫描整个 `region` 来建立索引。根据用户配置的 `region` 大小的不同，这个操作可能花费大量的时间和 I/O 资源。

当磁盘上的数据有索引时，内存中数据搜索的方式如下：它直接使用内存中的数据来搜索索引相关的详细数据。这个方案的优点是索引永远都是同步的，并且不需要额外的事务控制。

与基于表的索引相比，使用这种方法有两方面不同。一方面它很快，所有需要的索引数据都在内存中，因此可以执行很快的二分查找来定位行。另外一方面，它也需要大量额外的堆空间来维护索引。由于用户的需求和想要索引的数据量不同，`IHBase` 有时并不能建立用户需要的所有索引。

内存中的索引包含类型支持，并提供了更细粒度的排序和更高效的内存存储。支持的类型包括 `BYTE`、`CHAR`、`SHORT`、`INT`、`LONG`、`FLOAT`、`DOUBLE`、`BIG_DECIMAL`、`BYTE_ARRAY` 和 `CHAR_ARRAY`。数据总是以升序存储。如果需要按降序排列，用户需要像前文所述的把值按位反转。

索引的定义由 `IdxIndexDescriptor` 类完成，它定义了数据表的哪些列添加了索引以及索引的数据类型，该类型来自于上面的列表。

^① 与 `ITHBase` 类似，`IHBase` 最开始也是 `HBase` 的一个 `contrib` 项目。它因为同样的原因被转移到一个外部库。详情参见 GitHub 中该项目的页面。（<http://github.com/ykulbaklihbse>）它的 JIRA 问题原始文档在网上的 `HBASE-2037`（<https://issues.apache.org/jira/browse/HBASE-2037>）里。

用户可以通过 `IdxScan` 类来按索引访问数据，它通过定义 `Expression` 来扩展正常的 `Scan` 类。没有明确表达式的扫描默认是正常的扫描行为。表达式提供了用 `And` 和 `Or` 来构造基本的布尔逻辑构造。例如：

```
Expression expression = Expression
    .or(
        Expression.comparison(column family1,qualifer1,operator1,value1)
    )
    .or(
        Expression.and()
            .and(Expression.comparison(column Family2,qualifer2,operator2, value2))
            .and(Expression.comparison(column Family3,qualifer3,operator3, value3))
    );
```

例子中使用创建者模式风格的辅助方法生成一个结合了 3 个不同索引的复杂表达式。表达式最低级别的操作是 `Comparison`，它允许用户指定实际索引和一个类似于过滤器的语法来选择匹配比较值和运算符的值。表 9-2 列出了可能选择的运算符。

表 9-2 `Comparison.Operator` 可选值

运算符	描述
EQ	相等运算符
GT	大于运算符
GTE	大于等于运算符
LT	小于运算符
LTE	小于等于运算符
NEQ	不等于运算符

用户必须指定一个列族和现有索引的限定符，否则将抛出 `IllegalStateException` 异常。

`Comparison` 类有一个可选的 `includeMissing` 参数，它同 4.1.3 节的“单列值过滤器 (`SingleColumnValueFilter`)”中所描述的 `filterIfMissing` 参数类似。用户可以使用这个参数对扫描中应返回的行进行微调。

排列顺序由在表达式中第一个被处理的索引来定义，而其他索引则被用来和第一个索引求交集 (`and`)，或者求并集 (`or`)。换句话说，只有使用相同的索引时，复杂表达式的排序才是可预见的。

`IHBase` 相较于 `ITHBase` 的好处是它获得了相同的一致性保证——维护基于数据表中已存储的列的索引的一致性——但是，无需使用额外的表。同时，它具有以

下缺点。

- 它的入侵性很强，因为它需要额外的 JAR 文件和配置来替换重要的客户端类和服务端类。
- 它需要额外的资源，尽管它用内存交换了额外的 I/O 需求。
- 它在按照基于辅助索引定义的排列顺序查询数据时，其需要在数据表上做随机查找。
- 它在最新版本的 HBase^① 中可能无法使用。

协处理器

目前也有一些方法基于协处理器来实现索引方案^②。使用协处理器框架提供的服务器端的钩子函数可以实现类似于 ITHBase 和 IHBase 的索引，并且不用替换任何客户端类和服务端类。协处理器将为每一个 region 载入索引层，并维护索引。

代码可以利用扫描器钩子透明地遍历一个正常的数据表，也可以遍历这张表上的索引视图。索引的定义可能需要存放在外部模式中，并由一个协处理器基类读取，或者附加在一个列族的属性定义中。



由于这些工作都还处于早期阶段，编写本书时可能还没有太多相关的内容介绍。如果你感兴趣，可以留意在线问题跟踪系统以查看相关更新。

9.4 搜索集成

使用索引用户可以按照行键以外的顺序来遍历数据表。但用户仍然受限于是使用键或过滤器来筛选数据，或者直接遍历数据来查找所需的内容。一个非常普遍的需求是使用任意关键字来搜索数据，满足这种需求往往需要集成一个完整的搜索引擎。

常见的选择是基于 Apache Lucene 的解决方案，例如，使用 Lucene 或者 Solr（一个基于 Lucene 的高性能的企业级搜索服务器^③）。类似于索引解决方案，以下是一系列可行的方法。

客户端管理

客户端管理的实现需要使用 HBase 存储数据，同时使用 MapReduce 任务来建立索引，

① 截止到写这本书的时候，IHBase 只支持到 HBase 的 0.20.5 版本。

② 详情参见 JIRA 问题跟踪系统里的 HBASE-2038 (<http://issues.apache.org/jira/browse/HBASE-2038>)。

③ Solr 基于 Lucene，并扩展了 Lucene 以提供一个功能完整的搜索服务。详情请参见该项目网站 (<http://lucene.apache.org/>)。

还需使用 HBase 作为 Lucene 的后台存储。另一种实现方法是把数据表的更新也转发到邻近的索引服务器中。在 HBase 上实现通过索引查找数据的方法取决于使用 HBase 作为数据的存储还是作为索引的存储。

一个不错的客户端管理解决方案的实现是 Facebook 的收件箱搜索系统 (Facebook inbox search)。以下是其大致模式。

- 每一行是一个单独的收件箱，即每个用户在搜索表中都有一个单独的行。
- 列是消息中被索引的词语。
- 版本是消息 ID。
- 值包括附加信息，例如，词组在文件中的位置。

这个模式使得用户很容易在收件箱中搜索包含特写关键词的消息。布尔运算符，例如，and 或 or，可以在客户端逻辑中实现，并用来归并找到的文件列表。用户也可以有效地实现输入提示查询，用户可以开始键入一个词，搜索将找到所有包含以用户输入作为前缀的词的消息。

Lucene

独立于 HBase 使用的 Lucene 或者其派生的解决方案可以通过 MapReduce 来建立索引。一个外部托管的项目^①提供了 BuildTableIndex 类，这个类以前是 HBase 中 contrib 包的一部分。该类会扫描整个表，并建立 Lucene 索引，索引最终以 HDFS 上的目录形式来存储，索引的数量取决于使用的 reducer 数目。这些索引可以被下载到基于 Lucene 的服务器上，并且使用类似于 Lucene 提供的 MultiSearcher 类来进行本地访问。

另外一种方法是通过运行只有一个 reducer 的 MapReduce 作业，或者使用 Lucene 中的索引合并工具来合并多个索引。合并后的索引通常性能更好，但是索引的建立、合并，以及最后能够提供服务所需要经历的时间将更长。

在一般情况下，这种方法只使用 HBase 来存储数据。如果通过 Lucene 执行一个搜索，通常只返回匹配的行键。随机查找数据表需要显示文档。如果查找的文档数很多，查找过程可能需要相当长的时间。一个更好的解决方案是把搜索与存储的数据直接结合起来，从而避免额外的随机查找开销。

HBasene

Hbasene^②选择的方法是直接在 HBase 内部建立搜索索引，同时为用户提供 Lucene 的 API。它把每个文档的字段 (field)、词 (term) 存储在一个单独的行，同时将包

① 请到 GitHub 的项目页面查看详细信息并获取代码 (<https://github.com/akkumar/hbasene/tree/master/src/main/java/org/hbasene/Index/create/mapred>)。

② GitHub 页面有详细信息和源代码 (<https://github.com/akkumar/hbasene>)。

含这个词的文档存储在这一行的列中。

它同时也在同一张表中存储了其他支持 **Lucene** 查询的相关信息。它实现了一个 **IndexWriter**，该类可以直接把文档存储在 **HBase** 表中，就像使用正常的 **Lucene API** 插入文档一样。下面是一个 **HBase** 测试类的例子。

```
private static final String[] AIRPORTS = { "NYC","JFK","EWR","SEA",
      "SFO","OAK","SJC" };

private final Map<String,List<Integer>> airportMap =
    new TreeMap<String,List<Integer>>();

protected HTablePool tablePool;

protected void doInitDocs() throws CorruptIndexException,IOException {
    Configuration conf = HBaseConfiguration.create();
    HBaseIndexStore.createLuceneIndexTable("idxtbl",conf,true);
    tablePool = new HTablePool(conf,10);
    HBaseIndexStore hbaseIndex = new HBaseIndexStore(tablePool,conf,
        "idxtbl");
    HBaseIndexWriter indexWriter = new HBaseIndexWriter(hbaseIndex,"id")
    for(int i = 100;i >= 0;--i){
        Document doc = getDocument(i);
        indexWriter.addDocument(doc,new StandardAnalyzer(Version.LUCENE_30));
    }
}

private Document getDocument(int i){
    Document doc = new Document();
    doc.add(new Field("id","doc" + i,Field.Store.YES,Field.Index.NO));
    int randomIndex =(int)(Math.random() * 7.0f);
    doc.add(new Field("airport",AIRPORTS[randomIndex],Field.Store.NO,
        Field.Index.ANALYZED_NO_NORMS));
    doc.add(new Field("searchterm",Math.random() > 0.5f ?
        "always" : "never",
        Field.Store.NO,Field.Index.ANALYZED_NO_NORMS));
    return doc;
}

public TopDocs search() throws IOException {
    HBaseIndexReader indexReader = new HBaseIndexReader(tablePool, "idxtbl",
        "id");
    HBaseIndexSearcher indexSearcher = new HBaseIndexSearcher (indexReader);
    TermQuery termQuery = new TermQuery(new Term("searchterm","always"));
    Sort sort = new Sort(new SortField("airport",SortField.STRING));
    TopDocs docs = this.indexSearcher.search(termQuery
        .createWeight(indexSearcher),null,25,sort,false);
    return docs;
}
```

```

public static void main(String[] args) throws IOException {
    doInitDocs();
    TopDocs docs = search();
    // use the returned documents...
}

```

这个例子创建了一个简单的用于测试的索引，并在其上执行了查询。用户可能注意到类似于 Lucene API 有很多小的修改和不同，这些修改是为了支持以 HBase 为后台的索引写操作。



截至写本书的时候，这个项目更像是一个验证可行性的原型系统，而不是一个已经可以作为产品使用的实现。

协处理器

另一种方法实现了一个类似于基于 Lucene 的数据查询功能的数据表，该方法目前正在开发中^①，并基于协处理器。使用协处理器提供的钩子函数来维护索引，索引直接存储在 HDFS 上。每个 region 都有自己的索引，通过搜索分布在所有 region 上的索引以获取完整的结果。

这只是一个展示协处理器功能的例子。与使用协处理器建立辅助索引类似，用户可以选择存储实际索引的位置：既可以存储在另一个表中，也可以存储在其他的外部存储空间。这个框架提供了很大的选择余地，用户实现的代码可以按自己的需求作出选择。

9.5 事务

讨论 HBase 事务看起来有点违背用户对 HBase 的第一印象。但是，辅助索引的例子展示了，在一些应用场景下，用户可以对 HBase 提供的简单数据模型稍稍放心，并且可以引入一些在传统关系型数据库系统下常见的概念。

其中的一个概念就是事务，传统关系型数据库通常能够提供跨行和跨表的 ACID 保证。有时 HBase 也需要这些保证。例如，更新数据表和对应的辅助索引表时，就需要事务来保证一致性。

通常情况下，用户并不需要事务，因为范式化的数据模式能够被转化成单个表的存储，所以此时不需要分布式事务的支持，同时也不需要为保证一致性而产生额外的开销。

① HBASE-3529 (<https://issues.apache.org/jira/browse/HBASE-3529>)。

但是，如果用户还需要一致性支持，以下是几种可选的解决方案。

事务型 HBase

带索引的事务型 HBase (Indexed Transactional HBase) 项目有一些取代默认客户端类和服务端类的扩展类，它们增加了跨行甚至跨表的事务支持。在 region 服务器中，更准确地说，每个 region 都保持了一个事务的列表。该列表是由 `beginTransaction()` 调用初始化，并且由相应的 `commit()` 调用结束。每次读写操作都有一个事务 ID，以保护调用不受其他事务影响。

ZooKeeper

HBase 运行时需要一个 ZooKeeper 集群，它在集群的启动中扮演着种子或引导启动的角色。有一些可用的模板和方法展示了如何将 ZooKeeper 作为一个事务来控制后端。例如，Cages (<http://code.google.com/p/cages/>) 项目提供了一个锁的抽象概念，即跨多个资源的锁，并计划增加一个专门的事务类来把 ZooKeeper 当作分布式协调系统使用。

ZooKeeper 也提供了一个能够被用于实现两阶段提交协议的锁方案。它使用一个特定的 `znode` 来代表事务，并且每个参与的客户端对应一个孩子 `znode`。客户端可以使用自己的 `znode` 标志自己在事务中的那部分是成功还是失败。其他客户端可以监控同级的 `znode`，并采取适当的行动^①。

9.6 布隆过滤器

5.1.3 节介绍了在列族定义时声明的布隆过滤器 (Bloom filter) 语法，并指出布隆过滤器在某些特定的使用场景中更有意义。

使用布隆过滤器的根本原因是默认机制决定了一个存储文件是否包含特定的受限于可用块索引的行键，同时这个索引又是相当粗粒度的，该索引只存储了文件包含块的开始键。例如，系统使用默认的 64 KB 作为块大小，这样会将一个 1 GB 的存储文件分成 16384 个块，与索引到的行键数相同。

如果我们进一步假设每个单元格的平均大小是 200 字节，那么用户将处理存储在一个文件中的超过 500 万个单元格。如果用户随机查找一个行键，则这个行键很可能处在两个块开始键之间的位置。对于 HBase 来说，判断这个键是否真实存在的唯一方法是加载这个块，并且扫描它来查找这个键。

同时以下情况会使上述问题变得更复杂，对于一个典型的应用程序来说，用户通

① 可以在 ZooKeeper 项目 (<http://zookeeper.apache.org/doc/trunk/recipes.html#sc-recipes.twoPhasedCommit>) 页面找到更多详细信息。

常会以一定速率更新数据，这将导致内存中的数据被刷写到磁盘上，并且之后系统会把它们合并成更大的存储文件。由于 minor 合并仅合并最近几个存储文件，直到合并后的文件达到配置的最大大小。最终系统中将会有很多存储文件，所有的这些文件都是候选文件，其可能包含一些用户请求行键的单元格，如图 9-4 中的例子所示。

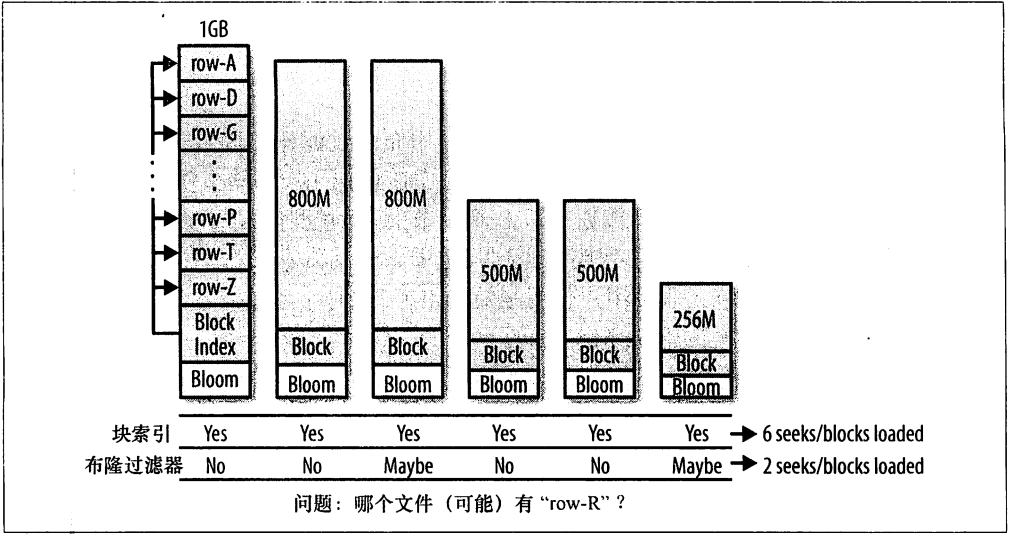


图 9-4 使用布隆过滤器能够帮助减少 I/O 操作的数量

这些文件都来自同一个列族，所以它们行键的分布很相似。尽管只有几个文件包含特定行的更新，文件的块索引还是覆盖了整个行键范围。块索引能确定文件中是否包含某个行键。region 服务器需要加载每一个块来检查该块中是否实际包含该行的单元格。

另一方面，使用布隆过滤器的好处是，用户可以立即判断一个文件是否包含特定的行键。这个过滤器的特性是：如果这个文件不包含这个行，它会给你一个明确的答复；但是如果文件包含这一行时，答复却可能有误，即它声称文件包含这个数据而实际却并非如此。错误肯定答复的数量可以被调整，通常被设置为 1%，这意味着过滤器中关于一个文件包含一个请求行的报告中有 1%是错误的，因此可能会有一个块被错误地加载和检查。



对单个的 get 操作来说，这并不能转化为即时的性能提升，因为 HBase 的读取是并行的，并且最终被读磁盘延时约束。减少不必要的块加载数量可以提高整个集群的吞吐率。

用户可以从例子中看出块加载数量大大减少了，这在负载很重的系统中会产生很大的影响。为了提高效率，用户还必须使用一种特定的更新模式：如果用户定期

修改所有行，那么大部分的存储文件都将包含用户查找行的数据。这种场景不适合使用布隆过滤器。但是，如果用户批量更新数据，使得一行数据每次只被写入到少数几个存储文件中，那么过滤器就能够为减少整个系统 I/O 操作的数量发挥很大作用。

用户还会在使用块缓存时发现另一个优点。因为加载更少的块将导致更少的缓存波动，所以缓存的命中率也会相应提高。由于服务器在大部分时候加载的都是包含请求数据的块，相关的数据将有更大的机会留在块缓存中，随后读操作可以使用这些数据。

除了更新模式，另一个决定是否在应用场景中使用布隆过滤器的因素是添加开销。每项会在过滤器中占用约 1 字节的存储空间。回到上面的例子中，存储文件的大小是 1 GB 时，假设用户只存储计数器类型数据（即 8 个字节的 long 型值），并且加上 Key-Value 信息（即它的坐标，或行主键、列族名、列限定词、时间戳和类型）的开销，那么每个单元格大概是 20 字节（假设用户使用很短的键）。此时布隆过滤器的大小将是存储文件的二十分之一，大概占用 51 MB 空间。

现在假设用户的单元格的平均大小是 1 KB，那么过滤器只需要 1 MB 空间。考虑到过滤器在今后需要发挥优化作用，与一个 1 GB 甚至更大的文件相比，一个几百 KB 的行级布隆过滤器的开销甚小，这种情况下使用过滤器是有用的。

最后一个是：使用行级还是行加列级的布隆过滤器？这取决于用户的使用模式。如果用户只做行扫描，那么使用更具体的行加列级的布隆过滤器将不会有任何帮助。即使用户使用行加列的读操作时，使用行级的布隆过滤器仍然可以减少需要检查的文件数量，但是反过来使用行加列级的布隆过滤器却不能为行扫描提供更好的性能。

当用户不能批量更新特定的一行，并且最后所有的存储文件都包含该行的一部分时，行加列级的布隆过滤器很有用。更具体的行加列级的过滤器能够识别出哪个文件包含用户请求的数据。显然，如果用户总是加载整行，那么这个过滤器仍然很难起到作用，因为 region 服务器无论如何都需要加载每个文件中匹配的块。

因为行加列级的过滤器将需要更多的存储空间，所以用户需要计算是否值得花费额外资源。还有一点很有趣，布隆过滤器能够容纳的元素数量有一个最大值。如果用户的存储文件中有太多单元格，实际的元素数量可能会超过最大值，并且需要回退到使用行级过滤器。

图 9-5 总结了不同级别布隆过滤器的选择标准。

根据用户的应用场景，使用布隆过滤器可能有益于提高整个系统的性能。如果可能的话，用户应当尽量使用行级布隆过滤器，因为它在额外的空间开销和利用选择过滤存储文件提升性能之间获得了很好的平衡。只有当用户使用行级布隆过滤器没有性能提

升的时候，再考虑使用开销更大的行加列级布隆过滤器。

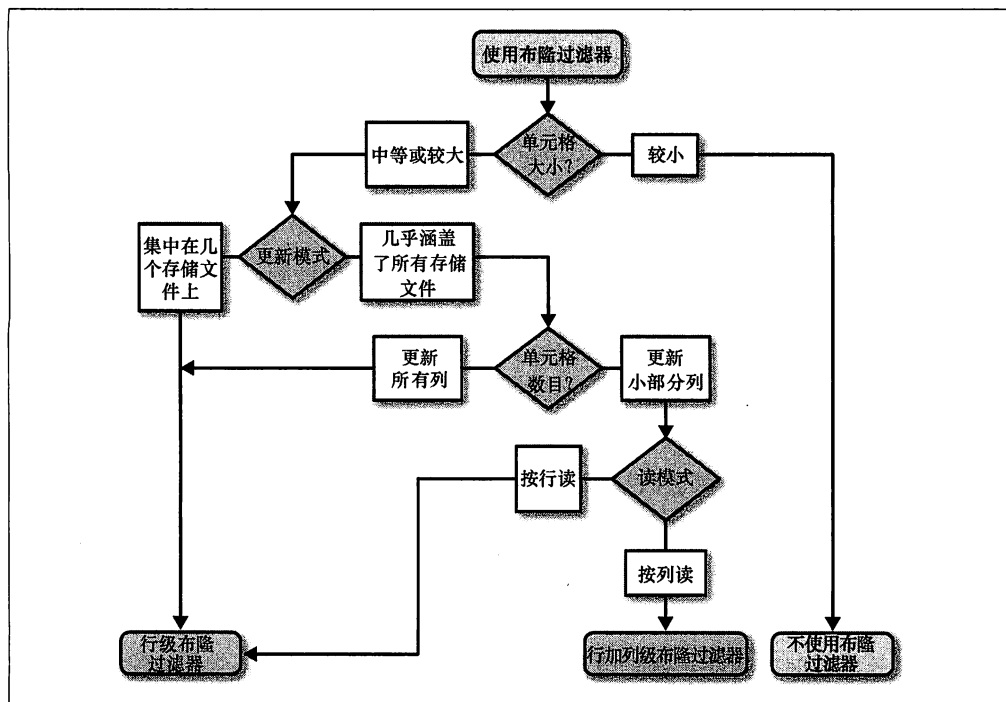


图 9-5 使用哪种布隆过滤器的选择标准

9.7 版本管理

我们需要在回顾 HBase 如何存储和检索数据以后，再来了解版本管理机制。用户使用时间戳时需要注意一些高级技巧（假设用户已经了解它们的行为模式），这些高级技巧可能适用于某些特定的使用场景。但是，同时这些内容也暴露了版本管理的的复杂性，用户也需要了解这些。

9.7.1 隐式版本控制

在用户确保具服务器上的时钟是同步的之前，有些问题就已经被指出了。其中一个可能出现的问题是，当用户跨服务器把数据存储在多行中时，使用隐式的时间戳可能让用户最终得到完全不同的时间集。

例如，假设用户使用 HBase URL 短网址服务，并且为现有用户存储 3 个新缩短过的 URL。所有的键都被认为是完全分布式的，所以 3 个新行最后在不同的 region 服务器上。进一步假设这些服务器的时间都相隔一个小时，如果用户通过扫描客户端来获取最近一小时更新的缩写 URL 列表，用户将会遗漏一些数据，因为它们被保存时的时间

戳与客户端认为的当前时间相差超过一小时。

用户可以通过在存储这些值时设置商定的或者共享的时间戳来避免这个问题。put 操作允许用户设置一个客户端的时间戳作为替代，并以此覆盖服务器的时间。显然，依靠服务器来完成这项工作更方便，但是，在一些情况下用户可能还需要使用这个方法^①。

region 被拆分暴露了服务器时间不一致引起的另一个问题。假设用户把一个值存储在一个比机群中所有其他服务器都超前一小时的服务器上，并且使用服务器隐式的时间戳。十分钟后这个 region 被拆分了，并且用户一半的数据更新被移到了另一个服务器上。5 分钟后，当用户再向同样的列中插入一个新值时，服务器会自动添加时间戳。现在新值被认为比之前添加的数据更老，因为第一个版本的时间戳比当前服务器的时间超前一小时。此时如果用户使用标准的 get 方法去检索这个值的最新版本，则会得到第一个之前存的值。

所有的服务器时间同步之后，用户还应该了解一些有趣的副作用。首先，一个特定的时间可能重现一系列的其他版本。这种情况发生在用户存储的版本数比列族级配置的版本数更多的时候。例如，当默认是保持一个单元或者一个值的最近 3 个版本时。

如果用户向同一列中插入 10 次新值，并且通过使用 Get 类的 setMaxVersions() 函数来要求保留所有版本的完整列表，用户将永远只能得到表模式中配置的那么多版本，即默认配置的最近 3 个版本。但是，当用户显式地删除最后两个版本时会发生什么呢？

例 9.1 删除显式时间戳的示例应用程序

```
for(int count = 1;count <= 6;count++){ ❶
    Put put = new Put(ROW1);
    put.add(COLFAM1,QUAL1,count,Bytes.toBytes("val-" + count)); ❷
    table.put(put);
}

Delete delete = new Delete(ROW1); ❸
delete.deleteColumn(COLFAM1,QUAL1,5);
delete.deleteColumn(COLFAM1,QUAL1,6);
table.delete(delete);
```

- ❶ 存储同一列 6 次。
- ❷ 使用循环变量把版本设为一个特殊的值。
- ❸ 删除最新的两个版本。

当你运行这个例子时，你应该可以看到以下输出：

```
After put calls...
KV: row1/colfam1:qual1/6/Put/vlen=5,Value: val-6
KV: row1/colfam1:qual1/5/Put/vlen=5,Value: val-5
```

① 一个不常见的例子是基于虚拟化的服务器。查看 http://support.ntp.org/bin/view/Support/KnownOsIssues#Section_9.2.2 列出的 NTP 的问题。NTP 是在虚拟机上常用的网络时间协议 (Network Time Protocol)。

```
KV: row1/colfam1:qual1/4/Put/vlen=5,Value: val-4
After delete call...
KV: row1/colfam1:qual1/4/Put/vlen=5,Value: val-4
KV: row1/colfam1:qual1/3/Put/vlen=5,Value: val-3
KV: row1/colfam1:qual1/2/Put/vlen=5,Value: val-2
```

有趣的现象是，用户使得版本 2 和版本 3 复活了。这是由于服务器把内部处理推迟到一个定义好的时间执行。该列的老版本仍然存在，因此删除较新的版本会使它们再次出现。这种情形只可能出现在 **major** 合并被执行之前，在这之后老版本会被永久删除，而保留的版本数基于已配置的最大保留版本数。



用户可以用示例代码中一些被注释掉的代码来强制执行刷写和 **major** 合并。如果用户重新运行这个例子，将会看到这样的结果：

```
After put calls...
KV: row1/colfam1:qual1/6/Put/vlen=5,Value: val-6
KV: row1/colfam1:qual1/5/Put/vlen=5,Value: val-5
KV: row1/colfam1:qual1/4/Put/vlen=5,Value: val-4
After delete call...
KV: row1/colfam1:qual1/4/Put/vlen=5,Value: val-4
```

由于老版本已被删除，它们不会再出现。

最后，在处理时间戳时，还有另外需要注意的一个问题：删除标记。在 **HBase** 中，删除操作的本质是添加一个带有特定时间戳的墓碑标记到存储中。在此基础上，它屏蔽直接指定的对应版本的数据，或者一个列删除标记会抹去比给定时间戳更老的所有版本的数据。例 9.2 用 **Shell** 展示了上述情况。

例 9.2 使用显式时间戳删除可以屏蔽之前的 **put** 操作

```
hbase(main):001:0> create 'testtable','colfam1'
0 row(s) in 1.1100 seconds

hbase(main):002:0> Time.now.to_i
=> 1308900346

hbase(main):003:0> put 'testtable','row1','colfam1:qual1','val1' ❶
0 row(s) in 0.0290 seconds

hbase(main):004:0> scan 'testtable'
ROW COLUMN+CELL
 row1column=colfam1:qual1,timestamp=1308900355026,value=val1
1 row(s) in 0.0360 seconds

hbase(main):005:0> delete 'testtable','row1','colfam1:qual1' ❷
0 row(s) in 0.0280 seconds
```

```

hbase(main):006:0> scan 'testtable'
ROW    COLUMN+CELL
0 row(s) in 0.0260 seconds

hbase(main):007:0> put 'testtable','row1','colfam1:qual1','val1',\
    Time.now.to_i - 50000 ③
0 row(s) in 0.0260 seconds

hbase(main):008:0> scan 'testtable'
ROW    COLUMN+CELL
0 row(s) in 0.0260 seconds

hbase(main):009:0> flush 'testtable' ④
0 row(s) in 0.2720 seconds

hbase(main):010:0> major_compact 'testtable'
0 row(s) in 0.0420 seconds

hbase(main):011:0> put 'testtable','row1','colfam1:qual1','val1',\
    Time.now.to_i - 50000 ⑤
0 row(s) in 0.0280 seconds

hbase(main):012:0> scan 'testtable'
ROW    COLUMN+CELL
row1 column=colfam1:qual1, timestamp=1308900423953,value=val1
1 row(s) in 0.0290 seconds

```

- ① 向新创建的表的列中存入一个值，运行扫描来验证结果。
- ② 删除列的所有值，这将会设置一个带有当前时间戳的删除标记。
- ③ 再一次把值存入列中，但是使用一个过去的时间戳，随后的扫描没有返回被屏蔽的值。
- ④ 通过对表的刷写和 major 合并来移除这个删除标记。
- ⑤ 再次存储带有过去时间戳的值，随后的扫描如预期一样显示了插入的值。

这个例子显示了在某些情况下，用户可能会看到一些意想不到的数据。但是，这些行为都是可以由 HBase 的架构来解释的，并且它们的发生也都是确定的。

9.7.2 自定义版本控制

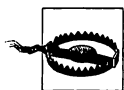
由于你可以指定自己时间戳的值，因此也可以创建自己的版本控制计划。在覆盖服务器端基于同步的服务器时间的时间戳时，用户可以不使用基于时间的版本。

例如，用户可以把时间戳和一个全局数字生成器^①结合起来使用，这个生成器提供了从

① 查看 `zk_idgen` 项目 (<http://sourceforge.net/projects/zkidgen/>)，它是一个基于 ZooKeeper 的数字生成器的例子。

“1”开始一直增加的顺序数字。每次用户插入一个新值时就会获得一个新数字，并且在调用 `put` 函数时会用到它。

用户必须为每个 `put` 操作都这么做，否则服务器将插入一个基于服务器端时间的时间戳。表或者列的描述中有一个标志可以表明你使用的是自定义的时间戳（即自定义的版本控制策略）。如果用户没有设置这个值，那么它将在后台被服务器时间替换掉。



当使用自己的时间戳值时，用户需要确保彻底地测试过自己的解决方案，因为这种方法并没有被广泛地用于生产中。

注意，负值的时间戳值也未经过测试，尽管它在 HBase 开发圈中被讨论过几次，但是它从来没有被确认能够正常工作。

确保同一个单元的两次单独更新不会使用相同的时间戳，这样会造成冲突。通常最后保存的值是可见的。

先抛开这些警告，下面是一些展示了自定义版本控制模式如何为全局表模式设计带来益处的例子。

记录 ID

使用这个技术的典型例子在 9.5 节中进行了讨论，即 Facebook 收件箱搜索应用 (Facebook inbox search)。它使用时间戳值存储消息 ID。由于这些 ID 随着时间而增长，并且在 HBase 中，版本的隐式排列顺序是降序排序，所以用户可以按时间排序获取到匹配搜索列的最后 10 个版本以得到最近的 10 条信息。

数字生成器

这里紧接着最初的例子，即使用分布式数字生成器。生成器和基于时间的数据戳看起来作用相同：按照一个单调递增的数升序排列所有存储的值。不过它们的区别更微妙，因为 Java 计时器使用的精度是毫秒，这意味着使用完全相同的时间存储两个值是不太可能的，但并不是完全不可能。如果用户需要一个全局完全唯一的版本控制解决方案，依赖数字生成器解决这个问题。

使用 HBase 的时间组件是利用其架构提供的额外维度的一种有趣方式。用户的自由度更少一些，因为它只接受 `long` 型值，这与行和列键支持任意二进制键不同。但是，它可以解决用户一些特定使用场景中的问题。

第 10 章

集群监控

一旦启动 HBase 集群，保证集群持续正常运行就显得非常必要。本章描述了如何使用一系列工具来监控集群的状态。

10.1 介绍

监控生产系统非常重要，系统通常会有多种监控指标，通过它们用户可以了解到当前状态的各种详细信息，HBase 系统也是如此。

HBase 实际上是从 Hadoop 继承了监控 API，然而 Hadoop 是一个面向批处理的系统，给用户的反馈信息往往不够及时。HBase 则要求更加实时，因为 HBase 常被用来处理随机在线访问请求，例如，在后台驱动网站。这些请求的响应时间需要维持在一定标准以下，以保持良好的用户体验——通常也被称作服务水平协议（SLA）。

对分布式系统来说，管理员要弄清楚系统的整体状态是一项艰巨的任务，这需要单独检查每台服务器的状态。即使对单机系统来说，当管理员处理一堆原始日志文件时，要搞清楚系统情况仍然十分困难。当系统崩溃时，了解问题出现的时间和地点将非常有用。但是当你面对 MB、GB 甚至 TB 量级的字符文件时，这种努力像大海捞针一样，可以说只有少部分人能够胜任这样的任务。即使用户拥有高超的日志阅读技巧，也将花费大量时间来推算和验证并找出问题的根源。

这显然这不是一个新问题，且多年来出现了各种可行的解决方案。这些解决方案可以归属于可视化和监控工具两类，许多工具包含其中一类或两类功能。它们将捕捉到的系统监控指标信息图形化，同时可以按时间筛选数据，通常以日、月和年等为时间单位将数据显示在可视化的图表中。这样能够很好地展示系统的最近信息，正如俗话说“一图抵千言”。

可视化可以很好地展示定量历史数据，但是对于超大粒度的时间间隔来说，了解系统当前的运行状态仍然很困难。此时监测支持系统的定性数据就非常重要了，它监听用

户的行为，检查数据点和一定时间范围内的监控指标。支持工具包通常都提供了很多出色的检测工具，因此用户所要做的就是利用它们来完成自己的工作。用户通常可以以插件的形式或者脚本的形式扩展和添加缺少的功能，同时用户还可以设定检查的频率，其频率可以设置为从几秒到几天的范围。

当检测显示系统出现问题或者完全失效时，对应的规避操作会自动执行：服务器可以被下线、重启或者修复。当问题依旧存在时，会有别的规则来提升问题的处理级别，例如，管理员将人工处理问题。具体措施包括发送电子邮件、短信或者拨打电话。

市场上有很多监控系统可供选择。HBase 基于 Java 的特性进行开发，并且和 Hadoop 关系密切，组合使用时性能更为可靠，但同时系统选择更为受限。HBase 原生支持的可视化系统是 Ganglia。为了监控系统，用户需要一个能够处理 HBase 进程导出的、基于 JMX^① 监控指标 API 的系统，通常此类型系统的例子是 Nagios。



用户应该建立一个完整的支持系统框架，以便在生产系统中使用，即使用户只是在使用 HBase 做原型研究或者概念验证。通过这种方式你可以开始对系统的各个数值有个良好的认识，从而可以正确地配置系统。处理没有监控和监控指标的集群和闭着眼开车一样。

在 HBase 集群中运行负载测试很有用，但是用户需要理解当前系统运行情况和性能之间的关系。使用可视化工具可以帮用户将服务器和子系统中的消息串联起来，这对理解测试结果很有价值。

10.2 监控框架

每个 HBase 进程（包括 master 和 region 服务器）都会提供一系列监控指标。这些监控指标随后可以被各种监控 API 和工具使用，包括 JMX 和 Ganglia。每种服务器都有多组监控指标，这些监控指标按子系统分组并隶属于一种服务器。例如，一组 Java 虚拟机（JVM）提供的监控指标深刻地反映了当前进程中很多值得关注的细节，包括垃圾回收统计和内存占用等。

10.2.1 上下文、记录和监控指标

HBase 使用 Hadoop 的监控框架，并继承了其所有的类和特性。这个框架基于 MetricsContext 接口来处理监控数据点的生成，并使用这些数据点监控和绘图。下面是

^① JMX 是 *Java Management Extensions* 的首字母缩写，这是一种基于 Java 技术，目的是方便用户构建监控和管理应用的解决方案。详细信息参见项目网站（<http://www.oracle.com/technetwork/java/javase/tech/javamangement-140525.htm>），用户也可以参考 10.2.5 节。

可用的实现列表。

GangliaContext

用来推送监控指标到 Ganglia，细节请参见 10.3 节。

FileContext

将监控指标写入磁盘上一个文件中。

TimeStampingFileContext

同样将监控指标写入磁盘上一个文件中，但是为每个监控指标添加一个时间戳前缀。这让文件的格式和日志记录非常类似。

CompositeContext

允许为监控指标生成不止一个上下文，例如，用户可以一次同时指定 Ganglia 和文件上下文。

NullContext

监控指标框架的关闭选项，使用这种上下文时，不生成也不聚合监控指标。

NullContextWithUpdateThread

不生成任何监控指标，但是启动聚合统计线程。这种上下文在通过 JMX 检索监控指标时使用，详见 10.4 节。

每个上下文都有唯一的名字，在外部配置文件中注明（详见 10.3.1 节的“HBase 相关步骤”部分），也被用来定义各种属性和实现 MetricsContext 接口的实现类。



另一个 HBase 继承自 Hadoop 监控指标框架的产物是使用其提供的 ContextFactory 来装载各种上下文类。配置文件名在这个类中被硬编码为 `hadoop-metrics.properties`，这是 HBase 使用和 Hadoop 完全一样的文件名，而不使用更为直观的 `hbase-metrics.properties` 的原因。

多重监控指标使用 MetricsRecord 分组，来描述一个具体的子系统。HBase 使用这些组别来保存 master、region 服务器或其他服务器的统计信息。每个组都有唯一的名字，完整的监控指标名称都是由上下文名称和实际监控指标名称组合而成的。

`<context-name>.<record-name>.<metric-name>`

上下文有内置的定时器来触发并将监控指标推送至目标——可能是文件、Ganglia 或者用户自己定制的解决方案。上下文配置选项中的 `period` 选项被用来设定上下文更新监控指标的时间间隔。具体上下文的实现可能含有额外的属性用来控制行为。图 10-1 显示了相关类的序列图。

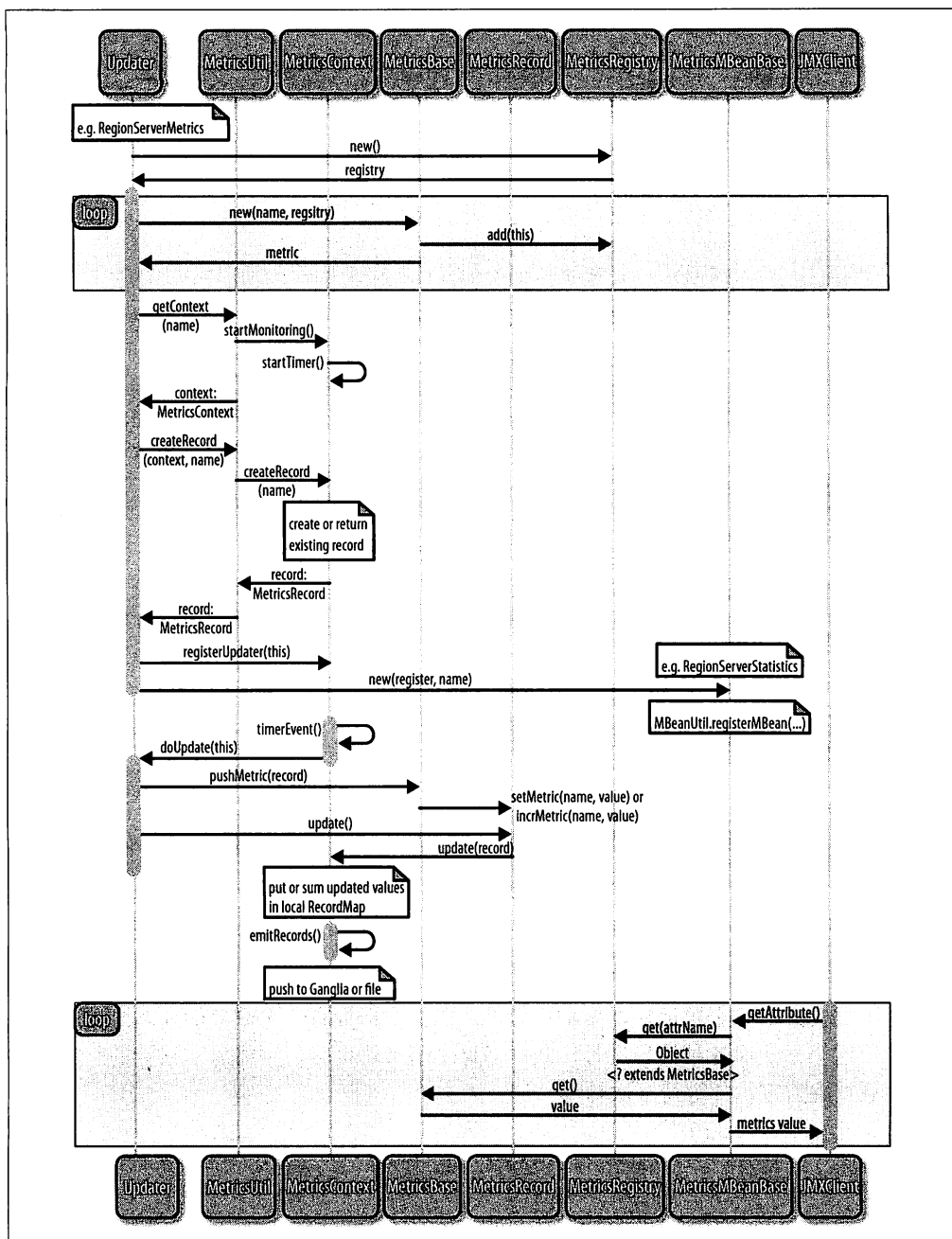


图 10-1 准备监控指标涉及类的序列图

在系统内部，监控指标是被基于 `MetricsBase` 的容器类追踪的，这些容器类包含各种当消息发生时需要调用的更新或者增加方法。反过来，框架跟踪每个监控指标消息的数

目，并将其与上次跟踪到现在所消耗的时间关联起来。

下面概述了 Hadoop 和 HBase 监控指标框架中各种可用的监控指标类型，并给出了相关缩写形式。这些缩写将在本章后续部分被引用。

整型值 (IV)

跟踪一个整型计数器，仅仅当值改变时此监控指标才更新。

长整型值 (LV)

跟踪一个长整型计数器，仅仅当值改变时此监控指标才更新。

速率 (R)

一个代表速率的浮点型值，可以是每秒操作或者消息数。它提供一个递增方法用来追踪操作次数，同时提供一个上次轮询时间戳来追踪消耗的时间。当监控指标被轮询时，将发生以下事件。

1. 速率以操作数目/秒为单位计算的消耗时间来统计。
2. 这个速率存储在之前设定的值域中。
3. 内部计数器重置为零。
4. 最后一次推送时间设置为当前时间。
5. 计算的速率返回给调用者。

字符串 (S)

这种上下文类型用来存储静态的、基于文本的信息，并用来报告 HBase 版本信息和构建时间等。这个值不会重置或者修改，一旦赋值，其在进程运行时一直保持原样。

时间变化整型 (TVI)

上下文会维护一个单调递增累加计数器。监控指标拥有一个简单的递增方法，框架使用这个方法对各种消息进行计数。当这个值被轮询时，它将返回累计的整型值，之后重置为零，并等待下次轮询。

时间变化长整型 (TVL)

和 TVI 一样，只有操作数是长整型值，主要作用于增速较快的计数器，否则可能会超过最大整数。同样，在获取检索之后，该值会被重置。

时间变化率 (TVR)

TVR 需要追踪操作数或者消息的数量，以及完成操作所用的时间，这些监控指标通常用来计算一次操作完成的平均时间。同时监控指标也会显示每个操作的最长和

最短时间。表 10-1 展示了这 4 个值在同一监控指标下是如何用不同前缀命名的。

表 10-1 基于时间变化率的监控指标反映的 4 个值

名称	简称	描述
Number Operations	NumOps	上次轮询至今实际事件的数目
Minimum Time	MinTime	完成一次事件的最短时间
Maximum Time	MaxTime	完成一次事件的最长时间
Average Time	AvgTime	每次事件完成的平均时间，每次事件完成时间的总和除以事件数

注意操作数和累积时间会在数据轮询时复位。操作数是由轮询上下文累加生成的，保持单调递增。与此相反，平均时间是一个相对值，由每次轮询间隔检索的监控指标计算出来。

每次操作的最大和最小操作时间不会复位，直到调用 `resetMinMax()` 函数。此操作可以通过 JMX（查看 10.4 节）或者某些扩展监控指标隐式触发来完成。

持续型时间变化率（PTVR）

PTVR 是 TVR 的扩展，它添加了对持续的周期性的监控指标的必要支持：因为这些长期运行的监控指标并不是每次轮询都会复位，因此它每次报告时都需要特殊处理。

表中简称那一列是实际监控指标名称的后缀。例如，当检索 HTable 提供 `increment()` 操作的监控指标时，你会看到 4 个值，分别是 `incrementNumOps`、`incrementMinTime`、`incrementMaxTime` 和 `incrementAvgTime`。

但也不是每个地方都可以获得这 4 个值，例如，基于上下文的监控指标只提供 `AvgTime` 和 `NumOps` 的值，但 JMX 可以访问这 4 个值。

讨论 HBase 提供的不同类型的监控指标时，用户可能会注意到引用它们的类型缩写，用户自己撰写支持工具时，可以参考它们。同时要记住，区分通过监控指标上下文还是 JMX 获取数据时，不同类型监控指标的行为有区别。

一些监控指标（例如，时间变化的监控指标）会在轮询时复位，但是对应的上下文会累计它们的值来作为单调递增计数器。通过 JMX 获取值时可以观察到复位的形为，因为这样可以直接获取监控指标的值，而非获取经上下文处理过的结果。

一个明显的例子就是 TVR 类监控指标中的 `NumOps`。通过监控指标上下文读取数据时，用户会得到一个递增的值，然而 JMX 仅会提供自上次轮询之后到现在的绝对数目。

其他监控指标仅在上次更新后数值再次变化时才生成数据。用户显然应当通过上下文而不是 JMX 来获取这些监控指标的值，后者只是简单地获取上次轮询后的新值。如果不设定一个轮询周期，JMX 取得的值将永不改变。更多细节请看 10.4 节。图 10-2 显示

了在每个监控指标周期，不同的监控指标是如何更新和生成数据的。JMX 总是读取原始的监控指标数据，这与基于上下文的聚合统计行为有显著差异。

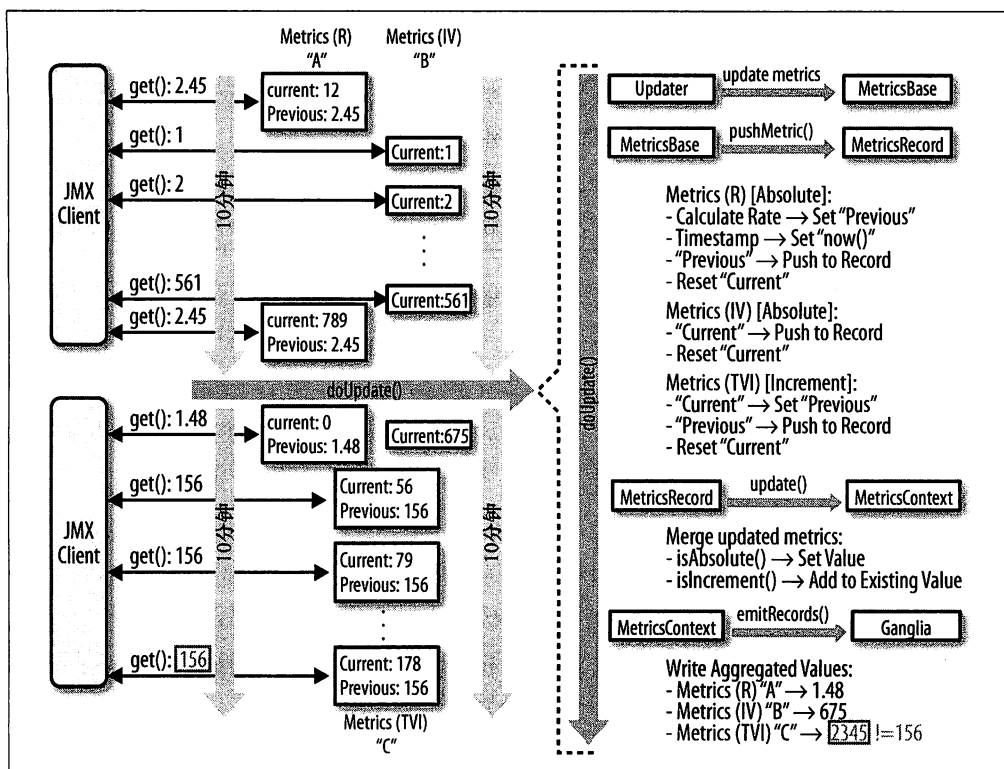


图 10-2 各种监控指标收集数据和（可选）复位的不同之处

HBase 也有些异常率监控指标，其更新的跨度超过了一般监控指标更新框架所使用的更新时间。



HBase 中有些长期运行的进程，需要保证监控指标一直持续到进程结束。这可以通过 `hbase.extendedperiod` 属性来控制，单位是秒。默认不会结束超期，但是提供的配置将其设成了 3600 秒，即 1 小时。

当前这些周期性持续的监控指标主要用于观察 region 服务器和 master 对应的合并、刷写和拆分操作的时间和变化率。在 region 服务器端，这个属性也会触发一些速率监控指标的复位，包括读、写和同步数据的延迟。

10.2.2 master 监控指标

master 进程导出的所有监控指标都和它在集群中的角色有关。因为 master 被设计成相当轻量级的系统进程，只负责一些集群范围的操作，与 region 服务器相比其提供的监

控信息很有限。表 10-2 列举了可用的监控指标。

表 10-2 master 提供的监控指标

监控指标	描述
<i>cluster requests</i> (R)	集群请求总数，累加所有 region 服务器统计的值
<i>split time</i> (PTVR)	重启后拆分预写日志的时间
<i>split size</i> (PTVR)	拆分预写日志的大小

10.2.3 region 服务器监控指标

region 服务器是管理实际数据读取和写入的部分，因此需要收集大量的监控指标信息。这些监控指标包含了服务器端整体架构中许多不同部分的细节信息，例如，块缓存 (block cache) 和内存存储 (in-memory store)。

本节我们会将监控指标进行分组并讨论其功能，而不是简单地列举所有监控指标，因为理解它们整体的含义比了解每个单独的数据点更重要。而且在每个组内部，它们的含义都非常明确，通常只需要很少的注释来解释它们。

块缓存监控指标

块缓存用来保存底层 HFile 从 HDFS 读取的存储块。这样用户能将一个块保存在内存中，直到空间不足时才被清除出内存。

count (LV) 监控指标反映了当前缓存中保存的块数目，*size* (LV) 监控指标是占用的 Java 堆空间大小，*free* (LV) 监控指标是堆空间为缓存保留的可用空间，*evicted* (LV) 监控指标统计了当堆空间受限时将被移除的块的数目。

块缓存追踪缓存命中 *hit* (LV) 和缓存失效 *miss* (LV) 的数目，以及命中率 *hit ratio* (LV)，其反映了命中缓存总数与请求缓存总数的关系。

最后，缓存命中数目与缓存命中率相似，仅仅是统计请求使用块缓存且命中的操作数目。参考 3.2.2 节“单行 grt”中的 `setCacheBlocks()` 方法。



所有读操作都会尝试使用缓存，不管用户是否指定过将使用的块保留在缓存中。使用 `setCacheBlocks()` 仅仅影响块的保留策略。

合并监控

当 region 服务器需要执行异步的或者手动调用的合并存储文件的日常管理任务时，它将会使用不同的监控指标来报告状态。*compaction size* (PTVR) 和 *compaction time* (PTVR) 分别代表需要合并的存储文件总大小（以字节为单位）和操作花费时间。注意，只有当合并操作完成之后，这些值才会被报告，因为只

有那时才能确定这些值。

compaction queue size (IV) 用来监测一个 region 服务器有多少文件当前正在排队等待合并。



这个值是另一个需要特别注意的参数。通常这个值很小，在零到十几之间变化。当用户遇到 I/O 问题时，通常会发现这个数上升很快。请查看图 10-5 中的示例。

用户需要注意 major 合并也会导致这个数字快速上升，因为这个操作会将所有存储文件添加到队列中。用户在查看图表时需要考虑这种情况。

memstore 监控指标

更新数据通常保存在 region 服务器的 memstore 中，并且将会通过之后的刷写写到磁盘上。memstore 监控指标提供了 *memstore size MB* (IV) 来表示服务器上所有 memstore 总共占用的堆大小，即所有在线 region 的 memstore 的总和。

flush queue size (IV) 是指将要被刷写的 region 的数目。*flush size* (PTVR) 和 *flush time* (PTVR) 分别表示被刷写到磁盘上的 memstore 大小和本次刷写所占用的时间。

与合并监控指标类似，这两个监控指标在每次刷写完成后进行更新。所以报告的内容只关注实际值，而不关注正在发生的内容。



与合并队列类似，在某些情况下用户可能发现刷写队列大小显著地上升了，例如，当用户的服务器 I/O 出现问题时。用户可以通过监控这个值来确定正常情况下数值的范围（通常也是个较小的值），同时设定一个合理阈值，当排队数超出限制时触发一个警告。

存储监控指标

store files (IV) 监控指标显示了所有存储文件的数目，涉及当前机器管理的所有 region 的存储文件。*stores* (IV) 显示了服务器上所有 region 的存储文件的数目。*store file index MB* (IV) 监控指标（以 MB 为单位）是所有存储文件的块索引和元数据索引的总和大小。

I/O 监控指标

region 服务器使用 3 个延迟监控指标来跟踪 I/O 性能，它们都以毫秒为单位。*fs read latency* (TVR) 报告文件系统的读延迟，例如，从存储文件中装载块时的延时。*fs write latency* (TVR) 和上面相似，这个监控指标收集所有写操作的数据，例如，包括写存储文件和预写日志。

fs sync latency (TVR) 监控指标统计了预写日志记录同步到文件系统的延迟。这个延时监控指标可以提供底层 I/O 性能的相关信息，用户应该密切关注。


其他监控指标

除了以上这些，region 服务器还提供了一些全局计数器作为监控指标。*read request count* (LV) 和 *write request count* (LV) 分别表示总的读操作（如 *get()*）和写操作（如 *put()*）的数目，这些操作由当前 region 服务器汇总其所有在线 region 的信息得出。

requests (R) 监控指标是自上次轮询之后目前每秒的请求数。*regions* (IV) 监控指标是目前 region 服务器在线的 region 数目。

10.2.4 RPC 监控指标

master 和 region 服务器还提供了 RPC 子系统的监控指标。它会自动监控每次不同客户端和服务端之间的操作，其中包括 master 和 region 服务器提供的 RPC 操作。




master 和 region 服务器使用的是一套共享的 RPC 监控系统，即用户可以在不同的服务器上看到相同的监控指标。不同之处在于更新这些监控指标时，进程调用的 RPC 不同。例如，在 master 端，用户不会发现 *increment()* 的监控指标被更新，这是因为这些方法是 region 服务器提供的。另一方面，你会在 master 上发现所有管理操作调用的监控指标，如 *enableTable* 和 *compactRegion*。

因为这些监控指标直接与客户端和管理 API 相关，用户可以通过相应的 API 调用来推测它们的意思。虽然命名规则不完全一致。一个值得注意的模式是，与 region 相关的 API 操作有额外的 region 后缀，例如，HBaseAdmin 提供的 *split()* 操作对应 *splitRegion* 监控指标。只有少数监控指标和 API 没关系。表 10-3 列出了一些监控指标，但是这些监控指标是由 RPC 系统自身提供的。

表 10-3 RPC 子系统提供的非 API 监控指标

监控指标	描述
<i>RPC Processing Time</i>	服务器端执行 RPC 消耗的时间，统计所有 RPC 调用的时间，并取平均值
<i>RPC Queue Time</i>	因为 RPC 需要使用排队系统，可能造成操作到达时间和操作实际执行时间的延迟，即排队时间



监控排队时间很有好处，因为它反映了服务器的负载。用户可以设置临界值，当这个监控指标的值超过临界值后便会触发报警。这是系统出现问题的早期预警信号。

剩下的监控指标都来自 **master** 和 **region** 服务器的 **RPC API**，包括 `regionServerStartup()` 和 `regionServerReport`，它们分别在 **region** 服务器初始化并向它的 **master** 节点报告时调用，以及常规报告状态时调用。

10.2.5 JVM 监控指标

当用户需要优化 **HBase** 部署时，调优 **JVM** 的参数需要专家级的技巧。用户可以在 11.1 节了解相关信息，本节将介绍用户可以从监控指标框架获取到的各个服务器进程的相关信息。每个 **HBase** 进程都会搜集和导出有用的 **JVM** 相关细节，例如，各种和服务性能相关的 **JVM** 内部参数，这些信息反过来可以用来帮助调优 **HBase** 集群部署。

提供的监控指标可以分为以下几类。

内存占用监控指标

用户可以得到正在使用的内存和保证可以由 **JVM** 使用的内存的信息^①，包括堆和非堆内存的使用情况。前者是由 **JVM** 按用户行为管理内存，同时会定时进行垃圾回收。后者是 **JVM** 为内部需求占用的内存。

垃圾回收监控指标

JVM 会按用户行为管理堆内存并运行垃圾回收。`gc count` 监控指标表示垃圾回收的次数，`gc time` 监控指标是上次轮询至今累计的垃圾回收占用的时间。



在垃圾回收过程中，某些阶段会导致 **JVM** 暂停，这些暂停在某些需要保证服务水平协议的系统中非常难以处理。

通常情况下，这种暂停只会持续几毫秒，但是有时候会增加到数秒。当暂停达到分钟级之后就会产生许多问题，有可能导致 **region** 服务器持有的 **ZooKeeper** 租约过期，促使 **master** 采取相应的规避动作。^②

用户可以使用此监控指标来追踪服务器当前情况和垃圾回收的时间。当用户观察到对应监控指标的值突然增加时就必须调查原因。超过 `zookeeper.session.timeout` 配置时间的暂停都应当被认为是系统错误。

① 参见线上官方文档 `MemoryUsage` (<http://download.oracle.com/javase/6/docs/api/java/lang/management/MemoryUsage.html>) 来了解使用“used”和分配的“committed”内存的含义。

② “**HBase** 开发小组亲切地称这种场景为朱丽叶停顿——**master**（罗密欧）提前假设 **region** 服务器（朱丽叶）死亡，而她其实只是在休眠，并因此采取了一些激烈的行动（恢复）。当服务器重新醒来，她发现出现了一个巨大的错误并结束了自己的生命。这种场景会产生一部伟大的戏剧，但却是一个痛苦的故障场景！”（<http://www.cloudera.com/blog/2011/02/avoiding-full-gcs-in-hbase-with-memstore-local-allocation-buffers-part-1/>）。

线程监控指标

这组监控指标反映了一组和 Java 线程相关的参数。用户可以观察到许多与线程可能的状态相关的计数器，包括新建、运行和阻塞等。

系统事件监控指标

最后，事件监控组包含了日志子系统收集的各种监控指标，其被归入到了 JVM 的监控指标类别中（因为没有更好的地方可以放置）。它提供了各种日志级别的消息数目。例如，*log error* 监控指标提供了自上次轮询至今，错误级别日志消息的数目。实际上所有的日志消息数目都显示的是上次轮询至今所累积的消息数。

用户可以使用这些监控指标为自己的支持系统提供数据，支持系统可以绘制时间趋势图或按固定阈值触发警告。理解这些数值并熟悉它们通常的范围非常重要，可以帮助用户在生产实践中合理利用它们。

10.2.6 info 监控指标

HBase 进程同样提供了一组称作 *info* 的监控指标。它们包含了进程的固化信息，且用户可以自动检查这些值。表 10-4 列举了这此监控指标及其详细描述信息。这些监控指标只能通过 JMX 来获取。

表 10-4 info 监控指标

监控指标	描述
<i>date</i>	HBase 编译日期
<i>version</i>	HBase 版本
<i>revision</i>	编译使用的源码库版本
<i>url</i>	源码库的 URL 地址
<i>user</i>	HBase 编译者
<i>hdfsDate</i>	HDFS 编译日期
<i>hdfsVersion</i>	当前使用的 HDFS 版本
<i>hdfsRevision</i>	HDFS 使用的源码库版本
<i>hdfsUrl</i>	HDFS 源码库 URL 地址
<i>hdfsUser</i>	HDFS 编译者

HDFS 引用 HBase 使用的 *hadoop-core-*<X.Y-nnnn>*.jar* 文件。这个 JAR 文件通常是安装包中已经提供的 JAR 文件，但是用户可以根据你的安装情况定制一个特定的 JAR 文件。返回的值如下所示。

```
date:Wed May 18 15:29:52 CEST 2011
version:0.91.0-SNAPSHOT
revision:1100427
url:https://svn.apache.org/repos/asf/hbase/trunk
user:larsgeorge

hdfsDate:Wed Feb 9 22:25:52 PST 2011
hdfsVersion:0.20-append-r1057313
hdfsRevision:1057313
hdfsUrl:http://svn.apache.org/repos/asf/hadoop/common/branches/branch-0.20-append
hdfsUser:Stack
```

这些值明显不能用图来显示，但是可以被管理员用来确认运行配置是否正确。

10.3 Ganglia

HBase 直接从 Hadoop 继承了对 Ganglia^① 的原生支持，同时提供了一个可以直接推送监控指标数据到 Ganglia 的客户端。Ganglia 由以下 3 部分组成。

Ganglia 监控守护进程 (gmond)

监控守护进程需要在每台需要监控的机器上运行。它搜集本地数据，准备统计信息，然后被其他系统拉取。它积极地监控通过单一或组播网络消息传播的主机变化情况。如果使用组播模式，每个监控守护进程可以获取集群完整状态，所有的服务器拥有同样的组播地址。

Ganglia 元数据守护进程 (gmetad)

元数据守护进程安装在一个中心节点上，作为整个集群的管理节点。元数据守护进程从一个或多个监控守护进程拉取数据来获取当前整个集群的状态，然后使用 RRDtool^② 将这些信息存放在一个用于轮询的时间序列数据库中。这些数据可以被其他客户端（如网页前端）以 XML 格式获取。

Ganglia 也支持数据报告守护进程的层次结构，每个层次树的元信息守护进程会合并它们的监控守护进程。更高层次的元信息守护进程再合并低层次元信息守护进程所合并的统计信息，以合并多个集群的统计信息。

GangliaPHP 前端展示网页

Ganglia 支持的网页前端从元信息守护进程处获取合并的统计信息，然后以 HTML 的形式展现。它使用 RRDtool 将时间序列数据展示成图像。

① Ganglia 是一种分布式可扩展的适合大规模集群使用的监控系统，参考项目网站 (<http://ganglia.info/>) 可获取项目历史和目标等相关信息。

② 参见 RRDtool 项目网站 (<http://www.mtrg.org/rrdtool/>) 来获取详细信息。

10.3.1 安装

Ganglia 的安装步骤分为两步：第一步是安装配置 Ganglia 本身，第二步是使 HBase 发送监控指标信息到 Ganglia。

1. Ganglia 相关步骤

用户应该尝试使用自身操作系统发行版支持的预编译二进制安装包。如果没有，可以从网站下载源代码，并在本地构建安装。例如，在基于 Debian 的系统中，用户可以按照以下步骤进行安装。

Ganglia 监控守护进程。在希望监控的所有节点执行以下步骤。

添加用户专用账号：

```
$ sudo adduser --disabled-login --no-create-home ganglia
```

从网站下载源代码包，解压到一个公共位置：

```
$ wget http://downloads.sourceforge.net/project/ganglia/ \
    ganglia%20monitoring%20core/3.0.7%20%28Fossett%29/ganglia-3.0.7.tar.gz
$ tar -xzf ganglia-3.0.7.tar.gz -C /opt
$ rm ganglia-3.0.7.tar.gz
```

安装有依赖关系的包：

```
$ sudo apt-get -y install build-essential libapr1-dev \
    libconfuse-dev libexpat1-dev python-dev
```

现在可以构建并安装二进制文件：

```
$ cd /opt/ganglia-3.0.7
$ ./configure
$ make
$ sudo make install
```

下一步是建立配置文件，可通过如下方式快速生成一个默认的配置文：

```
$ gmond --default_config > /etc/gmond.conf
```

按照如下内容在 */etc/gmod.conf* 文件中修改：

```
globals {
    user = ganglia
}

cluster {
    name = HBase
```

```
owner = "Foo Company"
url = "http://foo.com/"
}
```

global 定义了之前创建的用户账号, *cluster* 定义了集群的相关信息。默认 Ganglia 配置使用组播 UDP 消息, 且使用 IP 地址 239.2.11.71 来通信, 这适用于少于 120 个节点的集群。

组播与单播的对比

虽然默认的监控守护进程 (gmond) 使用 UDP 组播消息方式通信, 但是用户可能遇到一些没有广播环境或者受其他因素限制的环境。前者例如使用基于亚马逊云服务器的集群环境 EC2。

另一个已知因素是组播通常只能满足 120 个节点以下的集群要求。如果遇到了这些情况, 用户必须将通信模式从组播调整为单播。在 */etc/gmond.conf* 文件中修改以下选项:

```
udp_send_channel {
    # mcast_join = 239.2.11.71
    host = host0.foo.com
    port = 8649
    # ttl = 1
}

udp_recv_channel {
    # mcast_join = 239.2.11.71
    port = 8649
    # bind = 239.2.11.71
}
```

这个例子假定你在 master 节点使用 gmond 来接受其他机器上 gmond 进程的数据更新。

host0.foo.com 需要替换成 master 的主机名或者 IP 地址。在一个更大的集群中, 用户可以设置在每个物理机器上启动多个 gmond 进程。通过使用这种方式, 用户可以防止只有一个 gmond 处理提交的更新。

同样你需要修改 */etc/gmetad.conf* 文件来设定指定节点, 具体参考本章讨论单播模式的内容。

启动监控守护进程:

```
$ sudo gmond
```



通过连接本地来测试守护进程:

```
$ nc localhost 8649
```

这会打印当前集群状态的原始 XML，使用 *kill* 命令可以停止守护进程。

Ganglia 元数据守护进程。用户需要在所有元信息守护服务器的节点上执行以下步骤，用于合并下游的监控统计数据。在少于 100 个节点的集群中，通常只需要一台这样的元信息服务器。注意，因为服务器需要绘制图形，所以需要相应的处理性能。

添加专用的用户账号:

```
$ sudo adduser --disabled-login --no-create-home ganglia
```

下载源代码，解压到公共的目录:

```
$ wget http://downloads.sourceforge.net/project/ganglia/ \
  ganglia%20monitoring%20core/3.0.7%20%28Fossett%29/ganglia-3.0.7.tar.gz
$ tar -xzf ganglia-3.0.7.tar.gz -C /opt
$ rm ganglia-3.0.7.tar.gz
```

安装依赖项:

```
$ sudo apt-get -y install build-essential libapr1-dev libconfuse-dev \
  libexpat1-dev python-dev librrd2-dev
```

现在用户可以像这样编译并安装二进制文件:

```
$ cd /opt/ganglia-3.0.7
$ ./configure --with-gmetad
$ make
$ sudo make install
```

注意，额外的 `--with-gmetad` 参数是之后安装必需的。下一步是安装配置，先复制默认的 *gmetad.conf* 文件:

```
$ cp /opt/ganglia-3.0.7/gmetad/gmetad.conf /etc/gmetad.conf
```

修改 */etc/gmetad.conf* 文件:

```
setuid_username "ganglia"
data_source "HBase" host0.foo.com
gridname "<Your-Grid-Name>"
```

`data_source` 行必须包含一个或多个 *gmond* 的主机名或 IP 地址。



当使用单播模式时，用户需要指定 `data_source` 为实际的专用 `gmond` 服务器。如果不只一台服务器，用户需要将所有服务器都列出，以提供节点失效保障。

现在创建需要的路径，这些路径用来将采集的数据存储到轮询数据库中：

```
$ mkdir -p /var/lib/ganglia/xrds/  
$ chown -R ganglia:ganglia /var/lib/ganglia/
```

启动守护进程：

```
$ gmetad
```

需要使用 *Kill* 命令关闭守护进程。

GangliaWeb 前端。最后一步是安装 Web 前端。通常的场景是在运行 `gmetad` 进程的机器上进行安装。同时至少要保证它能读取由 `gmetad` 创建的轮询分时数据库。

首先，安装所需的库：

```
$ sudo apt-get -y install rrdtool apache2 php5-mysql libapache2-mod-php5 php5-gd
```

Ganglia 包含必备的 PHP 文件，用户可以按如下方式复制它们：

```
$ cp -r /opt/ganglia-3.0.7/web /var/www/ganglia
```

现在启动 Apache：

```
$ sudo /etc/init.d/apache2 restart
```

如果用户已经将 `ganglia` 子域名指定到了运行 `gmetad` 的服务器，则用户可以通过网址 <http://ganglia.foo.com/ganglia> 来浏览网页前端。因为用户还需要安装 HBase 并将其监控指标信息推送至 Ganglia，所以目前用户只会看到以简单图形显示的关于服务器的监控信息，推送 HBase 的监控数据是接下来将要讨论的内容。

2. HBase 相关步骤

HBase 和 Ganglia 集成的关键部分是 `GangliaContext` 类，该类会将服务器进程的监控指标信息发送到 Ganglia 监控守护进程。此外在 `conf/` 目录下有一个名为 `hadoop-metrics.properties` 的文件，需要修改配置文件来启用上下文。参考以下内容编辑配置文件：

```
# HBase-specific configuration to reset long-running stats  
#(e.g. compactions). If this variable is left out, then the default  
# is no expiration.  
hbase.extendedperiod = 3600  
  
# Configuration of the "hbase" context for ganglia
```

```
# Pick one: Ganglia 3.0(former)or Ganglia 3.1(latter)
hbase.class=org.apache.hadoop.metrics.ganglia.GangliaContext
#hbase.class=org.apache.hadoop.metrics.ganglia.GangliaContext31
hbase.period=10
hbase.servers=239.2.11.71:8649

jvm.class=org.apache.hadoop.metrics.ganglia.GangliaContext
#jvm.class=org.apache.hadoop.metrics.ganglia.GangliaContext31
jvm.period=10
jvm.servers=239.2.11.71:8649

rpc.class=org.apache.hadoop.metrics.ganglia.GangliaContext
#rpc.class=org.apache.hadoop.metrics.ganglia.GangliaContext31
rpc.period=10
rpc.servers=239.2.11.71:8649
```



之前提过 HBase 当前版本 (0.91.X) 只支持 Ganglia 3.0.x, 所以可否让用户在 GangliaContext 和 GangliaContext 31 中选择呢? 某些 HBase 的发行版本已经包含了对 Ganglia 3.1.x 支持的补丁。当用户确认自己的 HBase 版本已经有了相关支持 (如 CDH3 已经支持了) 后, 再使用这个对应的上下文。

当用户使用单播消息时, 之前默认设置的 239.2.11.71 组播地址需要被替换成指定 gmond 机器的主机名或 IP 地址。例如:

```
...
hbase.class=org.apache.hadoop.metrics.ganglia.GangliaContext
hbase.period=10
hbase.servers=host0.yourcompany.com:8649

jvm.class=org.apache.hadoop.metrics.ganglia.GangliaContext
jvm.period=10
jvm.servers=host0.yourcompany.com:8649

rpc.class=org.apache.hadoop.metrics.ganglia.GangliaContext
rpc.period=10
rpc.servers=host0.yourcompany.com:8649
```

一旦用户完成了这个配置文件的修改, 用户需要重启 HBase 集群。之后就不需要再做什么修改了, Ganglia 会自动帮用户获取所有监控指标数据。

10.3.2 用法

一旦用户刷新 Web 前端网页, 就会看到 Ganglia 的主页, 如图 10-3 所示。

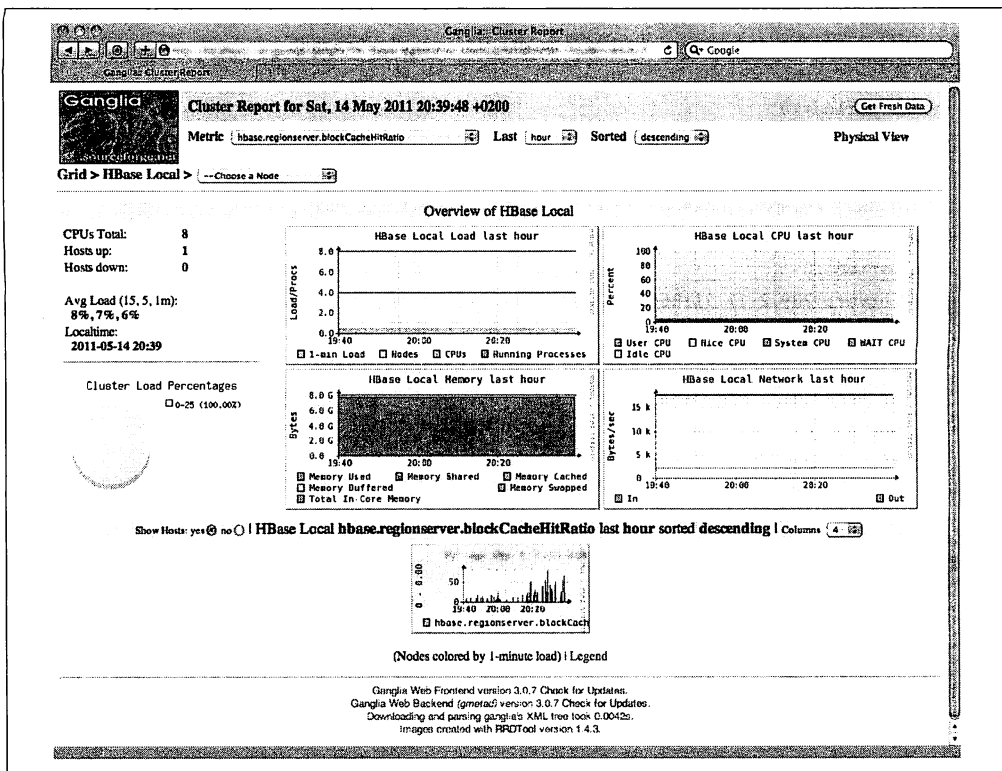


图 10-3 Ganglia 提供的基于 Web 前端的各种监控图表

用户可以在页面上更改监控指标、时间跨度和排序方式，系统会自动重载页面内容。在配置比较低的机器上，用户可能需要等待一段时间才能看到重绘的图像。图 10-4 展示了在下拉框中包括了所有可用的监控指标。

最后，图 10-5 是一个如何使用监控指标来查找问题根源的例子。这个图像显示了每天午夜一台负载过重的服务器的垃圾回收时间突然变长，这将导致合并队列长度也显著提高。

Ganglia 和它绘制出的图表是一种非常有效的、可回顾并查找问题根源的工具。不过它只能通过定量数据来提供帮助，例如，当问题已经发生后对集群问题进行分析。接下来我们将对用户展示如何使用定量支持系统来补充一些图表。



看起来显然是写负载过重导致了 I/O 扰动，但当读负载很重时，系统也会产生相似的行为（虽然不是很频繁）。例如，后台运行的 major 合并可能累积很多需要重写的存储文件。当没有明显的来自客户端的写负载时，这也会对读操作的响应时间有不利的影


```

getall
hbase.master.cluster_requests
hbase.master.splitSize_avg_time
hbase.master.splitSize_num_ops
hbase.master.splitTime_avg_time
hbase.master.splitTime_num_ops
hbase.regionserver.blockCacheCount
hbase.regionserver.blockCacheEvictedCount
hbase.regionserver.blockCacheFree
hbase.regionserver.blockCacheHitCachingRatio
hbase.regionserver.blockCacheHitCount
hbase.regionserver.blockCacheHitRatio
hbase.regionserver.blockCacheMissCount
hbase.regionserver.blockCacheSize
hbase.regionserver.compactionQueueSize
hbase.regionserver.compactionSize_avg_time
hbase.regionserver.compactionSize_num_ops
hbase.regionserver.compactionTime_avg_time
hbase.regionserver.compactionTime_num_ops
hbase.regionserver.flushQueueSize
hbase.regionserver.flushSize_avg_time
hbase.regionserver.flushSize_num_ops
hbase.regionserver.flushTime_avg_time
hbase.regionserver.flushTime_num_ops
hbase.regionserver.fsReadLatency_avg_time
hbase.regionserver.fsReadLatency_num_ops
hbase.regionserver.fsSyncLatency_avg_time
hbase.regionserver.fsSyncLatency_num_ops
hbase.regionserver.fsWriteLatency_avg_time
hbase.regionserver.fsWriteLatency_num_ops
hbase.regionserver.memstoreSizeMB
hbase.regionserver.readRequestsCount
✓ hbase.regionserver.regions
hbase.regionserver.requests
hbase.regionserver.storefileIndexSizeMB
hbase.regionserver.storefiles
hbase.regionserver.stores
hbase.regionserver.writeRequestsCount
jvm.metrics.gcCount
jvm.metrics.gcTimeMillis
jvm.metrics.logError
jvm.metrics.logFatal
jvm.metrics.logInfo
jvm.metrics.logWarn
jvm.metrics.memHeapCommittedM
jvm.metrics.memHeapUsedM
jvm.metrics.memNonHeapCommittedM
jvm.metrics.memNonHeapUsedM
jvm.metrics.threadsBlocked
jvm.metrics.threadsNew
jvm.metrics.threadsRunnable
jvm.metrics.threadsTerminated
jvm.metrics.threadsTimedWaiting
jvm.metrics.threadsWaiting
load fifteen

```

图 10-4 下拉框提供的各个监控指标入口

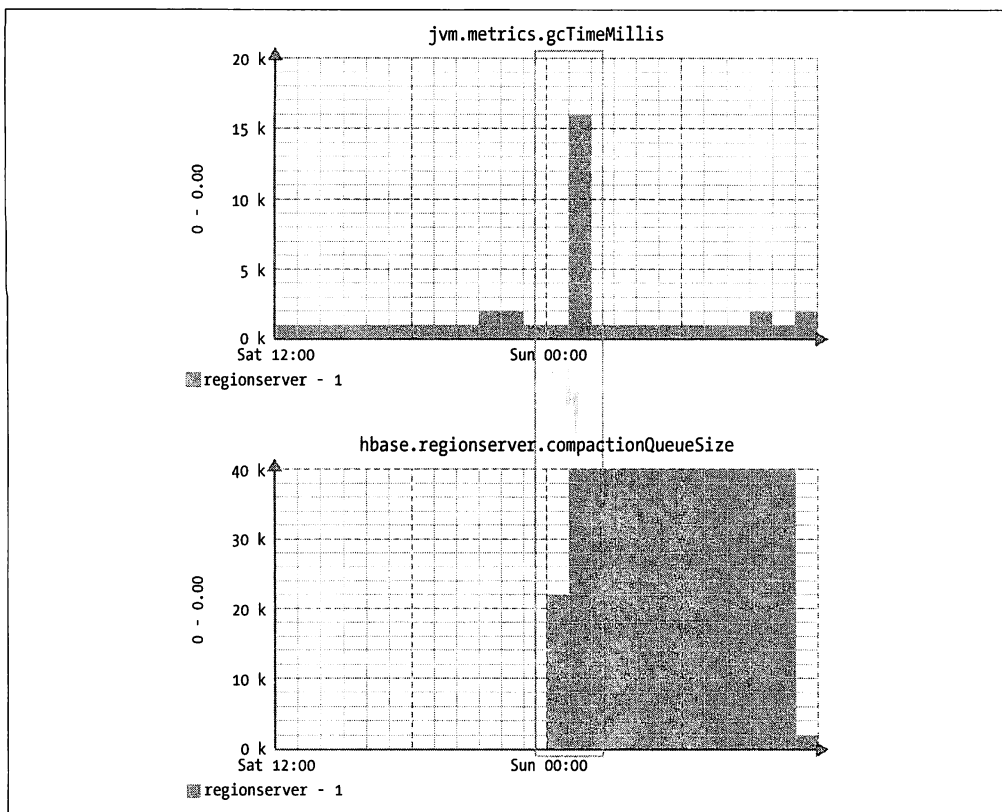


图 10-5 图形可以帮助用户整理相关事件的问题

10.4 JMX

Java Management Extensions 技术是 Java 应用程序导出当前状态的标准。除了之前已经看到的 Ganglia 和监控指标上下文提供的功能之外, JMX 还可以提供一些操作。这些操作允许用户在一个启用了 JMX 的 Java 进程上远程调用一些功能。

在通过 JMX 访问 HBase 进程之前, 用户必须启用它。此项工作可以通过在 `$HBASE_HOME/conf/hbase-env.sh` 配置文件中去掉和修改以下行注释来完成。

```
# Uncomment and adjust to enable JMX exporting
# See jmxremote.password and jmxremote.access in $JRE_HOME/lib/management to
# configure remote password access. More details at:
# http://java.sun.com/javase/6/docs/technotes/guides/management/agent.html
#
export HBASE_JMX_BASE="-Dcom.sun.management.jmxremote.ssl=false \
-Dcom.sun.management.jmxremote.authenticate=false"
export HBASE_MASTER_OPTS="$HBASE_JMX_BASE \
-Dcom.sun.management.jmxremote.port=10101"
```

```
export HBASE_REGIONSERVER_OPTS="$HBASE_JMX_BASE \
-Dcom.sun.management.jmxremote.port=10102"
export HBASE_THRIFT_OPTS="$HBASE_JMX_BASE \
-Dcom.sun.management.jmxremote.port=10103"
export HBASE_ZOOKEEPER_OPTS="$HBASE_JMX_BASE \
-Dcom.sun.management.jmxremote.port=10104"
```

这样会在没有安全保证的情况下启用 JMX 远程连接支持。假设大多数情况下 HBase 集群在防火墙外无法被访问，则没有必要进行安全验证。启用 JMX 的安全检查机制会使得安装过程稍微麻烦一些。^①用户需要重启 HBase 来激活这些修改。

当服务器启动时，它不仅会将监控指标注册到相应的上下文中，而且还会将它们导出为所谓的 JMX 属性。上文提到了如果用户想使用 JMX 读取监控指标的值，用户至少需要通过给 `period` 赋个合适的值来激活 `NullContextWithUpdateThread`。例如，最基本的 `hadoop-metrics.properties` 文件可能包含：

```
hbase.class=org.apache.hadoop.metrics.spi.NullContextWithUpdateThread
hbase.period=60

jvm.class=org.apache.hadoop.metrics.spi.NullContextWithUpdateThread
jvm.period=60

rpc.class=org.apache.hadoop.metrics.spi.NullContextWithUpdateThread
rpc.period=60
```

这将确保所有的监控指标每隔 10 秒更新一次，同时用户可以通过 JMX 属性检索到监控指标的值。不做这些将会导致所有的 JMX 属性没有实际作用。然而，用户也可以正常使用 JMX 操作。与此同时，如果用户已经启用了其他上下文，如 `GangliaContext`，这就足够了。

JMX 使用 `managed beans`（即 `MBeans`）的概念，用来提供特定的属性和操作集合。监控指标框架提供的监控指标上下文和 JMX 导出的 `MBeans` 有些相同的功能。这些 `MBeans` 用以下的方式进行声明：

```
hadoop:service=<service-name>,name=<mbean-name>
```

下面的 `MBeans` 是由 HBase 的各种进程提供的。

```
hadoop:service=Master, name=MasterStatistics
```

提供访问 `master` 监控指标的功能，如 10.2.2 节所介绍的内容。

```
hadoop:service=RegionServer, name=RegionServerStatistics
```

提供访问 `region` 监控指标的功能，如 10.2.3 节所介绍的内容。

① HBase 监控指标网页（<http://hbase.apache.org/metrics.html>）有如何添加密码及访问证书文件的介绍。

```
hadoop:service=HBase, name=RPCStatistics-<port>
```

提供访问 RPC 监控指标的功能，正如 10.2.4 节描述的。注意命名中的端口部分可能是动态的，在用户修改配置并指定 master 和 region 服务器绑定的端口时，这部分内容会随之变化。

```
hadoop:service=HBase, name=Info
```

提供访问常规监控指标的功能，如 10.2.6 节所描述的内容。

MasterStatistics、RegionServerStatistics 和 RPCStatistics 这些 MBeans 也提供了一个操作：resetAllMinMax。使用这个操作可以重置已经观测到的时间变化率（TVR）的最小和最大完成时间。

用户有很多方式来访问 JMX 属性和操作，以下介绍两种。

10.4.1 JConsole

Java 的发行版中有一个名为 JConsole 的帮助工具，它可以用来连接本地或远程的 Java 进程。假设用户在系统查找路径中已经包括了 \$JAVA_HOME 目录，用户可以使用以下方式来启动它：

```
$ jconsole
```

一旦应用打开，它会显示一个对话框来让用户选择连接本地还是远程的进程。图 10-6 展示了这个对话框。

用户可以用它连接本地或者远程的进程。因为用户已经配置了所有 HBase 进程监听特定的端口，所以推荐用户将它们作为远程进程访问。这样做的好处是，即使当进程 ID 改变时，用户仍然可以重新连接一个远程的服务器。但如果按本地连接方式的话，用户就不能这样做了，因为连接时受限于前面说的进程 ID。

通过使用 JMX 服务的 URL 可以连接远程 HBase 进程，其格式如下：

```
service:jmx:rmi:///jndi/rmi://<server-address>:<port>/jmxrmi
```

这里使用 Java Naming and Directory Interface（JNDI）注册并查找相应的必备细节信息，以及指定连接访问端口。某些情况下，用户可能在同一个物理服务器上运行多个 Java 进程，例如，Hadoop 的 NameNode 和 HBase 的 master，此时每个服务进程都需要分配一个唯一的端口。查阅 *hbase-env.sh* 文件中设定的各个进程的端口。例如，master 监听 10101 端口，region 服务器监听 10102 端口。因为用户只能在一个物理机器上运行一个 region 服务器，所以可以使所有 region 服务器使用一个相同的端口。这种情况下可以通过修改 <server-address>——为主机名或 IP 地址——来构造一个唯一的 *address:port* 对。

一旦连接到进程，用户会看到一个包含了它们各种细节信息的多标签页窗口。图 10-7 显示了连接进程之后的初始屏幕。不断更新的各种图表对了解服务器的当前状态来说非常有用。

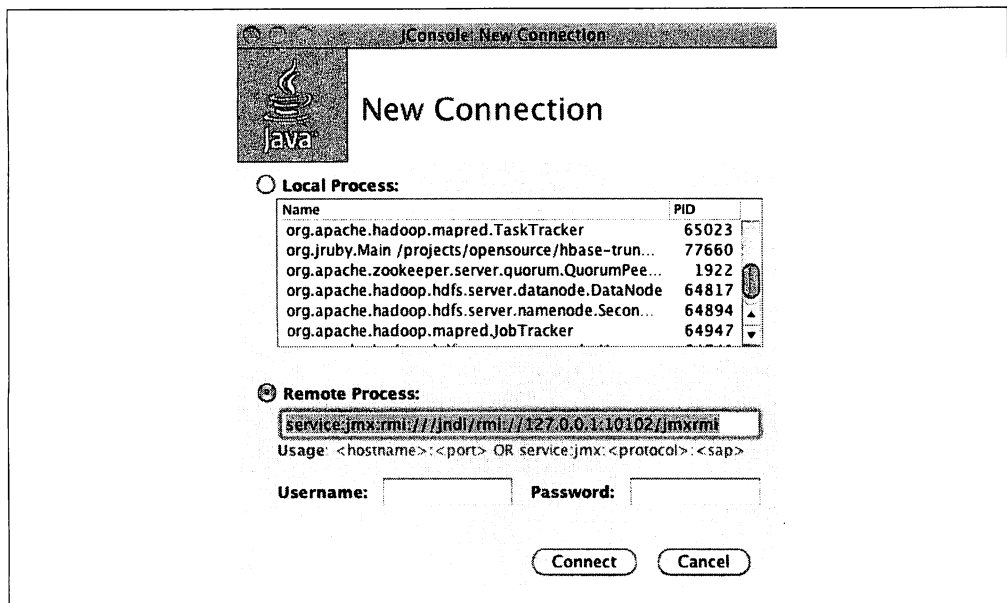


图 10-6 JConsole 启动时连接本地或者远程的进程

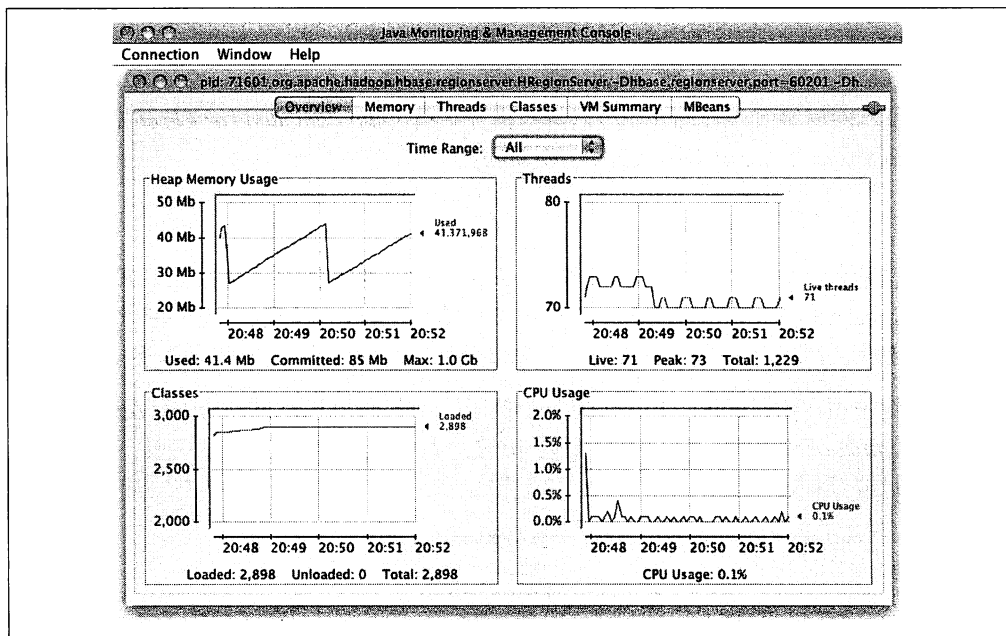


图 10-7 JConsole 提供了一个运行时的 Java 进程的内部情况

图 10-8 是一个 MBeans 标签页的截屏。用户可以观察已注册的 managed bean 提供的各种操作和属性。在图中用户可以看到 compactionQueueSize 监控指标的内容。

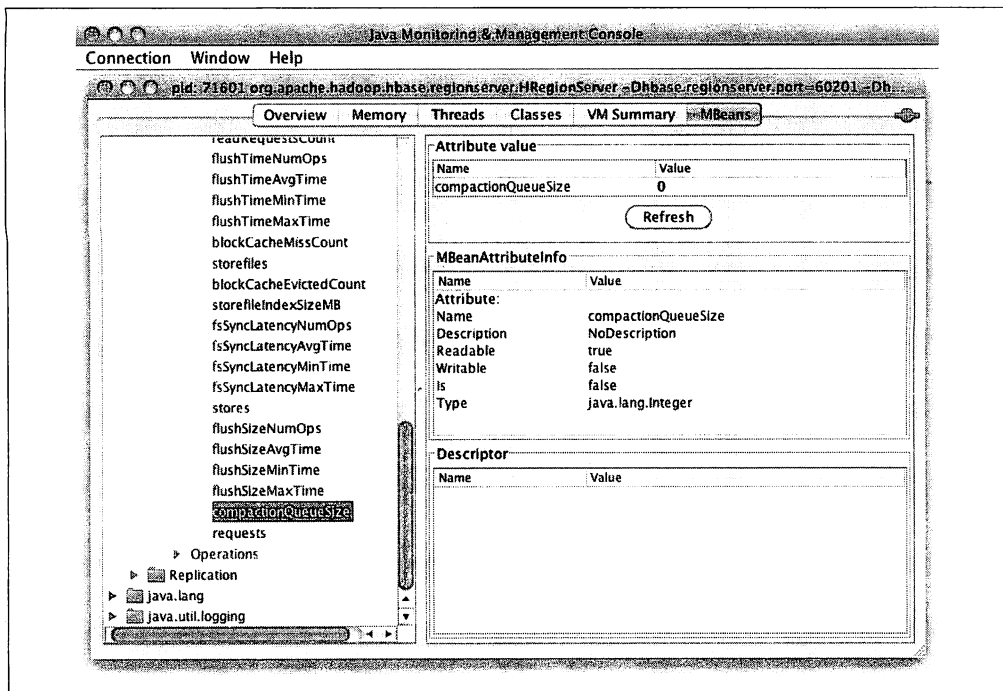


图 10-8 通过 MBeans 标签页观察各种 HBase 进程监控指标

各种选项及不同标签页内容的介绍可以查看官方文档 (<http://download.oracle.com/javase/6/docs/technotes/guides/management/jconsole.html>)。

10.4.2 JMX 远程 API

获取相同信息的另一种方式是 JMX Remote API，它通过使用远程方法调用 (remote method invocation, RMI)^① 来实现。还有很多有用的工具，它们也实现了可以访问远程受控 Java 进程的客户端。甚至 Hadoop 工程也在致力于为其添加一些基本支持。^②

下面我们将使用 JMXToolkit 工具作为一个例子，源代码可以从网站 <https://github.com/larsgeorge/jmxtoolkit> 获取。用户需要配置 git 命令行工具和 Apache Ant 工具。克隆源代码库，同时编译构建工具：

```
$ git clone git://github.com/larsgeorge/jmxtoolkit.git
Initialized empty Git repository in jmxtoolkit/.git/
...
$ cd jmxtoolkit
$ ant
Buildfile: jmxtoolkit/build.xml
```

① 参考官方文档中的详细信息 (<http://www.oracle.com/technetwork/java/javase/tech/Tindex-jsp-136424.html>)。

② 参见 HADOOP-4756 (<http://issues.apache.org/jira/browse/HADOOP-4756>)。

```
...
jar:
[jar] Building jar: /private/tmp/jmxtoolkit/build/hbase-jmxtoolkit.jar

BUILD SUCCESSFUL
Total time: 2 seconds
```

编译构建进程完成之后，用户可以通过调用-h 开关选项来观察提供的功能：

```
$ java -cp build/hbase-jmxtoolkit.jar \
  org.apache.hadoop.hbase.jmxtoolkit.JMXToolkit -h

Usage: JMXToolkit [-a <action>] [-c <user>] [-p <password>]
  [-u url] [-f <config>] [-o <object>] [-e regexp]
  [-i <extends>] [-q <attr-oper>] [-w <check>]
  [-m <message>] [-x] [-l] [-v] [-h]

-a <action>      Action to perform, can be one of the following
                  (default: query)

    create      Scan a JMX object for available attributes
    query       Query a set of attributes from the given objects
    check       Checks a given value to be in a valid range(see -w below)
    encode      Helps creating the encoded messages(see -m and -w below)
    walk        Walk the entire remote object list

...
-h              Prints this help
```

用户可以使用 JMXToolkit 来遍历或打印所有可用属性和操作集合。用户只需要知道希望获取的 MBeans 属性或操作的完整名字。不过由于目前还没有相应的列表，所以这并不是一项简单的工作。建立一个基本的配置文件可以帮助检索所有的信息列表。创建一个如下内容的属性文件。

```
$ vim hbase.properties
$ cat hbase.properties
;HBase Master
[hbaseMasterStatistics]
@object=hadoop:name=MasterStatistics,service=Master
@url=service:jmx:rmi:///jndi/rmi://${HOSTNAME1|localhost}:10101/jmxrmi
@user=${USER|controlRole}
@password=${PASSWORD|password}
[hbaseRPCMaster]
@object=hadoop:name=RPCStatistics-60000,service=HBase
@url=service:jmx:rmi:///jndi/rmi://${HOSTNAME1|localhost}:10101/jmxrmi
@user=${USER|controlRole}
@password=${PASSWORD|password}

;HBase RegionServer
[hbaseRegionServerStatistics]
```

```

@object=hadoop:name=RegionServerStatistics,service=RegionServer
@url=service:jmx:rmi:///jndi/rmi://${HOSTNAME2|localhost}:10102/jmxrmi
@user=${USER|controlRole}
@password=${PASSWORD|password}
[hbaseRPCRegionServer]
@object=hadoop:name=RPCStatistics-60020,service=HBase
@url=service:jmx:rmi:///jndi/rmi://${HOSTNAME2|localhost}:10102/jmxrmi
@user=${USER|controlRole}
@password=${PASSWORD|password}

;HBase Info
[hbaseInfo]
@object=hadoop:name=Info,service=HBase
@url=service:jmx:rmi:///jndi/rmi://${HOSTNAME1|localhost}:10101/jmxrmi
@user=${USER|controlRole}
@password=${PASSWORD|password}

;EOF

```

这个配置可以被作为参数送入工具中，以用来检索列出的 MBeans 的属性和操作。结果被存放在 *myjmx.properties* 中：

```

$ java -cp build/hbase-jmxtoolkit.jar \
  org.apache.hadoop.hbase.jmxtoolkit.JMXToolkit \
  -f hbase.properties -a create -x > myjmx.properties

$ cat myjmx.properties
[hbaseMasterStatistics]
@object=hadoop:name=MasterStatistics,service=Master
@url=service:jmx:rmi:///jndi/rmi://${HOSTNAME1|localhost}:10101/jmxrmi
@user=${USER|controlRole}
@password=${PASSWORD|password}
splitTimeNumOps=INTEGER
splitTimeAvgTime=LONG
splitTimeMinTime=LONG
splitTimeMaxTime=LONG
splitSizeNumOps=INTEGER
splitSizeAvgTime=LONG
splitSizeMinTime=LONG
splitSizeMaxTime=LONG
cluster_requests=FLOAT
*resetAllMinMax=VOID
...

```



这些命令假设用户以伪分布式和本地 HBase 实例方式来运行集群。当使用远程服务来运行时，用户需要简单地修改模板属性文件中的变量。例如，在上述命令中添加如下内容即可指定一个主机名（或者 IP 地址）到相应的主或从节点。

```
-DHOSTNAME1=master.foo.com -DHOSTNAME2=slave1.foo.com
```


当用户查看新创建的 *myjmx.properties* 文件时,会看到所有前文已经介绍过的监控指标。操作都是以* (即星号) 为前缀。

现在用户可以在命令行中使用这个工具和统计出的属性文件来请求监控指标的值。在下面的例子中, 第一个查询是请求相关属性的值, 第二个则触发了一个操作 (这种情况下没有返回值)。

```
$ java -cp build/hbase-jmxtoolkit.jar \
    org.apache.hadoop.hbase.jmxtoolkit.JMXToolkit \
    -f myjmx.properties -o hbaseRegionServerStatistics -q compactionQueueSize
compactionQueueSize:0
```

```
$ java -cp build/hbase-jmxtoolkit.jar \
    org.apache.hadoop.hbase.jmxtoolkit.JMXToolkit \
    -f myjmx.properties -o hbaseRegionServerStatistics -q *resetAllMinMax
```

创建完一个这样的属性文件后,用户可以检索单个值或者整个 MBeans 的值,同时也可以触发一些操作。这是个很重要的工具,它可以用来快速扫描受管理的进程并记录所有可用信息,查询 JMX MBeans 可以将用户从主观臆断中解脱出来。

JMXToolkit 和 Cacti

一旦 JMXToolkit JAR 被创建了,它就可以在一个 Cacti 服务器中使用。第一步,复制 JAR 包到 Cacti 的脚本目录 (可能安装时有所差异,用户需要确认目录正确)。下一步,解压脚本:

```
$ cd $CACTI_HOME/scripts
$ unzip hbase-jmxtoolkit.jar bin/*
$ chmod +x bin/*
```

一旦脚本解压完毕,用户可以测试其基本功能:

```
$ bin/jmxtkcacti-hbase.sh host0.foo.com hbaseMasterStatistics

splitTimeNumOps:0 splitTimeAvgTime:0 splitTimeMinTime:-1 splitTimeMaxTime:0\
splitSizeNumOps:0 splitSizeAvgTime:0 splitSizeMinTime:-1 splitSizeMaxTime:0 \
cluster_requests:0.0
```

JAR 也包含一组 Cacti 模版^①, 用户可以将其导入,并用来为 HBase 和 Hadoop 的 JMX MBeans 提供的变量进行初步的可视化工作。注意, 这些模版使用上述脚本并通过 JMX 获取监控指标的值。

在 Cacti 中, 创建图像与 Ganglia 相比更复杂, 后者可以动态添加来自于监控守护进程推送的监控指标。Cacti 拥有一系列 PHP 脚本, 这些脚本用来批量将集群中的服务器添加进来。

① 由于完成本书时模板已经有些陈旧了, 不过还是能与较新的 HBase 版本配合使用。

10.5 Nagios

Nagios 是一个被广泛使用的、用来获取与集群状态相关的定性数据的支持工具。它定期拉取当前的监控指标并且和阈值进行比较。一旦超过阈值，它将开启相应的规避动作，包括发送电子邮件、打电话、发送短信，使用各种方法执行各种触发脚本，必要时甚至重启物理服务器。

Nagios 中的典型检测项目可以是其自身以插件形式提供的，也可以是用户添加的脚本，且脚本必须返回特定程序退出值并将结果打印到标准输出。用户可以使用 JMX 将 Nagios 和 HBase 集成到一起，有很多可选的方法，其中包括前文提到过的 JMXToolkit。

JMXToolkit 的优点在于，用户制作了包含所有属性和操作的属性文件之后，就可以添加 Nagios 或者其他监控工具了，但是它们必须使用和 Nagios 相同的退出代码和标准输出消息。接下来，如果用户需要执行并修改检测一个不同的值时，只需要编辑一下属性文件。例如：

```
attributeXYZ=INTEGER|0:OK%3A%20%7B0%7D|2:WARN%3A%20%7B0%7D:80:<| \
1:FAILED%3A%20%7B0%7D:95:<
*operationABC=FLOAT|0|2::0.1:>=|1::0.5:>
```

使用之前安装 Cacti 相同的步骤，用户就可以将 Nagios 检测项连接到提供的 JMXToolkit 脚本上。如果用户要在属性文件中定义检测项，则仅仅需要设定查询的对象、属性或者操作。如果用户没有自定义检测项，则可以按照以下方式设定 Nagios 中的检测项：

```
$ bin/jmxtnagios-hbase.sh host0.foo.com hbaseRegionServerStatistics \
    compactionQueueSize "0:OK%3A%20%7B0%7D|2:WARN%3A%20%7B0%7D:10:>=| \
    1:FAIL%3A%20%7B0%7D:100:>"
OK: 0
```

注意，JMXToolkit 也有相应操作来将文本编码成合适的格式。

显然，使用 JMXToolkit 只是众多选择中的一种。关键的是，可视化描述集群和监控集群对于集群的维护是非常重要的，另一方面，它可以帮助用户更容易地追踪问题。推荐用户在项目的早期就部署实现以上两项功能，同时还推荐用户使用反映真实情况的负载来测试系统，因为这样做之后用户既熟悉了图形意义，又了解了如何分析它们。它还能帮助用户设定合理的阈值，并找出对应的上界和下界，这将为用户在随后的生产环境中减少很多麻烦。

现在，我们已经了解了如何安装和使用集群。为了使 HBase 能够如预期一样运行，还需要调整一些配置。这一章将会列出大量的技巧来帮助用户优化集群和反复验证其性能。

11.1 垃圾回收优化

用户需要调整的一组较为底层的 region 服务器启动参数是垃圾回收参数。注意，垃圾回收时 master 通常不会产生问题，这主要是由于 master 没有处理任何过重的负载并且实际的数据服务并不经过它。这些参数只需要被添加到 region 服务器的启动参数中。

用户可能会问什么要通过优化垃圾回收来使 HBase 有效率地运行。其主要原因是 JRE 在默认情况下会按照一般情况来估计用户的程序在做什么、它们怎么创建对象、如何分配堆去处理数据等。这些假设在多数情况下都是正确的。此外，JRE 能够运用启发式的算法来根据运行的进程进行调整。甚至当启发式的学习调整功能受限于具体实现时，JRE 也能够更好地处理某些特殊情况。

现在的底线是它不能很好地处理 region 服务器。主要原因是当 region 服务器处理特定的负载时，特别是写入量过大的负载，繁重的负载会迫使内存分配策略无法安全地只依赖 JRE 对程序行为的各种假设：用户需要使用 JRE 所提供的选项来调整垃圾回收策略以应对这些特殊情况。

对写入负载过大的情况来说，memstore 在不同时期创建并释放着各种不同大小的对象。因为数据是被存储在内存缓冲区内的，它们会被保留直到超过用户配置的最小刷写大小，用户可以在配置文件中使用 `hbase.hregion.memstore.flush.size` 来设置 region 的 memstore 刷写大小，此外在定义表时也可以对不同的表单独指定表的这个

属性。

一旦 `memstore` 大于这个值，数据就会被刷写到磁盘，并创建一个新的存储文件。因为写入磁盘的数据是由客户端在不同时间写入的，那么它们占据的 Java 堆空间很可能是不连续的，所以 Java 虚拟机的堆内存会出现孔洞。

数据会根据自身在内存中停留的时间被保存在 Java 堆中分代结构的不同位置：被快速插入且被刷写到磁盘的数据，通常会被分配到被称为年轻代（`young generation`）或新生代（`new generation`）的堆中。这种空间可以被迅速地回收，并且对内存管理没有影响。

另一方面，如果数据在内存中停留的时间过长，例如，向一个列族中插入数据的速度较慢时，对应的数据就很可能被提升为了老生代（`old generation`）或终生代（`tenured generation`）。年轻代和老生代的不同点在于空间大小：年轻代占用的空间在 128 MB 到 512 MB 之间，而老生代几乎占用了所有可以占用的堆空间，通常是好几 GB 的内存。



用户可以通过向 `hbase-env.sh` 配置文件中添加 `HBASE_OPTS` 或者 `HBASE_REGIONSERVER_OPTS` 变量来设置垃圾回收相关选项。后者仅仅影响 region 服务器进程（例如，相对于 master），并且也是推荐的修改方式。

指定新生代的空间可以通过以下两种方式完成：

```
-XX:MaxNewSize=128m -XX:NewSize=128m
```

另一种更简洁的方式是将之前的两个代码合并成一个简便的选项：

```
-Xmn128m
```



使用 128 MB 是一个好的开端，用户可以通过对 JVM 各指标的进一步观察来确认年轻代的大小是否满足需求。

注意，默认值对于多数 region 服务器面对的负载来说都太小，所以它必须增大。如果不这样做的话，用户可能会发现服务器 CPU 的使用量会急剧上升，因为从年轻代中收集对象会消耗大量的 CPU。

为了重复使用由于刷写数据到磁盘而产生（或由其他对象的创建和释放产生）的堆孔洞，新老生代都需要由 JRE 来维护。如果在某个时间内，应用程序需要的堆大小不适合这些碎片空间，那么 JRE 需要压缩堆内存碎片。这个操作包含了其他隐式操作，例如，将长时间存在的对象从年轻代提升并转移到老生代。如果这个操作失败，用户将

会在垃圾回收日志中看到提升失败的信息。



强烈建议在 JRE 日志中输出垃圾回收的详细信息。用户可以通过添加以下 JRE 选项来达到目的：

```
-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps \  
-Xloggc:$HBASE_HOME/logs/gc-$(hostname)-hbase.log"
```

一旦启用此日志选项，用户将会监测到伴随着长时间停顿的"concurrent mode failure"或者"promotion failed"信息。

请注意，该日志文件不会像其他日志一样按日期定时滚动存放不同文件中，用户需要自己手动管理（例如，使用基于 *cron* 的每日滚动转存任务）。

以上讨论的重写并整理堆中不同代的过程被称之为垃圾回收，并且用户可以通过不同的 JRE 参数来指定不同的垃圾回收实现策略，推荐的值是：

```
-XX:+UseParNewGC and -XX:+UseConcMarkSweepGC
```

第一个选项是设置年轻代使用 **Parallel New Collector** 垃圾回收策略：这将停止运行 Java 进程而去清空年轻代堆。与老生代相比，新生代很小，所以这个过程花费时间很短，通常只需要几百毫秒时间。

以上回收策略对于较小的年轻代来说是可以接受的，但是并不适合老生代：在最差的情况下，以上回收策略会造成数秒钟甚至几分钟的进程停顿。一旦停顿时间达到了 ZooKeeper 会话超时限制，这个服务器将被 master 认为已经崩溃并且随后会被抛弃。一旦 region 服务器从垃圾回收暂停中恢复之后，它会获知自己已经被抛弃，然后它会自行关闭。

这种情况可以通过使用并行标记回收器（Concurrent Mark-Sweep Collector, CMS）来缓解，这种回收策略通过上述例子中后面的选项启用。不同之处在于其工作时试图在不停止运行 Java 进程的情况下尽可能异步并行地完成工作。这种策略将增加 CPU 的负载，但是却可以避免重写老生代堆碎片时的停顿——除非发生提升失败，这种错误会迫使垃圾回收暂停运行 JAVA 进程并进行内存整理。

CMS 有一个额外的开关选项，这个选项控制着将在什么时候开始并发标记和清扫检查。这个值可以通过以下选项来设置：

```
-XX:CMSInitiatingOccupancyFraction=70
```

这个值是一个百分比，并指定后台线程何时启用，用户需要设定这个值以防止另一种情况发生，即并发模式失败。当后台进程为回收空间而标记和清理堆内存时，可能会发生堆空间不足（如回收碎片时）。在这种情况下，JRE 必须暂停运行 Java 进程并且通过释放对象来强制释放空间，或者将停留时间较长的对象转移到老生代。

将初始占用百分比设置为 70%意味着其比 region 服务器设置的 60%的堆占用率要大一点, 60%的堆占用率由默认的 20%块缓存和 40%的 memstore 组成。这样的配置允许在堆空间被占用完之前就开始并行垃圾回收过程, 同时这样的配置也不会使回收工作开始得太早而使回收过程频繁运行。

把上面的设置放在一起, 用户可以使用下列内容作为最开始的配置:

```
export HBASE_REGIONSERVER_OPTS="-Xmx8g -Xms8g -Xmn128m -XX:+UseParNewGC \  
-XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=70 -verbose:gc \  
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps \  
-Xloggc:$HBASE_HOME/logs/gc-$(hostname)-hbase.log"
```



注意, 在真实的硬件环境下, 不推荐使用 `-XX:+CMSIncrementalMode` 参数。

这些设置结合了编写本书时最好的实践经验。如果用户使用的是比 Java 6 更新的版本, 应当评估一下新垃圾回收机制的实现, 并选择一个合适的配置。

调整年轻代空间的大小是十分重要的, 这样生存期较长的对象不会过快地引起老年代产生内存碎片。从另一方面来讲, 年轻代空间也不能太大, 否则回收时会引起太长的停顿。虽然这些停顿不会使 region 服务器超时, 但停止几百毫秒会影响服务器响应延时。

同样, 当优化块缓存和 memstore 大小时, 用户应确保将占用百分比的初始值设置得稍大一些。同时, 用户必须合理地指定这两个值, 它们的和肯定不能大于 100%。用户还需要考虑管理一般 Java 类的开销等。按默认情况, 两个默认值和为 60%是比较合理的。更多的信息请参考 11.8 节。

11.2 本地 memstore 分配缓冲区

HBase 的 0.90. x 版本引入了一种高级机制来缓解 region 服务器内存碎片问题, 这个问题主要是 memstore 的扰动造成的 (不断创建和释放内存空间): 本地 memstore 分配缓冲区 (Memstore-Local Allocation Buffers, MSLAB)。

前面的章节解释了生存期长的 KeyValue 实例一旦刷写到磁盘, 就会在老年代的堆上产生孔洞。申请新空间时, 由于碎片过多导致没有足够大的连续空间分配, JRE 会退回到使用应用程序停止 (stop-the-world) 垃圾回收器, 这样会导致其重写整个堆空间并压缩剩余的可用对象。

减少这些压缩回收的关键是减少碎片, MSLAB 就是为此设计的。其关键在于只允许从堆中分配相同大小的对象。一旦这些对象分配并且最终被回收, 它们将在堆中留下固定大小的孔洞。之后调用相同大小的新对象将会重新使用这些孔洞:

这样就不会产生提升错误（promotion error），因此就不需要应用程序停止压缩回收了。

MSLAB 是许多大小固定的缓冲区，用来存储大小不同的 **KeyValue** 实例。当一个缓冲区不能放下一个新加入的 **KeyValue** 时，系统就认为这个缓冲区已经被占满了，然后创建一个新的固定大小的缓冲区。

这个特性默认是在 0.92 版中被启用的，而在 0.90 版中没有被启用。用户也可以通过 `hbase.hregion.memstore.mslab.enabled` 配置属性来覆盖这个属性。用户在使用新特性时需要充分测试以避免发生问题，使用 **MSLAB** 会推迟垃圾回收停顿的发生，这样会有很多好处，不过用户仍然需要处理长时间的垃圾回收停顿。如果用户仍然在经历这些停顿，那么用户可以考虑以几天或几周的频率在停顿发生前重启服务。



因为这个新特性还没有在生产环境中长时间运行以测试其功能，建议使用时仔细观察服务器的行为。

每一个被分配的、固定大小的缓冲区的大小都是由 `hbase.hregion.memstore.mslab.chunksize` 属性控制的。默认值是 2 MB，并且这是一个合理的开始值。根据 **KeyValue** 实例的大小，用户可能需要调整这个值：如果用户需要储存更大的单元格，例如，其大小为 100 KB，那么就需要增加 **MSLAB** 的大小以容纳更多的单元格。

同样也有一个存储缓冲区的上边界。这个是通过 `hbase.hregion.memstore.mslab.max.allocation` 属性来设置的，并且其默认值是 256 KB。任何大于这个值的单元格将会直接在 Java 堆中申请空间。如果用户存储了许多大于上限的 **KeyValue** 实例，用户将会更早遇到与堆内存碎片相关的停顿。

使用 **MSLAB** 是有代价的：它们将更加浪费堆空间，因为用户不太可能把缓冲区都用到最后一个字节，剩余的没有使用的空间则将被浪费。用户需要在压缩收集器造成的服务停止和额外空间消耗之间找到一个平衡。

最后，因为使用缓冲区需要额外的内存复制工作，所以使用缓冲区比直接使用 **KeyValue** 实例要稍慢一点。用户需要衡量这些工作负载是否会产生负面影响。

11.3 压缩

HBase 支持大量的压缩算法，并且可以支持列族级别上的数据压缩。除非有特殊原因，例如，使用已经压缩过的内容如 **JPEG** 图像，否则我们还是推荐开启压缩。对于其他的使用场景来说，压缩通常都会带来较好的性能，因为 CPU 压缩和解压消耗的时间比从磁盘中读取和写入更多数据消耗的时间更短。

11.3.1 可用的编解码器

用户可以从固定支持的压缩算法列表中选择一个算法。它们有不同的压缩质量和需求，选择时通常需要考虑它们的压缩率、CPU 消耗和其他安装需求。



目前 HBase 并没有提供嵌入式的压缩算法。HBase 提供的要么是 Java 本身的一部分，要么是操作系统级别的第三方类库。它们需要的支持库需要用户提前编译构建，不过有些算法已经集成在 HBase 中。

在比较压缩算法之前，用户可以参照表 11-1 或者查看 Google 在 2005 年发布的压缩算法比较信息。^①虽然该数据的时间比较久远，但是它们仍然能够展现每种压缩算法的特点。

表 11-1 压缩算法比较

算法	%压缩比	压缩	解压
GZIP	13.4%	21 MB/s	118 MB/s
LZO	20.5%	135 MB/s	410 MB/s
Zippy/Snappy	22.2%	172 MB/s	409 MB/s

注意，有一些算法拥有更好的压缩率，而另一些算法拥有更快的编码速度和非常快的解码速度。用户最好根据实际情况选择一个最适合的压缩算法。



在 2011 年 Snappy 可用之前，虽然 LZO 没有最好的压缩率，但它仍是被推荐的算法。GZIP 在压缩率上有优势，但它也是 CPU 密集型算法，其微弱的节省存储空间上的优势通常是无法弥补速度较慢和 CPU 使用率过高的劣势，所以不推荐使用 GZIP。

Snappy 拥有和 LZO 一样的质量，并有兼容的使用许可，而且在第一次测试时就表明在使用 Hadoop 和 HBase 时，使用 Snappy 比使用 LZO 性能稍好。因此，根据以上描述，用户应该更多考虑使用 Snappy 而非 LZO。

1. Snappy

用户可以通过 Google 以 BSD 许可协议发布的 Snappy 来使用 BigTable 所用的压缩算法（称为 Zippy）。与其他压缩算法相比，它在提高压缩速度以及达到合理压缩率两方面均进行了优化。

Snappy 的代码是用 C++ 编写的，HBase 的 0.92 版本包括了所需使用的 JNI 库。用

^① 演示的视屏可以在网上下载（<http://horfolk.cs.washington.edu/htbin-post/unrestricted/Colloq/details.cgi?id=437>）。

户首先需要通过使用包管理软件，如 apt、rpm 和 yum，或者直接从源代码构建它们，然后安装它们的本地可执行二进制文件，这样它们就能够被 JNI 库发现并调用了。^①

当用户设置好 Snappy 的支持后，用户必须安装 Snappy 本地二进制库到所有的 region 服务器上，只有这样它们才能被库正常使用。

2. LZO

Lempel-Ziv-Oberhumer（简称 LZO）是一个无损压缩算法，其专注于解压速度，并且使用 ANSI C 编写。和 Snappy 相似，它也需要 JNI 库才能使 HBase 能够调用它。

不幸的是，由于 LZO 的许可问题，HBase 不能将所需的 JNI 库集成到安装包：HBase 使用的是 Apache 许可证，而 LZO 使用的是不兼容的 GNU General Public (GPL) 许可。这就意味着，LZO 安装需要在 HBase 安装后单独进行安装^②。

3. GZIP

一般来说，GZIP 压缩算法的压缩比会比 Snappy 或者 LZO 高，但是速度较慢。虽然这看起来像缺点，但它能减少存储空间的开销。

这个性能问题可以通过使用本地操作系统 GZIP 库的方式得到缓解。HBase 使用的压缩库（由 Hadoop 提供）会自动检查是否有本地库可用。如果没有本地库可用^③，用户将会在日志文件中看见：“Got brand-new compressor”。它们表明载入本地版本失败，并返回默认的 Java 实现代码来替代。压缩仍然会运行但是速度会稍微变慢。

另外的劣势是 GZIP 需要消耗大量的 CPU 资源。这将会加重服务器的负载，同时这种负载应当被监控以免出现问题。

11.3.2 验证安装

一旦安装了一种 HBase 支持的压缩算法，强烈建议用户检查一下安装是否成功。在 HBase 中有好几种机制能够验证压缩算法是否正常。

1. 压缩测试工具

HBase 包含一个能够测试压缩设置是否正常的工具。用户可以通过输入 `./bin/hbase org.apache.hadoop.hbase.util.CompressionTest` 来使用它。这样会显示如何使用工具

① Java 使用 Java 本地接口（Java Native Interface, JNI）来与本地库或应用集成。

② 见维基百科页面“Using LZO Compression”（<http://wiki.apache.org/hadoop/UsingLzoCompression>），其中有在 HBase 中使用 LZO 的详细介绍。

③ Hadoop 项目有一个页面（http://hadoop.apache.org/common/docs/current/native_libraries.html）专门介绍编译和或安装本地库的方法，其中包括 GZIP 的内容。

的说明：

```
$ ./bin/hbase org.apache.hadoop.hbase.util.CompressionTest
Usage: CompressionTest <path> none|gz|lzo|snappy
```

For example:

```
hbase class org.apache.hadoop.hbase.util.CompressionTest file:///tmp/ testfile.gz
```

用户需要指定一个测试文件，系统会用所选的压缩算法创建并测试这个文件。例如，用户想在 HDFS 中使用测试文件检查 GZIP 是否安装可以运行：

```
$ ./bin/hbase org.apache.hadoop.hbase.util.CompressionTest \
/user/larsgeorge/test.gz gz
11/07/01 20:27:43 WARN util.NativeCodeLoader: Unable to load native-hadoop \
library for your platform... using builtin-Java classes where applicable
11/07/01 20:27:43 INFO compress.CodecPool: Got brand-new compressor
11/07/01 20:27:43 INFO compress.CodecPool: Got brand-new compressor
SUCCESS
```

工具报告成功之后，用户可以在定义列族时使用这种压缩算法。注意运行这条命令时，如果打印"Got brand-new compressor"信息，则意味着本地 GZIP 库没有被服务器找到，但是它可以使 Java 自带的 GZIP 相关代码进行解压缩。

对一种没有安装好的压缩类型尝试使用同样的操作将会得到一个异常：

```
$ ./bin/hbase org.apache.hadoop.hbase.util.CompressionTest \
file:///tmp/test.lzo lzo
Exception in thread "main" Java.lang.RuntimeException: \
Java.lang.ClassNotFoundException: com.hadoop.compression.lzo.LzoCodec
at org.apache.hadoop.hbase.io.hfile.Compression$Algorithm$1.getCodec()
at org.apache.hadoop.hbase.io.hfile.Compression$Algorithm.getCompressor
```

如果出现这种情况，需要返回并再次检查安装。在安装 JNI 或本地压缩库后，用户可能还需要重新启动 region 服务器。

2. 启动检查

即使压缩测试工具报告成功了，并且确认了压缩库已正确安装，用户仍旧可能在接下来的使用中遇到问题：由于 JNI 需要首先安装好本地库，如果缺失这一步，将会在添加新服务器时出现问题，导致新的服务器使用本地库打开含有压缩列族的 region 失败（参见 12.5.3 节中“基本安装检查表”的内容）。

这个问题可以通过在（默认未设定）hbase.regionserver.codecs 属性中设定所有需要的 JNI 库来缓解。如果其中一个本地库没有被找到，则整个 region 服务器将无法启动。这样系统会在启动过程中产生异常，以帮助用户快速发现问题，而不用面对启动之后的各种问题。

例如，通过以下配置，在 **region** 服务器启动的时候将会检查 Snappy 和 LZO 压缩库是否已经正确安装：

```
<property>
  <name>hbase.regionserver.codecs</name>
  <value>snappy,lzo</value>
</property>
```

如果在一些情下载入 JNI 库失败，服务器将会终止启动并出现一个 I/O 异常信息 "Compression codec <codec-name> not supported, aborting RS construction"。用户可以修复这些问题，并尝试再次启动 **region** 服务器守护进程。

用户可以在每一个 HBase 支持的压缩算法上完成这个测试，并且别忘了将修改过的配置文件复制到所有的 **region** 服务器上并重启它们。

11.3.3 启用压缩

启用压缩需要安装相应的 JNI 和本地压缩库（除非用户只想使用基于 Java 代码的 GZIP 压缩），就像前面描述的，在定义列族时，用户可以指定一个压缩算法。

用户在创建表的时候可以进行这些操作，可用的值在 5.1.3 节中有介绍。

```
hbase(main):001:0> create 'testtable',{ NAME => 'colfam1',COMPRESSION => 'GZ' }
0 row(s) in 1.1920 seconds

hbase(main):012:0> describe 'testtable'
DESCRIPTION                                ENABLED
{NAME => 'testtable',FAMILIES => [{NAME => 'colfam1', true
BLOOMFILTER => 'NONE',REPLICATION_SCOPE => '0',VERSIONS
=> '3',COMPRESSION => 'GZ',TTL => '2147483647',BLOCKSIZE
=> '65536',IN_MEMORY => 'false',BLOCKCACHE => 'true'}}}
1 row(s) in 0.0400 seconds
```

describe 命令用于读取用户新创建表的模式（schema）。用户可以看到，压缩方式被设置为 GZIP（要求使用 GZ 简称）。另外一种方式是通过 *alter* 命令对现有表进行启用、更改或禁用压缩算法。

```
hbase(main):013:0>create 'testtable2','colfam1'
0 row(s) in 1.1920 seconds

hbase(main):014:0>disable 'testtable2'
0 row(s) in 2.0650 seconds

hbase(main):016:0>alter 'testtable2',{ NAME => 'colfam1',COMPRESSION => 'GZ' }
0 row(s) in 0.2190 seconds
```

```
hbase(main):017:0>enable 'testtable2'
0 row(s) in 2.0410 seconds
```

注意，用户需要先禁用表。对于修改表操作来说，必须首先禁用表。最后，*enable* 命令会使表重新启用上线。

将压缩格式更改为 *NONE* 会使给定的列族禁用压缩。

操作延迟

注意，即使用户进行了启用、禁用或改变压缩算法的操作，这些操作并不会立竿见影。所有的存储文件实际都仍旧使用以前的压缩算法，或者没有使用压缩算法。而更改后的 *region* 会在刷写存储文件时使用新的压缩格式。

如果用户想强制将所有现成的文件都使用新设置的格式重写，那么可以在 Shell 中使用 *major_compact* '*tablename*>' 命令让 *major* 合并进程在后台运行。这将会重写所有的文件，并使用新的设置。记住，这可能会使资源十分紧张，请用户在确定可用的资源充足时再强制执行。同时也要注意，*major* 合并将会运行一段时间，这个是由存储文件的数量和大小来决定的，请耐心等待。

11.4 优化拆分和合并

HBase 内置的处理拆分和合并的机制一般是合理的，并且它们按照预期处理任务，但在有些情况下，还是需要按照应用需求对这部分功能进行优化以获得额外的性能改善。

11.4.1 管理拆分

通常 HBase 是自动处理 *region* 拆分的：一旦它们达到了既定的阈值，*region* 将被拆分成两个，之后它们可以接收新的数据并继续增长。这个默认行为能够满足大多数用例的需求。

其中一种可能出现问题的情况被称之为“拆分/合并风暴”：当用户的 *region* 大小以恒定的速度保持增长时，*region* 拆分会在同一时间发生，因为同时需要压缩 *region* 中的存储文件，这个过程会重写拆分之后的 *region*，这将会引起磁盘 I/O 上升。

与其依赖 HBase 自动管理拆分，用户还不如关闭这个行为然后手动调用 *split* 和 *major_compact* 命令。用户可以通过设置这个集群的 *hbase.hregion.max.filesize* 值或者在列族级别上把表模式中对应参数设置成非常大的值来完成。为防止手动拆分无法运行，最好不要将其设置为 *Long.MAX_VALUE*。用户最好将这个值设置为一个合理的上限，例如，100 GB（如果触发的话将会导致一个小时的 *major* 合并）。

手动运行命令来拆分和压缩 region 的好处是可以对它们进行时间控制。在不同 region 上交错地运行，这样可以尽可能分散 I/O 负载，并且避免拆分/合并风暴。用户可以实现一个可以，调用 `split()` 和 `majorCompact()` 方法的客户端，也可以使用 Shell 交互地调用相关命令，或者使用 `cron` 定时地运行它们。用户可以参见 `RegionSplitter` 类（0.90.2 版本添加进来的）的另一种拆分 region 的方法：其拥有滚动拆分（rolling split）的特性，用户可以使用该功能拆分正在长时间等待合并操作完成的 region（参见 `-r` 和 `-o` 命令行选项）。

另外一个手动管理拆分的优势是用户能够更好地在任意时间控制哪些 region 可用。这对于用户需要解决底层 bug 这种少数情况来说是十分有用的，例如，排查某一个 region 的问题。在使用自动拆分时，用户可能发现要检查的 region 已经被两个拆分后的子 region 替代了。这些子 region 有新的名字，并且客户端需要大量的时间重新定位 region，这使得查询所需要的信息变得更加困难。

11.4.2 region 热点

参考 10.2.3 节^①的内容，用户将会发现自己应用程序的模式是否会让特定的 region 成为热点。

如果存在这种情况，请参照第 9 章中讨论的内容，尤其是 9.1 节的内容：用户需采用盐析主键（salt key）或者使用随机的行键来把负载均衡到所有的服务器。

唯一可以缓解这种现象的途径就是手动地将热点 region 按特定的边界拆分出一个或多个新 region，然后将子 region 负载分布到多个 region 服务器上。用户可以为 region 指定一个拆分行键，即 region 被拆分为两部分的位置。用户可以指定 region 中任意的行键，这样用户也可以生成大小完全不同的两个 region。

这个只能在处理非完全连续的行键范围时起作用，因为采用连续的行键时，过一段时间插入的数据总会集中到最近生成的几个 region 上。

表热点

对于拥有很多 region 的表来说，大部分 region 分布并不均匀，即大多数 region 位于同一个 region 服务器上^②。这就意味着，即使用随机的 key 来写入数据，某一台 region 服务器的负载仍大于其他的 region 服务器。用户可以从 HBase Shell 或者使用 `HBaseAdmin` 类中的 API，并通过 5.2.4 节中介绍的 `move()` 函数显式地把 region 从一个 region 服务器移动到另一个 region 服务器。另外一个方法就是使用 `unassign()` 方法或者 Shell 命令简单地当前服务器移除受影响的表的 region，master 会立即将其部署到其他 region 服务器上。

① 作为另一种选择，用户也可以查看 master UI 页面的 TPS，参见 6.5.1 节中的“主页”部分。

② 在 HBase 0.92.0 中，开发人员已经完成了很多改善工作。

11.4.3 预拆分 region

管理拆分能够在集群负载增加时有效地进行负载控制。但是，用户仍然会面临的一个问题是，在用户初始创建一张新表之后，用户需要频繁地拆分 region，因为建立的新表通常只有一个 region，不推荐让单个 region 增长到太大。因此，在表创建时，最好就有较大数量的 region。用户可以在创建表时指定需要的 region 数目来达到预拆分的目的。

管理接口中的 `createTable()` 方法和 Shell 中的 `create` 命令都可以接受以列表形式提供的拆分行键作为参数，该参数在创建表的时候会被用来预拆分 region。HBase 提供了一个能帮助用户创建预拆分表的工具类 `RegionSplitter`。不含参数时它将会显示使用说明信息：

```
./bin/hbase org.apache.hadoop.hbase.util.RegionSplitter
usage: RegionSplitter <TABLE>
  -c <region count>      Create a new table with a pre-split number of
                           regions
  -D <property=value>    Override HBase Configuration Settings
  -f <family:family:...> Column Families to create with new table.
                           Required with -c
  -h                      Print this usage help
  -o <count>             Max outstanding splits that have unfinished
                           major compactions
  -r                      Perform a rolling split of an existing region
  --risky                 Skip verification steps to complete
                           quickly.STRONGLY DISCOURAGED for production
                           systems.
```

默认采用 `MD5StringSplit` 类将行键拆分到不同的段中。用户能够通过实现提供的 `splitAlgorithm` 接口来定义自己的算法，并且通过使用 `-D split.algorithm=<your-algorithm-class>` 参数将它融入到工具之中。以下例子使用了提供的算法并创建了一张预拆分的表：

```
./bin/hbase org.apache.hadoop.hbase.util.RegionSplitter \
-c 10 testtable -f colfam1
```

在 master 的 Web 界面中，用户能够通过点击刚刚创建的表名来查看生成的 region。

```
testtable,,1309766006467.c0937d09f1da31f2a6c2950537a61093.
testtable,0ccccccc,1309766006467.83a0a6a949a6150c5680f39695450d8a.
testtable,19999998,1309766006467.1eba79c27eb9d5c2f89c3571f0d87a92.
testtable,26666664,1309766006467.7882cd50eb22652849491c08a6180258.
testtable,33333330,1309766006467.cef2853e36bd250c1b9324bac03e4bc9.
testtable,3ffffffc,1309766006467.00365940761359fee14d41db6a73ffc5.
testtable,4ccccccc,1309766006467.f0c5045c304c2ff5338be27e81ae698e.
```

```
testtable,59999994,1309766006467.2d854f337aa6c09232409f0ba1d4964b.
testtable,66666660,1309766006467.b1ec9df9fd90d91f54cb18da5edc2581.
testtable,7333332c,1309766006468.42e179b78663b64401079a8601d9bd06.
```

或者使用 Shell 的 `create` 命令:

```
hbase(main):001:0>create 'testtable','colfam1',\
{ SPLITS => ['row-100','row-200','row-300','row-400'] }
0 row(s) in 1.1670 seconds
```

这将生成以下的 region:

```
testtable,,1309768272330.37377c4ab0a944a326ba8b6596a29396.
testtable,row-100,1309768272331.e6092cc777f58a08c61bf081aba14916.
testtable,row-200,1309768272331.63c9630a79b37ebce7b58cde0235dfe5.
testtable,row-300,1309768272331.eead6ad2ff3303ffe6a3126e0df3ff7a.
testtable,row-400,1309768272331.2bee7417fa67e4ac8c7210ce7325708e.
```

关于如何设定预拆分的 region 数量,用户可以先按照每个服务器 10 个 region 来进行预拆分,随着时间的推移观察数据的增长情况。先设置较少的 region 数目再稍后滚动拆分它们是一种更好的方法,因为过多的 region 通常会影响集群性能。

另一种方法是,用户可以基于 region 中最大的存储文件大小来决定预拆分 region 的数量,随着数据的增加,该大小会随之一起增加。用户希望最大的 region 正好能够跳过 major 合并,否则用户可能会面对前面所提到的 compaction 风暴。

如果用户将 region 预拆分的太小,可以通过增加 `hbase.hregion.majorcompaction` 的值来加大 major 合并的间隔。如果用户的数据规模增加过大,用户可以使用 `RegionSplitter` 工具在所有 region 上通过网络 I/O 执行安全的滚动拆分。

使用手动拆分和预拆分是高级概念,需要用户有谨慎的计划并仔细监控操作时 HBase 系统的运行情况。另一方面,这能够避免全局一致的数据增长造成的合并风暴,并可以通过手动拆分摆脱 region 热点的困扰。

11.5 负载均衡

master 有一个内置的叫做均衡器的特性。在默认的情况下,均衡器每五分钟运行一次,这是通过 `hbase.balancer.period` 属性设置的。一旦均衡器启动,它将会尝试均匀分配 region 到所有 region 服务器。启动均衡器时,均衡器首先会确定一个 region 分配计划,该计划用于描述 region 如何移动。然后通过迭代调用管理 API 中的 `unassign()` 方法开始移动 region。

均衡器有一个可以限制自身运行时间的上限,用户可以通过 `hbase.balancer.max.balancing` 属性来配置,默认设置为均衡器运行间隔周期的一半,即两分半钟。

用户可以通过均衡器开关来控制均衡器：使用 Shell 的 `balance_switch` 命令来更改均衡器的开启和关闭状态，或者使用 `balanceSwitch()` 接口来做同样的事情。当禁用均衡器的时候，它将不会如预期一样自动运行。

均衡器可以显式地使用 `balancer` 命令进行启动，同时也可以使用 API 中的 `balancer()` 方法。以上介绍的自动均衡过程会隐式地调用这个方法。HBase 会判断如果需要负载均衡就返回 `true`，返回 `false` 则意味着不能运行均衡器，原因要么是开关被关闭或者没有工作需要做（已经达到均衡了），也有可能其他工作阻止了其运行。例如，一个 `region` 处于事务列表中（参见 6.5.1 节的“主页”部分）：如果一个 `region` 正处于状态转换时，均衡操作将会被跳过。

除了依赖均衡器完成自己的工作，用户还可以使用 `move` 命令和 API 方法显式地将 `region` 移动到另一个服务器上。当用户想控制某张表特定 `region` 的确切位置时，这种方法是很有用的。详细内容请参见 11.4.2 节。

11.6 合并 region

当用户向相应的表中插入数据时，`region` 自动拆分的情况是很常见的。当然在某些特殊情况下，用户有可能需要合并 `region`，例如，用户删除大量数据并且想减少每个服务器管理的 `region` 数目。

HBase 集成了一个工具能够让用户在集群没有工作时合并两个相邻的 `region`。可以使用命令行工具来获得使用说明：

```
./bin/hbase org.apache.hadoop.hbase.util.Merge
Usage: bin/hbase merge<table-name><region-1><region-2>
```

以下例子中，有一张表的 `region` 超过一个，接下来将合并它们：

```
./bin/hbase shell

hbase(main):001:0>create 'testtable','colfam1',\
{SPLITS => ['row-10','row-20','row-30','row-40','row-50']}
0 row(s) in 0.2640 seconds

hbase(main):002:0>for i in '0'..'9' do for j in '0'..'9' do \
put 'testtable','row-#{i}#{j}','colfam1:#{j}','#{j}' end end
0 row(s) in 1.0450 seconds

hbase(main):003:0>flush 'testtable'
0 row(s) in 0.2000 seconds

hbase(main):004:0>scan '.META.',{ COLUMNS => ['info:regioninfo']}
```



```

ROW                                COLUMN+CELL
testtable,,1309614509037.612d1e0112 column=info:regioninfo,timestamp= 130...
406e6c2bb482eeaec57322.          STARTKEY => '',ENDKEY => 'row-10'
testtable,row-10,1309614509040.2fba column=info:regioninfo,timestamp=130...
fcc9bc6afac94c465ce5dcabc5d1.    STARTKEY => 'row-10',ENDKEY => 'row-20'
testtable,row-20,1309614509041.e7c1 column=info:regioninfo,timestamp=130...
6267eb30e147e5d988c63d40f982.    STARTKEY => 'row-20',ENDKEY => 'row-30'
testtable,row-30,1309614509041.a9cd column=info:regioninfo,timestamp=130...
e1cbc7d1a21blaca2ac7fda30ad8.     STARTKEY => 'row-30',ENDKEY => 'row-40'
testtable,row-40,1309614509041.d458 column=info:regioninfo,timestamp=130...
236feae097efcf33477e7acc51d4.    STARTKEY => 'row-40',ENDKEY => 'row-50'
testtable,row-50,1309614509041.74a5 column=info:regioninfo,timestamp= 130...
7dc7e3e9602d9229b15d4c0357d1.    STARTKEY => 'row-50',ENDKEY => ''
6 row(s) in 0.0440 seconds

hbase(main):005:0>exit

$ ./bin/stop-hbase.sh

$ ./bin/hbase org.apache.hadoop.hbase.util.Merge testtable \
testtable,row-20,1309614509041.e7c16267eb30e147e5d988c63d40f982. \
testtable,row-30,1309614509041.a9cde1cbc7d1a21blaca2ac7fda30ad8.

```

这个例子创建了一张有 5 个拆分点的表，并产生了 6 个 region。然后其插入一些记录并刷写数据，以保证有存储文件来进行之后的合并操作。用户通过扫描方法来获得 region 的名字，用户也可以使用 master 的 Web 界面，在 *User Tables* 部分中点击表名获得 region 列表。



注意，Shell 如何包装列值。region 名被拆分成了两行，需要分别复制粘贴。网页界面在这方面更加好用一些，因为它在独立的一行一列中显示了名字。

在上面的例子中，列值被缩简到开始和结束键的位置。用户可以发现 *create* 命令使用拆分键创建了 region。这个例子接下来需要退出 Shell 并停止 HBase 集群。注意 HDFS 要保持运行，因为它需要在每个 region 中读取存储文件并将它们合并成一个新的存储文件。

11.7 客户端 API：最佳实践

当用户通过客户端使用接口读写数据的时候，有许多优化方法可供选择来提升性能。这里是最佳实践选项的列表：

禁止自动刷写

当有大量的写入操作时，使用 `setAutoFlush(false)` 方法，确认 `HTable` 自动刷写的特性已经被关闭。否则 `Put` 实例将会被逐个传送到 **region** 服务器。通过 `HTable.add(Put)` 和 `HTable.add(Put)` 添加的 `put` 实例都会添加到一个相同的写入缓存中，如果用户禁用了自动刷写，这些操作直到写缓冲区被填满时才会被送出。如果要显式地刷写数据，用户可以调用 `flushCommits()` 方法。调用 `HTable` 实例的 `close` 方法也会隐式地调用 `flushCommits()`。

使用扫描缓存

如果 `HBase` 被用作一个 **MapReduce** 作业的输入源，请最好将作为 **MapReduce** 作业输入扫描器实例的缓存用 `setCaching()` 方法设置为比默认值 1 大得多的值。使用默认的值意味着 `map` 任务会在处理每条记录时都请求 **region** 服务器。例如，将这个值设置为 500，则一次可以传送 500 行数据到客户端进行处理。这里用户需要权衡传输数据的开销和内存的开销，因为缓存更大之后，无论是客户端还是服务器端都将消耗更多内存缓存数据，所以大的缓存并不一定最好。

限定扫描范围

当 `Scan` 被用来处理大量行时（特别是被用作 **MapReduce** 输入源时），注意哪些属性被选中了。如果 `Scan.addFamily()` 被调用了，那么特定列族中的所有列都将会被返回到客户端。如果只处理少数列，则应当只有这些列被添加到 `Scan` 的输入中，因为选择了过多的列将导致在大数据集上极大的效率损失，这可不是一件小事。

关闭 ResultScanner

这不会带来性能提升，但是会避免可能的性能问题。如果用户忘记关闭由 `HTable.getScanner()` 返回的 `ResultScanner` 实例，则有可能对服务器端造成影响。

一定要在 `try/catch` 的 `finally` 块中关闭 `ResultScanner`，例如：

```
Scan scan = new Scan();
// configure scan instance
ResultScanner scanner = table.getScanner(scan);
try {
    for(Result result : scanner){
        // process result...
    } finally {
        scanner.close(); // always close the scanner!
    }
    table.close();
```

块缓存用法

`Scan` 实例能够通过 `sScan` 实例能够通过 `setCacheBlocks()` 方法 `etCacheBlocks()` 方法来设

置使用 **region** 服务器中的块缓存。如果 **MapReduce** 作业中使用扫描，这个方法应当被设置成 **false**。对于那些频繁访问的行，建议使用块缓存。

优化获取行键的方式

当执行一个表的扫描以获取需要的行键时（没有列族、列名、列值和时间戳），在 **Scan** 中用 **setFilter()** 方法添加一个带 **MUST_PASS_ALL** 操作符的 **FilterList**。**FilterList** 中包含 **FirstKeyFilter** 和 **KeyOnlyFilter** 两个过滤器，在 4.1.3 节中提到过。使用以上组合的过滤器将会把发现的第一个 **KeyValue** 行键（也就是第一列的行键）返回给客户端，这将会最大程度地减少网络传输。

关闭 Put 上的 WAL

一个经常讨论的提高写吞吐量的方式是使用 **Put** 的 **writeToWAL(false)** 来关闭 **WAL**。这样服务器端就不会把这个 **Put** 写到 **WAL** 中，而只把它写到 **memstore** 里，不过一旦 **region** 服务器出现故障就会丢失数据。如果用户使用了 **writeToWAL(false)**，请特别小心。用户可能会发现把数据在集群间分布均匀后，关闭日志所带来的性能提升并不明显。

总而言之，最好是在写入数据时使用 **WAL**，并且如果特别关心吞吐量的话，就用批量导入（**bulk load**）技术，这个技术将在 12.2.3 节中进行介绍。

11.8 配置

用户可以使用许多的配置项来优化集群。2.6 节列举出了运行集群时用户需要更改和设置的内容。这里也有一些高级选项，用户需要根据应用的需求来调整它们。下面是一些常见的配置项和应当如何调整它们的说明。



主要的配置都放在了配置文件 **hbase-site.xml** 中。编辑这个文件，然后将文件复制到集群的所有服务器上，并重启所有服务器来使更新的配置生效。

减少 ZooKeeper 超时的发生

默认在 **region** 服务器和 **ZooKeeper** 集群之间的超时时间是 3 分钟，并使用 **zookeeper.session.timeout** 属性来设置。这意味着，如果服务器崩溃，**master** 将在 3 分钟后发现这个崩溃现象，并开始恢复数据。用户可以将这个时间设为 1 分钟或者更少，这样 **master** 就能够很快地发现这一故障。

在改变值之前，确认用户服务器上 **JVM** 的垃圾回收机制是受控的，因为长时间垃圾回收且回收运行时间超过 **ZooKeeper** 会话的超时上限可能导致 **region** 服务器被误认为崩溃。不过如果这是用户所需要的那就不必介意：用户可能希望在 **region** 服务器

进行长时间垃圾回收时启动数据恢复，这样数据可能会更快变为可用。

默认值特别长的原因是为了避免在大数据导入的情况下产生问题：写入大量数据时会对 `region` 服务器产生很大的压力，这样更有可能导致出现垃圾回收暂停的问题。参见 12.5.3 节中“稳定性问题”部分介绍的方法来检测这种停顿。

增加处理线程

`hbase.regionserver.handler.count` 属性定义了响应外部用户访问数据表请求的线程数。默认值 10 有些偏小，这是为了防止用户在客户端高并发使用较大写缓冲区的情况下使服务器端过载。将这个值设得小是为了优化单次请求涉及的数据量达到 MB 级别（如较大的写入和使用大缓存的扫描）场景，而当单次请求开销较小时（如 `get`、较小的 `put`、`increment` 和 `delete` 等操作）可以将工作线程数设得高一些。

如果客户端的请求开销较小时，用户将该属性设置为最大的客户端数目会比较安全。典型的例子就是，当一个集群服务于一个网站时，写请求一般不会使用缓存，同时大多数的请求都是读取数据。

将这个值设置得高也有可能产生问题，因为并发的写请求涉及到的数据累加起来之后很可能会对一个 `region` 服务器的内存造成巨大压力，这甚至会导致服务器端抛出 `OutOfMemoryError` 异常。`region` 服务器运行在可用内存过低的情况下时，其将会使 JVM 的垃圾回收器运行地更加频繁，同时随之发生的停顿也会更加明显（原因是内存都被写请求占用，无论垃圾回收器怎么尝试，它们都不能被回收）。一段时间后，集群的吞吐量就会受到影响，因为命中这个服务器的请求都会变慢，这样会使其内存紧张的情况更加严重。

增加堆大小

HBase 默认使用一组合理并且保守的配置，该配置可以满足大多数不同机型的测试需求。如果用户使用更好的服务器，则其可以给 HBase 分配 8 GB 或更大的内存空间。用户可以在 `hbase-env.sh` 文件中调整 `HBASE_HEAPSIZE` 的设置。

注意，使用 `HBASE_REGIONSERVER_OPTS` 而非全局的 `HBASE_HEAPSIZE`；在这种方式下，`master` 将会以默认 1 GB 的堆运行，`region` 服务器则会按用户单独指定的堆空间运行。

这个配置项在 `hbase-env.sh` 文件中，`hbase-site.xml` 文件则用于其他大多数属性。

启用数据压缩

用户应当为存储文件启用压缩，尤其推荐使用 Snappy 或者 LZO 压缩。这两个近乎平滑的压缩算法在大多数应用中都能够使应用性能得到提升。参见 11.3 节以获取所有有关压缩算法的信息。

增加 region 大小

更大的 region 可以减少集群总的 region 数目。一般来说，管理较少的 region 可以让集群的运行更平稳。一个 region 变热点后，用户可以手动拆分大的 region 并将负载分散到集群中，11.4 节有更详细介绍。

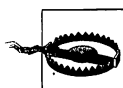
在默认情况下，region 的大小是 256 MB。用户可以配置 1 GB 或者更大的 region。注意，该参数的大小要仔细评估，大的 region 也意味着在高负载的情况下合并的停顿时间更长。

用户可以在 *hbase-site.xml* 文件中调整 `hbase.hregion.max.filesize` 属性的值。

调整块缓存大小

控制堆中块缓存大小的属性是一个用浮点数类型的百分比值，默认值是 20%（即 0.2）。改变 `perf.hfile.block.cache.size` 属性可以改变这个百分比值。仔细观察块缓存的使用情况（参见 10.2.3 节），看看是否存在许多块被换出的情况。如果存在这种情况，用户就可以考虑增加块缓存大小来容纳更多的块。

用户负载大多数为读请求是另一个增加块缓存大小的原因。此时 HBase 将更加需要块缓存，增加块缓存的大小可以帮助用户缓存更多的数据。



块缓存与 memstore 的上限不能超过 100%。用户需要为其他操作保留空间，不然服务器端可能面临内存紧张的问题。默认它们占用的堆空间量是 60%，这是一个合理的值。只有当用户确认有必要并且不会造成副作用时，用户才能为其设定超过这个上限的值。

调整 memstore 限制

内存存储占用的堆大小用 `hbase.regionserver.global.memstore.upperLimit` 属性来配置，默认值为 40%（设置为 0.4）。此外，`hbase.regionserver.global.memstore.lowerLimit` 属性（设置为 35% 或者 0.35）用于控制当服务器清空 memstore 之后剩余的大小。将上限和下限设置得接近一些以避免过度刷写。

当用户主要在处理读请求时，其可以考虑同时减少 memstore 的上下限来增加块缓存的空间。另一方面，当用户在处理许多写请求时应该检查日志文件（或者使用 10.2.3 节中提到的 region 服务器监控），如果刷写的数据量都很小，如 5 MB，用户就有必要通过增加内存存储的限制来降低过度 I/O 操作。

增加阻塞时存储文件数目

这个值是通过 `hbase.hstore.blockingStoreFiles` 属性来设置的，它决定了当存储文件的数目达到阈值时，更新操作（put、delete 等）将会被阻塞，并以此来给合并操作留出时间来减少存储文件的数目。当应用经常遇到大负载的突发写入请求时，用

户需要稍稍增加这个值来应对这种情况，其默认值是 7。

用户可以使用监控来观察 **region** 服务器管理的存储文件数目，如果文件数一直很高，那么用户可能不适合提高这个配置项的大小，因为这样做只会延迟由于服务器负载过重而产生的无法避免的问题。

增加阻塞倍率

属性 `hbase.hregion.memstore.block.multiplier` 的默认值为 2，它是一个用于阻塞来自客户端更新数据请求的安全阀值，当 `memstore` 达到属性 *multiplier* 乘以 *flush* 的大小限制时会阻止进一步的更新。

当有足够的存储空间时，用户可以增加这个值来更加平滑地处理写入突发流量：与阻塞更新操作来等待 `memstore` 完成刷写相比，用户可以临时接收更多的数据。

减少最大日志文件限制

设置 `hbase.regionserver.maxlogs` 属性使得用户能够控制基于磁盘的 **WAL** 文件数目，进而控制刷写频率。该参数的默认值是 32，对于写压力比较大的应用来说这个值有点高。降低这个值会强迫服务器更频繁地将数据刷写到磁盘上，这样已经刷写到磁盘上的数据所对应的日志就可以被丢弃了。

11.9 负载测试

在安装好集群之后，建议运行负载测试来验证集群功能。这些测试能够在用户修改集群设置或者表结构后提供对比的基准线。在用户的集群上执行一个负载测试可以告诉用户集群所能达到的性能，但是这并不能够取代使用实际用例负载的测试。

11.9.1 性能评价

HBase 有自己的性能评价工具，名为 **PE** (**Performance Evaluation**)。用户可以在不加命令行参数时使用它以获得使用帮助：

```
$. /bin/hbase org.apache.hadoop.hbase.PerformanceEvaluation
Usage: Java org.apache.hadoop.hbase.PerformanceEvaluation \
    [--miniCluster] [--nomapred] [--rows=ROWS] <command><nclients>

Options:
miniCluster    Run the test on an HBaseMiniCluster
nomapred       Run multiple clients using threads(rather than use map reduce)
rows           Rows each client runs. Default: One million
flushCommits   Used to determine if the test should flush the table.
                Default: false
writeToWAL     Set writeToWAL on puts. Default: True
```

Command:

```
filterScan      Run scan test using a filter to find a specific row based
                  on it's value(make sure to use --rows=20)
randomRead      Run random read test
randomSeekScan  Run random seek and scan 100 test
randomWrite     Run random write test
scan            Run scan test(read every row)
scanRange10     Run random seek scan with both start and stop row(max 10 rows)
scanRange100    Run random seek scan with both start and stop row(max 100 rows)
scanRange1000   Run random seek scan with both start and stop row(max 1000 rows)
scanRange10000  Run random seek scan with both start and stop row(max 10000 rows)
sequentialRead  Run sequential read test
sequentialWrite Run sequential write test
```

Args:

```
nclients      Integer. Required. Total number of clients(and HRegionServers)
                  running: 1 <= value <= 500
```

Examples:

To run a single evaluation client:

```
$ bin/hbase org.apache.hadoop.hbase.PerformanceEvaluation sequentialWrite 1
```

PE 默认是作为 MapReduce 作业来执行的，除非用户指定一个客户端或使用--nomapred 参数。用户可以通过上述信息来查看默认值，默认值都是较为合理的起始值，以下是运行测试的例子：

```
./bin/hbase org.apache.hadoop.hbase.PerformanceEvaluation sequentialWrite 1
11/07/03 13:18:34 INFO hbase.PerformanceEvaluation: Start class \
  org.apache.hadoop.hbase.PerformanceEvaluation$SequentialWriteTest at \
  offset 0 for 1048576 rows
...
11/07/03 13:18:41 INFO hbase.PerformanceEvaluation: 0/104857/1048576
...
11/07/03 13:18:45 INFO hbase.PerformanceEvaluation: 0/209714/1048576
...
11/07/03 13:20:03 INFO hbase.PerformanceEvaluation: 0/1048570/1048576
11/07/03 13:20:03 INFO hbase.PerformanceEvaluation: Finished class \
  org.apache.hadoop.hbase.PerformanceEvaluation$SequentialWriteTest \
  in 89062ms at offset 0 for 1048576 rows
```

命令行创建一个单独客户端，并且执行连续的写入测试。命令行将一直显示完成的进度直到打印最后的结果。当用户确定客户端服务器负载并不太大时，可以增加一定数量的客户端（也就是说线程或者 MapReduce 任务）。

这里不需要指定表名，不需要设定列族，PE 代码会自动生成表和对应的模式：一张拥有一个叫做 info 的列族的名字为 TestTable 的表。



用户需要在做读测试之前运行过写测试，写测试将会生成表并插入数据，该数据可供后面的读测试使用。

使用随机或顺序的读写测试可以帮助用户模拟特定的负载，但用户不能把它们混合在一起测试，也就是说用户只能把它们分开测试。

11.9.2 YCSB

Yahoo 的云服务基准测试系统（Yahoo! Cloud Serving Benchmark^①，YCSB）是一套用于比较不同系统的工具。虽然它主要用于比较不同系统之间的性能，但它同样也适用于对 HBase 集群进行超负荷测试。

安装

YCSB 只能够通过网络库获取，并需要自己编译二进制文件。首先要做的事情是复制网络库的内容：

```
$git clone http://github.com/brianfrankcooper/YCSB.git
Initialized empty Git repository in /private/tmp/YCSB/.git/
...
Resolving deltas: 100%(475/475),done.
```

这将会在当前目录下创建一个本地 *YCSB* 目录。接下来的步骤就是进入新创建的目录，复制 HBase 所需要的库并编译可执行文件：

```
$cd YCSB/
$cp $HBASE_HOME/hbase*.jar db/hbase/lib/
$cp $HBASE_HOME/lib/*.jar db/hbase/lib/
$sant
Buildfile: /private/tmp/YCSB/build.xml
...
makejar:
[jar] Building jar: /private/tmp/YCSB/build/ycsb.jar

BUILD SUCCESSFUL
Total time: 1 second

$sant dbcompile-hbase
...
BUILD SUCCESSFUL
Total time: 1 second
```

这个过程只有几秒钟的时间，然后会在 *build* 文件夹下留下一个可执行的 jar 文件。

在用户可以运行 YCSB 之前，需要创建所需的测试表，名字为 *usertable*。虽然表名在

① 参见 GitHub 库中的工程（<http://github.com/brianfrankcooper/YCSB>）来查看细节信息。

代码中是写死的，但是用户可以自由命名列族，例如：

```
./bin/hbase shell

hbase(main):001:0>create 'usertable','family'
0 row(s) in 0.3420 seconds
```

如果用户无参数运行 YCSB，YCSB 会为用户提供使用帮助：

```
$Java -cp build/ycsb.jar:db/hbase/lib/* com.yahoo.ycsb.Client
Usage: Java com.yahoo.ycsb.Client [options]
Options:
  -threads n: execute using n threads(default: 1)- can also be specified as the
              "threadcount" property using -p
  -target n: attempt to do n operations per second(default: unlimited)- can also
              be specified as the "target" property using -p
  -load: run the loading phase of the workload
  -t: run the transactions phase of the workload(default)
  -db dbname: specify the name of the DB to use(default: com.yahoo.ycsb. BasicDB)-
              can also be specified as the "db" property using -p
  -P propertyfile: load properties from the given file. Multiple files can
                  be specified,and will be processed in the order specified
  -p name=value: specify a property to be passed to the DB and workloads;
                multiple properties can be specified,and override any
                values in the propertyfile
  -s: show status during run(default: no status)
  -l label: use label for status(e.g. to label one experiment out of a whole
            batch)
```

Required properties:

```
  workload: the name of the workload class to use
            (e.g. com.yahoo.ycsb.workloads.CoreWorkload)
```

To run the transaction phase from multiple servers,start a separate client on each. To run the load phase from multiple servers,start a separate client on each;additionally,use the "insertcount" and "insertstart" properties to divide up the records to be inserted

测试运行中 HBase 集群的第一步是加载一定数量的记录，这些记录接下来会用于修改相同的行，或者往表中加入新的记录：

```
$Java -cp $HBASE_HOME/conf:build/ycsb.jar:db/hbase/lib/* \
com.yahoo.ycsb.Client -load -db com.yahoo.ycsb.db.HBaseClient \
-P workloads/workloada -p columnfamily=family -p recordcount=100000000 \
-s > ycsb-load.log
```

这个命令将会运行一段时间并创建记录，记录的结构由给定的导入文件控制，在这里是由 *workloada* 控制，它包含以下设置：

```
$cat workloads/workloada
# Yahoo! Cloud System Benchmark
```

```

# Workload A: Update heavy workload
# Application example: Session store recording recent actions
#
# Read/update ratio: 50/50
# Default data size: 1 KB records(10 fields,100 bytes each,plus key)
# Request distribution: zipfian

recordcount=1000
operationcount=1000
workload=com.yahoo.ycsb.workloads.CoreWorkload

readallfields=true

readproportion=0.5
updateproportion=0.5
scanproportion=0
insertproportion=0

requestdistribution=zipfian

```

用户可以参考 YCSB 的联机文档来查阅关于如何修改记录结构的详细说明，以外，用户也可以按照自己的意愿来调整设置。文件中指定了使用 load 语句时产生的数据大小以及列的数量。这个工具的输出将重定向到日志文件，内容如下：

```

YCSB Client 0.1
Command line: -load -db com.yahoo.ycsb.db.HBaseClient -P workloads/ workloada \
-p columnfamily=family -p recordcount=100000000 -s
[OVERALL],RunTime(ms),915.0
[OVERALL],Throughput(ops/sec),1092.896174863388
[INSERT],Operations,1000
[INSERT],AverageLatency(ms),0.457
[INSERT],MinLatency(ms),0
[INSERT],MaxLatency(ms),314
[INSERT],95thPercentileLatency(ms),1
[INSERT],99thPercentileLatency(ms),1
[INSERT],Return=0,1000
[INSERT],0,856
[INSERT],1,143
[INSERT],2,0
[INSERT],3,0
[INSERT],4,0
...

```

上述初始记录的写性能数据是十分有价值的。默认的记录数是 1000 行，我们增加了记录数来反映更真实的负载情况。用户可以在命令行上修改任何一个设置选项。如果用户经常使用同一种结构，那么用户可以通过命令行使用 -P 参数来引用它。

YCSB 性能测试的第二步是在准备好的表上执行负载测试作业，例如：

```
$Java -cp $HBASE_HOME:build/ycsb.jar:db/hbase/lib/* \
com.yahoo.ycsb.Client -t -db com.yahoo.ycsb.db.HBaseClient \
-P workloads/workloada -p columnfamily=family -p operationcount=1000000 -s \
-threads 10 > ycsb-test.log
```

与上述的载入步骤相同，用户需要根据自己的需求改变一些参数：增加（或者用你自己修改过的结构文件）测试操作的次数，并将并发的线程数设置为一个合理的值。如果用户使用了太多的线程，这将会使测试机（即运行 YCSB 的服务器）负载过重。在这样的情况下，最好的方法就是在不同的机器上同时运行相同的测试。

输出也被重定向到了日志文件，这样用户就能够在测试完成之后进行评估了。输出将包含以下这些行：

```
]$ cat transactions.dat
YCSB Client 0.1
Command line: -t -db com.yahoo.ycsb.db.HBaseClient -P workloads/workloada -p \
columnfamily=family -p operationcount=1000 -s -threads 10
[OVERALL],RunTime(ms),575.0
[OVERALL],Throughput(ops/sec),1739.1304347826087
[UPDATE],Operations,507
[UPDATE],AverageLatency(ms),2.546351084812623
[UPDATE],MinLatency(ms),0
[UPDATE],MaxLatency(ms),414
[UPDATE],95thPercentileLatency(ms),1
[UPDATE],99thPercentileLatency(ms),1
[UPDATE],Return=0,507
[UPDATE],0,455
[UPDATE],1,49
[UPDATE],2,0
[UPDATE],3,0
...
[UPDATE],997,0
[UPDATE],998,0
[UPDATE],999,0
[UPDATE],>1000,0
[READ],Operations,493
[READ],AverageLatency(ms),7.711967545638945
[READ],MinLatency(ms),0
[READ],MaxLatency(ms),417
[READ],95thPercentileLatency(ms),3
[READ],99thPercentileLatency(ms),416
[READ],Return=0,493
[READ],0,1
[READ],1,165
[READ],2,257
[READ],3,48
[READ],4,11
[READ],5,4
```

```
[READ],6,0
...
[READ],998,0
[READ],999,0
[READ],>1000,0
```

注意，YCSB 很难模拟应用负载，但使用它的测试仍旧是有意义的，因为它可以测试集群在不同负载下的表现。使用默认提供的结构文件或者自己创建的结构文件来模拟读写或读写混合的操作。

当用户运行批量作业时，例如，MapReduce 处理或扫描一个数据集合或整张表，同时也可以运行 YCSB 来测试各种操作之间的影响。



对写操作来说，与 HBase 提供的性能评估工具相比，YCSB 更加适合。因为它提供了更多的选项，并且能够将读写负载混合在一起。

一旦集群开始运转，用户可能会想要改变集群的大小或添加一些额外的机制来应对出现的故障，这些可能需要在集群正在提供服务时进行。有时用户需要将数据备份或迁移到不同的集群。接下来我们会介绍如何在对集群造成最小影响的情况下完成这些工作。

12.1 运维任务

本节介绍了操作集群时所必需的一些任务，包括增加和移除节点。

12.1.1 减少节点

用户可以在指定节点的 HBase 目录下使用以下命令停止集群中的一个 region 服务器。

```
$ ./bin/hbase-daemon.sh stop regionserver
```

region 服务器会先将它所有的 region 关闭，然后再把自己的进程停止。region 服务器在 ZooKeeper 中对应的临时节点（ephemeral node）将会过期。master 会注意到 region 服务器停止服务并将其按崩溃的服务器处理：master 会将这台服务器上的 region 重新分配到其他机器上。

让节点下线前先禁用负载均衡

如果在关闭节点时负载均衡还在运行，则在负载均衡和 master 恢复刚才下线的 region 服务器之间可能产生竞争。要避免这种情况发生，请先禁用负载均衡：使用 Shell 工具进行如下操作。

```
hbase(main):001:0> balance_switch false
true
0 row(s) in 0.3590 seconds
```

以上操作会关闭 balancer。如需启用，请输入如下命令：

```
hbase(main):002:0> balance_switch true
false
0 row(s) in 0.3590 seconds
```

以上停止 region 服务器的方法的坏处是，region 会下线一段时间，时间的长度由 ZooKeeper 超时时间决定。region 是按顺序关闭的：如果服务器上有很多 region，第一个被关闭的 region 要等所有 region 都关闭，且 master 注意到 region 服务器的 znode 被删除之后才能上线。

HBase 0.90.2 引入了一种可以让 region 服务器逐渐较少其负载并停止服务的方法。这项功能由 *graceful_stop.sh* 脚本完成。不带参数使用时会显示以下使用说明：

```
$ ./bin/graceful_stop.sh
Usage: graceful_stop.sh [--config &conf-dir>] [--restart] [--reload] \
        [--thrift] [--rest] &hostname>

thrift      If we should stop/start thrift before/after the hbase stop/ start
rest        If we should stop/start rest before/after the hbase stop/start
restart      If we should restart after graceful stop
reload      Move offloaded regions back on to the stopped server
debug       Move offloaded regions back on to the stopped server
hostname    Hostname of server we are to stop
```

如果用户要下线一个 region 服务器，可以输入以下内容：

```
$ ./bin/graceful_stop.sh HOSTNAME
```

HOSTNAME 是用户要卸载的 region 服务器。



传入 *graceful_stop.sh* 的 HOSTNAME 参数必须与 HBase 用于辨别 region 服务器的名字一致。用户可以在 master UI 列表中查看 HBase 是如何引用每个服务器的。region 服务器标识通常情况下会是 hostname，也有可能是一个全称域名（FQDN），例如，hostname.fooobar.com。用户需要把 HBase 使用的服务器名作为参数传入 *graceful_stop.sh* 脚本。

如果用户传入 IP 地址，脚本不会智能地将其转换为 hostname 或全称域名，同时会在检查阶段出现错误：这种平滑地卸载 region 服务器的方法不会继续执行。

graceful_stop.sh 脚本会把 region 从对应的服务器上一个个移出来以减少扰动。它会在移动到下一个 region 前先检查新位置上的 region 是否已经部署好，直到对应的要关闭的服务器上没有了 region。

此时脚本会让对应的服务器关闭。master 会察觉到服务器停止了服务，不过此时服务器

上的 region 已经都被转移并部署好了。同时，由于服务器关闭时没有 region，所以也不会有 WAL 切分的相关操作。

12.1.2 滚动重启

用户也可以使用 *graceful_stop.sh* 脚本来重启服务器，并将之前属于它的 region 移回原位（用户可能会选择后者以保持数据的局部性）。最简单的滚动重启可以使用以下脚本来完成：

```
$ for i in `cat conf/regionserver|sort`;do ./bin/graceful_stop.sh \
--restart --reload --debug $i;done &> /tmp/log.txt &
```

可以使用 *tail* 命令查看 */tmp/log.txt* 中脚本运行的进度。以上操作只对 region 服务器有效，同时请确保在执行上述操作之前已禁用负载均衡。

用户需要单独对 master 进行升级，同时建议最好预先完成这项工作。以下是实现滚动重启的步骤。

1. 解压用户要升级的发行版，将配置调整好并分发到整个集群。如果用户使用的是 0.90.2 版，还需要打上 HBASE-3744 和 HBASE-3756 补丁。
2. 运行 *hbck* 以确认集群数据的一致性（主要验证 meta 表），如果有不一致的情况修复一下。

```
$ ./bin/hbase hbck
```

3. 重启 master。

```
$ ./bin/hbase-daemon.sh stop master;./bin/hbase-daemon.sh start master
```

4. 关闭 region 均衡器。

```
$ echo "balance_switch false" | ./bin/hbase shell
```

5. 在每个 region 服务器上运行 *graceful_stop.sh* 脚本。例如：

```
$ for i in `cat conf/regionserver|sort`;do ./bin/graceful_stop.sh \
--restart --reload --debug $i;done &> /tmp/log.txt &
```

如果用户运行了 Thrift 或 REST 服务，请在运行脚本时传入 *--thrift* 或 *--rest* 选项，详情请参考之前介绍过的使用说明（即不加任何参数运行脚本时得到的使用说明）。

6. 再次重启 master。这样会清除其保存的崩溃的服务器列表，并同时启动负载均衡器。
7. 运行 *hbck* 以确认集群一致性。

12.1.3 新增服务器

HBase 的一种主要特性是内置的高扩展性。当用户集群负载加重时，用户需要通过添加服务器来满足新需求。添加新服务器是一种十分简单的操作，并可以在任何模式下完成该操作，分布式模式请参见 2.5.2 节。

1. 伪分布式模式

即使 HBase 的所有后台程序都在不同的进程中运行，在本地模式下运行 HBase 与扩展 HBase 看起来仍然是矛盾的。伪分布式模式是用户能使用的最接近于真实状态的运行情况的模拟。在开发或原型设计时，如果能在一台服务器上模拟整个集群的运行状态也是非常有用的。

因为所有进程都要分享本地资源，所以并非启动的服务进程越多性能越好。实际上，伪分布式模式适合少量数据的测试。此外，它几乎可以让用户测试 HBase 提供的结构特性。

例如，用户可以在 master 故障恢复场景中或 region 从一个服务器移动到另一个服务器的情况下进行实验。当然，这并不能完全替代在真实的集群上使用模拟的生产环境预期的负载进行测试。但是，它可以帮助用户使学习使用 HBase 提供的管理功能，如 HBase Shell。

用户也可以使用第 5 章中介绍的 HBase 的管理 API。用户可以使用它们来开发管理表结构或者调整集群负载的工具。许多生产环境下的应用都会用到这些功能，所以能在本地先对这些功能进行测试是非常有益的。



用户必须先按伪分布式模式配置自己的安装，同时作为备份 master 必须使用以下命令启动。以下脚本会启动一个新的 master 进程，但不会启动整个 HBase 本地集群。

添加本地备份 master。用户可以使用 `bin` 目录下的 `local-master-backup.sh` 脚本来启动本地设备的 master 进程如下所示。

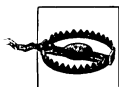
```
$ ./bin/local-master-backup.sh start 1
```

命令最后的数字指定了 master 的 RPC 端口和 Web UI 端口与默认配置值的偏移量，两端口的默认值分别是 60000 和 60010，偏移量会被加到相应的默认值上。在上面的命令中，一个新的 master 进程将会启动，并与通常情况一样读取与原 master 相同的配置，不过监听的端口号分别为 60001 和 60011。

换句话说，这些参数是必须填写的且不代表用户要启动几个备份 master，而只是指定了启动的 master 要使用的端口。启动多个 master 进程请按下面的方式输入：


```
$/bin/local-master-backup.sh start 1 3 5
```

这样会启动 3 个用于备份的 master，分别使用 60001、60003 和 60005 作为 RPC 监听端口，同时使用 60011、60013 和 60015 作为 Web UI 监听端口。



用户需要确定要使用的端口与其他进程没有冲突。例如，不要使用 30 作为偏移量，因为这会使 master 使用 60030 为 RPC 端口，而这个端口已被 region 服务器用作 UI 端口。

启动脚本也会在进程使用的日志文件名中添加这个偏移量，这样就可以将它的日志与其他本地进程使用的日志文件区分开。例如，偏移量为 1 时，其日志文件名为：

```
logs/hbase-${USER}-1-master-${HOSTNAME}.log
```

注意，文件名中多出的 1。如果使用 10 作为偏移量，则文件名中添加的应为 10。

用户想要停止备份 master 时，可以把 start 命令替换为 stop 命令，如下所示：

```
$/bin/local-master-backup.sh stop 1
```

用户需要指定需要停止的备份 master 端口的偏移量，并且可以一次停止一个或多个备份 master：用户指定的端口偏移量用于停止对应的 master。

添加本地 **region** 服务器。与之相似，用户可以启动附加的本地 region 服务器。用户所需使用的脚本为 *local-regionserver.sh*，同时参数类型与 *local-master-backup.sh* 脚本相同：用户可以指定端口偏移量列表用于启动和停止对应的服务进程。

不同的是，region 服务器默认使用 60200 作为 RPC 端口，和 60300 作为 Web UI 端口。例如：

```
$/bin/local-regionserver.sh start 1
```

这条命令会启动一个额外的 region 服务器，并指定 60201 端口为 RPC 监听端口，以及 60301 端口为 web UI 监听端口。同时为日志文件名添加这个偏移量，如下所示：

```
logs/hbase-${USER}-1-regionserver-${HOSTNAME}.log
```

与此同时，用户需要确定这些端口没有被其他进程占用，否则用户可能会遇到 `java.net.BindException: Address already in use` 异常。

启动多个 region 服务器可以通过添加多个偏移量来完成：

```
$/bin/local-regionserver.sh start 1 2 3
```



用户不必使用 1 作为第一个端口的偏移量。因这个偏移量只是被简单地加到了默认端口号上，所以用户可以随意指定自己想要的偏移量。

通过把 `start` 换成 `stop` 来停止一个额外的 `region` 服务器可以使用如下命令：

```
$ ./bin/local-regionservers.sh stop 1
```

这条命令会停止使用以 1 为偏移量的 `region` 服务器。它的端口为 60201 和 60301。如果用户之前用 1 作为偏移量启动过 `region` 服务器，那么它会被关闭。

2. 完全分布式集群

在运行一个 HBase 集群时，典型的操作就是向现有集群添加新的服务器，且通常添加 `region` 服务器，因为它们承担了集群的大部分负载。对 `master` 用户来说，其可以启动一个可选的备份实例。

添加一个备份 master。 用户通过添加备份 `master` 可以避免因 `master` 出现故障而造成 HBase 单点失效，这是一种典型做法，备份 `master` 节点通常在不同的服务器上启动，所以在最坏的情况下，当前 `master` 失效后，系统可以切换到备份 `master` 上。

`master` 进程使用 ZooKeeper 来协商哪一个是当前活动的进程：ZooKeeper 中有一个所有 `master` 进程都会竞争的专用 `znode`，第一个竞争到的 `master` 会创建这个 `znode`，也意味着它在竞争中胜出。这个过程发生在集群启动时，竞争获胜的 `master` 会成为当前提供服务的 `master`。其他的 `master` 进程只是轮询检查这个 `znode`，当它消失时再次竞争。

`/hbase/master znode` 是临时 `znode`，同样类型的 `znode` 也被 `region` 服务器用于报告它们的存在状态。当 `master` 创建 `znode` 失败时，ZooKeeper 会注意到会话过期并删除这个 `znode`，以及触发新 `master` 的选举。

在多台机器上启动服务时，要求其 HBase 配置应与集群其他服务器一致（参见 2.6 节介绍的详细信息）。通常 `master` 服务器的配置与集群的其他机器相同，一旦用户确认配置没有问题，用户可以使用以下命令在用户指定的服务器上启动一个备份 `master`：

```
$ ./bin/hbase-daemon.sh start master
```

假设用户已经有 `master` 正在运行了，这条命令就会启动一个新的 `master` 进程等待 ZooKeeper 中的 `znode` 被移除^①。如果用户想让这个过程更自动化一些，并指定一个专门的服务器来运行当前的 `master`，那么所有其他的 `master` 都将被认为是备份 `master`，用户可以使用 `--backup` 参数：

```
$ ./bin/hbase-daemon.sh start master --backup
```

这会让新启动的 `master` 等待一个专用的 `master`，即使用 `start-HBase.sh` 脚本启动的，或

① 如之前所写的，新启动的 `master` 也没有页面 UI 可用。也就是说，访问它的“info”端口不会得到任何反馈。

不加`--backup`参数启动的 master 在 ZooKeeper 中创建`/hbase/master`这个 znode。一旦创建完成,新启动的备份 master 就会进入轮询选举阶段。因为现在已经有 master 在运行了,它们会如之前解释的那样进入空闲模式。



如果用户启动了不只一个 master,同时用户经历过故障恢复,那么就不太容易知道现在正在对外提供服务的 master 是哪一个是了。所以也不知道对应的 Web UI 在哪台服务器上。用户需要使用 `http://hostname:60010` 在所有可能的 master 服务器上尝试查找当前提供服务的 master^①。

从 HBase 0.90.x 开始,用户也可以在 `conf` 目录下添加 `backup-masters` 文件来指定备份服务器。这个文件与 `regionserver` 文件相似,其中按行列出了需要启动备份 master 的服务器的主机名。参考 2.6.6 节,我们可以假设有 3 个备份 master 运行在 ZooKeeper 所在的服务器上。在这种情况下,配置文件 `conf/backup-masters` 中的内容大致如下:

```
zk1.foo.com
zk2.foo.com
zk3.foo.com
```

在一个规模较小的集群中将这 master 进程配置到 ZooKeeper 运行的服务器上是很好的选择,因为 master 被设计为集群运行时的协调者,所以不需要太多资源。



用户可以启动足够的备份 master 来达到自己要求的处理失效的标准。启动得太多并没有什么坏处,不过启动得太少则可能为当前系统留下隐患。不过用户也可以采用一些监控手段来报告 master 节点失效。你可以尝试修复节点并重新将它添加到集群中。总之,启动两到三个备份节点比较合适。

注意, `backup-masters` 中的节点会使用 `--backup` 选项来启动备份 master 进程。并且在 `start-hbase.sh` 脚本启动主 master 和 region 服务器之后,备份 master 才会启动。此外,用户也可以运行 `hbase-backup.sh` 脚本来初始化启动备份服务器。

添加 region 服务器。添加一个新的 region 服务器是运行集群时的常用操作。首先,用户需要修改 `conf` 目录下的 `regionserver` 文件,这样可以使启动脚本能够添加新服务器。^②简单地在文件中添加一行,内容为对应服务器的主机名即可。

用户修改配置文件之后,还需要将其复制到集群中所有的机器上。同时,用户需要保证新添加的机器上已经安装了 HBase,且配置正确。

① 问题跟踪系统中有一项来修正这个不便之处,即今后这一点会得到改善。不过到目前为止,用户可以使用脚本来从 ZooKeeper 中读取当前 master 的主机名,并指定一个固定的 DNS 项到当前的 master 上。

② 注意,有些 HBase 的发行版不要求这些,因为它们不使用 `start-hbase.sh` 脚本。

之后用户拥有多种方式来启动新的 **region** 服务器进程。一种是通过在 **master** 机器上运行 *start-hbase.sh* 脚本，它会跳过所有已经启动 **region** 服务器的服务。由于新添加的节点中没有 **region** 服务器进程，它将正常启动 **region** 服务器守护进程。

另一种方法是，使用启动脚本直接在新节点上启动，具体操作如下：

```
$ ./bin/hbase-daemon.sh start regionserver
```



以上操作必须在用户要添加新的 **region** 服务器进程的节点上使用。

新启动的 **region** 服务器进程会启动，并使用自己的主机名在 **ZooKeeper** 中创建对应的 **znode** 来进行注册，然后它会加入集群，以及被分配 **region**。

12.2 数据任务

当用户使用 **HBase** 集群时，将会处理遍布在一张或多张表中的大量数据。有时为了备份数据而需要移动全部或部分数据到归档数据中。以下是几种完成这项任务所需的方法。

12.2.1 导入/导出

HBase 发布了一些有用的工具，其中两个是支持导入和导出的 **MapReduce** 作业。它们可以将部分或全部的表写入到 **HDFS** 文件中，且随后可以再载入它们。这些工具包含在 **HBase** 的 **JAR** 文件中，用户可以通过 *hadoop jar* 命令来获得这些工具列表。

```
$ hadoop jar $HBASE_HOME/hbase-0.91.0-SNAPSHOT.jar
An example program must be given as the first argument.
Valid program names are:
  CellCounter: Count cells in HBase table
  completebulkload: Complete a bulk data load.
  copytable: Export a table from local cluster to peer cluster
  export: Write table data to HDFS.
  import: Import data written by Export.
  importtsv: Import data in TSV format.
  rowcounter: Count rows in HBase table
  verifyrep: Compare the data from tables in two different clusters.
  WARNING: It doesn't work for incrementColumnValues'd cells since the
           timestamp is changed after being appended to the log.
```

在命令后面添加 **export** 参数名就可以显示该命令的用法：

```
$ hadoop jar $HBASE_HOME/hbase-0.91.0-SNAPSHOT.jar export
ERROR: Wrong number of arguments: 0
Usage: Export [-D <property=value>]* <tablename> <outputdir> \
  [<versions> [<starttime> [<endtime>]]] \
  [^[regex pattern] or [Prefix] to filter]]

Note: -D properties will be applied to the conf used.
For example:
-D mapred.output.compress=true
-D mapred.output.compression.codec=org.apache.hadoop.io.compress. GzipCodec
-D mapred.output.compression.type=BLOCK
Additionally, the following SCAN properties can be specified
to control/limit what is exported..
-D hbase.mapreduce.scan.column.family=<familyName>
```

用户可以看到该命令提供了多种选项。其中只有 `tablename` 和 `outputdir` 两个参数是必需的。其他参数都是可选参数。表 12-1^①列出了所有可用的选项。

表 12-1 导出工具的参数

名字	描述
tablename	要导出的表的名称
outputdir	导出数据存放在 HDFS 中的路径
versions	每列备份的版本数量，默认值为 1
starttime	开始时间，进一步限制保存的版本。3.5.1 节详细介绍了这部分所使用的 <code>setTimeRange()</code> 方法的详细信息
endtime	扫描所使用的时间范围的结束时间
regexprefix	当以 “^” 开始时，该选项被当做正则表达式来匹配行键，否则会被当做行键的前缀



`regex` 参数使用了 `RowFilter` 和 `RegexStringComparator`，这两部分均在 4.1.2 节的“行过滤器 (`RowFilter`)”中做了介绍，而 `prefix` 则使用了 4.1.3 节的“前缀过滤器 (`PrefixFilter`)”所讨论的 `PrefixFilter`。

用户必须从左到右指定参数，不能省略在这之间的任何一个参数。换句话说，如果想指定一个行键过滤器，用户必须指定版本以及开始和结束时间。如果不需要这些参数，用户只需要将它们设定为它们的最小值和最大值，例如，将开始时间设为 0，而将结束时间设为 9223372036854775807（由于时间是一个长整型数）。这样会保证时间范围不会被考虑在内。

① 问题跟踪系统已经开启了一项内容，该项用来使用一个更现代的命令行解析参数。这会影响未来任务中的参数设定方式。



尽管用户提供了 HBase JAR 文件，用户还需要几个额外的依赖来保证能够顺利执行 MapReduce 作业。它需要以下几个 JAR 文件：*zookeeper-xyz.jar*、*guava-xyz.jar* 和 *google-collections-xyz.jar*。用户必须确保 MapReduce 作业可以访问到这些文件。其中一个方法是将它们添加到 *\$HADOOP_HOME/conf/hadoop-env.sh* 的 *HADOOP_CLASSPATH* 变量中。

执行以下命令就会启动 MapReduce 作业，并且打印出进度：

```
$ hadoop jar $HBASE_HOME/HBase-0.91.0-SNAPSHOT.jar export \
testtable /user/larsgeorge/backup-testtable
11/06/25 15:58:29 INFO mapred.JobClient: Running job: job_201106251558_0001
11/06/25 15:58:30 INFO mapred.JobClient: map 0% reduce 0%
11/06/25 15:58:52 INFO mapred.JobClient: map 6% reduce 0%
11/06/25 15:58:55 INFO mapred.JobClient: map 9% reduce 0%
11/06/25 15:58:58 INFO mapred.JobClient: map 15% reduce 0%
11/06/25 15:59:01 INFO mapred.JobClient: map 21% reduce 0%
11/06/25 15:59:04 INFO mapred.JobClient: map 28% reduce 0%
11/06/25 15:59:07 INFO mapred.JobClient: map 34% reduce 0%
11/06/25 15:59:10 INFO mapred.JobClient: map 40% reduce 0%
11/06/25 15:59:13 INFO mapred.JobClient: map 46% reduce 0%
11/06/25 15:59:16 INFO mapred.JobClient: map 53% reduce 0%
11/06/25 15:59:19 INFO mapred.JobClient: map 59% reduce 0%
11/06/25 15:59:22 INFO mapred.JobClient: map 65% reduce 0%
11/06/25 15:59:25 INFO mapred.JobClient: map 71% reduce 0%
11/06/25 15:59:28 INFO mapred.JobClient: map 78% reduce 0%
11/06/25 15:59:31 INFO mapred.JobClient: map 84% reduce 0%
11/06/25 15:59:34 INFO mapred.JobClient: map 90% reduce 0%
11/06/25 15:59:37 INFO mapred.JobClient: map 96% reduce 0%
11/06/25 15:59:40 INFO mapred.JobClient: map 100% reduce 0%
11/06/25 15:59:42 INFO mapred.JobClient: Job complete: job_201106251558_0001
11/06/25 15:59:42 INFO mapred.JobClient: Counters: 6
11/06/25 15:59:42 INFO mapred.JobClient:   Job Counters
11/06/25 15:59:42 INFO mapred.JobClient:     Rack-local map tasks=32
11/06/25 15:59:42 INFO mapred.JobClient:     Launched map tasks=32
11/06/25 15:59:42 INFO mapred.JobClient:   FileSystemCounters
11/06/25 15:59:42 INFO mapred.JobClient:     HDFS_BYTES_WRITTEN=3648
11/06/25 15:59:42 INFO mapred.JobClient:   Map-Reduce Framework
11/06/25 15:59:42 INFO mapred.JobClient:     Map input records=0
11/06/25 15:59:42 INFO mapred.JobClient:     Spilled Records=0
11/06/25 15:59:42 INFO mapred.JobClient:     Map output records=0
```

一旦作业结束，可以检查文件系统中的导出数据。以下使用了 *hadoop dfs* 命令（每一行都缩减了部分内容以保持水平页面宽度）：

```
$ hadoop dfs -lsr /user/larsgeorge/backup-testtable
drwxr-xr-x  - ...      0 2011-06-25 15:58 _logs
-rw-r--r--  1 ...      114 2011-06-25 15:58 part-m-00000
-rw-r--r--  1 ...      114 2011-06-25 15:58 part-m-00001
-rw-r--r--  1 ...      114 2011-06-25 15:58 part-m-00002
-rw-r--r--  1 ...      114 2011-06-25 15:58 part-m-00003
-rw-r--r--  1 ...      114 2011-06-25 15:58 part-m-00004
-rw-r--r--  1 ...      114 2011-06-25 15:58 part-m-00005
-rw-r--r--  1 ...      114 2011-06-25 15:58 part-m-00006
-rw-r--r--  1 ...      114 2011-06-25 15:58 part-m-00007
-rw-r--r--  1 ...      114 2011-06-25 15:58 part-m-00008
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00009
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00010
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00011
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00012
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00013
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00014
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00015
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00016
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00017
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00018
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00019
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00020
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00021
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00022
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00023
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00024
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00025
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00026
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00027
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00028
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00029
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00030
-rw-r--r--  1 ...      114 2011-06-25 15:59 part-m-00031
```

每一个 *part-m-nnnnn* 文件都包含导出一部分数据，将这些文件合并到一起就会形成整张表的备份。现在可以使用 *hadoop distcp* 命令将这个文件夹复制到另一个集群，并在那儿进行导入操作。

用户也可以使用可选参数实现增量备份：将开始时间设置为上一次备份的时间。作业仍然会扫描全表，但是只导出从上次备份之后修改过的部分。

只导出一列值的最新版本在大多数场景下是可以接受的，但是如果需要整张表的备份，就需要将 *versions* 设置为 2147483647，这意味着备份所有的版本。

导入数据是反向操作，我们首先可以运行不带参数的命令来获得详细使用说明，然后可以使用 `tablename` 和 `inputdir` 这两个参数来启动作业，`inputdir` 指包含导出文件的文件夹。

```
$ hadoop jar $HBASE_HOME/hbase-0.91.0-SNAPSHOT.jar import
ERROR: Wrong number of arguments: 0
Usage: Import <tablename> <inputdir>

$ hadoop jar $HBASE_HOME/hbase-0.91.0-SNAPSHOT.jar import \
  testtable /user/larsgeorge/backup-testtable
11/06/25 17:09:48 INFO mapreduce.TableOutputFormat: Created table instance \
  for testtable
11/06/25 17:09:48 INFO input.FileInputFormat: Total input paths to process : 32
11/06/25 17:09:49 INFO mapred.JobClient: Running job: job_201106251558_0003
11/06/25 17:09:50 INFO mapred.JobClient: map 0% reduce 0%
11/06/25 17:10:04 INFO mapred.JobClient: map 6% reduce 0%
11/06/25 17:10:07 INFO mapred.JobClient: map 12% reduce 0%
11/06/25 17:10:10 INFO mapred.JobClient: map 18% reduce 0%
11/06/25 17:10:13 INFO mapred.JobClient: map 25% reduce 0%
11/06/25 17:10:16 INFO mapred.JobClient: map 31% reduce 0%
11/06/25 17:10:19 INFO mapred.JobClient: map 37% reduce 0%
11/06/25 17:10:22 INFO mapred.JobClient: map 43% reduce 0%
11/06/25 17:10:25 INFO mapred.JobClient: map 50% reduce 0%
11/06/25 17:10:28 INFO mapred.JobClient: map 56% reduce 0%
11/06/25 17:10:31 INFO mapred.JobClient: map 62% reduce 0%
11/06/25 17:10:34 INFO mapred.JobClient: map 68% reduce 0%
11/06/25 17:10:37 INFO mapred.JobClient: map 75% reduce 0%
11/06/25 17:10:40 INFO mapred.JobClient: map 81% reduce 0%
11/06/25 17:10:43 INFO mapred.JobClient: map 87% reduce 0%
11/06/25 17:10:46 INFO mapred.JobClient: map 93% reduce 0%
11/06/25 17:10:49 INFO mapred.JobClient: map 100% reduce 0%
11/06/25 17:10:51 INFO mapred.JobClient: Job complete: job_201106251558_0003
11/06/25 17:10:51 INFO mapred.JobClient: Counters: 6
11/06/25 17:10:51 INFO mapred.JobClient:   Job Counters
11/06/25 17:10:51 INFO mapred.JobClient:     Launched map tasks=32
11/06/25 17:10:51 INFO mapred.JobClient:     Data-local map tasks=32
11/06/25 17:10:51 INFO mapred.JobClient:   FileSystemCounters
11/06/25 17:10:51 INFO mapred.JobClient:     HDFS_BYTES_READ=3648
11/06/25 17:10:51 INFO mapred.JobClient:   Map-Reduce Framework
11/06/25 17:10:51 INFO mapred.JobClient:     Map input records=0
11/06/25 17:10:51 INFO mapred.JobClient:     Spilled Records=0
11/06/25 17:10:51 INFO mapred.JobClient:     Map output records=0
```



用户可以使用导入作业将数据存储到一个不同的表中，只要这张表使用的是相同的模式，就可以在命令行中指定任意一个不同的表名。

MapReduce 作业读取导出的文件数据，并将其存储到指定的表中。最后，导出/导入工具的

组合是每张表的组合。如果用户拥有多张表，用户需要分开执行每张表的作业。

使用 DistCp

用户必须使用 HBase 提供的工具才能操作一张表。用户似乎也可以使用 *hadoop distcp* 命令复制 HDFS 上的整个 */hbase* 目录以备份数据。但是我们不推荐使用这种方法，因为这个操作在复制文件的过程中会忽略文件的状态：用户可能在 *memstore* 的刷写操作过程中复制数据，这样会导致数据中混杂着新旧多种文件。

这个操作也会导致用户忽略在内存中还没有被刷写的数据。低层次的复制操作只操作那些已经持久化的数据。一种解决办法是禁止对这张表的写操作，明确地刷写这张表的内存，然后再复制 HDFS 文件。

即使使用这种方法，用户仍需要仔细监测刷写操作执行到的位置，这里可能会出现问題，用户需要特别注意。

12.2.2 CopyTable 工具

另一个工具是 CopyTable，这个工具主要被用来引导集群的复制。用户可以使用该工具将一张已经存在的表从主集群复制到从集群。下面是该工具的命令行选项：

```
$ hadoop jar $HBASE_HOME/hbase-0.91.0-SNAPSHOT.jar copytable  
Usage: CopyTable [--rs.class=CLASS] [--rs.impl=IMPL] [--starttime=X]  
               [--endtime=Y] [--new.name=NEW] [--peer.adr=ADR] <tablename>
```

Options:

rs.class	hbase.regionserver.class of the peer cluster specify if different from current cluster
rs.impl	hbase.regionserver.impl of the peer cluster
starttime	beginning of the time range without endtime means from starttime to forever
endtime	end of the time range
new.name	new table's name
peer.adr	Address of the peer cluster given in the format hbase.zookeeper.quorum:hbase.zookeeper.client.port:zookeeper.znode.parent
families	comma-seperated list of families to copy

Args:

tablename	Name of the table to copy
-----------	---------------------------

Examples:

```
To copy 'TestTable' to a cluster that uses replication for a 1 hour window:  
$ bin/hbase org.apache.hadoop.hbase.mapreduce.CopyTable \  
  --rs.class=org.apache.hadoop.hbase.ipc.ReplicationRegionInterface
```

```
--rs.impl=org.apache.hadoop.hbase.regionserver.replication.ReplicationRegionServer
--starttime=1265875194289 --endtime=1265878794289
--peer.addr=server1,server2,server3:2181:/hbase TestTable
```

在用法输出信息的最后有一个例子，用户可以根据这个示例来设置自己的复制过程。用法输出信息也包含参数说明，你可能已经注意到了开始和结束时间选项，用户可以参照上文提到的导出/导入工具中相同参数的用法来使用这两个参数。

此外，用户可以使用 `families` 参数来限制所需复制列族的数量。复制操作只会考虑列值的最新版本。下面是在同一个集群中复制表的一个示例：

```
$ hadoop jar $HBASE_HOME/hbase-0.91.0-SNAPSHOT.jar copytable \
--new.name=testtable3 testtable
11/06/26 15:20:07 INFO mapreduce.TableOutputFormat: Created table instance for testtable3
11/06/26 15:20:07 INFO mapred.JobClient: Running job: job_201106261454_0003
11/06/26 15:20:08 INFO mapred.JobClient: map 0% reduce 0%
11/06/26 15:20:19 INFO mapred.JobClient: map 6% reduce 0%
11/06/26 15:20:22 INFO mapred.JobClient: map 12% reduce 0%
11/06/26 15:20:25 INFO mapred.JobClient: map 18% reduce 0%
11/06/26 15:20:28 INFO mapred.JobClient: map 25% reduce 0%
11/06/26 15:20:31 INFO mapred.JobClient: map 31% reduce 0%
11/06/26 15:20:34 INFO mapred.JobClient: map 37% reduce 0%
11/06/26 15:20:37 INFO mapred.JobClient: map 43% reduce 0%
11/06/26 15:20:40 INFO mapred.JobClient: map 50% reduce 0%
11/06/26 15:20:43 INFO mapred.JobClient: map 56% reduce 0%
11/06/26 15:20:46 INFO mapred.JobClient: map 62% reduce 0%
11/06/26 15:20:49 INFO mapred.JobClient: map 68% reduce 0%
11/06/26 15:20:52 INFO mapred.JobClient: map 75% reduce 0%
11/06/26 15:20:55 INFO mapred.JobClient: map 81% reduce 0%
11/06/26 15:20:58 INFO mapred.JobClient: map 87% reduce 0%
11/06/26 15:21:01 INFO mapred.JobClient: map 93% reduce 0%
11/06/26 15:21:04 INFO mapred.JobClient: map 100% reduce 0%
11/06/26 15:21:06 INFO mapred.JobClient: Job complete: job_201106261454_0003
11/06/26 15:21:06 INFO mapred.JobClient: Counters: 5
11/06/26 15:21:06 INFO mapred.JobClient:   Job Counters
11/06/26 15:21:06 INFO mapred.JobClient:     Launched map tasks=32
11/06/26 15:21:06 INFO mapred.JobClient:     Data-local map tasks=32
11/06/26 15:21:06 INFO mapred.JobClient: Map-Reduce Framework
11/06/26 15:21:06 INFO mapred.JobClient:   Map input records=0
11/06/26 15:21:06 INFO mapred.JobClient:   Spilled Records=0
11/06/26 15:21:06 INFO mapred.JobClient:   Map output records=0
```

复制过程还需要保证目标表已经存在：可以使用 **Shell** 来获得原表的定义，并且使用这个定义来创建目标表。用户可以省略那些没有包含在复制命令中的列族。

示例使用了一个可选参数 `new.name`，它允许你指定一个不同于原始表名的新表名。复制的表会存储在同一个集群中，这是由于没有使用 `peer.adr` 参数。



注意，上述两种工具只提供行级别的原子性。换句话说，如果用户导出或复制的表在操作过程中被别的客户端修改过，那么用户就无法辨别出哪些数据被复制到了新的地方。

尤其是当处理多张表的时候，例如，存在二级索引的情况，用户需要保证在客户端已经复制了一个所有表的一致性视图。一种解决方法是使用开始和结束时间参数。这样用户可以执行一个只处理最近更新数据的二次更新作业。

12.2.3 批量导入

HBase 拥有多种导入数据的方法。最直接的方法有两种，一种是在 MapReduce 作业中使用 `TableOutputFormat` 类（见第 7 章），另一种是使用普通的客户端 API。但是，这两种方法并不一定是最有效率的方法。

另一种高效导入大量数据的方法是批量导入（`bulkimport`）。批量导入使用一个 MapReduce 作业将表数据输出为 HBase 内部数据格式，然后直接将数据文件载入到正在执行的集群中。使用批量导入特性比只使用 HBase API 占用更少的 CPU 和网络资源。



HBase 载入数据必须在极短的时间内执行完，但是导入的数据可能会非常大。这会给集群带来额外的负载，而且有可能会使集群过载。批量导入就是一种通过避免 region 服务器扰动来解决这个问题的方法。

1. 批量导入过程

HBase 批量导入过程包含两个步骤：

准备数据

批量导入的第一步是利用一个使用 `HFileOutputFormat` 的 MapReduce 作业来生成 HBase 数据文件。`HFileOutputFormat` 会直接将数据写成 HBase 的内部存储格式，这样它们可以被高效地载入到集群中。

为了高效运行此功能，必须配置 `HFileOutputFormat` 使得它输出的 `HFile` 能够适合单一的 region：所有输出需要被载入到 HBase 的作业都会使用 Hadoop 的 `TotalOrderPartitioner` 类来进行分区，该类会将 map 输出的数据划分到键空间不相交的区间中，这些区间与表的 region 范围相对应。

`HFileOutputFormat` 包含一个简便的函数 `configureIncrementalLoad()`，这个方法会

基于当前表的 **region** 的边界自动配置 `TotalOrderPartitioner`。

载入数据

在使用 `HFileOutputFormat` 准备好数据之后,我们可以使用 `completebulkload` 工具将数据载入到集群中。它会遍历一遍准备好的数据,决定每一个文件属于哪个 **region**。然后与接收这些 `HFile` 文件的 **region** 服务器进行交互,移动文件到其存储目录,最后使得客户端可以使用这些数据。

如果 **region** 的边界在批量导入或准备数据和完成之间的过程中有改动, `completebulkload` 工具会自动地根据新的边界将这些数据文件进行拆分。这一步并不是最高效的,用户需要减少准备数据和载入数据之间的延迟,特别是当还有其他客户端在同时使用其他方法导入数据时。

这个原理利用了服务器上已经存在的合并读取 (`merge read`) 方式来扫描 `memstore`, 以及硬盘上 (更确切地说是文件系统中的) 的存储文件来获取整行的 `KeyValue`。将新生成的批量导入的文件添加进来并加以处理,这与内存文件刷写会生成新的存储文件的过程相似。

更重要的是,所有文件都是按照其对应的 `KeyValue` 实例所拥有的时间戳来排序的 (见 8.4 节)。换句话说,用户可以批量导入新旧的版本的列值,而 **region** 服务器会将这些文件恰当地进行排序。最终的结果是,用户即刻获得了所存储的行的一致并连贯的视图。

2. 使用 `importtsv` 工具准备数据

HBase 发行了一个命令行工具 `importtsv`,这个工具可以使用制表符拆分数据 (简称 TSV) 格式的文件执行批量导入。该工具默认使用 HBase 的 `put()` API 一行一行地向 HBase 插入数据。

用户也可以使用 `importtsv.bulk.output` 选项,这样 `importtsv` 工具会使用 `HFileOutputFormat` 来生成文件。然后这些文件可以被批量载入到 HBase 中去。用户可以无参数地运行这个工具来打印出简短的用法信息:

```
$ hadoop jar $HBASE_HOME/hbase-0.91.0-SNAPSHOT.jar importtsv
Usage: importtsv -Dimporttsv.columns=a,b,c <tablename> <inputdir>
```

Imports the given input directory of TSV data into the specified table.

The column names of the TSV data must be specified using the `-Dimporttsv.columns` option. This option takes the form of comma-separated column names, where each column name is either a simple column family, or a `columnfamily:qualifier`. The special column name `HBASE_ROW_KEY` is used to designate that this column should be used as the row key for each imported record. You must specify exactly one

column to be the row key, and you must specify a column name for every column that exists in the input data.

By default `importtsv` will load data directly into HBase. To instead generate HFiles of data to prepare for a bulk data load, pass the option:

```
-Dimporttsv.bulk.output=/path/for/output
```

Note: if you do not use this option, then the target table must already exist in HBase

Other options that may be specified with `-D` include:

```
-Dimporttsv.skip.bad.lines=false - fail if encountering an invalid line
```

```
'-Dimporttsv.separator=|' - eg separate on pipes instead of tabs
```

```
-Dimporttsv.timestamp=currentTimeAsLong - use the specified timestamp for the import
```

```
-Dimporttsv.mapper.class=my.Mapper - A user-defined Mapper to use instead\
of org.apache.hadoop.hbase.mapreduce.TsvImporterMapper
```

以上使用信息解释得非常详尽，所以只需要简单地运行这个工具并指定要求的选项即可。它会启动一个从 HDFS 中读取数据的作业，并准备好批量导入所需的存储文件。

3. 完全批量载入工具

无论是使用带 `importtsv.bulk.output` 选项的 `importtsv` 工具，还是其他使用 `HFileOutputFormat` 的 MapReduce 作业的工具，它们都需要使用 `completebulkload` 工具将数据导入到正在运行的集群中。

而 `completebulkload` 工具只需要用户提供 `importtsv` 的输出路径或 MapReduce 作业的输出路径，同时还需要导入的表名。例如：

```
$ hadoop jar $HBASE_HOME/hbase-0.91.0-SNAPSHOT.jar completebulkload \
-conf ~/my-hbase-site.xml /user/larsgeorge/myoutput mytable
```

如果配置文件没有包含在 CLASSPATH 中，用户可以使用可选 `-conf config-file` 参数来指定一个包含合适的 HBase 参数的文件。此外，当不通过 HBase 管理 ZooKeeper 时，CLASSPATH 必须包含拥有 ZooKeeper 配置文件的文件夹。



如果 HBase 不包含目标表，该工具会自动为用户创建一个。

`completebulkload` 工具执行得非常快，这之后集群就可以看到这些新数据了。

4. 高级用法

虽然 `importtsv` 在很多情况下都非常有用，但是高级用户可能会希望自己编写程序来生

成数据，或者导入其他格式的数据。为了实现这些，用户需要仔细阅读 `ImportTsv.java` 类，并查看 `HFileOutputFormat` 的 `JavaDoc`。

导入过程中的批量载入也可以使用用户所写的代码：查看 `LoadIncrementalHFiles` 类来获取更多的信息。

12.2.4 复制

HBase 复制功能的架构已经在 8.8 节详细讨论过了。下面我们来看一下如何在两个集群之间开启表的复制功能。

首先，修改 `conf` 文件夹中的 `hbase-site.xml` 配置文件以为整个集群打开该功能。

```
<configuration>
  <property>
    <name>hbase.zookeeper.quorum</name>
    <value>zk1.foo.com,zk2.foo.com,zk3.foo.com</value>
  </property>
  <property>
    <name>hbase.rootdir</name>
    <value>hdfs://master.foo.com:8020/hbase</value>
  </property>
  <property>
    <name>hbase.cluster.distributed</name>
    <value>true</value>
  </property>
  <property>
    <name>hbase.replication</name>
    <value>true</value>
  </property>
</configuration>
```

这个示例增加了一个新的属性 `hbase.replication`，当该属性设为 `true` 时，复制支持就会启用。它只打开了所必需的底层功能。换句话说，用户不会在集群上看到任何设置和功能上的区别。用户还需要复制修改过的配置文件到集群上所有的机器，并且重启所有服务器。

现在，用户可以修改已存在的表了（用户需要先将该表禁用），或者创建一个复制范围（`replication scope`）被设置为 1 的表（也可以查看 5.1.3 节中该属性的值范围）：

```
hbase(main):001:0> create 'testtable1','colfam1'
hbase(main):002:0> disable 'testtable1'
hbase(main):003:0> alter 'testtable1',NAME => 'colfam1',\
  REPLICATION_SCOPE => '1'
hbase(main):004:0> enable 'testtable1'

hbase(main):005:0> create 'testtable2',{ NAME => 'colfam1',\
  REPLICATION_SCOPE => 1}
```

设置好范围，为主集群作为复制源做好准备。现在，添加从（也叫对等）集群，并启动复制：

```
hbase(main):006:0> add_peer '1','slave-zk1:2181:/hbase'
hbase(main):007:0> start_replication
```

第一条命令是为从集群设置 ZooKeeper 集群信息，这样使得修改可以被同步到从集群上。第二条命令真正地将修改过的记录发布到从集群上。为了使工作能够按照预期进行，用户必须保证已经在从集群上创建了一个相同的表的副本：表可以是空的，但是必须有相同的表的模式和表名。



用户可以使用运行两个本地集群的方法（这部分内容在 12.3.1 节进行了详细描述）来进行开发和原型设计，并将第二个本地集群配置为从集群的地址。

```
hbase(main):006:0> add_peer '1','localhost:2181:/hbase-2'
```

需要修改第二个集群的 *conf.2* 目录文件夹下 *hbase-site.xml* 中的一个参数：

```
<property>
  <name>hbase.replication</name>
  <value>true</value>
</property>
```

增加这个属性可以使该集群作为从集群。

由于复制功能已经启用，用户可以在主集群上增加数据，然后过一会，用户就能够在从集群上相同名字的表中看到这些数据。

从集群已经不需要再做进一步修改了。从集群上的复制功能使用普通的客户端 API 来执行局部数据修改。移除从集群并停止复制可以使用相反的命令来完成：

```
hbase(main):008:0> stop_replication
hbase(main):009:0> remove_peer '1'
```

需要注意一点是，停止复制仍会完成所有已在队列里的修改的复制，但是之后所有的处理都被停止了。

最后，当只有几行数据时，用户可以在 Shell 中查看并简单验证两个集群上复制的数据的正确性，但是一个系统级的比较需要消耗更多的计算量。这也是提供 *Verify Replication* 工具的原因，用户可以通过 *hadoop jar* 命令调用 *verifyrep* 来执行：

```
$ hadoop jar $HBASE_HOME/HBase-0.91.0-SNAPSHOT.jar verifyrep
Usage: verifyrep [--starttime=X] [--stoptime=Y] [--families=A] <peerid>
      <tablename>
```

Options:

```
starttime    beginning of the time range
              without endtime means from starttime to forever
stoptime     end of the time range
families     comma-separated list of families to copy
```

Args:

```
peerid       Id of the peer used for verification,must match the one given
              for replication
tablename    Name of the table to verify
```

Examples:

```
To verify the data replicated from TestTable for a 1 hour window with peer #5
$ bin/HBase org.apache.hadoop.HBase.mapreduce.replication.VerifyRepli cation \
--starttime=1265875194289 --stoptime=1265878794289 5 TestTable
```

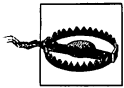
该命令必须在主集群上执行，并且需要提供从集群的 ID（建立复制流时提供的）和表名。其他选项可指定时间范围和列族。

12.3 额外的任务

除了运维任务和数据任务，我们还有额外的任务用于设置和运行测试或生产的 HBase 集群。我们将在下面的章节讨论额外的任务。

12.3.1 集群共存

为了测试，让两个不同的 HBase 实例运行在同一个物理机器上是非常有用的。例如，当用户需要在开发机上设计复制数据的原型系统时，这会非常有用。



不推荐在分布式集群上运行多个 HBase 实例，包括其任何组件，这样是未经过测试的。任何 HBase 进程都不可以在生产环境中共享同一个服务器，而且这也不是它的设计的一部分。

假设按照第 2 章所说的，安装了一个本地的 HBase，并且运行在单机模式下，用户可以先按照如下方法来复制配置文件夹：

```
$ cd $HBASE_HOME
$ cp -pR conf conf.2
```

下一步是修改新的 *conf.2* 目录中的 *hbase-env.sh* 文件。

```
# Where log files are stored. $HBASE_HOME/logs by default.
export HBASE_LOG_DIR=${HBASE_HOME}/logs.2

# A string representing this instance of hbase. $USER by default.
export HBASE_IDENT_STRING=${USER}.2
```


这样可以保证没有重复的本地文件名。然后用户还需要修改 *hbase-site.xml* 文件：

```
<configuration>
  <property>
    <name>hbase.zookeeper.quorum</name>
    <value>localhost</value>
  </property>
  <property>
    <name>hbase.rootdir</name>
    <value>hdfs://localhost:8020/hbase-2</value>
  </property>
  <property>
    <name>hbase.tmp.dir</name>
    <value>/tmp/hbase-2-${user.name}</value>
  </property>
  <property>
    <name>hbase.cluster.distributed</name>
    <value>true</value>
  </property>
  <property>
    <name>zookeeper.znode.parent</name>
    <value>/hbase-2</value>
  </property>
  <property>
    <name>hbase.master.port</name>
    <value>60100</value>
  </property>
  <property>
    <name>hbase.master.info.port</name>
    <value>60110</value>
  </property>
  <property>
    <name>hbase.regionserver.port</name>
    <value>60120</value>
  </property>
  <property>
    <name>hbase.regionserver.info.port</name>
    <value>60130</value>
  </property>
</configuration>
```

加粗的属性包含所要求的修改。用户需要给两个集群分配不同的端口，这样两个集群就会被明确地分隔开。操作第二个集群需要指明新的配置路径：

```
$ HBASE_CONF_DIR=conf.2 bin/start-hbase.sh
$ HBASE_CONF_DIR=conf.2 ./bin/hbase shell
```

```
$ HBASE_CONF_DIR=conf.2 ./bin/stop-hbase.sh
```

第一条命令启动第二个本地集群，中间那条命令启动了一个 Shell 并连接到了集群上，最后一条命令停止了集群。

12.3.2 端口要求

HBase 进程在启动时会绑定到两个不同的端口：一个是用于处理 RPC 的端口，另一个是用于 Web UI 的端口。这适用于 master 和每一个 region 服务器进程。由于用户在每台服务器上只运行一种进程类型，所以用户只需要在每台服务器上考虑分配两个端口给 HBase，除非其运行在一个非分布式集群上。表 12-2 列出了所有的默认端口。

表 12-2 HBase 守护进程使用的默认端口

节点类型	端口	描述
master	60000	master 用来监听客户端请求的 RPC 端口，可以使用 <code>hbase.master.port</code> 属性来配置
master	60010	master 进程用来监听 Web UI 的端口，可以使用 <code>hbase.master.info.port</code> 属性来配置
region 服务器	60020	region 服务器用来监听客户端请求的 RPC 端口，可以使用 <code>hbase.regionserver.port</code> 属性来配置
region 服务器	60030	region 服务器进程用来监听 Web UI 请求的端口，可以使用 <code>hbase.regionserver.info.port</code> 属性来配置

此外，如果用户想配置一个防火墙，需要保证防火墙允许访问 Hadoop 子系统使用的端口，例如，MapReduce 和 HDFS 的端口，以使 HBase 守护进程可以访问它们^①。

12.4 改变日志级别

HBase 进程默认的日志级别为 DEBUG，这在安装和设计阶段是非常有用的。它可以让用户在系统出现问题的时候在日志文件中搜寻到更多信息，这部分内容在 12.5.2 节中已经讨论过了。

在生产环境中，用户可以将日志级别降低，例如，降低到 INFO 或 WARN 级别。这个操作可以通过编辑 `conf` 目录下的 `log4j.properties` 文件来实现。下面是一个修改 HBase 类的日志级别的示例：

^① Hadoop 使用相似的端口分配方式，但是，因为它的进程种类更多，所以它占用了更多端口。请查阅网上的相关日志内容（<http://www.cloudera.com/blog/2009/08/hadoop-default-ports-quick-reference/>）。

```
...
# Custom Logging levels
log4j.logger.org.apache.zookeeper=INFO
#log4j.logger.org.apache.hadoop.fs.FSNamesystem=DEBUG
log4j.logger.org.apache.hadoop.hbase=INFO
# Make these two classes INFO-level. Make them DEBUG to see more zk debug.
log4j.logger.org.apache.hadoop.hbase.zookeeper.ZKUtil=INFO
log4j.logger.org.apache.hadoop.hbase.zookeeper.ZooKeeperWatcher=INFO
#log4j.logger.org.apache.hadoop.dfs=DEBUG
# Set this class to log INFO only otherwise its OTT
...
```

这个文件需要被复制到所有服务器上，然后重启集群使得修改生效。

如果想要暂时改变日志级别，或者修改了属性文件但是不想立即重启，用户可以使用 **Web UI** 和其日志级别页面。这部分内容在 6.5.3 节中已经讨论过了。由于 UI 日志级别修改只作用于载入该页面的服务器，所以需要分别修改集群中每一台服务器的日志级别。

12.5 故障处理

本节内容主要帮助用户处理故障集群，以及修复运行不正常工作的集群。

12.5.1 HBase Fsck

HBase 有一个叫做 *hbck* 的工具，其功能由 **HBaseFsck** 类来实现。它提供了许多可以影响其行为的命令行参数。用户可以使用 **-h** 参数来查看帮助。

```
$ ./bin/HBase hbck -h
Unknown command line option : -h
Usage: fsck [opts]
where [opts] are:
  -details Display full report of all regions.
  -timelag {timeInSeconds} Process only regions that have not experienced
    any metadata updates in the last {{timeInSeconds} seconds.
  -fix Try to fix some of the errors.
  -sleepBeforeRerun {timeInSeconds} Sleep this many seconds before checking
    if the fix worked if run with -fix
  -summary Print only summary of the tables and status.
```

在运行 *hbck* 时，**details** 参数可以打印出更为详细的输出，而 **summary** 参数打印出的输出较少。当没有参数时，则按正常情况打印输出，例如：

```
$ ./bin/HBase hbck
Number of Tables: 40
```

```

Number of live region servers: 19
Number of dead region servers: 0
Number of empty REGIONINFO_QUALIFIER rows in .META.: 0
Summary:
  -ROOT- is okay.
    Number of regions: 1
    Deployed on: host1.foo.com:60020
  .META. is okay.
    Number of regions: 1
    Deployed on: host4.foo.com:60020
  testtable is okay.
    Number of regions: 15
    Deployed on: host7.foo.com:60020 host14.foo.com:60020
  ...
  testtable2 is okay.
    Number of regions: 1
    Deployed on: host11.foo.com:60020
  0 inconsistencies detected.
Status: OK

```

其他参数如 `timelag` 和 `sleepBeforeRerun` 在使用说明中都有详细介绍。它们可以用于检查数据的子集，也可以通过延迟再次检查的时间来报告任何剩下的问题。

一旦开始运行，`hbck` 工具会扫描 `.META.` 表来收集其持有的所有相关信息。它还会扫描 HBase 使用的 HDFS 中的 `root` 目录。然后它会比较收集的信息来报告相关的一致性和完整性问题。

一致性检查

一致性检查以 `region` 为单位。它会检查 `region` 是否同时存在于 `.META.` 表和 HDFS 中，并检查其是否只被指派给唯一的 `region` 服务器。

完整性检查

完整性检查以表为单位。它将 `region` 与表细节信息来进行比较以找到缺失的 `region`。同时也会检查 `region` 起止键范围中的空洞或重叠情况。

`fix` 选项可以用来修复以上这些情况。随着时间的推移，这个功能将被继续增强，更多的问题也将被修复。在本书写作期间，可以修复的问题如下。

- 如果 `.META.` 没有被分配，则会将其分配到一个新的服务器上。
- 如果 `.META.` 被分配到多个 `region` 服务器上，其会被重新分配。
- 如果用户表的 `region` 没被分配，则其会被重新分配。

- 如果用户表的 region 被分配到多个 region 服务器上，其会被重新分配。
- 如果它当前所处的服务器与 .META. 表中所述的服务器不一致，重新分配用户表 region 到新的服务器上。



用户需要知道有时 *hbck* 报告的不一致问题只是暂时的。例如，当 region 在某些日常处理流程时变为不可用时，*hbck* 也会报告这些不一致状态。用户可通过添加 *details* 参数来得到更多运行信息，同时多次运行这条命令来确定问题不是暂时的。

12.5.2 日志分析

在少数情况下，用户需要参考由多种 HBase 进程创建的日志文件来查找问题。日志文件包括了许多信息，一些是打印的运行时系统信息，还有一些可能是警告和错误信息。某些信息只是描述集群的暂时行为，并不意味着集群出现了长期的问题。此外，还有一些在进程结束时打印的系统错误信息。

表 12-3 展示了 HBase、ZooKeeper 和 Hadoop 创建的日志文件。user 字段在实际的文件名中被启动进程的用户 ID 替换，同时 hostname 被进程运行时的服务器名替换。

表 12-3 不同服务种类创建的日志文件

服务器类型	Log 文件
HBase Master	<i>\$HBASE_HOME/logs/HBase-<user>-master-<hostname>.log</i>
HBase RegionServer	<i>\$HBASE_HOME/logs/HBase-<user>-regionserver-<hostname>.log</i>
ZooKeeper	只在控制台打印日志
NameNode	<i>\$HADOOP_HOME/logs/hadoop-<user>-namenode-<hostname>.log</i>
DataNode	<i>\$HADOOP_HOME/logs/hadoop-<user>-datanode-<hostname>.log</i>
JobTracker	<i>\$HADOOP_HOME/logs/hadoop-<user>-jobtracker-<hostname>.log</i>
TaskTracker	<i>\$HADOOP_HOME/logs/hadoop-<user>-jobtracker-<hostname>.log</i>

显然，这些路径可以通过修改对应的系统配置文件来进行修改。

当用户分析日志文件时，应当先从 master 日志文件开始，因为它起到协调整个集群的作用。该文件包括了负载均衡器（balancer）在内等后台操作的运行提示信息：

```
2011-06-03 09:12:55,448 INFO org.apache.hadoop.HBase.master.Hmaster: balance \
hri=testtable,mykey1,1308610119005.dbccd6310dd7326f28ac09b60170a84c.,\
src=host1.foo.com,60020,1308239280769,dest=host3.foo.com,60020,1308239274789
```

当 region 拆分时，应当会报告如下信息：

```
2011-06-03 09:12:55,344 INFO org.apache.hadoop.HBase.master.ServerManager: \
Received REGION_SPLIT:
testtable,myrowkey5,1308647333895.0b8eeffeba8e2168dc7c06148d93dfcf.:
Daughters;testtable,myrowkey5,1308647572030.bc7cc0055a3a4fd7a5f56df6f27a696b.,
testtable,myrowkey9,1308647572030.87882799b2d58020990041f588b6b31c.
from host5.foo.com,60020,1308239280769
```

大部分日志信息是 INFO 级别的信息，它们很好地描述了集群正在进行的工作和正处于的状态。用户可以通过查看这些信息回溯时间，同时了解集群之前在做什么。**master** 通常只是简单有规律地输出这些信息，所以当用户观察某段日志时，很可能会发现一些相同的且不断重复的运行模式。

如果出现错误，这些模式将会改变：日志信息中出现一条 WARN（warning 的简写）信息甚至 ERROR 级别的信息。用户需要找出这些模式，并在正常模式被打断前重新进行设置。



我们在 10.2.5 节中介绍了一些有用的监控指标，它们是一些系统级的统计信息：error 日志事件监控指标给出了一张图，该图显示了从什么时段开始集群打印的错误信息开始增加。找到图中曲线开始上升的时间点，并按这个时间点查找输出日志。

一旦用户找到了开始输出 ERROR 信息的时间点，用户就可以确定问题的根源了。许多后续的信息都有附加的其他损害：它们是原始问题产生的副作用。

并非所有集群行为改变的相关信息都使用更高级别的日志打印出。下面是一个 region 状态变化时间过长的日志信息。

```
2011-06-21 09:19:20,218 INFO org.apache.hadoop.hbase.master.Assignmen tManager:
Regions in transition timed out:
testtable,myrowkey123,1308610119005.dbccd6310dd7326f28ac09b60170a84c.
state=CLOSING,ts=1308647575449
```

```
2011-06-21 09:19:20,218 INFO org.apache.hadoop.hbase.master.Assignment Manager:
Region has been CLOSING for too long,this should eventually complete or the
server will expire,doing nothing
```

日志的级别是 info，因为系统最终会修复这个问题。但是，它也可能表明集群开始遇到了许多更严重的问题，如节点超载。用户在分析日志时，需要确定正常工作模式被打断的时间点。

一旦用户调查完 **master** 日志，就可以开始查看 **region** 服务器日志了。使用监控指标找到异常日志出现的时间点，然后开始仔细检查该服务器。

一旦用户发现错误信息，使用线上资源查找^①公共邮件列表（参见 <http://hbase.apache.org/mail-lists.html>）。这些问题很有可能以前都被发现并讨论过，特别是重复出现的问题，例如，之前提到的服务器过载场景：甚至错误也有固定的模式。

以下是一个错误信息的例子，这个错误是由于 region 服务器在 ZooKeeper 集群中的会话超时引起的。

```
2011-06-09 15:28:34,836 ERROR
org.apache.hadoop.HBase.regionserver.HRegionServer:
ZooKeeper session expired
2011-06-09 15:28:34,837 ERROR
org.apache.hadoop.HBase.regionserver.HRegionServer:
java.io.IOException: Server not running,aborting
...
```

用户可以在日志中查询"ERROR"和"aborting"来查找停止工作的服务器出错的原因。

12.5.3 常见问题

以下是用户安装集群时可能遇到的常见问题的列表。

1. 基本安装检查表

本节提供了一些用户在启动 HBase 之后需要检查的项目，在检查这些项目之后用户可以进行更深入的问题分析和性能调优。

文件句柄。DataNode 进程和 HBase 进程中的 `ulimit -n` 值应当被设得稍高一些。用户可以使用以下命令来检查当前 `ulimit` 设置。

```
$ cat /proc/<PID of JVM>/limits
```

用户可以看到该限制值被设置得较高，安全的设置是 32000，甚至更多。2.2.2 节中的“文件句柄和进程限制”详细说明了如何配置这些值。

DataNode 连接数。DataNode 需要配置一个更高的收发器数目，至少为 4096 或更大。设置为 16000 或更大也没有什么特别的坏处。参见 2.2.2 节中的“DataNode 处理线程数”。

压缩。压缩应当一直被打开，除非存储的信息已经被预压缩过了。用户可以参见 11.3 节所讨论的细节内容。保证用户已经检查了这些条目，这样 region 服务器可以加载对应的压缩库。否则，可能会出现以下错误：

```
hbase(main):007:0> create 'testtable',{ NAME => 'colfam1',COMPRESSION => 'LZO' }
ERROR: org.apache.hadoop.hbase.client.NoServerForRegionException: \
```

① 用户可以使用 Hadoop 搜索服务，即 <http://search-hadoop.com/>。

```
No server address listed in .META. for region \
testtable2,,1309713043529.8ec02f811f75d2178ad098dc40b4efcf.
```

在服务器的日志文件中，用户可以找到这个问题的根源（以下显示的内容被简化了，并且通过换行来保持文本宽度）：

```
2011-07-03 19:10:43,725 INFO org.apache.hadoop.hbase.regionserver.HRegion: \
    Setting up tabledescriptor config now ...
2011-07-03 19:10:43,725 DEBUG org.apache.hadoop.hbase.regionserver.HRegion:\
    Instantiated testtable,,1309713043529.8ec02f811f75d2178ad098dc40b4efcf.
2011-07-03 19:10:43,839 ERROR org.apache.hadoop.hbase.regionserver.handler.\
    OpenRegionHandler: Failed open of region=testtable,,1309713043529. \
    8ec02f811f75d2178ad098dc40b4efcf.
java.io.IOException: java.lang.RuntimeException: \
    java.lang.ClassNotFoundException: com.hadoop.compression.lzo.LzoCodec
    at org.apache.hadoop.hbase.util.CompressionTest.testCompression
    at org.apache.hadoop.hbase.regionserver.HRegion.checkCompressionCodecs
    ...
```

缺少压缩库引发了这个错误，region 服务器试图打开一个 region，而这个 region 中的列族配置使用了 LZO 压缩。

垃圾回收/内存调优。常用的垃圾回收配置我们在 11.1 节中已经讨论过了。如果内存足够，用户应当把 region 服务器的堆大小调整为 4 GB，甚至调整为 8 GB。推荐的回收策略应当可以应对各种堆大小。

如果用户把 region 服务器和 MapReduce Task Tracker 一起运行时，就需要考虑共享系统中可能出现的资源竞争。修改 *mapred-site.xml* 文件来减少 region 服务器上的计算单元位数，这样用户可以将更多的内存分配给 region 服务器。预先计算好内存的分配情况，包括 Task Tracker、region 服务器和 Child Task（参考 *mapred-site.xml* 和 *hadoop-env.sh* 两个文件）所需分配的内存，以保证 region 服务器拥有足够的内存，同时也保证系统有足够的内存，参考 2.2 节。如果资源紧张，用户可能需要考虑把 MapReduce 与 HBase 分开。

最后，HBase 也是 CPU 密集型的系统。即使内存足够也要保证 CPU 不太紧张，使用简单的命令，例如，*top* 来检查 CPU 使用率以确定是否需要减少计算单元，或使用第 10 章中介绍的方法来监控服务器状态。

2. 稳定性问题

在极端情况下，region 服务器可能会关闭或意外结束。用户可以检查以下内容。

- 检查 JVM 版本是否为 1.6.0u18（此版本有已知的问题影响 HBase 进程）。
- 检查 region 服务器日志的最后几行——其中应当包括 aborting（或 abort）信息。

最后有可能出现的情况是服务器的 ZooKeeper 会话过期超时。如果是这种情况，则需要检查以下几项。

ZooKeeper 问题。确保 ZooKeeper 能正常提供 HBase 所需的重要的协调服务。同时保证 HBase 进程可以以正常的频率与 ZooKeeper 通信也十分重要。

确定 region 服务器和 ZooKeeper 没有开始使用交换分区

如果服务器开始使用交换分区 (*swap*)，请求的某些资源可能会开始超时，同时 region 服务器会丢失其 ZooKeeper 会话，这会导致服务器关闭。用户可以使用 Ganglia 来监控交换内存的使用量，或在对应的服务器上运行以下命令：

```
$ vmstat 20
```

当服务器上有负载时（如 MapReduce 作业）：确保“si”和“so”列为 0。这些列反映了交换出或入的内存情况。同时运行以下命令：

```
$ free -m
```

此命令可以确保没有交换空间被使用（交换列应为 0），并且考虑调整内核 swappiness 值（/proc/sys/vm/swappiness）为 5 或 10。这会防止在总内存分配量小于可用内存时使用交换区。

检查网络问题

如果网络不稳定，region 服务器会丢失它们到 ZooKeeper 的连接并停止工作。

检查 ZooKeeper 机器部署情况

千万不要将 ZooKeeper 节点部署到 TaskTracker 或 DataNode 上。在较小的集群中，最好将其与 NameNode、SecondaryNameNode 或 JobTracker 部署到一起（例如，小于 40 个节点的集群）。其他进程将加剧机器的负担，并导致 ZooKeeper 超时。

最好只部署一个 ZooKeeper 与 NameNode 或 JobTracker 共享，而不是将以上三者与其他进程一起使用。

检查垃圾回收产生的停顿

检查 region 服务器日志中的"slept"，例如，用户可能会看到"We slept 65000ms instead of 10000ms"。如果用户看到了这些日志，则很可能是发生了长时间的垃圾回收或繁重的内存交换。如果是垃圾回收停顿，请参考本节“基本安装检查表”中的优化选项。

监控慢磁盘

HBase 在 DataNode 中读写块时遇到了磁盘变慢的情况，且没有做特别的降级处

理。如果这个块正好在 META region 中，则这种情况有可能会影响整个集群的性能，并造成合并变得缓慢。此时，用户需要使用监控工具检查重要的系统指标项是否正常。

“Could not obtain block”错误。通常情况下这是 DataNode 接收器的问题，在本节的“基本安装检查表”部分进行了讨论。反复检查对应的接收器值，同时检查 DataNode 日志，看看其中有没有“exceeds the limit”项，这些日志表明接收器出现了问题。检查 region 服务器和 data node 的日志以发现“Too many open files”错误。

HBase 配置属性

本节列出了 HBase 支持的所有配置和默认属性，以及配置的使用说明。使用配置时需要引用 *hbase-site.xml* 文件。为了便于查找，下面列出的属性按字母顺序排列，如何调优更重要的属性的细节请看 11.8 节。



属性的描述来自于 *hbase-default.xml* 文件。为了方便读者，类型、默认值和单位都已经填加进去。

`hbase.balancer.period`

在 master 节点中运行 region 负载均衡器的周期。

类型: int

默认值: 300000 (5 分钟)

单位: 毫秒

`hbase.client.keyvalue.maxsize`

设置 KeyValue 实例大小的上限，这是为了协助设置存储文件中单个条目存储的上限。这种做法有利于避免 region 过大但不能被拆分的现象，最好将其设置为最大的 region 大小。如果用户想绕开这个检查，可以将这个参数设置为 0 或更少。

类型: int

默认值: 10485760

单位: 字节

`hbase.client.pause`

客户端暂停时间。最常用做失败的 get 和 region 查询等操作重试前等待的时间。

类型: long

默认值: 1000 (1 秒)

单位: 毫秒

`hbase.client.retries.number`

最大重试次数。例如, **region** 查询、**get** 和 **update** 操作等发生错误时最大重试的值。

类型: int

默认值: 10

单位: 数值

`hbase.client.scanner.caching`

扫描器调用 **next** 方法的时候发现本地客户端内存的数据已经取完, 就会向服务器端发起请求, 该值就是扫描器调用 **next** 方法一次性从服务器端返回的最大行数。该值越大, 扫描器整体的返回速度就越快, 但同时依赖的内存也就越多, 并且当请求的数据没有在内存中命中的话, **next** 方法的返回时间可能会更长, 因此要避免这个时间长于扫描器超时的时间, 即 `hbase.regionserver.lease.period`。

类型: int

默认值: 1

单位: 数值

`hbase.client.write.buffer`

HTable 客户端写缓冲区的默认字节大小。该值越大消耗的内存越多——由于服务器端也需要消耗内存来处理传入的数据, 客户端与服务器端都会消耗更多的内存——较大的缓冲区大小有助于减少 **RPC** 调用的次数。例如, 服务器端的内存消耗大概等于 `hbase.client.write.buffer * hbase.regionserver.handler.count` 的值。

类型: long

默认值: 2097152

单位: 字节

`hbase.cluster.distributed`

HBase 集群的运行模式。该值为 **false** 时, 集群是单机模式; 该值为 **true** 时, 集群是分布式模式。如果将该值设置为 **false**, 则 HBase 与 ZooKeeper 的守护进程将运行在同一个 JVM 中。

类型: boolean

默认值: false

hbase.coprocessor.master.classes

HMaster 进程默认使用的协处理器是 `org.apache.hadoop.hbase.coprocessor.MasterObserver`，在这个配置中协处理器之间用逗号分隔，协处理器中实现的方法将按照配置顺序执行。用户可以通过继承 `MasterObserver` 来实现自己的协处理器，只需将其添加到 HBase 的 classpath 中，并添加可用的类名。

类型: 类名

默认值: 无

hbase.coprocessor.region.classes

协处理器之间使用逗号分隔，这些协处理器默认会被所有的表加载，并按照顺序执行。用户可以实现自己的协处理器，只需将其添加到 HBase 的 classpath 中，并在此配置完整类名。用户也可以根据需求通过设置 `HTableDescriptor` 来选择性地加载协处理器。

类型: 类名

默认值: 无

hbase.defaults.for.version.skip

将当前参数设置为 true 可以跳过 `hbase.defaults.for.version` 检查。将该参数设置为 true，其会在上下文中发挥作用，这一点不同于其在 maven 下的使用方法，即在 IDE 中通过 maven 使用 HBase。用户也可以将该参数设置为 true，以避免因 *hbase-default.xml* 中的版本匹配检查不通过而抛出的运行时异常。

类型: boolean

默认值: false

hbase.hash.type

HashFunction 中使用的散列算法，其支持两个值: `murmur(MurmurHash)` 和 `jenkins(JenkinsHash)`，并应用于布隆过滤器中。

类型: string

默认值: murmur

hbase.hregion.majorcompaction

region 中所有 `HStoreFile` 的 major 合并的周期。默认值是 1 天。将其设置为 0 可以禁用 major 合并。

类型: long

默认值: 86400000 (1 天)

单位: 毫秒

`hbase.hregion.max.filesize`

HStoreFile 的最大值。**region** 中任何一个列族的存储文件如果超过了这个上限, 就会被拆分成两个 **region**。

类型: long

默认值: 268435456 ($256 \times 1024 \times 1024$)

单位: 字节

`hbase.hregion.memstore.block.multiplier`

如果 **memstore** 达到了 `hbase.hregion.memstore.block.memstore` 乘以 `hbase.hregion.flush.size` 的大小, 就会阻塞更新操作。这是为了预防在更新高峰期会导致的失控。如果不设上界, 刷写的时候会花费很长的时间来合并或者拆分, 最坏的情况还会引发 OOME 异常。

类型: int

默认值: 2

单位: 数值

`hbase.hregion.memstore.flush.size`

如果内存的大小达到这个阈值, **memstore** 的数据就会刷写到磁盘中。这个值由一个线程每隔 `hbase.server.thread.wakefrequency` 检查一次。

类型: long

默认值: 67108864 ($1024 \times 1024 \times 64L$)

单位: 字节

`hbase.hregion.memstore.mslab.enabled`

启动本地 **memstore** 分配缓冲区 (MemStore-Local Allocation Buffer, MSLAB), 这个特性是为了防止在大量写负载的时候堆的碎片过多。这有利于降低 Full 垃圾回收的频率。

类型: boolean

默认值: true

hbase.hregion.preclose.flush.size

当我们要关闭一个 memstore 的大小大于这个值的 region 时，此时会先运行“预刷写”操作，清理这个需要关闭的 memstore，然后再将这个 region 下线。在关闭 region 时，关闭标签会触发一次清空内存的刷写。在 region 处于下线过程中时，我们就无法再对其进行任何写操作了。如果一个内存存储中的内容很大，刷写磁盘操作会消耗很多时间。预刷写操作意味着在 region 被打上关闭标签之前，会先把写 memstore 清空。这样在最终执行关闭操作的时候，带关闭标签的刷写操作会很快。

类型: long

默认值: 5242880 (1024×1024×5)

单位: 字节

hbase.hstore.blockingStoreFiles

如果一个 HRegion 中存储文件的数量(每次 MemStore 刷写到磁盘便会产生一个存储文件)达到一个阈值，该 HRegion 就会强制阻塞客户端的写请求，直到完成一次存储文件的合并，或者阻塞到 hbase.hstore.blockingWaitTime 超时。

类型: int

默认值: 7, 硬编码: -1

单位: 数值

hbase.hstore.blockingWaitTime

当一个 HRegion 的 Storefile 数量达到 hbase.hstore.blockingStoreFiles 设置的值后，其会阻塞客户端写请求。超过当前设置的时间时，即使合并没有完成，也会停止阻塞写请求。

类型: int

默认值: 90000

单位: 毫秒

hbase.hstore.compaction.max

每次 minor 合并处理的最大 HstoreFile 数目。

类型: int

默认值: 10

单位: 数值

`hbase.hstore.compactionThreshold`

当一个 HStore 含有多于这个值的 HStoreFile（每一次 memstore 刷写产生一个 HStoreFile）时，会执行一个合并操作，把这个 HStoreFiles 写成一个。这个值越大，合并消耗的时间越长。

类型：int

默认值：3，硬编码：2

单位：数值

`hbase.mapreduce.hfileoutputformat.blocksize`

MapReduce HfileOutputFormat 可以直接写 HFile 格式的存储文件。这个值是 HFile 的 blocksize 的最小值。通常在 HBase 写 HFile 的时候，blocksize 是由表模式（HColumnDescriptor）决定的，但是在 MapReduce 写 HFile 的时候，我们无法访问表模式，所以从配置中获取 blocksize。这个值越小，索引文件就越大，随机访问需要获取的数据就越小。如果用户的数据都很小，而且需要更快地随机访问，可以把 blocksize 调低。

类型：int

默认值：65536

单位：字节

`hbase.master.dns.interface`

当使用 DNS 的时候，master 用来上报 IP 地址的网络接口名。

类型：string

默认值：“default”

`hbase.master.dns.nameserver`

当使用 DNS 的时候，region 服务器使用的主机名或者 IP 地址。master 用它来确定需要进行通信的主机名。

类型：string

默认值：“default”

`hbase.master.info.bindAddress`

HBase Master 的 Web UI 绑定的地址。

类型：String

默认值：0.0.0.0

`hbase.master.info.port`

HBase Master 的 Web UI 服务端口。如果不想启动 UI 实例，则可以将当前参数设置为-1。

类型: int

默认值: 60010

单位: 数值

`hbase.master.kerberos.principal`

例如,“hbase/_HOST@EXAMPLE.COM”。HMaster 进程运行时需要使用 Kerberos 验证名, 验证名可以在 `user/hostname@DOMAIN` 中获取。如果 “_HOST” 被用作主机名, 在实际运行的时候可以使用主机名来替代。

类型: string

默认值: 无

`hbase.master.keytab.file`

HMaster 服务器验证登录使用的 Kerberos keytab 完整文件路径。

类型: string

默认值: 无

`hbase.master.logcleaner.plugins`

WAL/HLog 清理程序, 类名之间以逗号分隔, 类会被 LogsCleaner 服务顺序调用, 以删除最早的 HLog 文件。用户可以实现自己的清理程序, 只需要在 HBase 的 classpath 中设置完整的类名即可。

类型: string

默认值: `org.apache.hadoop.hbase.master.TimeToLiveLogCleaner`

`hbase.master.logcleaner.ttl`

HLog 文件在 *oldlogdir* 目录中最长的生命周期, 一旦超过这个值, HLog 就会被 master 的线程清理掉。

类型: long

默认值: 600000

单位: 毫秒

`hbase.master.port`

HBase Master 应该绑定的端口。

类型: int

默认值: 60000

单位: 数值

`hbase.regions.slop`

HBase 的负载均衡因子, 如果某台 **region** 服务器中加载的 **region** 数量达到了 “平均值+ (平均值×均衡因子)” 会自动进行负载均衡。默认值是 20%。

类型: 无

默认值: 0.2

单位: 浮点数 (百分比)

`hbase.regionserver.class`

要使用的 **RegionServer** 接口, 主要用于客户端代理^①以连接远程 **region** 服务器。

类型: 类名

默认值: `org.apache.hadoop.hbase.ipc.HRegionInterface`

`hbase.regionserver.dns.interface`

region 服务器使用 DNS 时用来报告 IP 地址的网络接口名。

类型: string

默认值: “default”

`hbase.regionserver.dns.nameserver`

region 服务器使用 DNS 时所用的主机名或者 IP 地址, **region** 服务器用其来确定和 **master** 进行通信的主机名。

类型: string

默认值: “default”

`hbase.regionserver.global.memstore.lowerLimit`

所有 **region** 的 **memstore** 所占用的内存总和达到堆的 35%时, **HBase** 会强制刷写数据到磁盘中。默认值是堆的 35%。当这个值与 `hbase.regionserver.global.memstore.upperLimit` 相等时, 更新操作由于 **memstore** 限制被阻塞时系统会以尽可能小的刷写

① 这里的代理指的是这段过程调用过程中客户端的接口类, 用于表征服务器端提供的方法。不同于上网时用的代理。——译者注

量刷写数据。

类型: float

默认值: 0.35, 硬编码: 0.25

单位: 浮点数 (百分比)

`hbase.regionserver.global.memstore.upperLimit`

单个 **region** 服务器的全部 **memstore** 的最大值。一旦超过这个值, 一个新的更新操作会被挂起, 强制执行刷写操作。默认值是堆的 40%。

类型: float

默认值: 0.4

单位: 浮点数 (百分比)

`hbase.regionserver.handler.count`

RegionServer 中 **RPC** 监听器实例的数量。对于 **master** 来说, 这个属性是 **master** 受理的处理线程 (**handler**) 数量。

类型: int

默认值: 10

单位: 数值

`hbase.regionserver.hlog.reader.impl`

负责实现 **HLog** 文件读取的类。

类型: 类名

默认值: `org.apache.hadoop.hbase.regionserver.wal.SequenceFileLogReader`

`hbase.regionserver.hlog.writer.impl`

负责实现 **HLog** 文件写入的类。

类型: 类名

默认值: `org.apache.hadoop.hbase.regionserver.wal.SequenceFileLogWriter`

`hbase.regionserver.info.bindAddress`

HBase RegionServer 的 **Web UI** 的地址。

类型: string

默认值: 0.0.0.0

`hbase.regionserver.info.port`

HBase RegionServer 的 Web UI 的端口，设置为-1 可以禁用 HBase RegionServer 的 Web UI。

类型：int

默认值：60030

单位：数值

`hbase.regionserver.info.port.auto`

该属性用于指定 Master 或 RegionServer 是否要动态搜索一个要绑定的端口。当 `hbase.regionserver.info.port` 已经被占用的时候，可以搜索一个空闲的端口来绑定。这个功能在测试的时候很有用。默认为关闭。

类型：boolean

默认值：false

`hbase.regionserver.kerberos.principal`

例如，“`hbase/_HOST@EXAMPLE.COM`”。HRegionServer 进程运行时需要使用 Kerberos 验证名。验证名应该是 `user/hostname@DOMAIN` 格式。如果“_HOST”被用做了主机名，可以使用实际运行的主机名来替代它。在 `hbase.regionserver.keytab.file` 中一定要指定默认的验证文件。

类型：string

默认值：空

`hbase.regionserver.keytab.file`

HRegionServer 验证登录使用的 Kerberos keytab 完整文件路径。

类型：string

默认值：空

`hbase.regionserver.lease.period`

HRegionServer 中租约期限，默认值是 60 秒。默认情况下，客户端必须在这个时间内发送一条信息来刷写租约，否则视为死掉。

类型：long

默认值：60000（1 分钟）

单位：毫秒

`hbase.regionserver.logroll.period`

无论当前日志中有多少记录，达到这个时间间隔系统都会自动滚动已经提交的日志。

类型: long

默认值: 3600000

单位: 毫秒

`hbase.regionserver.msginterval`

消息从 **RegionServer** 发送到 **HBase Master** 的时间间隔，单位是毫秒。

类型: int

默认值: 3000 (3 秒)

单位: 毫秒

`hbase.regionserver.nbreservationblocks`

储备的内存块的数量。当发生 OOME 异常的时候，可以用这些内存在 **region** 服务器停止工作之前做清理操作。

类型: int

默认值: 4

单位: 数值

`hbase.regionserver.optionallogflushinterval`

将 **HLog** 同步到 **HDFS** 的间隔。即使 **HLog** 没有累积到阈值，但是一旦到了时间窗口的末尾，也会触发同步。默认是 1 秒，单位是毫秒。

类型: long

默认值: 1000 (1 秒)

单位: 毫秒

`hbase.regionserver.port`

HBase RegionServer 绑定的端口。

类型: int

默认值: 60020

单位: 数值

hbase.regionserver.regionSplitLimit

region 的数量达到这个值后就不会再拆分了。这不是一个 **region** 数量的硬性限制，但是起到了一定的限制作用，到了这个阈值就应该停止拆分。默认设置为 `MAX_INT`，即不阻止拆分。

类型: `int`

默认值: `2147483647`

单位: 数值

hbase.rest.port

HBase REST 服务器的端口。

类型: `int`

默认值: `8080`, 硬编码: `9090`

单位: 数值

hbase.rest.readonly

定义 **REST** 服务器的运行模式。`false` 意味着所有的 **HTTP** 方法 (**GET**、**PUT**、**POST** 和 **DELETE**) 都是被允许的, `true` 意味着只有 **GET** 方法是被允许的。

类型: `boolean`

默认值: `false`

hbase.rootdir

这个目录是 **region** 服务器的共享目录, 用来持久存储 **HBase** 的数据。**URL** 必须完全正确, 其中包含了文件系统的 **scheme**。例如, 要表示 **HDFS** 中的 `/hbase` 目录, **HDFS** 实例的 **namenode** 需运行在服务器 `namenode.example.org` 的 `9090` 端口, 则需要将这个属性设置为 `hdfs://namenode.example.org:9090/hbase`。默认情况下, **HBase** 是写到 `/tmp` 的, 如果不修改这个配置, 数据会在集群重启时丢失。

类型: `string`

默认值: `file:///tmp/hbase-${user.name}/hbase`

hbase.rpc.engine

`org.apache.hadoop.hbase.ipc.RpcEngine` 的实现, 用于客户端/服务器 **RPC** 调用。

类型: 类名

默认值: org.apache.hadoop.hbase.ipc.WritableRpcEngine

hbase.server.thread.wakefrequency

服务线程的睡眠时间间隔, 单位是毫秒。例如, 日志滚动线程的睡眠时间间隔。

类型: int

默认值: 10000 (10 秒)

单位: 毫秒

hbase.tmp.dir

本地文件系统的临时文件夹, 可以修改为一个更为持久的目录 (*/tmp* 目录会在重启时被清除)。

类型: string

默认值: /tmp/hbase-\${user.name}

hbase.zookeeper.dns.interface

当使用 DNS 时, ZooKeeper 服务器用来上报其 IP 地址的网络接口名。

类型: string

默认值: “default”

hbase.zookeeper.dns.nameserver

使用 DNS 时, ZooKeeper 服务器使用的主机名或 IP 地址, ZooKeeper 服务器用它来确定和 master 进行通信的主机名。

类型: string

默认值: “default”

hbase.zookeeper.leaderport

ZooKeeper 用来选择主节点的端口。详情见 http://hadoop.apache.org/zookeeper/docs/r3.1.1/zookeeperStarted.html#sc_RunningReplicatedZooKeeper。

类型: int

默认值: 3888

单位: 数值

hbase.zookeeper.peerport

ZooKeeper 节点内部通信使用的端口。详情见 <http://hadoop.apache.org/zookeeper/docs/>

r3.1.1/zookeeperStarted.html#sc_RunningReplicatedZooKeeper.

类型: int

默认值: 2888

单位: 数值

hbase.zookeeper.property.clientPort

ZooKeeper 的 *zoo.cfg* 配置文件中的属性。ZooKeeper 面向客户端服务的端口。

类型: int

默认值: 2181

单位: 数值

hbase.zookeeper.property.dataDir

ZooKeeper 的 *zoo.cfg* 配置文件中的属性。ZooKeeper 元数据快照的存储目录。

类型: string

默认值: \${hbase.tmp.dir}/zookeeper

hbase.zookeeper.property.initLimit

ZooKeeper 的 *zoo.cfg* 配置文件中的属性。初始化同步阶段可使用的 tick 的数量限制。

类型: int

默认值: 10

单位: 数值

hbase.zookeeper.property.maxClientCnxns

ZooKeeper 的 *zoo.cfg* 配置文件中的属性。ZooKeeper 集群中的单个节点接受的单个客户端（以 IP 进行区分）的请求的并发数。这个值可以适当调高一点，以避免在单机模式和伪分布式模式中出现连接问题。

类型: int

默认值: 30

单位: 数值

hbase.zookeeper.property.syncLimit

ZooKeeper 的 *zoo.cfg* 配置文件中的属性。发送一个请求到获得承认之间的 tick 的

数量限制。

类型: int

默认值: 5

单位: 数值

`hbase.zookeeper.quorum`

ZooKeeper Quorum 中的服务器列表, 使用逗号分隔。例如, 默认“`host1.mydomain.com, host2.mydomain.com, host3.mydomain.com`”设置为本地主机, 协助伪分布式模式使用。在完全分布式模式下, 用户需要把所有的 ZooKeeper Quorum 节点都添加进去。如果在 `hbase-env.sh` 文件中设置了 `HBASE_MANAGES_ZK`, 此列表中的节点就是我们将会启动或停止 ZooKeeper 服务的节点。

类型: string

默认值: localhost

`hfile.block.cache.size`

分配给存储文件 HFile/StoreFile 的块缓存占启动虚拟机最大堆 (-Xmx 设置) 的比例。默认值是 20%, 设置为 0 就是不分配。

类型: float

默认值: 0.2

单位: 浮点数 (百分比)

`zookeeper.session.timeout`

ZooKeeper 会话超期时间。HBase 把这个值传递给 ZooKeeper Quorum 作为建议的会话最大超时时间。详情见 http://hadoop.apache.org/zookeeper/docs/current/zookeeperProgrammers.html#ch_zkSessions。如果客户端超时, 服务器端会做相应处理并反馈给订阅事件的客户端。单位是毫秒。

类型: int

默认值: 180000

单位: 毫秒

`zookeeper.znode.parent`

HBase 在 ZooKeeper 中的根 znode。所有的 HBase 对应要操作 ZooKeeper 的 znode 都会用这个目录作为相对路径。默认情况下, 所有 HBase 的 ZooKeeper 文件路径都是相对路径, 所以都会去这个目录下面进行操作。

类型: *string*

默认值: */hbase*

zookeeper.znode.rootserver

到保存根 *region* 位置的 *znode* 的路径, 这个值是由 *master* 来更新, 客户端和 *region* 服务器来读取的。如果将其设置为一个相对地址, 父目录就是 *\${zookeeper.znode.parent}*。默认情况下, 根 *region* 位置的存储路径是 */hbase/root-region-server*。

类型: *string*

默认值: *root-region-server*

HBase 一直在发展，本节主要介绍 HBase 近期的发展计划。

HBase 0.92.0

这一版的主题是协处理器。于 2011 年第三季度发布，其中增加了以下几个重要特性。

协处理器

这是 HBase 一个主要的新特性，协处理器可以帮助用户编写代码并在每个 region 中执行，且直接返回计算结果。详情见 4.3 节。

分布式日志拆分

在 region 服务器中，WAL 被并行、分布式地进行恢复。这使 HBase 与 BigTable 在这一点上类似。

UI 中展现任务

目前很难观察集群后台在执行什么任务，如合并或拆分。这个新特性可以帮助用户在 master 和 region 服务器的 Web UI 中观察当前集群正在执行的任务的状态。详情见 6.5 节。

性能提升

性能提升涉及多种改进，由量变引起质变，有超过 260 个优化问题解决方案被打包到 0.92.0 版本中（完整清单参见 <https://issues.apache.org/jira/browse/HBASE/fixforversion/12314223>）。

在本书出版的过程中，0.92.0 版本仍旧在发展。有关最新的动向用户可以到官方网址查看其特性列表。

HBase 0.94.0

HBase 下一个版本的计划是安全特性，这一特性会在 0.94 版本中体现出来。除此之外，还有其他一些重量级功能特性仍处于研发状态，详情可见 <https://issues.apache.org/jira/browse/HBASE/fixforversion/12316419>。

安全（Security）

在 HBase 中增加 Kerberos 验证。

辅助索引（Secondary Indexes）

通过协处理器增加辅助索引，允许用户创建和管理表上基于列的索引。

搜索集成（Search Integration）

本特性使用户可以创建和管理搜索索引，例如，按 region 的基于 Apache Lucene 索引，这样用户可以在行或列中搜索数据。

HFile 格式第二版（HFile v2）

新的存储格式，克服了现有存储格式的缺点。

这个版本还有一些很有趣的功能特性，例如，插件式块缓存特性，该特性使用户可以在 JRE 堆内存之外管理一块内存，有利于减少内存垃圾——这一点恰好是 HBase 集群在读写压力比较大的情况下最值得关注的问题。

更多的相关内容可以到 JIRA 平台中查看。

版本升级

升级 HBase 需要制定非常谨慎详细的计划，尤其是生产集群。滚动重启可以帮助用户不停机升级，详情见 12.1.2 节。



依据用户将要使用的 Hbase 版本，用户需要先升级底层 Hadoop 版本并使 Hadoop 版本与 HBase 依赖的版本进行匹配，有关 Hadoop 的升级指南可在 Hadoop 官方网站查阅。

升级到 HBase 0.90.x

由于用户使用的 HBase 版本可能不同，集群中从旧版本升级到新版本需要不同的步骤。下面列出了常规的升级方案。

原始版本 0.20.x 或 0.89.x

0.90.x 系列向下兼容，可以直接读取 0.20.x 版本产生的数据。0.90.x 与 0.89.x 会通过 MD5 散列算法（而非 Jenkins 散列）计算出 region 名，并写入指定目录——这意味着，一旦 0.20.x 向上升级，就无法再回退到 0.20.x 系列了。

升级时一定要先从 *conf* 目录中移除 *hbase-default.xml* 这个文件，与 0.20.x 不同的是，0.90.x 将这个文件默认打包到了 JAR 中，读者可以到 *src* 目录中进行查找，见 *\$HBASE_HOME/src/main/resources/hbase-default.xml* 或者见附录 A。

升级后，用户需要通过终端检查 *.META.* 的结构。以前有人建议以 16 KB 的 *MEMSTORE_FLUSH_SIZE* 运行。在 Shell 中执行以下命令：

```
hbase(main):001:0> scan '-ROOT-'
```

以上命令可以输出当前的 *.META.* 结构。检查 *MEMSTORE_FLUSH_SIZE* 是否被设置为了 16 KB (16384)。如果是，需要改变这种情况，默认的新值是 64 MB (67108864)。运行

`$HBASE_HOME/bin/set_meta_memstore_size.rb` 脚本，这会对 `.META.` 结构进行必要的修改，如果不改变上述参数的值，集群运行会变慢^①。

0.90.x 之间升级

这种情况比较简单，用户只需要简单地安装新版本，然后使用 12.1.2 节介绍的过程重启 `region` 服务器即可。

升级到 HBase 0.92.0

滚动重启是不可能的，两个版本之间的引导协议已经发生了变化。用户需要提前同步准备安装，然后关闭集群，并重新以新版本启动集群，此时不需要迁移数据。

① 更多细节请查看 “HBASE-3499 Users upgrading to 0.90.0 need to have their `.META.` table updated with the right `MEMSTORE_SIZE`” (<http://issues.apache.org/jira/browse/HBASE-3499>)。

除了 Apache 提供的版本，用户还可以有其他的选项，下面我们就罗列一下可用的安装版本。

Cloudera 的 Hadoop 分支

Cloudera 的版本（简称 CDH）基于 Apache Hadoop 最新的稳定版，并打入了很多的补丁，做了较多的移植和更新。Cloudera 提供了非常多的部署程序：源代码、二进制 tar 文件、RPM、Debian 软件包、VMware 镜像和在云中运行 CDH 的脚本。CDH 开源，发行版基于 Apache 2.0 的许可序列号，详情见 <http://www.cloudera.com/hadoop/>。

为了部署方便，Cloudera 提供了 *yum* 和 *apt* 库。CDH 可以做到一行命令就在每台机器上安装和配置 Hadoop 和 HBase，需要快速启动的用户可以自动使用整个集群而无需人工干涉。

CDH 管理跨组件的版本，并提供了一个稳定的包含兼容的一组软件包的平台。CDH3 包含以下软件包（其中很多软件包在本书中都介绍过）：

- HDFS——分布式文件系统
- MapReduce——强大的并行数据处理框架
- Hadoop Common——支持 Hadoop 子项目的实用工具集
- HBase——用于随机读写的 Hadoop 数据库
- Hive——大数据集类 SQL 查询和表
- Pig——数据流语言和编译器
- Oozie——相互独立的 Hadoop 作业工作流程

- Sqoop——将数据库数据仓库与 Hadoop 集成
- Flume——高可靠、可配置的流式数据收集框架
- ZooKeeper——分布式应用系统的协同服务
- Hue——访问 Hadoop 的桌面程序
- Whirr——一套在云上运行 Hadoop 和 HBase 的库

对于 HBase CDH 解决了集群安装的可靠性问题，且拥有 HFDS 的所有补丁来保证持久性。而 Hadoop 项目本身并没有在 0.20.x 系列中为一台服务器崩溃而不丢数据的情况提供支持。

要下载 CDH，请访问 <http://www.cloudera.com/downloads/>。

Hush SQL Schema

HBase URL 短地址（即 Hush）结构可以用 SQL 来表示：

```
CREATE TABLE user (
  id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
  username CHAR(20) NOT NULL,
  credentials CHAR(12) NOT NULL,
  roles CHAR(10) NOT NULL, // could be a separate table "userroles", but \
    for the sake of brevity it is folded in here, eg. "AU" == "Admin,User"
  firstname CHAR(20),
  lastname CHAR(30),
  email VARCHAR(60),
  CONSTRAINT pk_user PRIMARY KEY (id),
  CONSTRAINT idx_user_username UNIQUE INDEX (username)
);
```

```
CREATE TABLE url (
  id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
  url VARCHAR(4096) NOT NULL,
  refShortId CHAR(8),
  title VARCHAR(200),
  description VARCHAR(400),
  content TEXT,
  CONSTRAINT pk_url (id),
)
```

```
CREATE TABLE shorturl (
  id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
  userId INTEGER,
  urlId INTEGER,
  shortId CHAR(8) NOT NULL,
  refShortId CHAR(8),
  description VARCHAR(400),
  CONSTRAINT pk_shorturl (id),
  CONSTRAINT idx_shorturl_shortid UNIQUE INDEX (shortId),
)
```

```

        FOREIGN KEY fk_user (userId) REFERENCES user (id),
        FOREIGN KEY fk_url (urlId) REFERENCES url (id)
    )

CREATE TABLE click (
    id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
    timestamp DATETIME,
    shortId CHAR(8) NOT NULL,
    category CHAR(2),
    dimension CHAR(4),
    counter INTEGER UNSIGNED,
    CONSTRAINT pk_clicks (id),
    FOREIGN KEY fk_shortid (shortId) REFERENCES shortid (id);
)

```

对比 HBase 和 BigTable

HBase 整体上实现了第 1 章中描述的所有的 BigTable 特性。不同的是，因为 BigTable 论文本身的描述不是非常详细，而且依赖于其他的开源项目，因此两者的工作机制略有不同。

HBase 使用的时间戳单位是毫秒——BigTable 使用的单位是微妙。这并不是大问题，原因与 Java 和 C 语言的特性有关，这两种语言最优的计时器精度不同。

需要指出的是，两者使用了不同的压缩算法。HBase 使用的是 Java 提供的压缩算法，也可以使用 LZO。^①BigTable 有一种两阶段的压缩算法，分别使用 BMDiff 和 Zippy。

HBase 中有协处理器，这不同于 BigTable 中提供的 Sawzall（类似于协处理器的框架）。^②Google 的协处理器实现的细节并不详细，因此有更多未知的差异。另一方面，HBase 支持服务器端的过滤器，可以帮助减少从服务器端返回到客户端的数据量。

HBase 主要工作在 Hadoop 分布式文件系统（HDFS）上，而 BigTable 使用 GFS。但是，HBase 也可以工作在其他文件系统上，还要感谢 Hadoop 提供了 FileSystem 插件，例如，Amazon S3（类似于 HDFS）和 EBS。

HBase 不能映射存储文件到内存中，BigTable 则可以。目前有正在进行中的工作来优化 HBase 的 I/O 性能并广泛使用的 Java 的 New I/O（NIO），这些将会使 HBase 性能得到提升。

BigTable 还有一个特性叫做 *locality groups*，该特性可以使客户端可以将特定的列族组合，并使它们分享属性，如压缩，这在多个组合列经常同时被访问时非常有好处，因为它们的数据会被放到相同的存储文件中。在 BigTable 中，列族被用于统计和访问控

① 在写这本书时，Google 将 Zippy 发布在了 Apache 许可下，命名为 Snappy。详情见链接 <http://code.google.com/p/snappy/>。

② Jeff Dean 在 LADIS '09 有一个关于协处理器的谈话（<http://www.scribd.com/doc/2163448/Dean-Key-note-Ladis2009>, 66-67 页）。

制，而在 HBase 中，则是截然不同的概念跟用法。

两个系统中都有块缓存，BigTable 还实现了键/值缓存 (key/value cache)，用于被经常访问的热点单元格。

两个系统对提交日志的处理和实现也略有不同，BigTable 有两种处理慢写的提交日志，并且可以自动切换两种模式。这个特性也可以在 Hbase 中实现，但目前还没有一个相应的议题被讨论过，所以也一直被大家忽视了。

相比而言，HBase 提供了跳过写提交日志的方式，这种模式会带来性能上的提升，但前提是要保证可以接受服务器崩溃后数据丢失的情况。

METADATA 表在 BigTable 中可以用来存储二级信息，例如，与每个表段 (tablet) 相关的事件信息。这些信息可以用来分析表段的转换、拆分和合并等。HBase 早就实现了类似的功能，但是因为性能不佳而去掉了。

region 拆分是相似的，但是合并有所不同。HBase 提供了一个工具来手动合并 region，而 BigTable 中提供了自动合并 region 的逻辑。合并 region 在 HBase 中是一个细致的工作，目前需要运维人员决定如何操作最优。

另外一个细小的差距是，BigTable 的 master 提供了存储文件的垃圾回收。这样做的一个原因可能是在 BigTable 中，存储文件在 METADATA 表中被跟踪。对于 HBase 而言，清理是由 region 服务器完成的，在 region 服务器做完拆分后并没有专门记录文件的位置。

BigTable 可以在内存中映射存储文件，这使得在查询时没有磁盘寻道。HBase 中可以在列族一级配置在内存中 (in-memory) 属性，该属性被用于 LRU 算法中^①以保证其尽可能不被淘汰。

两者的合并算法也不相同。例如，合并时也包含内存数据刷写。而其他的则是相同的，只是命名不同。

region 名存储在 HBase 的 meta 表中，结合了表名、起始行键和一个 ID，而在 BigTable 中，则使用表标识符和终止行键组合。这种设计在存储文件中定位数据位置时稍稍有些影响 (详情见 8.4 节)。

最后，值得注意的是，HBase 拥有两个独立的目录表，即 -ROOT- 和 .META.。在 BigTable 中只有 root 表，由于两种系统中它们都只包含一个 region/表段，BigTable 中 root 被存储为 meta 表的一部分。在 METADATA 表中，第一个表段为 root 表，其余表段都为 meta 表段。这些仅仅是实现的细节。

① 见 Wikipedia 中 Cache algorithms 一节。

关于作者

Lars George 从 2007 年开始参与 HBase 的项目，到 2009 年已经成为 HBase comitter。他参加了各种 Hadoop 用户组会议以及一些大型的会议，如在布鲁塞尔举行的 FOSDEM，同时他还在慕尼黑创办了 Munich OpenHUG 会议。目前他在 Cloudera 工作，担任解决方案架构师，在欧洲地区提供 Hadoop 与 HBase 的技术支持、咨询和培训。

关于封面

《HBase 权威指南》封面上的动物是克莱兹代尔马。它起源于苏格兰地区，其历史可以追溯到 19 世纪初，是进口的佛兰德种马与当地母马杂交产生的品种。这种马的繁殖是为了满足当地农民的需要，以及在全国各地运输煤炭以及其他货物。由于它作为负重马非常可靠，20 世纪初，克莱兹代尔马开始被出口到许多国家，包括澳大利亚、新西兰、加拿大和美国。机械时代的到来降低了对这一品种的需求，虽然 20 世纪后期对该马的需求量出现了小幅上升，但这种马仍然被认为是容易灭绝的品种。

现代的克莱兹代尔马略大于原来的苏格兰马，品种标准高度 162~183 cm（约 64~72 英寸），重量约 725~998 kg（约 1600~2200 磅）。不过，马的外观在其整个发展历史中基本没变。克莱兹代尔马与其他品种相比，有非常鲜明的特征，尤其是其腿部和高昂的步伐。它的身体通常是赤褐色的、棕色或黑色的，白色或红棕色的鬃毛。亮白的脸和腿与深色的身体形成鲜明对比，不过腿是黑色的马也并不罕见。众所周知的是，它的脚的大小刚好适合盘子大小的马蹄铁。

虽然在很大程度上拖拉机取代了马的地位，但克莱兹代尔马仍然是农业生产中不可或缺的生产物资，同时也用于骑、运输服务和旅游等。在美国，克莱兹代尔马是 Anheuser Busch 酿酒公司营销活动使用的马队中最常选用的品种。

