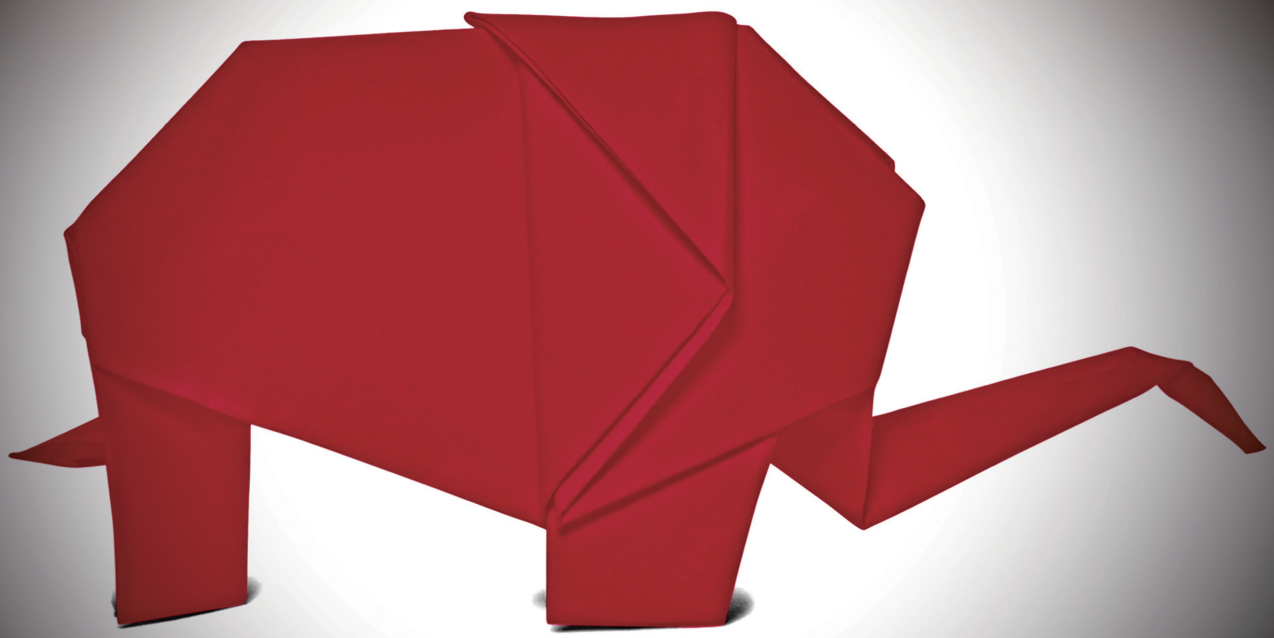
Join the discussion @ p2p.wrox.com

Wrox Programmer to Programmer™



Professional Hadoop Solutions

# Hadoop高级编程

## ——构建与实现大数据解决方案

Boris Lublinsky
[美] Kevin T. Smith 著
Alexey Yakubovich
穆玉伟 靳晓辉 译

清华大学出版社

# Hadoop 高级编程

## ——构建与实现大数据解决方案


Boris Lublinsky
[美] Kevin T. Smith 著
Alexey Yakubovich
穆玉伟 靳晓辉 译



清华大学出版社

北 京

Boris Lublinsky, Kevin T. Smith, Alexey Yakubovich
Professional Hadoop Solutions
EISBN：978-1-118-61193-7
Copyright © 2013 by John Wiley & Sons, Inc., Indianapolis, Indiana
All Rights Reserved. This translation published under license.

**本书中文简体字版由** Wiley Publishing, Inc. **授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。**

**北京市版权局著作权合同登记号 图字：**01-2013-8907

Copies of this book sold without a Wiley sticker on the cover are unauthorized and illegal.

**本书封面贴有** Wiley **公司防伪标签，无标签者不得销售。**
**版权所有，侵权必究。侵权举报电话：**010-62782989　13701121933

**责任编辑：王　军　刘伟琴**
**装帧设计：孔祥峰**
**责任校对：曹　阳**
**责任印制：**

**出版发行：清华大学出版社**
**网　　址：**http://www.tup.com.cn，http://www.wqbook.com
**地　　址：北京清华大学学研大厦** A **座**　　**邮　　编：**100084
**社 总 机：**010-62770175　　**邮　　购：**010-62786544
**投稿与读者服务：**010-62776969，c-service@tup.tsinghua.edu.cn
**质　量　反　馈：**010-62772015，zhiliang@tup.tsinghua.edu.cn
**印 刷 者：**
**装 订 者：**
**经　　销：全国新华书店**
**开　　本：**185mm×260mm　　**印　　张：**28　　**字　　数：**681 **千字**
**版　　次：**2014 **年 月第** 1 **版**　　**印　　次：**2014 **年 月第** 1 **次印刷**
**印　　数：**1~4000
**定　　价：**59.80 **元**

**产品编号：**

# 译 者 序

自 2004 年左右 Google 发布 GFS 和 MapReduce 论文以来，伴随着国内移动通信和互联网的爆炸式发展，人们开始被信息所淹没，大数据开始推动各个领域发生深刻的变化，也产生了全新的机遇。Hadoop 生态系统作为 GFS 和 MapReduce 最成功的开源实现和演化，为我们分析和挖掘大数据提供了强大而易用的工具。

大数据如此重要，译者在工作中也不可避免地使用 Hadoop 等系统来存储用户数据，并在这些海量数据的基础上进行大量的运算和挖掘，以分析用户行为、支持产品决策。在学习或使用 Hadoop、GFS、MapReduce 等众多分布式存储/计算平台时，自己的一个最大体会就是中文资料太少，好的资料基本上都是英文资料。因而，看到本书英文版之后，就非常迫切地希望能将它翻译为中文，以便和国内同仁们一起分享。

本书的作者都是大数据和 Hadoop 领域的资深专家，有着多年的相关领域经验。本书各章提供了很多例子，除了一些入门级的例子之外，很多例子都来自实际项目，让我们有机会领略 Hadoop 在实际环境中的企业级应用。此外，本书紧跟最新技术，视野开阔，书中介绍的很多最新的项目，例如 Oozie、DSL 等都代表了 Hadoop 生态系统演化的最新流行方向，目前还很少有其他的中文出版物进行过系统阐述。这也是译者翻译并力荐本书的主要原因。

翻译的过程很有收获，但又颇为辛苦，译者在此过程中力求做到专业术语的准确、一致，对一些难以用中文描述的概念，也进行了反复推敲，力争做到准确而易于理解。尽管如此，由于时间仓促以及水平有限，错误之处在所难免，敬请广大读者批评指正。

参与本书翻译工作的还有李露熙、张宇欣和龙伟。

当人们都在讨论大数据的时候，真正理解大数据的精髓，并掌握 Hadoop 中一系列的强大工具来驾驭大数据，是一件非常快乐而有意义的事情。希望读者通过阅读本书能早日步入 Hadoop 的殿堂，领略大数据之美！

译 者

2014 年 5 月于北京

# 作 者 简 介

Boris Lublinsky 是诺基亚的一名资深架构师，他在诺基亚积极参与了无数聚焦于技术架构、面向服务架构(Service-Oriented Architecture，SOA)和集成项目等企业级应用的所有阶段。他也是诺基亚架构委员会的一名活跃会员。Boris 是多家行业杂志中 80 多篇出版物的作者，同时也是 *Service-Oriented Architecture and Design Strategies* (Indianapolis: Wiley, 2008)一书的合著者。此外，他是 InfoQ 在 SOA 和大数据领域的一名编辑，并经常在行业会议上演讲。在过去的两年里，他参与了多个基于 Hadoop 和 Amazon Web Services (AWS) 应用的设计和实现。他现在是芝加哥地区 Hadoop 用户组的活跃成员、工作组织者和撰稿人。

Kevin T. Smith 是 Novetta Solutions 公司应用事业解决方案(ASM)部门的技术解决方案和推广总监，他提供策略性的技术领导并为客户开发创新性的、聚焦于数据的、高度安全的解决方案。他经常在技术会议上演讲，是无数 Web 服务、云计算、大数据和网络安全相关技术文章的作者。他撰写了很多技术书籍，包括 *Applied SOA: Service-Oriented Architecture and Design Strategies* (Indianapolis: Wiley, 2008)、*The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management* (Indianapolis: Wiley, 2003)、*Professional Portal Development with Open Source Tools* (Indianapolis: Wiley, 2004)、*More Java Pitfalls* (Indianapolis: Wiley, 2003)以及其他书籍。

Alexey Yakubovich是Hortonworks的一名系统架构师。他在Hadoop/Big Data环境中为不同的公司和项目工作了5年：PB量级存储、流程自动化、自然语言处理(Natural Language Processing，NLP)、来自移动设备的数据流的数据科学和社交媒体。更早些时候，他工作于SOA、Java 2企业版(J2EE)、分布式应用和代码生成等技术领域。他通过解决Hilbert第一问题的最后部分获得了数学博士学位。他是MDA OMG工作组的一名成员，并参与和出席美国芝加哥地区的Hadoop用户组。

# 技术编辑简介

Michael C. Daconta 是 Advanced Technology for InCadence Strategic Solutions (http://www.incadencecorp.com)的副总裁，他目前为政府和商业客户指导多个高级技术项目。他是一位著名的作者、演讲者和专栏作家，撰写或与其他人共同撰写了 11 部技术书籍(涉及语义 Web、XML、XUL、Java、C++和 C 等主题)、无数的杂志文章和在线专栏文章。他还为 *Government Computer News* 撰写月度“ Reality Check ”专栏文章。他获得了纽约大学的计算机学士学位、Nova Southeastern 大学的计算机科学硕士学位。

Michael Segel 从事 IT 行业已经超过了 20 年。他从 2009 年起集中关注大数据领域，同时拥有 MapR 和 Cloudera 认证。Segel 创立了美国芝加哥地区 Hadoop 用户组，并活跃于美国芝加哥大数据社区。他获得了俄亥俄州立大学工程学院 CIS 专业的理学学士学位。在空闲时，他把时间花在遛狗上。

Ralph Perko是美国西北太平洋国家实验室可视化分析组的一名软件架构师和工程师。他目前参与多个Hadoop项目，帮助发明、开发用于处理和可视化大规模数据集的新方法。他拥有 16 年的软件开发经验。

# 致　　谢

感谢和我一起工作的许多同事。他们总是督促我发挥到极限、询问我的解决方案，同时也让他们自己进步。本书采纳了他们的很多想法，希望能因此而更好。

非常感谢 Dean Wampler，他为第 13 章提供了 Hadoop 特定领域语言的内容。

感谢本书的合著者 Kevin 和 Alexey，他们把自己的 Hadoop 视角带到本书中，使得本书的内容更好、更平衡。

感谢技术编辑们对本书内容的很多有价值的建议。

感谢 Wiley 的编辑人员 Bob Elliott 决定出版本书；感谢我们的编辑 Kevin Shafer 不懈地工作来修正任何不一致之处；还感谢很多其他人，他们创造了您这时正在阅读的最终产品。

——Boris Lublinsky

首先，我想感谢本书的合著者以及 Wrox 出版社伟大团队付出的努力。这是我的第 7 个出书项目，我想要特别感谢我的好妻子 Gwen，以及我亲爱的孩子们 Isabella、Emma 和 Henry，他们容忍我很多个日日夜夜和周末一直都呆在自己的电脑前。

真诚地感谢阅读这些章节早期草稿的很多人，他们提供了很多建议、编辑、深入的观察和想法——特别是 Mike Daconta、Ralph Perko、Praveena Raavichara、Frank Tyler 和 Brian Uri。我对自己的公司 Novetta Solutions 心存感激，尤其想感谢 Joe Pantella 和 Novetta 执行团队的其他成员，他们支持我撰写本书。

有几件事情可能已经影响了我今年的新书撰写。有一段时间，华盛顿 Redskins(译者注：一支橄榄球队)看起来势不可挡，如果他们确实能进入“超级碗”(译者注：美国职业橄榄球队冠军赛)，这将会危及本书的截止时间。然而，Redskins 的赛季和另一个我参与其中的赛季被突然结束了。因而，我们要为本书的按时完成而感谢 Redskins 球队和 POJ 球队。

最后，特别感谢 CiR、Manera，以及过去、现在和未来的那个人。

——Kevin T. Smith

感谢诺基亚的同事们，撰写本书时我还在那里工作，他们的建议和知识为我撰写的几章提供了非常专业的素材。

感谢本书的合著者 Boris 和 Kevin，他们使本书以及我的参与成为可能。

感谢 Wiley 的编辑人员出版本书，他们提供了所有必要的帮助和指导来使本书变得更有价值。

——Alexey Yakubovich

# 前　言

在这个技术不停地改变的快节奏世界中，我们已经被信息淹没。我们在不断生成和存储大量的数据。随着网络上的设备不断丰富，我们已经看到了信息格式和数据多样性的惊人增长——大数据。

但是我们要直面它——如果我们忠实于自己的话，多数组织还不能积极有效地管理这大量的数据，我们也还不能使这些信息发挥优势，从而更好地做出决策，更聪明地做生意。我们已经被大批量的数据所淹没，但同时我们又渴求知识。这会导致公司损失生产力、损失机会，并损失收入。

在过去的10年中，很多技术承诺帮助处理和分析我们拥有的大批量信息，而这些技术多数都出现了不足。我们知道这一点，因为作为专注于数据的程序员，我们已经都尝试过了。很多方法都是受专利保护的，导致供应商被锁定。一些方法看起来很有希望，但无法扩展以处理大型数据集，还有很多是宣传很好但不能满足预期，或者在关键时刻还没有准备好。

然而，Apache Hadoop 登场之后，一切都不一样了。当然这里也有宣传的因素，但它是一个开源项目，已经在大规模可扩展商业应用中取得了不可思议的成功。尽管学习曲线有些陡峭，但这是我们第一次可以轻松地编写程序并对大规模数据进行分析——以一种以前我们做不到的方式。MapReduce 算法可以使开发人员处理分布在可扩展机器集群上的数据，基于该算法，我们以过去无法做到的方式，在进行复杂数据分析方面取得了很大的成功。

关于 Hadoop 的书并不缺少。人们写了很多，而且其中很多书都很好。那么，为什么还要编写本书呢？当我们开始使用 Hadoop 时，希望有一本书不只是介绍 API，还要解释 Hadoop 生态系统的诸多部分如何共同工作，并可用于构建企业级的解决方案。我们在寻找这样一本书，它可以带领读者学习数据设计及其对实现的影响，并解释 MapReduce 的工作原理，以及如何用 MapReduce 来重新表述特定的业务问题。我们在寻找如下问题的答案：

- MapReduce 的强项和弱点是什么，以及我们如何自定义它以便更好地满足自己的需求？
- 为什么我们需要在 MapReduce 之上有一个额外的协调层，以及 Oozie 是怎么满足这个需求的？
- 我们如何使用特定领域语言(Domain-Specific Language，DSL)来简化 MapReduce 开发？

- 每个人都在讲的实时 Hadoop 是什么，它可以做什么，以及它不能做什么？它的工作原理是什么？
- 我们如何确保 Hadoop 应用程序的安全，我们需要考虑什么，我们必须考虑什么安全隐患，以及处理这些问题有哪些方法？
- 我们如何将自己的 Hadoop 应用程序迁移到云中，以及这样做有哪些重要的考虑因素？

当开始 Hadoop 探险时，我们不得不夜以继日地浏览整个 Internet 和 Hadoop 源代码，与人们交谈并尝试使用这些代码来找到这些问题的答案。然后我们决定通过撰写本书来分享自己的发现和经验，并希望本书能够成为读者理解和使用 Hadoop 的一个良好的开端。

# 本书读者对象

本书是由程序员写给程序员的。本书的作者是开发企业级解决方案的技术人员，我们对于本书的目标是为使用 Hadoop 的其他开发人员提供可靠的、实用的建议。本书的目标人群是试图更好地理解和利用 Hadoop——不只是做简单的数据分析，同时也将 Hadoop 用作企业级应用的基础——的软件架构师和开发人员。

因为 Hadoop 是一个基于 Java 的框架，所以本书包含了大量的代码示例，需要读者熟悉 Java。此外，作者假设读者在一定程度上熟悉 Hadoop，并有一些入门级的 MapReduce 知识。

尽管本书在设计上希望读者从头到尾以逐个模块的方式阅读，但某些章节可能更适合特定的人群。想要理解 Hadoop 数据存储能力的数据设计人员更可能从第 2 章中受益。从 MapReduce 开始的程序员们最有可能关注第 3～第 5 章，以及第 13 章。意识到不使用像 Oozie 这样的 Workflow 系统会导致复杂性的开发人员，最可能关注第 6～第 8 章。那些对实时 Hadoop 感兴趣的人们会关注第 9 章。有兴趣在实现中使用亚马逊云的人们可能会关注第 11 章，而在乎安全性的人们可能关注第 10 章和第 12 章。

# 本书涵盖的内容

现在，每个人都在处理大数据。很多企业正在最大限度地实施大规模可扩展性数据分析工作，其中多数企业都在尝试使用 Hadoop 来完成该目标。本书集中讲述构建基于 Hadoop 高级企业级应用的架构和方法，并为此涵盖了如下主要 Hadoop 组件：

- 基于 Hadoop 的企业级应用的架构蓝图
- 基础的 Hadoop 数据存储和组织系统
- Hadoop 的主要执行框架(MapReduce)
- Hadoop 的 Workflow/Coordinator 服务器(Oozie)
- 实现基于 Hadoop 的实时系统的技术

? 在云环境中运行 Hadoop 的方式
? 确保 Hadoop 应用安全的技术和架构

# 本书的组织结构

本书被编排为 13 章。

第 1 章(“大数据和 Hadoop 生态系统”)介绍了大数据，以及 Hadoop 用作大数据实现的方法。在该章中，我们学习 Hadoop 如何解决大数据带来的挑战，以及哪些 Hadoop 核心组件可以共同工作来创建丰富的 Hadoop 生态系统，适合解决很多现实世界的问题。我们也学习了多个可用的 Hadoop 发行版，以及新出现的用于大数据应用的架构模式。

任何大数据实现的基础都是数据存储设计。第 2 章(“Hadoop数据存储”)涵盖了Hadoop支持的分布式数据存储。该章讨论了两个主要的Hadoop数据存储机制——HDFS和HBase——的架构和API，并提供了何时使用哪种机制的一些建议。这里我们学习了HDFS(联盟)和HBase新文件格式以及协处理器的最新发展。该章也涵盖了HCatalog(Hadoop元数据管理解决方案)和Avro(一个序列化/组装框架)，以及它们在Hadoop数据存储中扮演的角色。

作为主要的 Hadoop 执行框架，MapReduce 是本书的主要议题之一，包含在第 3～第 5 章中。

第 3 章(“使用 MapReduce 处理数据”)介绍了 MapReduce 框架。涵盖了 MapReduce 架构、它的主要组件和 MapReduce 编程模型。该章也重点介绍了 MapReduce 应用的设计、设计模式以及 MapReduce 的一般注意事项。

第 4 章(“自定义 MapReduce 执行”)建立在第 3 章的基础之上，涵盖了自定义 MapReduce 执行的重要方法。我们学习 MapReduce 执行可以被自定义的一些方面，并使用工作代码示例来揭示如何做到这一点。

最后，在第 5 章(“构建可靠的 MapReduce 应用程序”)中，我们学习了构建可靠的 MapReduce 应用程序的途径，包括测试和调试，以及使用内置的 MapReduce 工具(例如，日志和计数器)来查看 MapReduce 的内部执行。

尽管 MapReduce 自身具有强大的功能，但实际的解决方案通常需要将多个 MapReduce 应用组合到一起，这涉及很多的复杂性。通过使用 Hadoop Workflow/Coordinator 引擎——Oozie——可以显著地简化这种复杂性，Oozie 将在第 6～第 8 章中描述。

第 6 章(“使用 Oozie 自动化数据处理”)介绍了 Oozie。这里我们学习 Oozie 的整体架构、它的主要组件，以及每个组件的编程语言。我们也学习 Oozie 的整体执行模型，以及可以和 Oozie 服务器交互的方式。

第 7 章(“使用 Oozie”)建立在第 6 章所学知识的基础之上并展示了一个实用的从无到有使用 Oozie 开发实际应用的示例。该示例演示了如何在解决方案中使用不同的 Oozie 组件，并演示了设计和实现途径。

最后，第 8 章(“高级 Oozie 特性”)讨论一些高级特性，并展示了扩展 Oozie 和将它与

其他企业级应用集成的方法。在该章中，我们学习了一些开发人员需要知道的建议和技巧——例如，Oozie 代码的动态生成如何帮助开发人员克服一些现存的、任何其他方法都不能解决的 Oozie 缺点。

当今与大数据相关的最热门的趋势之一是进行“实时分析”的能力。该主题在第 9 章(“实时 Hadoop”)中讨论。该章开始提供了一些目前使用的实时 Hadoop 应用示例，并展示了这些实现的整体架构需求。我们将学习构建这样一些实现的 3 种主要方法——基于 HBase 的应用程序、实时查询和基于流的处理。

该章提供了两个基于HBase的实时应用——一个假想的图片管理系统和一个基于 Lucene、使用HBase作为后端的搜索引擎。我们也会学习实现实时查询的整体架构，以及两个具体的产品——Apache Drill和Cloudera的Impala——实现实时查询的方式。该章同时涵盖了另一种类型的实时应用——复杂事件处理，包括它的整体架构，以及HFlame和Storm实现该架构的方式。最后，该章提供了实时查询、复杂事件处理和MapReduce之间的一个对比。

Hadoop应用程序开发中一个经常被忽略但至关重要的主题是Hadoop安全。第 10 章(“Hadoop安全”)深入讨论了与大数据分析和Hadoop——确切地说是Hadoop安全模型和最佳实践——相关的安全性考虑。这里我们学习Rhino项目——一个支持开发人员扩展Hadoop安全能力(包括加密、认证、授权、单点登录(Single-Sign-On，SSO)和审计)的框架。

基于云的 Hadoop 使用需要有趣的架构决策。第 11 章(“在 AWS 上运行 Hadoop 应用”)描述了这些挑战，并涵盖了在亚马逊 Web 服务(Amazon Web Service，AWS)云上运行 Hadoop 的不同方法。该章也讨论了一些权衡选择并考察了一些最佳实践。我们将学习弹性 MapReduce(Elastic MapReduce，EMR)和可以用于补充 Hadoop 功能的额外 AWS 服务(例如 S3、CloudWatch、Simple Workflow 等)。

除了确保 Hadoop 自身的安全，Hadoop 实现通常和其他企业级组件集成——数据经常被导入到 Hadoop 并被导出。第 12 章(“为 Hadoop 实现构建企业级安全解决方案”)涵盖了如何确保使用 Hadoop 的企业级应用尽可能安全，并提供了示例和最佳实践。

作为本书的最后一章，第 13 章(“Hadoop 的未来”)浏览了一些当前和未来将发生的 Hadoop 行业趋势和创新。这里我们学习了可用来简化 MapReduce 开发的 Hadoop DSL 及其使用，以及新的 MapReduce 资源管理系统(YARN)和 MapReduce 运行时扩展(Tez)。我们也学习了最重要的 Hadoop 发展方向和趋势。

# 使用本书需要的条件

本书中展示的所有代码都用 Java 实现。因此，要使用这些代码，读者需要 Java 编译器和开发环境。所有的开发都在 Eclipse 中完成，但由于每个项目都有一个 Maven pom 文件，因而把代码迁移到任何您所选择的开发环境都足够简单。

所有的数据访问和 MapReduce 代码都在 Hadoop 1(Cloudera CDH 3 发行版和 Amazon

EMR)和 Hadoop 2(Cloudera CDH 4 发行版)上测试通过。因此，这些代码应当可以在任何 Hadoop 发行版上运行。Oozie 代码在最新版的 Oozie(例如，可以从 Cloudera CDH 4.1 发行版获得)上测试通过。

示例的源代码都以 Eclipse 项目进行组织(每章一个项目)，并可以从 Wrox 网站 www.wrox.com/go/prohadoopsolutions 下载获得。

## 源代码

在学习本书中的示例时，读者可以选择手工输入所有的代码，也可以使用随书的源代码文件。本书的源代码可以从 www.wrox.com 下载获得。具体对本书而言，专门的代码下载位于 www.wrox.com/go/prohadoopsolutions 的 Download Code 链接。

读者也可以在 www.wrox.com 按照本书英文版的 ISBN(本书英文版的 ISBN 是 978-1-118-61193-7)搜索本书来找到这些代码。当前所有的 Wrox 图书代码下载的完整列表位于 www.wrox.com/dynamic/books/download.aspx。

在整个选定的各章中，读者也可以在需要时，从代码清单的标题和文本中找到代码文件名称的引用。

www.wrox.com 上的大多数代码都压缩在.ZIP、.RAR 文档或适合特定平台的类似文档中。一旦下载了这些代码，只需要使用合适的压缩工具将其解压即可。

注意：

由于许多图书的标题都很类似，因此按 ISBN 搜索是最简单的，本书英文版的 ISBN 是 978-1-118-61193-7。

另外，读者也可以通过网址 www.wrox.com/dynamic/books/download.aspx 进入 Wrox 代码下载主页面，找到本书以及所有其他 Wrox 图书的可用代码。

## 勘误表

尽管我们已经尽了各种努力来保证文章或代码中不出现错误，但是错误总是难免的，如果您在本书中发现了错误，例如拼写错误或代码错误，请告诉我们，我们将非常感激。通过勘误表，可以让其他读者避免受挫，当然，这还有助于提供更高质量的信息。

要找到本书英文版的勘误表，可以登录 http://www.wrox.com/go/prohadoopsolutions，单击 Errata 链接。在这个页面上可以查看到 Wrox 编辑已提交和粘贴的所有勘误项。

如果您发现的错误在我们的勘误表里还没有出现的话，请登录 www.wrox.com/contact/techsupport.shtml 并完成那里的表格，把您发现的错误发送给我们。我们会检查您的反馈信息，如果正确，我们将在本书的勘误表页面张贴该错误消息，并在本书的后续版本加以修订。

## p2p. wrox.com

要与作者和同行讨论，请加入 p2p.wrox.com 上的 P2P 论坛。这个论坛是一个基于 Web 的系统，便于您张贴与 Wrox 图书相关的消息和相关技术，与其他读者和技术用户交流心得。该论坛提供了订阅功能，当您感兴趣的主题在论坛上有新帖子发布时，系统会向您发送电子邮件。Wrox 的作者、编辑和其他业界专家和读者都会到这个论坛上来探讨问题。

在 http://p2p.wrox.com 上，有许多不同的论坛，它们不仅有助于阅读本书，还有助于开发自己的应用程序。要加入论坛，可以遵循下面的步骤：

(1) 进入 p2p.wrox.com，单击 Register 链接。

(2) 阅读使用协议，并单击 Agree 按钮。

(3) 填写加入该论坛所需要的信息和自己希望提供的其他可选信息，单击 Submit 按钮。

(4) 您会收到一封电子邮件，其中的信息描述了如何验证账户，完成加入过程。

**注意：**

不加入 P2P 也可以阅读论坛上的消息，但要张贴自己的消息，就必须先加入该论坛。

加入论坛后，就可以张贴新消息，响应其他用户张贴的消息。可以随时在 Web 上阅读消息。如果要让该网站给自己发送特定论坛中的消息，可以单击论坛列表中该论坛名旁边的 Subscribe to this Forum 图标。

要想了解更多的有关论坛软件的工作情况，以及 P2P 和 Wrox 图书的许多常见问题的解答，就一定要阅读 FAQ，只需要在任意 P2P 页面上单击 FAQ 链接即可。

# 目　录

第 1 章

# 大数据和 Hadoop 生态系统

**本章内容提要**

- 了解大数据带来的挑战
- 初识 Hadoop 生态系统
- 熟悉 Hadoop 发行版
- 使用基于 Hadoop 的企业级应用

每个人都在说——我们生活在“大数据”时代。说不定你也听说过这句话。在由科技驱动的当今世界，计算能力极大提升、电子设备更加普及、访问 Internet 已经易如反掌，并且用户能够传递和收集的数据比以往任何时候都多。

一些组织正在以惊人的速度产生数据。据报道，仅 Facebook 每天就会收集 250TB 的数据。根据汤姆森路透社新闻分析部门的报导，数字化数据的产生量已经不止翻了一倍，从 2009 年的近一百万 PB(等于约 10 亿 TB)到 2015 年预计的 7.9ZB(1ZB 等于一百万 PB)，以及到 2020 年的约 35ZB。其他研究组织给出的估计甚至更高！

随着众多组织开始收集和产生海量数据，它们已经认识到了数据分析的益处。但海量数据的管理问题也让它们很挠头。这是个全新的挑战。怎样才能有效地存储规模如此庞大的数据？怎样才能有效地处理它们？怎样才能以一种有效的方式分析我们的数据？既然已知数据量将会持续增长，那么该如何构建一个能够与之相适应的解决方案呢？

并不是只有学术研究者和数据科学家遇到了大数据带来的这些挑战。在几年前的一段 Google+谈话中，著名的计算机图书出版社 Tim O’Reilly 引用了 Alistair Croll 的观点，他说“拥有海量数据但没有足够思路的公司将会被那些刚刚创始的、只拥有相对少量数据但思路更加广阔的公司所取代……”简言之，Croll 所表达的意思是，除非我们的企业理解自己所拥有的数据，否则它将无法同能够做到这一点的企业竞争。

许多企业意识到对商业竞争、态势感知、生产率、科学和革新相关的大数据进行分析

可以获得巨大的利益。大多数组织赞同 O'Reilly 和 Croll 的观点，因为竞争正在驱动着对大数据的分析。这些组织相信，当今公司的生存将取决于它们存储、处理和分析海量信息的能力，以及对大数据挑战的掌控。

如果我们正在阅读本书，则非常有可能熟知这些挑战、熟悉 Apache Hadoop，并且知道 Hadoop 可以用于解决这些问题。本章讲述大数据带来的机遇和挑战。同时给出了 Hadoop 及其软件组件生态系统的高层概述，它们可以一起被用来构建可扩展的、分布式的数据分析解决方案。

## 1.1 当大数据遇见 Hadoop

一些组织将“人力资本”视为无形资产，这是其成功的关键因素，它们大多认为员工是其最宝贵的财富。另一个通常不会在公司资产负债表上列出的关键资产就是公司所拥有的信息。一些因素能够加强组织所拥有信息的效力，它们包括信息的可信度、体量、可访问性，以及该组织在合理的时间内利用所掌握的信息做出智能决策的能力。

我们很难掌握各种组织产生数字信息的绝对数量。IBM 指出仅仅过去两年就产生了世界上 90%的数字化数据。众多组织正在收集、产生和存储可能成为战略资源的数据。由 Michael Daconta、Leo Obrst 和 Kevin T. Smith 在十余年之前撰写的一部书籍 *The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management* (Indianapolis: Wiley，2004)中的一句格言说，“拥有最优质信息，知道如何查找它们，而且能够快速利用它们的组织才会取得胜利。”

知识就是力量。而问题是随着海量数据的不断收集，传统的数据库工具已经不能够足够快速地管理或处理这些信息了。其结果是很多组织正在被数据淹没。这些组织没能很好地利用数据，而且没有足够快速地梳理出数据中的头绪，自然也无法体会数据所呈现的威力。

术语“大数据”用于描述非常庞大的数据集，它们大到对典型的和传统的数据存储、管理、查询、分析及其他处理方法产生了挑战。大数据典型的特征在于数字信息的量级——可以来自多种源和数据格式(结构化的和非结构化的)，而且处理和分析数据可以发现其中的内涵和模式，这些有助于做出明智的决策。

大数据带来的挑战是什么？应该如何存储、处理和分析如此大量的数据才能够从信息的海洋中识别出模式和知识呢？

分析大数据需要庞大的存储和大规模的运算，这些需要大量的处理能力。随着过去十年数字信息量开始增长，各种组织尝试用不同的方法来解决这些问题。一开始，人们将焦点放在为单台机器增加更多的存储、处理能力和内存上——但很快发现，单台机器上的分析技术无法扩展。随着时间的推移，很多人意识到了分布式系统的潜力(将任务分配到多台机器上)，但这样的数据分析解决方案通常复杂性高、易出错或根本不够快。

2002 年，在开发 Nutch 项目(一个搜索引擎项目，专注于爬取、索引和搜索 Internet 网

页)的过程中，Doug Cutting 和 Mike Cafarella 正在为处理海量信息努力寻求解决方案。要实现 Nutch 对存储和处理的需求，他们知道需要一个可靠的、分布式的计算方案，该方案能够良好扩展，足够用于保存工具将要收集到的海量网站数据。

一年以后，Google 发表了关于 Google 文件系统(Google File System，GFS)和 MapReduce 的论文，它们是用于处理大数据集的算法和分布式编程平台。认识到 Google 分布式处理以及使用机器集群进行存储方案的前景后，Cutting 和 Cafarella 将此成果作为构建 Nutch 分布式平台的基础，其结果就是我们现在所熟知的 Hadoop 分布式文件系统(Hadoop Distributed File System，HDFS)和 Hadoop 的 MapReduce 实现。

2006 年，在与相同的“大数据”难题——为搜索引擎所需要的海量信息建立索引——奋战之后，且在考察了 Nutch 项目进展的基础上，Yahoo!聘请了 Doug Cutting，并且迅速决定引入 Hadoop 作为其解决搜索引擎挑战的分布式框架。Yahoo!将 Nutch 中的存储和处理部分抽出，形成 Hadoop，作为一个开源 Apache 项目，而 Nutch Web 爬虫自身仍作为独立的项目。此后不久，Yahoo!开始将 Hadoop 推广成为增强各种生产应用分析能力的手段。该平台如此高效，以致于 Yahoo!将其搜索和广告合并成一个单元以更好地利用 Hadoop。

在过去的 10 年中，Hadoop 以搜索引擎相关的需求为起点，演化为用于解决大数据挑战的最通用的计算平台之一。它正在迅速成为下一代基于数据应用的基础。市场研究公司 IDC 预测，截止 2016 年，Hadoop 将会驱动价值高达 230 亿美元的大数据市场。随着第一家以 Hadoop 为核心的公司 Cloudera 于 2008 年成立，若干基于 Hadoop 的创业公司已经吸引了数亿美元的风险投资。简单来说，众多组织都发现 Hadoop 为大数据分析提供了一个行之有效的方案。

### 1.1.1　Hadoop：直面大数据的挑战

Apache Hadoop 通过简化数据密集、高度并行的分布式应用的实现来应对大数据带来的挑战。全球诸多企业、大学和其他组织都在使用 Hadoop，它允许把分析任务划分为工作片段，并分派到上千台计算机上，提供快速的分析时间和海量数据的分布式存储。Hadoop 为存储海量数据提供了一种经济的方式。它提供了一种可扩展且可靠的机制，用一个商用硬件集群来处理大量数据。而且它提供新颖的和更先进的分析技术，允许对不同结构的数据进行复杂的分析处理。

Hadoop 从以下几个方面区别于之前的分布式方案：

- 数据预先就是分布式的。
- 为了保证可靠性和可用性，数据在整个计算机集群中进行备份。
- 数据处理力图在数据存储的位置进行，从而避免产生带宽瓶颈。

此外，Hadoop 提供一种简单的编程方式，将之前分布式实现中存在的复杂性进行抽象。其结果是，Hadoop 为数据分析提供了一种强大的机制，包含以下几个方面：

- 海量存储——Hadoop 允许应用使用成千上万的计算机和 PB 数量级的数据。在过去的十年里，计算机专家已经意识到廉价的“商用”系统可以一起用于高性能计算应

用，而这些运算以前只能由超级计算机来处理。将数以百计的“小型”计算机配置为集群，就能以相对低廉的价格获得总体上远远超过单个超级计算机的计算能力。Hadoop 可以利用超过数千台机器的集群，以企业可以接受的价格提供庞大的存储和处理能力。

- 支持快速数据访问的分布式处理——Hadoop集群提供高效存储海量数据能力的同时，还提供快速的数据访问。在Hadoop之前，并行计算应用在集群中的机器之间分布执行任务时面临着困难。这是因为此种集群执行模型依赖于需要极高I/O性能的共享数据存储。Hadoop把程序执行移向数据。将应用移向数据缓解了许多高性能挑战。此外，Hadoop应用通常被设计为顺序地处理数据。这避免了随机数据访问(磁盘寻道操作)，进一步降低了I/O负载。
- 可靠性、失效转移和可扩展性——过去，当使用机器集群时，并行应用的实现者们需要费尽心思来处理可靠性问题。尽管单一机器的可靠性相当高，但随着集群规模的增长，失效概率也在增加。在一个大集群(成千上万台机器)中，每天出现失效并不鲜见。鉴于 Hadoop 的设计和实现方式，一台机器失效(或者一组机器失效)将不会导致不一致的结果。Hadoop 检测失效并重试执行(使用不同的节点)。此外，Hadoop 内置的可扩展性允许无缝地向集群添加额外的(修理好的)服务器，并且将它们用于数据存储和程序执行。

对于多数 Hadoop 用户来说，Hadoop 最重要的特性是业务逻辑程序与框架支持代码的清晰分离。对于想要关注业务逻辑的用户，Hadoop 隐藏了框架的复杂性，为解决困难问题需要进行的复杂的、分布式的计算提供了一个简单易用的平台。

### 1.1.2 商业世界中的数据科学

Hadoop 存储和处理海量数据的能力通常与“数据科学”相关。尽管该术语由 Peter Naur 在 20 世纪 60 年代提出，但它直到最近才得到广泛接受。雪城大学的 Jeffrey Stanton 将其定义为“一个新兴领域的工作，关注于大量信息的收集、准备、分析、可视化、管理和保存”。

遗憾的是，在商业领域，该术语通常与商业分析学交替使用。但实际上，这两个学科截然不同。

商业分析师研究现有商业运营中的模式，以寻求对其进行改进。

数据科学的目标是提取数据的含义。数据科学家的工作基于数学、统计分析、模式识别、机器学习、高性能计算、数据仓库以及更多。他们基于收集到的信息进行分析，寻找趋势、统计特性和新的商业机会。

在过去的几年中，许多更熟悉数据库和编程的商业分析师已经成为数据科学家，他们使用 Hadoop 生态系统中基于 SQL 的高级工具(例如 Hive 或实时 Hadoop 查询)，并用分析的方法做出明智的商业决策。

**不只是“一个大数据库”**

我们在本书的后面会更多地体会到这一点，但在此之前，让我们打消 Hadoop 仅是为数据分析师提供的“一个大数据库”的概念。由于 Hadoop 的某些工具(例如 Hive 和实时 Hadoop 查询)为较熟悉数据库查询的人使用 Hadoop 提供了较低的进入门槛，因此有些人对 Hadoop 生态系统的了解仅限于其中一些以数据库为中心的工具。

不过，如果尝试解决的问题超出了数据分析的范畴，而涉及真正的“数据科学”问题，那么数据挖掘 SQL 会明显失去作用。大多数这种问题需要线性代数和其他一些复杂的数学应用，并不能将它们很好地翻译成 SQL。

这意味着基于 SQL 的工具尽管很重要，但也仅是使用 Hadoop 的诸多方法之一。利用 Hadoop 的 MapReduce 编程模型，不仅能够解决数据科学问题，还可以显著地简化企业级应用的创建和部署。有很多方法可以做到这一点，而且我们可以使用多种工具，这通常必须与其他功能组合，且需要软件开发技巧。例如，使用基于 Oozie 的应用协作(我们将在本书的后面学到关于 Oozie 的更多知识)，能够简化多种应用的合并，并以一种非常灵活的方式将多种工具的工作串联起来。在本书中，我们将看到在企业中使用 Hadoop 的实用技巧，以及关于如何将正确的工具应用到正确场景中的指南。

驱动当前 Hadoop 开发的目标是为数据科学家提供更好的支持。Hadoop 提供强劲的计算平台、高度可扩展性、并行执行支持，非常适合于创建新一代强大的数据科学和企业级应用。实现者能够同时使用可扩展的分布式存储和 MapReduce 处理能力。众多企业正在使用 Hadoop 解决商业问题，一些知名的案例包括：

- 增强银行和信用卡公司的诈骗检测能力——很多公司正在利用 Hadoop 检测交易欺诈。通过在大的商用硬件集群上进行分析，银行依赖 Hadoop 将分析模型应用到其客户的全部交易上，几乎可以实时地检测到正在发生的诈骗行为。
- 社交媒体市场分析——有些公司目前正在将 Hadoop 用于品牌管理、营销活动和品牌保护。通过从各种 Internet 信息源，例如博客、留言板、新闻、tweet 和社交媒体中监测、收集和聚合数据，一些公司正在使用 Hadoop 抽取和聚合有关它们产品、服务和竞争对手的信息，发现其中的模式并揭示对理解自身业务有重要意义的未来趋势。
- 用于零售商品布局的购物模式分析——零售企业正在通过Hadoop，基于商店的位置和附近人群的购物模式，确定某个特定商店里最适合进行销售的商品。
- 交通模式识别用于城市发展规划——城市发展规划通常依赖于交通模式，并以此来确定路网扩展的需求。通过在每天的不同时段监测交通并发现模式，城市开发者可以定位交通的瓶颈，这有助于他们决定是否需要建设额外的街道/便道来避免高峰时段出现交通拥堵。
- 内容优化和拼接——一些公司致力于内容优化，以支持将不同格式的内容渲染到不同的设备上。许多媒体公司需要处理大量不同格式的内容。同样，内容拼接模型需

要考虑反馈和增益。

- 网络分析和调优——对产生的大量数据进行实时分析，包括使用事务数据、网络性能数据、小区基站信息、设备级数据以及其他格式的后台数据，有助于公司降低运营成本，并提升网络的用户体验。
- 大规模数据转换——《纽约时报》需要把(从 1851 年到 1980 年所有的)1100 万篇文章由原始报纸扫描而来的图片格式转换为 PDF 文件。使用 Hadoop，能够在 24 小时内将 4TB 的扫描文章转换为 1.5TB 的 PDF 文档。

这种例子还有很多。企业正在将 Hadoop 用于制定战略决策，并且开始明智地使用它们的数据。其结果是，数据科学已经进入商业世界。

**大数据工具——不只为商业服务**

尽管这里的大多数例子关注的都是商业，但 Hadoop 在科学界和公共部门中也得到广泛应用。

科技美国基金会(Tech America Foundation)最近的一项研究指出，医学研究人员已经证明使用大数据分析方法，聚合取自癌症病人的信息，可以改善治疗效果。警察部门正在使用大数据来开发模型，预测可能会发生犯罪的时间和地点，用来降低犯罪率。相同的调查表明，能源官员正在利用大数据工具分析有关能源消耗的数据和潜在的电网故障问题。

总之，大数据分析正在被应用于发现模式和趋势，并在以前所未有的方式提高效率和指导决策制定。

## 1.2 Hadoop 生态系统

当架构师和开发人员讨论软件的时候，他们通常立刻根据特定的用途来描述软件工具。例如，他们可能会说 Apache Tomcat 是 Web 服务器，MySQL 是数据库。

不过，当遇到 Hadoop 时，事情变得有些复杂。Hadoop 包含大量工具，这些工具可以协同工作。其结果是，Hadoop 可以用于很多事情，而且人们通常基于自己使用 Hadoop 的方式来定义它。

对于一些人来说，Hadoop 是一个数据管理系统，将海量的结构化和非结构化数据聚集在一起，这些数据涉及传统企业数据栈的几乎每一个层次，其定位是在数据中心占据核心地位。对于其他一些人，Hadoop 是大规模并行执行框架，把超级计算机的能力带给大众，致力于加速企业级应用的执行。还有些人将 Hadoop 视为开源社区，专门创造用于解决大数据问题的工具和软件。由于 Hadoop 提供如此广泛的功能，可以适用于解决大量问题，因此很多人也认为 Hadoop 是基础框架。

当然，Hadoop 提供所有这些功能，因此应该将 Hadoop 归类为一个生态系统，它包含大量的组件，从数据存储到数据集成、数据处理以及数据分析师的专用工具。

## 1.3　Hadoop 核心组件

尽管 Hadoop 生态系统仍在增长，但图 1-1 展示了其核心组件。

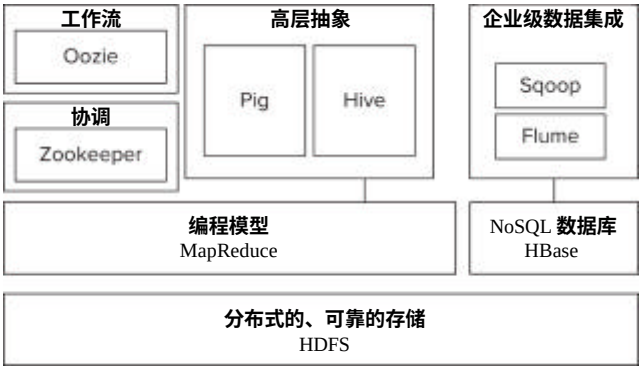



图 1-1　Hadoop 生态系统的核心组件

从图 1-1 所示框图的底部开始，Hadoop 生态系统包含以下组件：

- HDFS——Hadoop 生态系统的基础组件是 Hadoop 分布式文件系统(HDFS)。HDFS 的机制是将大量数据分布到计算机集群上，数据一次写入，但可以多次读取用于分析。它是其他一些工具的基础，例如 HBase。
- MapReduce——Hadoop 的主要执行框架即 MapReduce，它是一个用于分布式并行数据处理的编程模型，将作业分为 mapping 阶段和 reduce 阶段(因此而得名)。开发人员为 Hadoop 编写 MapReduce 作业，并使用 HDFS 中存储的数据，而 HDFS 可以保证快速的数据访问。鉴于 MapReduce 作业的特性，Hadoop 以并行的方式将处理过程移向数据，从而实现快速处理。
- HBase——一个构建在 HDFS 之上的面向列的 NoSQL 数据库，HBase 用于对大量数据进行快速读取/写入。HBase 将 Zookeeper 用于自身的管理，以保证其所有组件都正在运行。
- Zookeeper——Zookeeper 是 Hadoop 的分布式协调服务。Zookeeper 被设计成可以在机器集群上运行，是一个具有高度可用性的服务，用于 Hadoop 操作的管理，而且很多 Hadoop 组件都依赖它。
- Oozie——一个可扩展的 Workflow 系统，Oozie 已经被集成到 Hadoop 软件栈中，用于协调多个 MapReduce 作业的执行。它能够处理大量的复杂性，基于外部事件(包括定时和所需数据是否存在)来管理执行。
- Pig——对 MapReduce 编程复杂性的抽象，Pig 平台包含用于分析 Hadoop 数据集的执行环境和脚本语言(Pig Latin)。它的编译器将 Pig Latin 翻译为 MapReduce 程序序列。
- Hive——类似于 SQL 的高级语言，用于执行对存储在 Hadoop 中数据的查询，Hive 允许不熟悉 MapReduce 的开发人员编写数据查询语句，它会将其翻译为 Hadoop 中

的 MapReduce 作业。类似于 Pig，Hive 是一个抽象层，但更倾向于面向较熟悉 SQL 而不是 Java 编程的数据库分析师。

Hadoop 生态系统还包含一些用于与其他企业级应用进行集成的框架：

- Sqoop 是一个连通性工具，用于在关系型数据库和数据仓库与 Hadoop 之间移动数据。Sqoop 利用数据库来描述导入/导出数据的模式，并使用 MapReduce 实现并行操作和容错。
- Flume 是一个分布式的、具有可靠性和高可用性的服务，用于从单独的机器上将大量数据高效地收集、聚合并移动到 HDFS 中。它基于一个简单灵活的架构，提供流式数据操作。它借助于简单可扩展的数据模型，允许将来自企业中多台机器上的数据移至 Hadoop。

在图 1-1 所示核心组件的基础上，Hadoop 生态系统还在增长，并不断提供更加新颖的功能和组件，例如：

- Whirr——它是一组静态库，让用户能够在 Amazon EC2、Rackspace 或任何虚拟基础架构之上构建 Hadoop 集群。
- Mahout——一个机器学习和数据挖掘的库，提供用于聚类、回归测试和统计建模常见算法的 MapReduce 实现。
- BigTop——一个正式的流程和框架，用于对 Hadoop 的子项目和相关组件进行打包和互操作性测试。
- Ambari——该项目致力于简化 Hadoop 的管理，提供对 Hadoop 集群进行供应、管理和监控的支持。

每天都有更多的成员加入到 Hadoop 大家庭。仅在撰写本书的过程中就有三个新的 Apache Hadoop 孵化器项目被添加进来！

**项目加入 Apache 的过程**

如果不了解 Apache 软件基金会(Apache Software Foundation)的工作方式，或者对各种项目以及它们彼此之间的关系存在疑惑的话，那么可以说 Apache 以一种有组织的方式支持项目的创建、成熟和退出。Apache 的会员由个人组成，并且他们一起来管理该组织。

项目均以“ 孵化器 ”项目开始。Apache 孵化器的创建就是为了帮助新项目加入 Apache。它提供监管和检查，并“ 过滤 ”创建新项目和为已有项目创建子项目的提议。孵化器援助被培养的项目，评估项目的成熟度，并负责让项目从孵化器中“ 结业 ”，加入到 Apache 项目或子项目中。孵化器也可能基于各种原因让项目从中退出。

要查看孵化器中项目(当前的、已结业的、暂停的和退出的)的完整列表，请访问 http://incubator.apache.org/projects/index.html。

当前大多数 Hadoop 出版物不是专注于描述该生态系统中的单一组件，就是 Hadoop 中业务分析工具(例如 Pig 和 Hive)的使用方法。尽管这些主题很重要，但它们会迅速地集中到一个话题上，即帮助架构师们构建基于 Hadoop 的企业级应用，或是复杂的分析应用。

## 1.4　Hadoop 发行版

尽管 Hadoop 是一组 Apache(以及现在的 GitHub)项目集合，但目前很多公司开始致力于帮助人们使用 Hadoop。这些公司大都从打包 Apache Hadoop 发行版开始，确保所有软件能够协同工作，并提供支持。而现在它们开始开发额外的工具来简化 Hadoop 的使用，并扩展其功能。有些扩展还涉及所有权，并被当成差异化的功能。有些成了 Apache Hadoop 家庭中新项目的基础。还有些是使用 Apache 2 许可证的 GitHub 开源项目。尽管所有这些公司均从 Apache Hadoop 发行版开始，但它们对一些问题的见解是不同的，包括 Hadoop 到底是什么、应该向什么方向发展以及如何完成其使命。

这些公司最大的不同之一在于对 Apache 代码的使用。除 MapR 之外，所有人都认为 Hadoop 应该由 Apache 项目所编写的代码来定义。相反，MapR 认为 Apache 代码是一个参考实现，并且基于 Apache 提供的 API 给出了自己的实现。此种方式让 MapR 可以引入许多新特性，尤其是围绕着 HDFS 和 HBase，使得这两个基本的 Hadoop 存储机制具备了更高的可靠性和性能。其发行版还额外引入了网络文件系统(NFS)对 Hadoop 的高速访问，这极大地简化了 Hadoop 与其他企业级应用的集成。

Amazon 和 Microsoft 发布了两个有趣的 Hadoop 发行版。两者均提供了可以运行在各自云(Amazon 与 Azure)上且预先打包好的 Hadoop 版本，以做到平台即服务(Platform as a Service，PaaS)。它们同时都提供了扩展，允许开发者除使用 Hadoop 原生的 HDFS 之外，还可以将 HDFS 映射到各自的存储机制(Amazon 的 S3，Azure 的 Windows Azure 存储)。Amazon 还提供了将 HBase 内容保存到 S3 和从 S3 导出内容到 HBase 的功能。

表 1-1 列出了主流 Hadoop 发行版的主要特性。

表 1-1　不同的 Hadoop 厂商

| 厂　商 | Hadoop 特性 |
| --- | --- |
| Cloudera CDH、Manager 和 Enterprise | 基于Hadoop 2，CDH(编写本书时为4.1.2版本)包括HDFS、YARN、HBase、MapReduce、Hive、Pig、Zookeeper、Oozie、Mahout、Hue和其他开源工具(包括实时查询引擎——Impala)。Cloudera Manager免费版包括CDH所有组件，加上一个支持最多50个集群节点的基础Manager。Cloudera Enterprise将CDH与一个更复杂的Manager组合，支持不限量的集群节点、主动监控和额外的数据分析工具 |
| Hortonworks Data Platform | 基于 Hadoop 2，该发行版(编写本书时为 2.0 Alpha 版本)包括 HDFS、YARN、HBase、MapReduce、Hive、Pig、HCatalog、Zookeeper、Oozie、Mahout、Hue、Ambari、Tez 和 Hive 的实时版(Stinger)以及其他开源工具。为 Hortonworks 提供高可用性支持、高性能 Hive ODBC 驱动和用于大数据的 Talend Open Studio |
| MapR | 基于 Hadoop 1，该发行版(编写本书时为 M7 版本)包括 HDFS、HBase、MapReduce、Hive、Mahout、Oozie、Pig、ZooKeeper、Hue 和其他开源工具。它还包括直接 NFS 访问、快照和用于“高可用性”的镜像，有版权的 HBase 实现(与 Apache API 完全兼容)，以及 MapR 管理控制台 |

(续表)

| 厂　商 | Hadoop 特性 |
| --- | --- |
| IBM InfoSphere BigInsights | 编写本书时，它基于 Hadoop 1 且有两个可用版本。基础版包括 HDFS、HBase、MapReduce、Hive、Mahout、Oozie、Pig、ZooKeeper、Hue 和一些其他开源工具，以及 IBM 安装程序和数据访问工具的基础版本。企业版增加了复杂的作业管理工具、与主要数据源相互集成的数据访问层和 BigSheets(类似于电子表格的界面，用于在集群中操作数据) |
| GreenPlum 的 Pivotal HD | 编写本书时，它基于 Hadoop 2，包括 HDFS、MapReduce、Hive、Pig、HBase、Zookeeper、Sqoop、Flume 和其他开源工具。有版权的高级数据库服务(ADS)，基于 HAWQ 技术，扩展了 Pivotal HD Enterprise，增加了丰富的、经过验证的、并行的 SQL 处理能力 |
| Amazon Elastic MapReduce (EMR) | 编写本书时，它基于 Hadoop 1。Amazon EMR 是一个 Web 服务，允许用户简单有效地处理海量数据。它使用托管的 Hadoop 框架，该框架运行在 Amazon Elastic Compute Cloud(Amazon EC2)和 Amazon Simple Storage Service(Amazon S3)基于 Web 的基础设施之上。它包括 HDFS(带有 S3 支持)、HBase(有备份恢复的版权)、MapReduce、Hive(增加对 Dynamo 的支持)、Pig 和 Zookeeper |
| Windows Azure HDInsight | 基于 Hortonworks Data Platform(Hadoop 1)，它运行在 Azure 云上。它集成了 Microsoft 管理控制台，用于简易部署和与 System Center 集成。它可以通过 Hive Excel 插件与 Excel 集成。可以通过 Hive Open Database Connectivity(ODBC)驱动程序与 Microsoft SQL Server 分析服务(SSAS)、PowerPivot 和 Power View 集成。Azure Marketplace 允许客户连接到数据、智能挖掘算法和用户防火墙之外的人。Windows Azure Marketplace 提供来自可信第三方提供者的数百个数据集 |

当然，众多发行版的存在可能会让人困惑，“我该使用哪个发行版呢？”当为公司/部门选取特定发行版时，需要考虑如下因素：

- 技术细节——应该包括 Hadoop 版本、包含的组件、涉及所有权的功能组件等。
- 易于部署——应该有可用的工具包来管理部署、版本更新、补丁等。
- 易于维护——涉及集群管理、多中心支持、灾难恢复支持等。
- 成本——包括实现某个特定版本所需要的费用、计费模式和许可证。
- 企业应用集成支持——包括对 Hadoop 应用与企业的其他应用进行集成的支持。

选择某个特定的发行版取决于计划使用 Hadoop 来解决的特定问题集合。本书的讨论致力于做到不评价各个发行版的优劣，因为作者们意识到每个发行版都有自己的价值。

## 1.5 使用 Hadoop 开发企业级应用

面对着大数据带来的挑战，我们需要重新思考为数据分析构建应用的方法了。构建应用的传统方式是将数据保存在数据库中，而这通常无法适用于大数据处理。其原因如下：

- 传统应用建立在事务型数据库访问的基础上，这一点 Hadoop 并不支持。
- 鉴于Hadoop中保存数据的量级，实时访问仅对存储在集群上的局部数据具有可行性。
- Hadoop的海量数据存储能力允许将数据集的各个版本保存起来，与传统的覆盖数据的方式截然相反。

其结果是，典型的基于 Hadoop 的企业级应用看上去类似于图 1-2 所示。在这些应用中，有数据存储层、数据处理层、实时访问层和安全层。实现这样一个架构不仅需要了解相关 Hadoop 组件的 API，而且还要了解它们的功能和局限性，以及每个组件在整个架构中所扮演的角色。

如图 1-2 所示，数据存储层由源数据和中间数据两部分组成。源数据是可以从外部数据源获取的数据，外部数据源包括企业级应用、外部数据库、执行日志和其他数据源。中间数据是 Hadoop 执行得到的结果。它可以用于 Hadoop 实时应用，也可以发送给其他应用和最终用户。

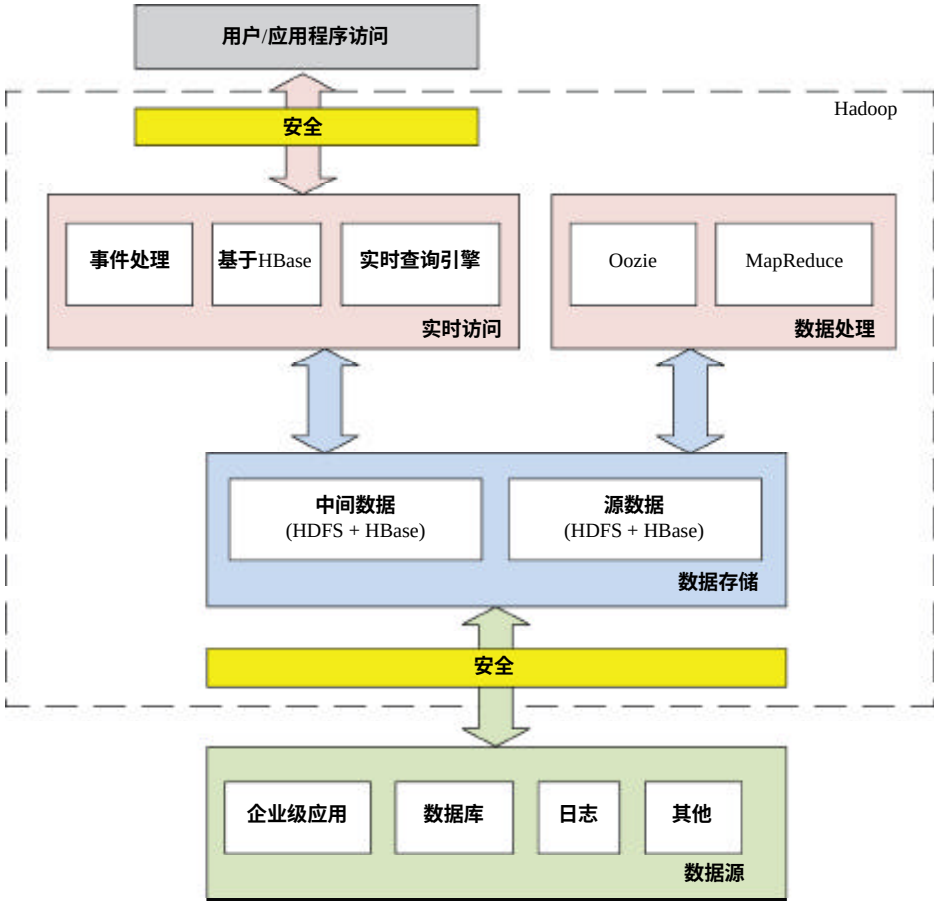



图 1-2　概念上的 Hadoop 企业级应用

可以使用不同的机制将源数据传输到 Hadoop，包括 Sqoop、Flume、直接将 HDFS 挂载为网络文件系统(NFS)以及 Hadoop 实时服务和应用。在 HDFS 中，新数据不会覆盖已有

数据，而是创建数据的新版本。了解这点很重要，因为 HDFS 是“一次性写入”文件系统。

对于数据处理层来说，Oozie 用于组合 MapReduce 作业，处理源数据并将其转换为中间数据。与源数据不同，中间数据不是版本化的，而是会被覆盖的，因此只存在有限数量的中间数据。

对于实时访问层来说，Hadoop 实时应用同时支持对数据的直接访问和基于数据集的程序执行。这些应用可以用于读取基于 Hadoop 的中间数据以及将源数据保存到 Hadoop 中。它们还可以用于服务用户，以及将 Hadoop 与企业的其他应用进行集成。

由于源数据和中间数据的清晰分离(源数据用于存储和初步处理，中间数据用于发送和集成)，该架构没有任何对事务的要求，允许开发人员构建几乎任意复杂的应用。同时，中间预处理显著减少了所需要的数据量，这让实时数据访问具备了可行性。

### Hadoop 扩展性

尽管许多出版物都在强调 Hadoop 为业务开发人员隐藏了基础框架的复杂性，但要知道 Hadoop 的可扩展性并没有得到足够的宣传。

Hadoop 的实现方式从设计上允许开发者简单无缝地将新功能合并到 Hadoop 执行中。Hadoop 支持为 MapReduce 的不同执行阶段显式指定类名，这使得开发者能够将执行与特定的问题需求进行适配，从而确保每个作业都以最低的成本和最高的性能来执行。

自定义 Hadoop 执行主要包括以下几点：

- 自定义 Hadoop 并行化问题执行的方式，包括执行的划分方式和位置
- 支持新的输入数据类型和位置
- 支持新的输出数据类型
- 自定义输出数据的位置

本书用大量篇幅专门描述此类自定义的方式，以及实用的实现方法。这些全部基于作者们的劳动成果。

随后我们将发现，本书涵盖了图 1-2 所示的基于 Hadoop 企业级应用的所有层面。

第 2 章描述构建数据层的方法。我们会学习到用于构建数据层的可选方案，包括 HDFS 和 HBase(两者的结构和 API)。还将看到有关两者的一些对比分析，以及关于如何选择某个方案、或是使用两者组合的指南。该章同时涵盖了 Avro——Hadoop 的全新序列化/封送处理框架，并介绍了它在存储和访问数据时所扮演的角色。最后，我们会学到 HCatalog，以及它在广告和数据访问方面的用法。

有关数据处理的描述占据了本书的大部分篇幅。对于应用的数据处理部分，我们会发现作者们推荐使用 MapReduce 和 Oozie。

### 本书为何如此关注 MapReduce 代码

可能此时你正在问为什么本书将大量的焦点放在 MapReduce 代码上，而非重点关注可以简化 MapReduce 编程的高级语言？我们可以在网络上和 Hadoop 社区中找到大量关于此

主题的类似讨论。此种讨论中抛出的一个观点就是 MapReduce 代码较之提供相同功能的 Pig 代码通常更大(在代码行数上)。尽管这是个不争的事实，但还是要考虑一些额外的事情：

- 并不是所有事情都可以用高级语言来表达。有些事情其实需要使用老式的 Java 代码来实现。
- 如果正在编写的代码仅需要执行一次，那么代码行数对我们来说可能很重要。但如果正在编写企业级应用，则需要考虑其他因素，包括性能、可靠性和安全性。通常来说，使用 MapReduce 代码更容易实现这些特性，因为它提供更多的选项。
- 通过支持自定义执行，MapReduce 支持用户进一步提升应用的性能、可靠性和安全性。

在第 3 章中，我们会学到MapReduce架构、它的主要组件和编程模型。该章还涵盖了MapReduce应用设计、一些设计模式和常见的MapReduce注意事项。该章还描述了MapReduce执行的实际工作原理。如前所述，MapReduce最强的特性之一是能够支持自定义执行。第 4 章包含这些自定义选项的细节和丰富的真实案例。第 5 章通过演示构建可靠MapReduce应用的方案，圆满结束了对MapReduce的讨论。

尽管 MapReduce 自身功能强大，但实用的解决方案通常需要将多个 MapReduce 应用组合起来，这会引入相当多的复杂性。使用 Hadoop 的 Workflow/协作引擎可以显著地简化 MapReduce 应用的集成。

### Oozie 的价值

Oozie 是一个被严重低估的 Hadoop 组件。极少(如果有的话)Hadoop 书籍讨论这一极其有价值的组件。本书不仅演示了 Oozie 能做什么，而且还提供了一个完备的例子，向读者展示了如何将 Oozie 的功能用于解决实际问题。同 Hadoop 的其他组件类似，Oozie 的功能具有极高的可扩展性。我们会学到扩展 Oozie 功能的不同方法。

最后，一个没有得到足够重视的挑战是 Hadoop 执行与其他企业处理的集成。使用 Oozie 来协调 MapReduce 应用并利用 Oozie API 暴露这些 Hadoop 处理过程，我们便已经为自己找到了一种将 Hadoop 处理与其他企业处理过程进行集成的优雅方案。

第 6 章描述了 Oozie 是什么、它的架构、主要组件、编程语言和总体的 Oozie 执行模型。为更好地解释每个 Oozie 组件的功能和角色，第 7 章给出了一个使用 Oozie 的完备的实际应用。第 8 章完成对 Oozie 的讨论，其中展示了一些高级的 Oozie 特性，包括自定义 Workflow 活动、动态 Workflow 生成和 uber-jar 支持。

实时访问层在第 9 章讨论。该章从业界使用的实时 Hadoop 应用示例开始，之后给出了此种实现对整体架构的要求。接着，我们会学习到构建此类实现的三种主要方法——基于 HBase 的应用、实时请求和基于流的处理。该章涵盖了整体架构，并提供了基于 HBase 实时应用的两个完整示例。接下来描述了一种实时查询架构，并讨论了两个具体实现——Apache Drill 和 Cloudera 的 Impala。我们还会看到实时查询与 MapReduce 的对比。最后，我们会学到基于 Hadoop 的复杂事件处理，以及两个具体实现——Storm 和 HFlame。

开发企业级应用需要大量信息安全相关的计划和策略。第 10 章聚焦于 Hadoop 的安全模型。

鉴于云计算的优势，很多组织都乐于在云上运行其 Hadoop 实现。第 11 章关注在 Amazon 云上使用 EMR 实现运行 Hadoop 应用，并讨论可用于补充 Hadoop 功能的其他 AWS 服务(例如 S3)。其中涵盖了在云中运行 Hadoop 的不同方法，并讨论不同方法的取舍和最佳实践。

除 Hadoop 自身安全之外，Hadoop 实现也经常与其他企业级组件集成——数据经常被导入 Hadoop 或从 Hadoop 导出。第 12 章聚焦于如何以最佳方式保障使用 Hadoop 的企业级应用的安全，并提供了增强企业级 Hadoop 应用整体安全性的示例和最佳实践。

## 1.6 小结

本章提供了大数据与 Hadoop 之间关系的高层次概述。我们已经了解了大数据、它的价值和它给企业带来的挑战，包括数据的存储和处理。本章已经把 Hadoop 和它的一些历史介绍给了读者。了解了 Hadoop 的特性之后，我们会明白为何 Hadoop 如此适用于大数据处理。本章还展示了 Hadoop 主要组件的概况，并给出了 Hadoop 如何简化数据科学和企业级应用构建的示例。

我们已经了解到一些关于主要 Hadoop 发行版的知识，以及很多组织倾向于选择特定厂商发行版的原因——这是因为它们不想处理单个 Apache 项目之间的兼容问题，或是可能需要厂商的支持。

最后，本章讨论了一种层次化的方案模型，它可以用于开发基于 Hadoop 的企业级应用。

第 2 章开始进入 Hadoop 的使用细节，并讨论如何存储数据。

# 第 2 章

# Hadoop 数据存储

**本章内容提要**

- **了解** Hadoop **分布式文件系统**(Hadoop Distributed File System，HDFS)
- **理解** HBase
- **为自己的应用选择最适合的数据存储**

wrox.com **为本章提供的代码下载**

**可以在**www.wiley.com/go/prohadoopsolutions**的**Download Code**选项卡中找到**wrox.com **为本章提供的代码下载。这些代码在**Chapter 2 Code**下载文件中，并依据本章中的名称逐一进行命名。**

Hadoop **高效处理数据的基础是其数据存储模型。本章讨论** Hadoop **中存储数据的不同方案，特别是** Hadoop **分布式文件系统**(HDFS)**和** HBase**。本章探讨每种方案的优势与不足，并给出一个决策树，用于针对给定的问题选择最优方案。我们还会学到关于** Apache Avro **的知识，它是用于数据序列化的** Hadoop **框架，可以与基于** Hadoop **的存储紧密集成。本章还涵盖了不同的数据访问模型，这些模型可以在** Hadoop **存储之上实现。**

## 2.1 HDFS

HDFS **是** Hadoop **的分布式文件系统的实现。它的设计目标是存储海量数据，并为分布在网络中的大量客户端提供数据访问。要想成功使用** HDFS**，首先必须了解其实现方式和工作原理。**

### 2.1.1 HDFS **架构**

HDFS **的设计思想基于** Google **文件系统**(Google File System，GFS)**。它的实现解决了**

存在于众多分布式文件系统(例如网络文件系统(NFS))中的大量问题。具体来说，HDFS 的实现解决了以下问题：

- 能够保存非常大的数据量(TB 级或 PB 级)，HDFS 的设计支持将数据散布在大量机器上，而且与其他分布式文件系统(例如 NFS)相比，它支持更大的文件尺寸。
- 可靠地存储数据，为应对集群中单台机器的故障或不可访问，HDFS 使用数据复制的方法。
- 为更好地与 Hadoop 的 MapReduce 集成，HDFS 允许数据在本地读取并处理(数据本地性问题会在第 4 章中详细讨论)。

HDFS为获得可扩展性和高性能而进行的的设计也是有代价的。HDFS只适用于特定类型的应用——它不是通用的分布式文件系统。大量额外的决策和取舍主导了HDFS的架构和实现，它们包括以下方面：

- HDFS 针对高速流式读取性能做了优化，随之而来的代价是降低了随机查找性能。这意味着，如果应用程序需要从 HDFS 读取数据，那么应该避免查找，或者至少最小化查找的次数。顺序读取是访问 HDFS 文件的首选方式。
- HDFS 仅支持一组有限的文件操作——写入、删除、追加和读取，不支持更新。它假定数据一次性写入 HDFS，然后多次读取。
- HDFS 不提供本地数据缓存机制。缓存的开销非常大，以至于应该单纯地从源文件重新读取数据，而这对于通常顺序读取大数据文件的应用程序来说并不是个问题。

HDFS 被实现为一种块结构的文件系统。如图 2-1 所示，单个文件被拆分成固定大小的块，而这些块保存在 Hadoop 集群上。一个文件可以由多个块组成，这些块存储在不同的 DataNode(集群中的单独一台机器)上；对于每个块来说，保存在哪个 DataNode 上是随机选取的。其结果是，访问某个文件通常需要访问多个 DataNode，这意味着 HDFS 支持的文件大小远远超过单机的磁盘容量。

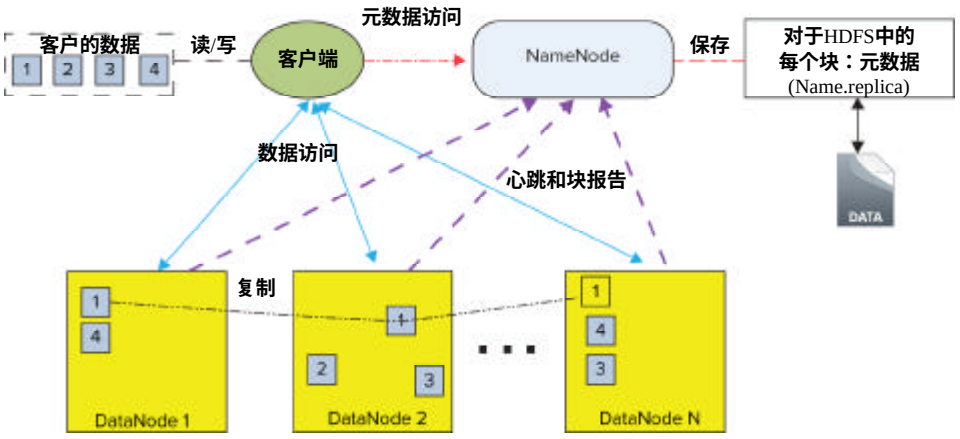



图 2-1　HDFS 架构

DataNode 在其本地文件系统上以单独文件的形式保存各个 HDFS 数据块，但它并不了解 HDFS 文件自身的信息。为了进一步提高吞吐量，DataNode 不会将所有文件创建在相同

的文件夹中。相反，它使用启发式算法来确定每个目录下最优的文件数目，并适当地创建子目录。

对这种块结构文件系统的要求之一是能够可靠地存储、管理和访问文件元数据(关于文件和块的信息)，并提供对元数据存储的快速访问。与HDFS文件自身(一次写入、多次读取的访问模型)不同，元数据结构可以被大量客户端同时修改。始终保持这些信息的同步很重要。HDFS通过引入称为NameNode的专用特殊机器来解决该问题，NameNode上保存了整个集群文件系统的所有元数据。这意味着HDFS实现了一种主/从架构。单独的NameNode(即主服务器)管理文件系统名称空间并规范客户端对文件的访问。集群中单独主控节点的存在极大地简化了系统架构。NameNode充当了单独仲裁者和全部HDFS元数据存储库的角色。

由于每个文件元数据的量相对较小(仅包含文件名、访问权限和各个块的位置)，因此 NameNode 将所有元数据保存在内存中，从而保证快速的随机访问。元数据存储在设计上很紧凑。其结果是，拥有 4GB 内存的 NameNode 能够支持巨大数目的文件和目录。

元数据存储也是持久化的。整个文件系统名称空间(包括块到文件的映射以及文件系统属性)包含在一个名为FsImage的文件中，该文件保存在NameNode的本地文件系统中。NameNode还使用事务日志来持久化地记录发生在文件系统元数据(元数据存储)中的每一次改动。该日志保存在NameNode本地文件系统上的EditLog文件中。

**从属 NameNode**

如前所述，HDFS 的实现基于主/从架构。一方面，这种方式极大地简化了 HDFS 的整体架构。但另一方面，这也产生了一个故障单点，即 NameNode 失效事实上意味着 HDFS 失效。为从某种程度上缓解这一问题，Hadoop 实现了从属 NameNode。

从属 NameNode 不是一个“备用 NameNode”。它不能接管主 NameNode 的功能。它为主 NameNode 提供检查点机制。除了保存 HDFS NameNode 的状态之外，它还在磁盘上维护两个数据结构，用于持久化存储当前的文件系统状态；这两个数据结构分别是一个镜像文件和一个编辑日志。镜像文件代表 HDFS 元数据在某个时间点的状态，而编辑日志是一个事务日志(与数据库体系结构中的日志相比)，记录了自从镜像文件创建之后文件系统元数据的每一次更改。

在启动(重启)的过程中，NameNode 通过读取镜像文件并接着重播编辑日志来重建当前状态。显然，编辑日志越大，重播它所需要的时间就越长，因此启动 NameNode 所需要的时间也越长。为提升 NameNode 的启动性能，编辑日志会定期轮转，即通过将编辑日志应用于现有镜像来产生新的镜像文件。这个操作相当耗费资源。为最小化创建检查点所产生的影响并简化NameNode的功能，检查点的创建通常由运行在独立机器上的从属 NameNode 守护进程来完成。

作为检查点创建的结果，从属NameNode以前面所述镜像文件的格式保存了主NameNode持久化状态的一个(过期的)副本。在编辑日志保持相对较小的情况下，可以使用从属NameNode恢复文件系统的状态。要知道在这种情况下会有一定数量的元数据(以及对应的内容数据)丢失，因为保存在编辑日志中的最新更改是不可用的。

有一项正在进行的工作是创建一个真正的备份 NameNode，它能够在主节点发生故障时实现接管。本章后面会讨论在 Hadoop 2 中引入的 HDFS 高可用性实现。

为保持 NameNode 的内存占用可控，HDFS 块的默认大小为 64MB——从数量级上大于多数其他块结构文件系统的块大小。大数据块的额外优势是允许 HDFS 将大量数据顺序地存储在磁盘上，以支持对数据进行高速流式读取。

HDFS 中较小的块

关于Hadoop的误解之一是认为较小的块(小于块大小)在文件系统中仍然会占用整个块。事实并非如此。较小的块只占用它们所需要大小的磁盘空间。

但这并不意味着大量小文件可以有效地利用 HDFS。无论块大小是多大，其元数据在 NameNode 中所占的内存完全相同。其结果是，数目众多的 HDFS 小文件(小于块大小)会占用大量的 NameNode 内存，从而给 HDFS 的可扩展性和性能带来负面影响。

在现实系统中，几乎不可能避免出现较小的HDFS块。比较大的可能性是某个给定的HDFS文件会占用一些完整的块和一些较小的块。这会成为一个问题吗？考虑到大多数HDFS文件会相当庞大，整个系统中此类较小块的数量将相对较少，因此通常没有问题。

HDFS 文件组织的缺点是一个文件需要多个 DataNode 来提供服务，这意味着如果这些机器中的任何一台失效的话，该文件就变得不可用。为了避免此问题，HDFS 在多台机器(默认为三台)上对每个块进行复制。

HDFS中数据复制的实现是写操作的一部分，采用数据管道的形式。当客户端向HDFS文件写入数据时，数据首先写入到本地文件。当本地文件积累到一整块数据时，客户端会向NameNode请求用于保存块副本的DataNode列表。客户端接着以 4KB为单位将数据块从其本地存储写入到第一个DataNode(参见图 2-1)。该DataNode在本地文件系统中保存接收到的块，并将这部分数据转发到列表中的下一个DataNode。下一个接收到数据的DataNode重复相同的操作，直到副本集合中的最后一个节点接收到数据。这个DataNode仅在本地保存数据，不再进行转发。

在块写入的过程中，如果某个 DataNode 失效了，那么它将被从管道中移除。在这种情况下，当前块的写操作完成之后，NameNode 会重新复制该块，以补偿由于 DataNode 失效而造成的副本缺失。当文件关闭时，临时本地文件中的剩余数据会通过管道发送到 DataNode。客户端接下来通知 NameNode 文件已关闭。此时，NameNode 将此文件创建操作提交到持久化存储。如果 NameNode 在文件关闭之前发生崩溃，那么该文件会丢失。

块大小和复制因子的默认值由 Hadoop 配置指定，但可以以逐个文件为基础覆盖该默认值。应用程序可以指定块大小、副本数量以及特定文件在其创建时的复制因子。

HDFS 最强大的特性之一就是对副本放置的优化，这对 HDFS 的可靠性和性能来说至关重要。NameNode 负责做出块复制相关的所有决定，它会周期性地(每 3 秒钟)接收来自每个 DataNode 的心跳和块报告。心跳用于确保 DataNode 功能正常，而块报告可以验证

DataNode 上的块列表与 NameNode 中的信息是否一致。DataNode 在启动时要做的首要事情之一就是将块报告发送到 NameNode。这允许 NameNode 迅速构建出整个集群中块分布的图景。

HDFS 数据复制的最重要特性叫做机架感知。运行在计算机集群上的大型 HDFS 实例通常跨越许多个机架。通常情况下，相同机架上机器之间的网络带宽(以及与之相关联的网络性能)远大于不同机架上机器之间的网络带宽。

NameNode 通过 Hadoop 机架感知进程确定各个 DataNode 所属的机架 ID。一种简单的策略是将各个副本分别放置于不同的机架上。这种策略能够在整个机架失效时防止数据丢失，且将副本均匀地分布到集群中。它也允许在读取数据时使用源自多个机架的带宽。但由于在这种情况下，写操作必须将块传输到多个机架上，因此写入性能会受影响。

机架感知策略的一个优化方案是让占用的机架数少于副本数，以减少跨机架写入流量(进而提高写入性能)。例如，当复制因子为 3 时，将两个副本置于同一个机架上，并将第三个副本放在另一个不同的机架上。

为最小化全局带宽消耗和读取时延，HDFS 会尝试从最靠近读取者的副本获取数据，以满足读请求。如果某个副本存在于与读取者节点相同的机架上，那么该副本将被用来响应读请求。

如前所述，每个DataNode会周期性地向NameNode发送心跳消息(参见图2-1)，NameNode利用这些消息来发现DataNode失效(基于心跳消息的缺失)。NameNode将最近没有心跳的DataNode标识为死机，并且不再向其派发任何新的I/O请求。由于位于死机DataNode上的数据对HDFS不再可用，因此DataNode死机可能造成某些块的复制因子降低到其设定值之下。NameNode随时跟踪需要重新复制的块，并在必要时启动复制操作。

类似于多数其他现有文件系统，HDFS 支持传统的层次化文件组织结构。它支持某个目录下文件的创建和删除、在不同目录之间移动文件等。它还支持用户配额和读/写权限。

### 2.1.2　使用 HDFS 文件

既然了解了 HDFS 的工作方式，本小节就开始讨论如何使用 HDFS 文件。用户应用程序使用 HDFS 客户端访问 HDFS 文件系统——HDFS 客户端是一个库，暴露了 HDFS 文件系统接口，这些接口隐藏了前面所述的 HDFS 实现中的大部分复杂性。用户应用程序不必了解文件系统元数据和存储位于不同的服务器上，或者说不必了解块有多个副本。

**访问 HDFS**

Hadoop 提供了访问 HDFS 的多种方法。FileSystem(FS) shell 命令提供了丰富的操作，支持访问和操作 HDFS 文件。这些操作包括查看 HDFS 目录、创建文件、删除文件、复制文件等。此外，HDFS 的典型安装会配置一台 Web 服务器，通过可配置的 TCP 端口暴露 HDFS 名称空间。这允许用户通过使用 Web 浏览器遍历 HDFS 名称空间并查看其中文件的内容。由于本书的焦点是编写 Hadoop 应用程序，因此后续讨论将集中于 HDFS Java API。

访问 HDFS 要依赖一个 FileSystem 对象实例。FileSystem 类是用于通用文件系统的一个抽象基类(除了 HDFS 之外，Apache 提供用于其他文件系统的 FileSystem 对象实现，包括 KosmosFileSystem、NativeS3FileSystem、RawLocalFileSystem 和 S3FileSystem)。它可能被实现为分布式文件系统，或者被实现为使用本地已连接磁盘的“本地”版本。本地版本用于小规模的 Hadoop 实例和测试。所有可能会使用 HDFS 的用户代码都要用到 FileSystem 对象。

我们可以通过向构造函数传递一个新的 Configuration 对象来创建 FileSystem 对象实例。假定 Hadoop 配置文件(hadoop-default.xml 和 hadoop-site.xml)在类路径中可用，则代码清单 2-1 中所示的代码创建了一个 FileSystem 对象(如果仅在 Hadoop 集群的一个节点上完成执行，那么配置文件总是可用的。如果要在远程机器上完成执行，那么必须显式地将配置文件加入到应用程序的类路径中)。

**代码清单 2-1：创建 FileSystem 对象**

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
```

另外一个重要的 HDFS 对象是 Path，它代表文件系统中文件或目录的名字。可以使用一个代表 HDFS 上文件/目录位置的字符串来创建 Path 对象。将 FileSystem 和 Path 对象进行组合，可以实现对 HDFS 文件和目录的多种编程操作。代码清单 2-2 展示了一个例子。

**代码清单 2-2：操作 HDFS 对象**

```
Path filePath = new Path(file name);
……
if(fs.exists(filePath))
   //do something
……
if(fs.isFile(filePath))
   //do something
……
Boolean result = fs.createNewFile(filePath);
……
Boolean result = fs.delete(filePath);
……
FSDataInputStream in = fs.open(filePath);
FSDataOutputStream out = fs.create(filePath);
```

代码清单 2-2 中的最后两行展示了如何基于文件路径创建 FSDataInputStream 和 FSDataOutputStream 对象。这两个对象是 Java I/O 包中 DataInputStream 和 DataOutputStream 的子类，这意味着它们支持标准的 I/O 操作。

**注意：**

除 DataInputStream 之外，FSDataInputStream 还实现了 Seekable 和 PositionedReadable 接口，从而实现了用于查找和从给定位置读取的方法。

至此，应用程序能够以和读/写本地数据系统相同的方式从 HDFS 读取数据，或者将数据写入到 HDFS。

**写租约**

当以写入方式打开文件时，打开文件的客户端会被授予一个独占的写租约，用于保护该文件。这意味着在该客户端完成操作之前，没有其他客户端能够写入该文件。为确保没有“不辞而别的”客户端占用租约，租约会周期性地过期。租约的使用有效地确保了两个应用程序不会同时写入一个给定文件(与数据库中的写锁相似)。

租约的生命周期由软限制和硬限制来限定。在软限制期间，写入者可以独占地访问文件。如果软限制过期且客户端没能关闭文件或者更新租约(通过向 NameNode 发送心跳)，那么其他客户端可以抢占该租约。如果硬限制(一小时)过期且客户端没能更新租约，那么 HDFS 认为客户端已经退出，并自动替写入者关闭文件，然后恢复租约。

写入者的租约并不会阻止其他客户端读取该文件。一个文件可能有多个并发的读取者。

### 2.1.3　Hadoop 特定的文件类型

除“普通”文件之外，HDFS 还引入了一些特定的文件类型(例如 SequenceFile、MapFile、SetFile、ArrayFile 和 BloomMapFile)，它们提供更加丰富的功能，且通常会简化数据处理。

SequenceFile 提供了用于二进制键/值对的持久化数据结构。这里，键和值的不同实例必须代表相同的 Java 类，但大小可以不同。类似于其他 Hadoop 文件，SequenceFile 只能追加。

当使用普通(文本或二进制)文件保存键/值对(MapReduce使用的典型数据结构)时，数据存储并不知道键和值的布局，它必须在通用存储之上的读取器中实现。SequenceFile的使用提供了一种存储机制，原生支持键/值结构，从而使用了这种数据布局的实现更加简单。

SequenceFile有三种可用格式：无压缩、记录压缩和块压缩。前两种以基于记录的格式保存(如图 2-2 所示)，而第三种使用基于块的格式(如图 2-3 所示)。


图 2-2　基于记录的 SequenceFile 格式


图 2-3　基于块的 SequenceFile 格式

对于序列文件来说，特定格式的选取决定了该文件在硬盘驱动器上的大小。块压缩的文件通常最小，而无压缩的最大。

在图 2-2 和图 2-3 中，头部(header)包含了有关 SequenceFile 的一般信息，如表 2-1 所示。

表 2-1　SequenceFile 头部(header)

| 字　段 | 描　述 |
| --- | --- |
| Version | 一个 4 字节的数组，包含三个字符(SEQ)和一个序列文件版本号(4 或 6)。当前使用的是版本 6。为向后兼容而支持版本 4 |
| Key Class | 键的类名，使用读取器提供的键的类名对其进行验证 |
| Value Class | 值的类名，使用读取器提供的值的类名对其进行验证 |
| Compression | 键/值压缩标志 |
| Block Compression | 块压缩标志 |
| Compression Codec | CompressionCodec 类。该类仅在键/值或块压缩标志为 true 时使用。否则，该值被忽略 |
| Metadata | 元数据(可选)是一个键/值对列表，可以用于向文件添加用户特定的信息 |
| Sync | sync 标识符 |

**注意：**

同步(sync)是个专门的标记，用于在 SequenceFile 内部更快速地进行查找。sync 标记在 MapReduce 实现中还有一个特殊作用——数据分割只能在 sync 边界处进行。

如表 2-2 所示，记录中包含了键和值的实际数据，以及它们的长度。

表 2-2　记录布局

| 字　段 | 描　述 |
| --- | --- |
| Record Length | 记录的长度(字节) |
| Key Length | 键的长度(字节) |
| Key | 字节数组，包含记录的键 |
| Value | 字节数组，包含记录的值 |

在基于块的情况下，header 和 sync 服务于相同的目的(与基于记录的 SequenceFile 格式的情况相同)。实际的数据包含在块中，如表 2-3 所示。

表 2-3　块布局

| 字　　段 | 描　　述 |
| --- | --- |
| Keys Lengths Length | 在这种情况下，某个给定块中的所有键都保存在一起。该字段指定压缩后的键-长度大小(以字节为单位) |
| Keys Lengths | 字节数组，包含压缩的键-长度块 |
| Keys Length | 压缩后的键大小(以字节为单位) |
| Keys | 字节数组，包含块中压缩后的键 |
| Values Lengths Length | 在这种情况下，某个给定块中的所有值都保存在一起。该字段指定压缩后的值-长度大小(以字节为单位) |
| Values Lengths | 字节数组，包含压缩的值-长度块 |
| Values Length | 压缩后的值大小(以字节为单位) |
| Values | 字节数组，包含块中压缩后的值 |

所有格式均使用相同的 header，其中包含着可以由读取器识别的信息。header(参见表 2-1)包含了键和值的类名(被读取器用来实例化这些类)、版本号以及压缩信息。如果启用了压缩，则 header 中会增加 Compression Codec class name 字段。

SequenceFile 的元数据是一系列键/值文本对，可以包含关于 SequenceFile 的额外信息，文件读取器/写入器会使用这些信息。

无压缩格式和记录压缩格式的写操作的实现非常类似。每一次对 append()方法的调用都会向 SequenceFile 添加一条记录，其中包含整条记录的长度(键的长度加值的长度)、键的长度以及键和值的原始数据。压缩和无压缩版本之间的不同在于是否使用特定的编解码器对原始数据进行了压缩。

块压缩格式可以达到更高的压缩率。直到达到某个阈值(块大小)后，数据才会被写入，此时所有的键将被一起压缩。值以及键和值长度的列表也会被压缩。

Hadoop 提供了用于 SequenceFiles 的特殊读取器(SequenceFile.Reader)和写入器(SequenceFile.Writer)。代码清单2-3展示了使用SequenceFile.Writer的一小段代码。

**代码清单 2-3：使用 SequenceFile.Writer**

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
Path path = new Path("fileName");
SequenceFile.Writer sequenceWriter = new SequenceFile.Writer(fs, conf, path,
     Key.class,value.class,fs.getConf().getInt("io.file.buffer.size",
     4096),fs.getDefaultReplication(), 1073741824, null,new Metadata());
……..
sequenceWriter.append(bytesWritable, bytesWritable);
…….......
IOUtils.closeStream(sequenceWriter);
```

一个最简化的 SequenceFile 写入器构造函数(SequenceFile.Writer(FileSystem fs, Configuration conf, Path name, Class keyClass, Class valClass))需要文件系统的类型、Hadoop 配置、路径(文件位置)以及键和值的类定义。前面示例中使用的构造函数支持指定额外的文件参数，包括以下几个：

- int bufferSize——如果不定义，则使用默认的缓冲区大小(4096)。
- short replication——使用默认复制。
- long blockSize——使用值 1073741824(1024MB)。
- Progressable progress——使用 None。
- SequenceFile.Metadata metadata——使用一个空的元数据类。

写入器一旦创建完成，就可以用来向文件添加键/记录对了。

SequenceFile 的局限之一是无法基于键值进行查找。其他 Hadoop 文件类型(MapFile、SetFile、ArrayFile 和 BloomMapFile)通过在 SequenceFile 之上增加基于键的索引克服了这个限制。

如图 2-4 所示，MapFile 实际上并非文件，而是目录，其中包含两个文件——一个数据(序列)文件，包含 map 中所有的键和值；一个较小的索引文件，包含一部分键。可以通过按顺序添加条目来创建 MapFile。MapFile 通常利用其索引来高效地搜索和检索文件的内容。

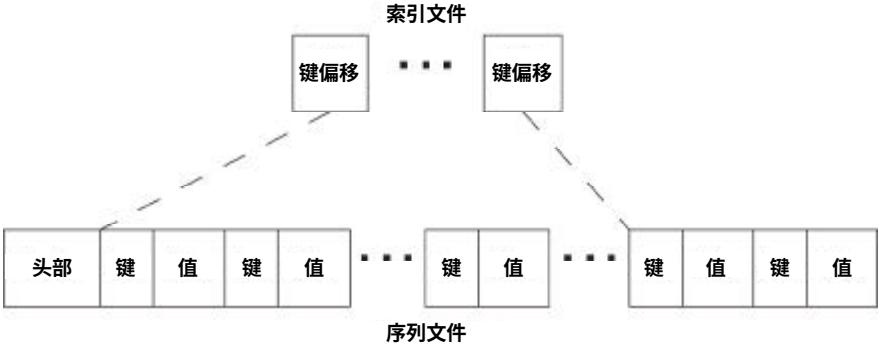



图 2-4 MapFile

索引文件中包含键和一个LongWritable对象，LongWritable对象保存了该键对应记录的起始字节位置。索引文件并不包含所有键，而是只包含其中一部分。我们可以使用写入器的setIndexInterval()方法设置indexInterval。索引会被完全读入内存，因此对于很大的map，有必要设置索引跳跃值，使得索引文件足够小，以致能够完全加载到内存中。

类似于 SequenceFile，Hadoop 提供了用于 map 文件的特殊读取器(MapFile.Reader)和写入器(MapFile.Writer)。

SetFile和ArrayFile是用于实现特定键/值类型的MapFile变体。SetFile是一种MapFile，用于代表没有值的键集合(值由NullWritable实例来代表)。ArrayFile处理键/值对，其键为连续的long(长整型)值。它维护一个内部计数器，并在每次进行追加调用时增加。该计数器的值会被用作键。

这两种文件类型对于保存键(而非值)很有用。

**布隆过滤器**

布隆过滤器是一种空间利用率高的、概率性的数据结构，用于测试一个元素是否是某个集合的成员。测试的结果是该元素确定不在集合中，或者可能在集合中。

布隆过滤器的基础数据结构是比特向量。误报的可能性取决于元素集合的大小和比特向量的大小。

尽管会有误报，但布隆过滤器在表示集合时相比于其他数据结构(例如自平衡二叉搜索树、单词查找树、哈希表或者简单的数组或链表)有着很强的空间优势。大多数数据结构至少要保存数据条目本身，这需要的存储空间可能是从少量比特位(对于小整数)到任意数量的比特位，例如对于字符串(单词查找树是个特例，因为前缀相同的元素之间可以共享存储)。

布隆过滤器的这种优势一部分源于其紧凑性(继承自数组)，还有一部分源于其概率性本质。

最后，BloomMapFile 通过添加动态的布隆过滤器(参见补充内容“布隆过滤器”)扩展了 MapFile 实现，为键提供快速的成员资格测试。它还提供了键搜索操作的一个快速版本，尤其适用于稀疏的 MapFile。写入器的 append()操作会更新 DynamicBloomFilter，当写入器关闭时，DynamicBloomFilter 会被序列化。当创建读取器时，该过滤器会被加载到内存中。读取器的 get()操作首先利用过滤器检查键的成员资格，如果键不存在的话，它立即返回空值，不再进行任何进一步的 I/O 操作。

**数据压缩**

在 HDFS 文件中存储数据时，一个需要考虑的重要因素就是数据压缩——将数据处理中的计算负载从 I/O 转化为 CPU。一些出版物提供了对在 MapReduce 实现中使用压缩时，计算与 I/O 之间相互权衡的系统评估，其结果显示数据压缩的益处取决于数据处理作业的类型。对于大量读操作(I/O 是瓶颈)应用(例如，文本数据处理)，压缩会节省 35%～60%的性能开销。另一方面，对于计算密集型应用(CPU 是瓶颈)，数据压缩带来的性能提升微不足道。

但这并不意味着数据压缩对此类应用没有益处。Hadoop 集群的资源均是共享的，其结果是，一个应用 I/O 负载的降低将会提升使用该 I/O 的其他应用的性能。

这意味着总是需要数据压缩吗？答案是“否”。例如，如果正在使用文本文件或自定义二进制输入文件，那么可能就不需要数据压缩，因为压缩后的文件不能被分割(在第 3 章中会学到更多)。另一方面，对于 SequenceFile 及其衍生的文件类型，压缩总是需要的。最后，压缩用于 shuffle 和 sort 的中间文件总是有意义的(在第 3 章中会学到更多)。

记住数据压缩的结果在很大程度上取决于待压缩数据的类型和压缩算法。

### 2.1.4 HDFS 联盟和高可用性

当前 HDFS 实现的主要局限在于单个 NameNode。由于所有文件元数据都保存在内存中，因此 NameNode 的内存量决定了 Hadoop 集群上可用文件的数量。为克服单个 NameNode 内存的限制并能够水平地扩展名称服务，Hadoop 0.23 引入了 HDFS 联盟(HDFS Federation)，它基于多个独立的 NameNode/名称空间。

以下是 HDFS 联盟的主要优势：

- 名称空间可扩展性——HDFS 集群存储可以水平扩展，但名称空间不能。通过向集群添加更多的 NameNode 来扩展名称空间，大规模部署(或者使用大量小文件的部署)会因此受益。
- 性能——文件系统操作的吞吐量受到单个 NameNode 的限制。向集群添加更多的 NameNode 可以扩展文件系统读/写操作的吞吐量。
- 隔离——在多用户环境下，单个 NameNode 无法支持隔离。实验性应用可能会让 NameNode 过载，并拖慢关键的生产应用。使用多个 NameNode，可以将不同类别的应用和用户隔离到不同的名称空间。

如图 2-5 所示，HDFS 联盟的实现基于多个独立 NameNode 的汇集，它们之间不需要进行协调。所有 NameNode 均将 DataNode 作为公共存储，用于保存块。每个 DataNode 都会向集群中的所有 NameNode 注册。DataNode 周期性地发送心跳和块报告，并处理来自 NameNode 的命令。

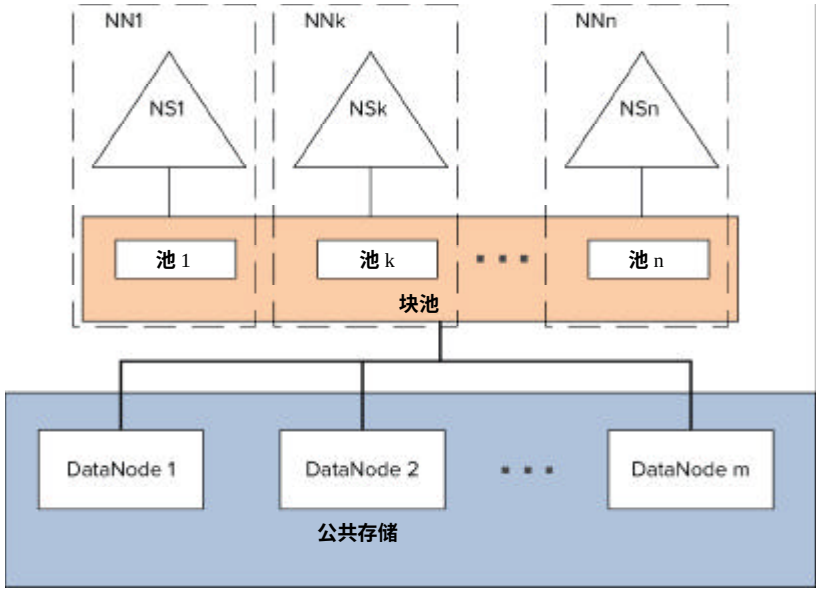



图 2-5 HDFS 联盟 NameNode 架构

名称空间在块的集合——块池——上操作。尽管块池只能用于特定的名称空间，但实际的数据可以被分配到集群中的任意 DataNode 上。每个块池都是独立管理的，这允许名称空间在不必与其他名称空间协调的情况下为新块生成块 ID。某个 NameNode 的失效不会

影响 DataNode 服务集群中的其他 NameNode。

名称空间和其块池一起被称为名称空间卷。这是一个自包含的管理单元。当删除 NameNode/名称空间时，DataNode 上相对应的块池也会被删除。在集群升级时，每个名称空间卷会被作为一个单元来升级。

HDFS 联盟的配置是向后兼容的，且允许已有的单一 NameNode 配置能够在不进行任何改动的情况下正常工作。新配置的设计使得集群中所有节点的配置相同，不必根据其中节点的类型来部署不同的配置。

尽管 HDFS 联盟解决了 HDFS 扩展性的问题，但它并不解决 NameNode 可靠性问题(事实上，它使事情变得更糟——这种情况下单个 NameNode 失效的概率更高)。图 2-6 展示了一种新的 HDFS 高可用性架构，包含两台配置为 NameNode 的独立机器，且在任意时间点只有其中一台处于活动状态。活动的 NameNode 负责响应集群中的所有客户端操作，而另外一台(备机)仅作为从属，维护着足够的状态信息，并在需要时提供快速的故障转移。为保持两个节点的状态同步，该实现要求两个节点均可以访问共享存储设备上的某个目录。

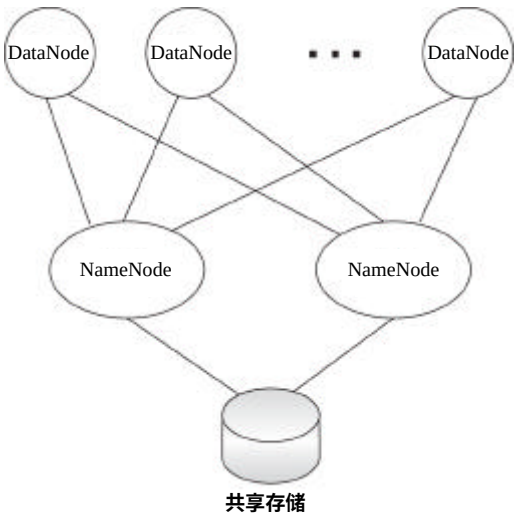



图 2-6　HDFS 故障转移架构

当活动节点进行任何名称空间修改时，它会向一个位于共享目录下的日志文件持久性地写入一条修改记录。备用节点持续观察该目录的变化，并且将改动应用于自身的名称空间。当发生故障转移时，备机在确保已经读取了所有改动之后，将自身切换为活动状态。

为支持快速的故障转移，备用节点也需要了解集群中块位置的最新信息。这可以通过配置 DataNode 来实现，让其同时向两台 NameNode 发送块位置信息和心跳。

目前，仅支持手工故障转移。Hortonworks公司提交到1.1版主干和分支中的核心Hadoop的补丁消除了该局限。该解决方案基于Hortonworks故障转移控制器，它会自动选择一个活动的NameNode。

HDFS为存储大量数据提供了非常强大和灵活的支持。一些特殊文件类型(类似于SequenceFile)非常适合支持MapReduce实现。MapFile及其衍生类型(Set、Array和BloomMap)在快速数据访问方面表现良好。

不过，HDFS只支持一组有限的访问模式——写、删除和追加。尽管从技术上讲，可以将更新实现为覆盖，但这种粒度的实现(覆盖仅能工作在文件级别上)在大多数情况下都成本过高。此外，HDFS的设计专门针对支持大量顺序读取，这意味着随机访问数据会造成显著的性能开销。而且最后，HDFS并不适用于较小的文件。尽管从技术上讲，HDFS支持这些文件，但它们的存在会造成NameNode内存的显著开销，因而降低了Hadoop集群内存容量的上限。

为了克服诸多局限，Hadoop 以 HBase 的形式引入了一个更加灵活的数据存储和访问模型。

## 2.2 HBase

HBase 是一个分布式的、版本化的、面向列的、多维度的存储系统，在设计上具备高性能和高可用性。为能够成功地使用 HBase，首先必须了解其实现方法和工作原理。

### 2.2.1 HBase 架构

HBase是Google BigTable架构的开源实现。类似于传统的关系型数据库管理系统(RDBMS)，HBase中的数据以表的形式组织。然而与RDBMS不同的是，HBase支持非常松散的结构定义，且不提供任何连接、查询语言或SQL。

**注意：**

尽管 HBase 不支持实时连接和查询，但可以通过 MapReduce 很轻松地实现批量连接和/或查询。事实上，更高层的系统(例如 Pig 和 Hive)可以很好地支持这些功能，它们使用一种受限的 SQL 方言来执行这些操作。我们会在本章的后面学到关于 Pig 和 Hive 的更多知识。

HBase 的主要关注点是大稀疏表上的创建、读取、更新和删除(CRUD)操作。目前，HBase 不支持事务(但提供有限的锁支持和一些原子化操作)和二级索引(一些社区项目正试图实现该功能，但它不是 HBase 实现的核心组成部分)。其结果是，大多数基于 HBase 的实现正在使用高度非规范化的数据。

类似于 HDFS，HBase 实现了主/从(HMaster/域服务器)架构，如图 2-7 所示。

HBase 利用 HDFS 作为其持久化数据存储。这允许 HBase 利用 HDFS 提供的所有高级特性，包括校验、复制和故障转移。HBase 数据管理由分布式的域服务器实现，域服务器由 HBase 主控服务器(HMaster)进行管理。

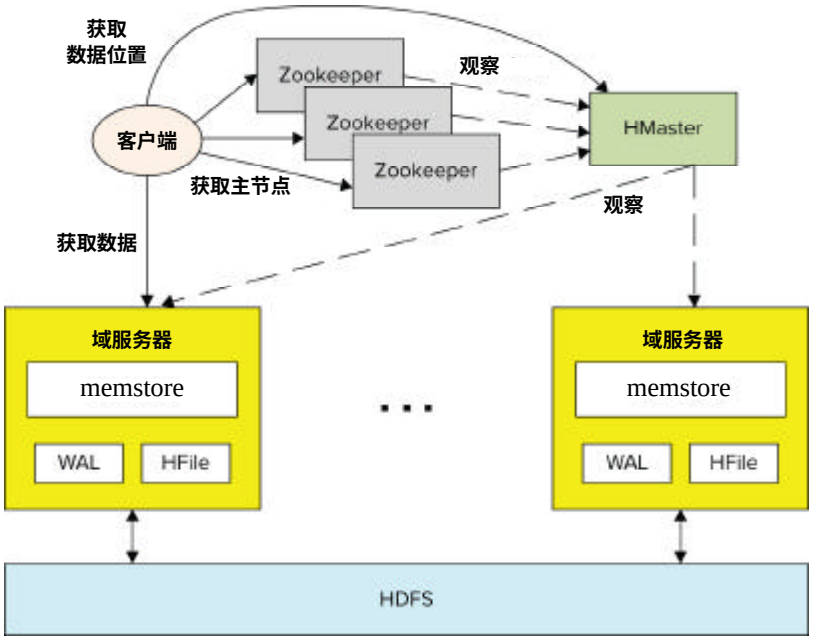



图 2-7　HBase 高层架构

域服务器的实现包含以下主要组件：

? memstore 是 HBase 的内存数据缓存实现，它通过直接从内存提供尽可能多的数据来提升 HBase 的整体性能。memstore 保存以键/值格式存储的数据在内存中的修改。预写日志(WAL)记录了对数据的所有改动。这在主存储发生意外时很重要。如果服务器崩溃，那么可以高效地重播该日志，将服务器恢复到崩溃之前的状态。这也意味着如果向 WAL 写入记录失败，那么整个操作都会被认为是失败的。

**注意：**

HBase 的优化技巧之一就是禁止写入 WAL。这体现了性能和可靠性之间的折衷。当域服务器在写操作完成之前发生失效时，禁止写入 WAL 将导致无法进行恢复。应该谨慎地使用这种优化方案，除非数据丢失可接受，或者写操作可以基于其他数据源进行“重播”的情况。

? HFile 是 HBase 专用的 HDFS 文件格式。域服务器中的 HFile 实现负责从 HDFS 读取 HFile，以及将 HFile 写入到 HDFS。

### Zookeeper

Zookeeper是一个多副本的同步服务，具备最终一致性。它具有鲁棒性，因为持久化的数据分布在多个节点上(该节点集合称为ensemble)，并且连接到其中任意一个节点(即一台特定的“服务器”)的客户端可以在该服务器失效时进行迁移。只要绝大多数节点正在工作，Zookeeper节点组成的ensemble就是存活的。

Zookeeper 的主节点由 ensemble 中的所有节点通过协商来动态选取。如果主节点失效，

余下的节点就会选择一个新的主节点。主节点可以授权写操作。这保证了写操作按照顺序进行(即写操作是线性的)。每次客户端向 ensemble 进行写操作时，大多数节点会保存该信息。这意味着每次写操作都会使服务器与主节点同步。

Zookeeper应用的一个典型示例就是分布式内存计算，其中某些数据会在客户端节点之间共享，而且出于同步的考虑，必须以一种非常小心的方式来访问/更新数据。Zookeeper 提供了用于构建自定义同步原语的库，以及运行分布式服务器的能力，而这避免了在使用中心化的消息存储库时会遇到的单点故障问题。

分布式的 HBase 实例依赖于处在运行状态的 Zookeeper 集群(关于该服务的描述，请参见补充内容“ Zookeeper ”)。所有参与的节点和客户端必须能够访问正在运行的 Zookeeper 实例。默认情况下，HBase 管理一个 Zookeeper“ 集群 ”——HBase 将 Zookeeper 进程的启动和停止作为启动/停止自身进程的一部分。由于 HBase 主控节点可能会被重新分配，因此客户端启动时会向 Zookeeper 查询 HBase 主控节点和-Root-表的当前位置。

如图 2-8 所示，HBase 使用一种自动分片和分发方案来应对大量数据(与 HDFS 基于块的设计和快速数据访问相比)。

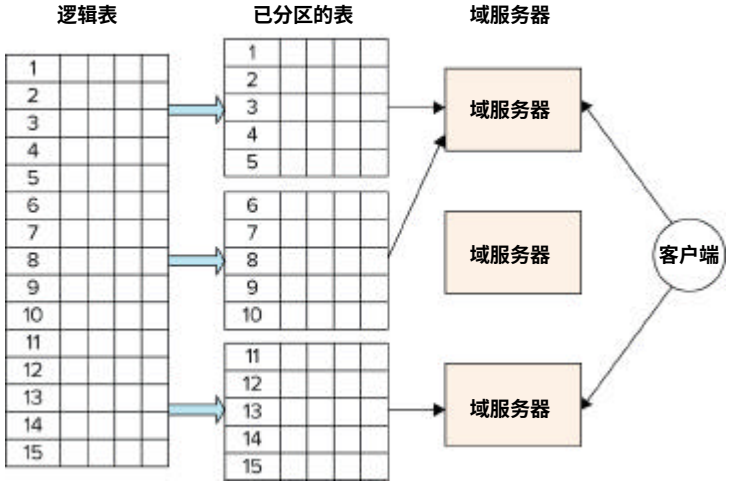



图 2-8　表分片和分发

为保存任意长度的表，HBase 将表划分成域，每个域包含已排序的(根据主键)、范围连续的行。这里，术语“ 连续的 ”并不意味着一个域中会包含给定间隔内的所有键。而是意味着保证会将某个间隔内的所有键划分到相同的域，而在键空间中可能有任意数量的空洞。

域的分裂方式不取决于键空间，而取决于数据的大小。某个特定表的数据分区大小可以在表创建时配置。这些域会“ 随机地 ”散落在域服务器上(单个域服务器可以保存某个特定表的任意数目的域)。它们可能也会被来回移动，以达到负载均衡和故障转移的目的。

当向表中插入新记录时，HBase确定该记录应该去到哪个域服务器(基于键值)并在其上插入它。如果域的大小超出了预先定义的值，那么它会自动分裂。域分裂是一个相当昂贵的操作。为了避免这样的操作，可以在表创建过程中进行预先分裂，或者在任意时间点手

动进行(本章后面会介绍更多相关内容)。

当读取/更新一条记录(或一组记录)时，HBase 确定哪些域应该包含该数据，并将客户端重定向到适当的域。从这个角度来说，域服务器实现了实际的读/更新操作。

如图 2-9 所示，HBase 利用一个专门的表(.META.)将特定的键/表对解析到特定的域服务器。此表包含一个可用的域服务器列表，以及一个用户表描述符的列表。每个描述符为包含在特定域中的特定表指定了键的范围。

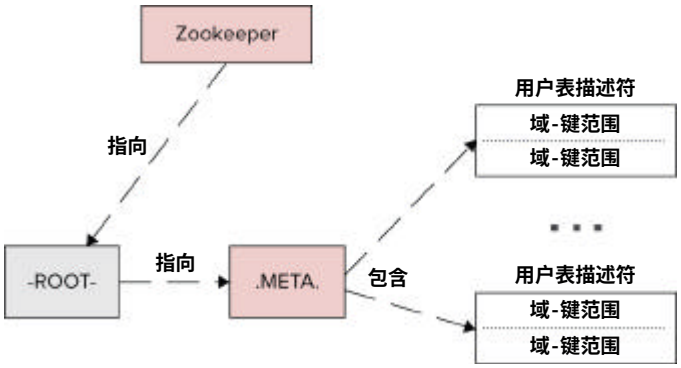



图 2-9　HBase 中的域服务器解析

另外一个专门的 HBase 表(-ROOT-)包含一个.META.表描述符的列表，用于发现.META.表。-ROOT-表的位置保存在 Zookeeper 中。

如图 2-10 所示，HBase 表是一个稀疏的、分布式的、持久化的多维度有序映射表。第一个映射级别是键/行值。如前所述，行键总是有序的，这是表分片以及高效读取和扫描(按顺序读取键/值对)的基础。

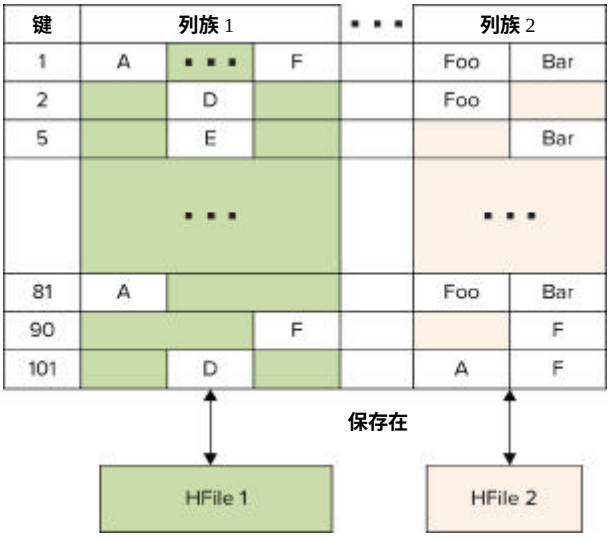



图 2-10　行、列族和列

**注意：**

需要知道的一个事实是 HBase 对字节数组进行操作。HBase 数据的所有组件——键、列族名和列名——均被其当成未解析的字节数组。这意味着所有内部值的比较以及相应的排序，均以字典顺序进行。记住这点非常重要，尤其是在行键设计时要避免出现意料之外的情况。一个典型的例子是整数键的使用。如果没有将它们左补齐为相同的长度，那么作为 HBase 排序的结果，举例来说，键 11 将出现在键 5 之前。

HBase所使用的第二个映射级别基于列族(列族最初由用于快速分析查询的柱状数据库引入)。在这种情况下，数据不像传统RDBMS那样一行接一行地保存，而是以列族的形式保存。HBase使用列族进行基于访问模式(和大小)的数据分割。

列族在 HBase 实现中扮演着特殊的角色。它们定义了保存和访问 HBase 数据的方式。每个列族都保存在一个单独的 HFILE 中。在表设计的过程中，这是一个需要记住的重要考虑因素。建议为每种数据访问类型创建一个列族——即应该把通常一起读/写的数据放入相同的列族。

在表创建的过程中会定义一系列列族(尽管以后可以修改)。不同的列族也可以使用不同的压缩机制，这可能是一个重要因素，例如当为元数据和数据分别创建了单独的列族时(一种常见的设计模式)。在这种情况下，元数据通常相对较小，且不需要压缩；而数据可能会非常大，且压缩通常能够提升 HBase 吞吐量。

鉴于这样的存储组织，HBase 实现了合并读。对于一个给定的行，它读取所有的列族文件，并在将结果发回给客户端之前将它们进行合并。其结果是，如果总是一起处理整行的话，那么一个单独的列族通常提供最佳的性能。

最后一个映射级别基于列。HBase 将列作为键/值对的动态映射。这意味着列不在表创建时定义，而是在写/更新操作过程中动态建立。其结果是，HBase 表中的每个行/列族可以包含任意多的列。列中包含实际的值。

从技术上讲，HBase还支持一种映射级别——每个列值的版本。HBase不区分写入和更新操作——更新其实就是新版本的写入。默认情况下，HBase为给定列的值保存最近三个版本(同时自动删除更老的版本)。版本的深度可以在建表时控制。版本的默认实现是数据插入的时间戳，但可以很容易地使用定制化的版本值将其改写。

以下是 HBase 支持的 4 种主要数据操作：

- Get(获取)——为特定行(或多行)返回列族/列/版本的值。可以将其进一步细化为应用于特定的列族/列/版本。重要的是要意识到，如果 Get 被细化为针对单独的列族，那么 HBase 就不必实现合并读了。
- Put(存入)——如果键不存在的话，向表添加新的一行(或多行)；如果键存在的话，更新现有的一行(或多行)。类似于 Get，可以限定 Put 应用于特定的列族/列。
- Scan(扫描)——允许在多行(某个键范围)上迭代以查找特定值，该特定值可以包括整行或其任意子集。

? Delete(删除)——删除一行(或多行)，或删除表中的任何部分。HBase 不会就地修改数据，其结果是，Delete 操作通过创建被称为墓碑的新标识符来完成。这些墓碑，以及死去的值，会在主要合并时被清理掉(我们将在本章后面学到关于合并的内容)。

可以选择为 HBase 上的 Get 和 Scan 操作配置过滤器(本章后面会介绍过滤器)，过滤器会应用在域服务器上。它们提供了一种 HBase 读优化技术，能够提高 Get/Scan 操作的性能。过滤器是一种适用于读取域服务器上数据的有效条件，且只有符合过滤条件的行(或行中的某些部分)才会被发回客户端。

HBase 支持很多种类型的过滤器，从行键到列族，以及从列到值过滤器。此外，利用布尔操作，可以将过滤器以链式组合。应当注意过滤器并不会减少读取数据的数量(仍然经常需要全表扫描)，但可以显著地减少网络流量。过滤器的实现必须在服务器上部署，而且需要重启服务器。

除通用的 Put/Get 操作之外，HBase 支持以下专有操作：

? 原子条件操作(包括原子比较并设置)支持在检查保护之下执行服务器端更新，以及原子的比较并删除操作(执行服务器端守护的 Delete 操作)。

? 原子“计数器”累加操作，保证操作的同步。在这种情况下，同步在域服务器内而非客户端上实现。

这是 HBase 数据的逻辑视图。表 2-4 描述了 HBase 用来存储表数据的格式。

表 2-4　文件中的表数据布局

| 键 | 时　间　戳 | 列族：列名 | 值 |
|---|---|---|---|
| 行键 | 最后一次更新的时间戳 | 列族：列 | 值 |

该表解释了 HBase 用于处理稀疏表和任意列名的机制。HBase 显式地保存每一列，它们在适当的列族文件中为特定的键而定义。

由于 HBase 使用 HDFS 作为持久化机制，因此它从不覆盖数据(HDFS 不支持更新)。其结果是，每次 memstore 写入磁盘的时候，它并不覆盖已有的存储文件，而是创建新文件。为避免存储文件过多，HBase 实现了一个称为合并的处理过程。

存在两种合并类型：次要合并和主要合并。次要合并通常收集多个较小的相邻存储文件并将它们重写为一个。次要合并不会删除已执行 Delete 操作或已过期的数据单元；只有主要合并会将其删除。有时，次要合并收集所有存储文件，在这种情况下它实际上成为了主要合并。

在主要合并运行之后，每个存储上将只有一个存储文件，这通常会提升性能。合并不会实现域的合并。

为在文件中更快速地进行键查找，HBase 使用了布隆过滤器，允许进行行或行/列级别的检查，并有可能在读取时过滤整个存储文件。这种过滤在键较稀疏的情况下尤其有用。布隆过滤器在文件持久化时生成，并保存在每个文件的尾部。

### 2.2.2 HBase 结构设计

在设计 HBase 结构时，需要考虑以下一般性原则：

? 访问 HBase 数据最高效的方式是使用基于行键的 Get 或 Scan 操作。HBase 不支持任何二级键/索引。这意味着在理想情况下，行键的设计应该包含特定表所需要的全部访问模式。这通常意味着需要使用复合行键来容纳更多的数据访问模式(本章后面讨论了行键设计的最佳实践)。

? 一条一般性的原则是将每个表的列族数限定为不超过10～15(记住 HBase 将每个列族保存在独立的文件中，因此大量的列族会导致需要读取和合并多个文件)。同样，基于这样一个事实，即列族名会显式地与每个列名保存在一起(参见表2-4)，因此应该最小化列族的大小。如果列中数据量小的话，通常推荐使用单个字母的名字。

? 尽管 HBase 对给定行中列的大小不强加任何限制，但应该考虑以下问题：

- 行不是可分割的。其结果是，较大的列数据(接近于域大小)通常表明该类型数据不应该保存在 HBase 中。
- 每个列的值均与其元数据一起保存(行键、列族名、列名)，这意味着非常小的列数据会导致较低的存储利用率(即元数据比实际表数据占用了更多的空间)。这也意味着不应该使用较长的列名。

? 在高而窄(上百万的键与有限数量的列)和扁而宽(有限数量的键与上百万的列)的表设计之间进行选择时，通常推荐使用前者。这是由于以下原因：

- 在极端情况下，扁而宽的表可能终结为每个域上的一行，这对于性能和可扩展性都是不利的。
- 与大量的读操作相比，表扫描通常更高效。其结果是，假如仅需要行数据的一个子集，则高而窄的设计提供更好的整体性能。

HBase 设计的主要优势之一是在多台域服务器之间分布式地执行请求。然而，利用该设计的优势并确保在应用执行时没有“热点”(高负载的)服务器出现，则可能需要特殊的行键设计方案。通常建议避免将单调递增序列(例如，1、2、3，或时间戳)作为大量 Put 操作的行键。通过将键值随机化使得它们不按顺序排列，可以缓解由于单调递增键引发的单个域上的拥塞。

数据本地性也是 HBase Get/Scan 操作在设计上的一个重要考虑因素。应该将经常一起读取的行放置在一起，这意味着它们必须有相邻的键。

在大量 Scan 操作的情况下，一般的规则是使用顺序的键，尤其是可以将大数据块导入用于数据填充的情况。对于大量并行写操作(随机的、单独的键访问)，推荐使用随机键。

以下是一些特殊的行键设计模式：

? 键“盐化(salting)”——这意味着为顺序键添加一个随机值前缀，允许将顺序键“分桶”到多个域中。

- 键字段的交换/提升(例如“反转域名”)——Web 分析中一种常见的设计模式是将域名用作行键。在这种情况下，使用反转域名作为键有助于保持每个站点页面的信息彼此靠近。
- 键的完全随机化——这种情况的一个例子是使用 MD5 散列。

关于键设计的最后考虑因素是键大小。基于表 2-4 中所示的信息，键值与每个列值一同保存，这意味着键的大小应该足以支持高效查找，但不应过长。过大的键会对 HBase 存储文件的大小产生负面影响。

### 2.2.3　HBase 编程

HBase 提供原生的 Java、REST 和 Thrift API。此外，HBase 支持命令行界面和通过 HBase 主控页面的 Web 访问，在其他 HBase 书籍中有关于这两者的很好描述，这里不再讨论。由于本书针对从应用程序访问 HBase 的开发者，因此这里仅讨论 Java 接口(可以分为两个主要部分——数据操作和管理)。

对HBase所有编程方式的数据操作访问均通过HTableInterface或实现了HTableInterface的HTable类完成。两者都支持之前描述的全部HBase主要操作，包括Get、Scan、Put和Delete。

代码清单 2-4 展示了如何基于表名创建 HTable 类的实例。

**代码清单 2-4：创建 HTable 实例**

```
Configuration configuration = new Configuration();
HTable table = new HTable(configuration, "Table");
```

有一件事情需要记住，HTable 的实现是单线程的。如果需要多线程访问 HBase，那么每个线程必须创建自己的 HTable 类实例。解决该问题的一种方式是使用 HTablePool 类。这个类的目的是在 HBase 集群中池化客户端 API(HTableInterface)实例。代码清单 2-5 展示了创建和使用 HTablePool 的一个例子。

**代码清单 2-5：创建和使用 HTablePool 实例**

```
Configuration configuration = new Configuration();
HTablePool _tPool = new HTablePool(config, mSize);
……
HTableInterface table = _tPool.getTable("Table");
……
_tPool.putTable(table);
……
_tPool.closeTablePool("testtable");
```

一旦得到了合适的类或接口，之前描述的所有 HBase 操作(包括 Get、Scan、Put 和 Delete)都是可用的方法。

代码清单 2-6 中的例子展示了在表“table”中对键“key”进行 Get 操作的实现。

**代码清单 2-6：实现 Get 操作**

```
HTable table = new HTable(configuration, "table");
Get get = new Get(Bytes.toBytes("key"));
Result result = table.get(get);
NavigableMap<byte[], byte[]> familyValues = result.getFamilyMap
   (Bytes.toBytes "columnFamily"));
for(Map.Entry<byte[], byte[]> entry : familyValues.entrySet()){
   String column = Bytes.toString(entry.getKey);
   Byte[] value = entry.getValue();
……
}
```

这里，在获取整行之后，仅有列族“ columnFamily ”的内容是必需的。HBase 以可导航 map 的形式返回该内容，对其进行迭代可以获取各个列的名字和值。

Put 和 Delete 利用相同的模式实现。HBase 还以多重 Get/Put 的形式提供了重要的优化技巧，如代码清单 2-7 所示。

**代码清单 2-7：实现多重 Put 操作**

```
Map<String, byte[]> rows = …….;
HTable table = new HTable(configuration, "table");
List<Put> puts = new ArrayList<Put>();
for(Map.Entry<String, byte[]> row : rows.entrySet()){
   byte[] bkey = Bytes.toBytes(row.getKey());
   Put put = new Put(bkey);
put.add(Bytes.toBytes("family"),
Bytes.toBytes("column"),row.getValue());
   puts.add(put);
}
table.put(puts);
……
```

在此代码片段中，假定输入格式为包含键和值的map，应该将其写入到“ family ”列族的“ column ”列中。与发送个别的Put请求不同，这里创建了一个Put操作的列表，并作为一个单独的请求发送。使用多重Get/Put所获得的实际性能提升可能差异显著，但大体上，这种方案似乎总是会带来更好的性能。

HBase 提供的最强大的操作之一是 Scan，它允许对一系列连续的键进行迭代。Scan 允许指定多个参数，包括起始键和结束键、需要获取的列族和列的集合以及需要应用于数据的过滤器。代码清单2-8(代码文件：类 BoundinBoxFilterExample)展示了一个如何使用 Scan 的例子。

**代码清单 2-8：实现 Scan 操作**

```
Put put = new Put(Bytes.toBytes("b"));
put.add(famA, coll1, Bytes.toBytes("0.,0."));
put.add(famA, coll2, Bytes.toBytes("hello world!"));
```

```
hTable.put(put);
put = new Put(Bytes.toBytes("d"));
put.add(famA, coll1, Bytes.toBytes("0.,1."));
put.add(famA, coll2, Bytes.toBytes("hello HBase!"));
hTable.put(put);
put = new Put(Bytes.toBytes("f"));
put.add(famA, coll1, Bytes.toBytes("0.,2."));
put.add(famA, coll2, Bytes.toBytes("blahblah"));
hTable.put(put);
// Scan data
Scan scan = new Scan(Bytes.toBytes("a"), Bytes.toBytes("z"));
scan.addColumn(famA, coll1);
scan.addColumn(famA, coll2);
WritableByteArrayComparable customFilter = new BoundingBoxFilter("-1.,-1.,
   1.5,1.5");
SingleColumnValueFilter singleColumnValueFilterA = new
   SingleColumnValueFilter(famA, coll1, CompareOp.EQUAL, customFilter);
singleColumnValueFilterA.setFilterIfMissing(true);
SingleColumnValueFilter singleColumnValueFilterB = new
   SingleColumnValueFilter(famA, coll2, CompareOp.EQUAL,
   Bytes.toBytes("hello HBase!"));
singleColumnValueFilterB.setFilterIfMissing(true);
FilterList filter = new FilterList(Operator.MUST_PASS_ALL, Arrays
   .asList((Filter) singleColumnValueFilterA,
   singleColumnValueFilterB));
scan.setFilter(filter);
ResultScanner scanner = hTable.getScanner(scan);

for (Result result : scanner) {
   System.out.println(Bytes.toString(result.getValue(famA, coll1)) + " , "
                      + Bytes.toString(result.getValue(famA, coll2)));
}
```

**在此代码片段中，首先使用样本数据对表进行填充。完成后，创建一个** Scan **对象。接下来，创建用于扫描的起始键和结束键。这里需要记住的是：起始键是包含在扫描范围内的，而结束键不包含在内。代码接着显式地指定了扫描操作将只读取列族** famA **中的列** coll1 **和** coll2**。最后，指定了一个过滤器列表，其中包含由** HBase **提供的两个过滤器——一个自定义边界框过滤器(代码清单** 2-9**，代码文件：类** BoundingBoxFilter**)和一个“ 标准 ”字符串比较过滤器。一旦所有的设置都已完成，代码就会创建** ResultScanner**，可以对其进行迭代来获取结果。**

**代码清单** 2-9**：边界框过滤器的实现**

```
public class BoundingBoxFilter extends WritableByteArrayComparable{

   private BoundingBox _bb;
   public BoundingBoxFilter(){}
```

```
public BoundingBoxFilter(byte [] value) throws Exception{

   this(Bytes.toString(value));
}

public BoundingBoxFilter(String v) throws Exception{
   _bb = stringToBB(v);
}

private BoundingBox stringToBB(String v)throws Exception{

   …….
}

@Override
public void readFields(DataInput in) throws IOException {

   String data = new String(Bytes.readByteArray(in));
   try {
        _bb = stringToBB(data);
   } catch (Exception e) {
      throw new IOException(e);
   }
}

private Point bytesToPoint(byte[] bytes){

   …….
}

@Override
public void write(DataOutput out) throws IOException {

   String value = null;
   if(_bb != null)
      value = _bb.getLowerLeft().getLatitude() + "," +
         _bb.getLowerLeft().getLongitude() +
      "," + _bb.getUpperRight().getLatitude() + "," +
         _bb.getUpperRight().getLongitude();
   else
      value = "no bb";
   Bytes.writeByteArray(out, value.getBytes());
}

@Override
public int compareTo(byte[] bytes) {

   Point point = bytesToPoint(bytes);
   return _bb.contains(point) ? 0 : 1;
}
```

```
}
```

**注意：**

如这里所示，使用 Get、多重 Get 和 Scan 可以实现 HBase 读操作。Get 仅用于某一时刻需要从表中真正地读取单独一行的情况。多重 Get 和 Scan 是更优的实现。当使用 Scan 时，利用 scan.setCaching(HBASECASHING)(其中 HBASECASHING 指定需要缓存的行数)设置缓存值可以显著地提升性能。缓存的大小显著地依赖于对数据进行的处理。如果处理一批记录需要较长的时间，则客户端可能会在处理完成之前向域服务器请求下一批数据，这种情况下超时会导致异常(例如，UnknownScannerException)。如果数据处理速度很快，那么可以将缓存设置得更高。

自定义边界框过滤器(代码清单 2-9)继承自 Hadoop 的 WritableByteArrayComparable 类并利用了用于地理空间计算的 BoundingBox 和 Point 类。在与本书配套的英文网站上可以找到这两个类(代码文件：类 BoundingBox 和类 Point)。以下是该类的主要方法，每个过滤器都必须实现这些方法：

? CompareTo 方法负责决定是否将某条记录包含在结果中。

? readFields和write方法分别负责从输入流中读取过滤器参数和将它们写入到输出流。

过滤是一种非常强大的机制，但如前所述，它对于提升性能帮助不大。真正的大表扫描即使使用了过滤器，也会很慢(仍然需要读取每条记录，并测试是否满足过滤器条件)。过滤更有助于提高网络利用率。过滤在域服务器上完成，其结果是，只有满足过滤条件的记录才会返回给客户端。

HBase 提供了相对多的过滤类，从一组列值过滤器(测试特定列的值)到列名过滤器(过滤列名)、列族过滤器(过滤列族名)以及行键过滤器(过滤行键值或数量)。代码清单2-9展示了如何实现自定义过滤器，用于需要达到特定过滤效果的情况。关于自定义过滤器的一个问题是它们需要服务器部署。这意味着为了能够使用，必须将自定义过滤器(和所有辅助的类)一起打成 jar 包，并添加到 HBase 的类路径中。

如前所述，HBase API 提供对数据访问和 HBase 管理的支持。例如，我们能够看到如何使用 HBase 结构管理器中的管理功能。代码清单 2-10(代码文件：类 TableCreator)展示了一个表创建器类，该类演示了通过编程管理 HBase 的主要功能。

**代码清单 2-10：TableCreator 类**

```
public class TableCreator {

   public static List<HTable> getTables(TablesType tables, Configuration
      conf)throws Exception{

      HBaseAdmin hBaseAdmin = new HBaseAdmin(conf);
      List<HTable> result = new LinkedList<HTable>();
      for (TableType table : tables.getTable()) {
         HTableDescriptor desc = null;
```

```
            if (hBaseAdmin.tableExists(table.getName())) {
                if (tables.isRebuild()) {
                    hBaseAdmin.disableTable(table.getName());
                    hBaseAdmin.deleteTable(table.getName());
                    createTable(hBaseAdmin, table);
                }
                else{
                    byte[] tBytes = Bytes.toBytes(table.getName());
                    desc = hBaseAdmin.getTableDescriptor(tBytes);
                    List<ColumnFamily> columns = table.getColumnFamily();
                          for(ColumnFamily family : columns){
                              boolean exists = false;
                              String name = family.getName();
                              for(HColumnDescriptor fm :
                                  desc.getFamilies()){
                              String fmName = Bytes.toString(fm.getName());
                              if(name.equals(fmName)){
                                  exists = true;
                                  break;
                              }
                          }
                          if(!exists){
                              System.out.println("Adding Family " + name +
    " to the table " + table.getName());
                              hBaseAdmin.addColumn(tBytes,
                                  buildDescriptor(family));
                          }
                      }
            }
            } else {
                createTable(hBaseAdmin, table);
            }
            result.add( new HTable(conf, Bytes.toBytes(table.getName())) );
        }
        return result;
    }

    private static void createTable(HBaseAdmin hBaseAdmin,TableType table)
            throws Exception{
        HTableDescriptor desc = new HTableDescriptor(table.getName());
        if(table.getMaxFileSize() != null){
            Long fs = 1024l * 1024l * table.getMaxFileSize();
            desc.setValue(HTableDescriptor.MAX_FILESIZE, fs.toString());
        }
        List<ColumnFamily> columns = table.getColumnFamily();
              for(ColumnFamily family : columns){
                  desc.addFamily(buildDescriptor(family));
              }
              hBaseAdmin.createTable(desc);
    }
```

```
    private static HColumnDescriptor buildDescriptor(ColumnFamily family){

          HColumnDescriptor col = new HColumnDescriptor(family.getName());
          if(family.isBlockCacheEnabled() != null)
             col.setBlockCacheEnabled(family.isBlockCacheEnabled());
          if(family.isInMemory() != null)
             col.setInMemory(family.isInMemory());
          if(family.isBloomFilter() != null)
             col.setBloomFilterType(BloomType.ROWCOL);
          if(family.getMaxBlockSize() != null){
             int bs = 1024 * 1024 * family.getMaxBlockSize();
             col.setBlocksize(bs);
          }
          if(family.getMaxVersions() != null)
             col.setMaxVersions(family.getMaxVersions().intValue());
          if(family.getTimeToLive() != null)
             col.setTimeToLive(family.getTimeToLive());
          return col;
    }
}
```

**所有对管理功能的访问均通过** HBaseAdmin **类完成，它可以使用配置对象来创建。该对象提供了广泛的** API**，从获取对** HBase **主控节点的访问，到检查某个特定的表是否存在且已启用，再到创建和删除表。**

**可以基于**HTableDescriptor**创建表。该类允许操作特定表的参数，包括表名、最大文件尺寸等。它还包含了一个**HColumnDescriptor**类的列表，其中每个**HColumnDescriptor**类对应一个列族。该类允许设置特定列族的参数，包括名称、最大版本数目、布隆过滤器等。**

**注意：**

**如前所述，表分裂是相当昂贵的操作，应该尽可能避免。在预先知道键数量的情况下，**HBaseAdmin **类允许使用以下两个方法创建预分裂表：**

```
admin.createTable(desc, splitKeys);
admin.createTable(desc, startkey, endkey, nregions);
```

**第一个方法需要表描述符和键的字节数组，其中每个键指定域的起始键。第二个方法需要表描述符、起始键和结束键以及域的数量。两个方法都会创建一个预先分裂为多个域的表，这样可以提高该表的使用性能。**

### 异步 HBase API

StumbleUpon **公司提出了另外一种** HBase API **实现——异步** HBase**。该实现与** HBase **自有的客户端**(HTable)**截然不同。异步** HBase **的核心是** HBaseClient**，它不仅线程安全，而且支持对任何** HBase **表的访问**(**与之相比较的** HTable**，每个实例仅支持一个表**)**。**

**该实现允许以完全异步/非阻塞的方式访问**HBase**。这使得吞吐量产生了巨大的提高，**

尤其是对于Put操作。Get/Scan操作不会产生如此巨大的提高，但仍然会有明显的提高。

此外，异步 HBase 产生更少的锁争用(几乎比输入繁重的作业少 4 倍)，同时使用较少的内存和少得多的线程(标准 HBase 客户端需要非常多的线程来保证良好地运行，而过度的上下文切换会导致糟糕的 CPU 使用率)。

最后，异步 HBase 也努力尝试与 HBase 的所有版本兼容。对于标准客户端，应用程序必须使用与服务器完全相同的 HBase jar 版本。任何微小的升级都需要重启使用了更新后 jar 包的应用程序。异步 HBase 支持 0.20.6 版本(或许更早)及以后所有现行的 HBase 发布版。至今为止，它仅在推出 0.90 版本(引入了 Get 远程过程调用，或称 RPC 中的向后不兼容)时需要应用程序更新一次。

尽管异步 HBase 具备所有这些优势，但它尚未在 HBase 开发社区中得到广泛采纳，原因如下：

- 它是异步的。很多程序员对于这种编程范式感到不适应。尽管从技术上讲，这些 API 可以用于编写同步调用，但这必须在完全异步 API 之上完成。
- 这些 API 仅有非常有限的文档。其结果是，为了使用它们，需要通读源代码。

HBase 提供了非常强大的数据存储机制，具有丰富的访问语义和特性。然而，它并不是万能的解决方案。当决定使用 HBase 还是传统的 RDBMS 时，需要考虑以下因素：

- 数据大小——如果有数亿(甚至数十亿)行，HBase 是个不错的选择。如果只有数千/数百万行，则使用传统的 RDBMS 可能是更好的选择，因为所有数据可能都会落在单独一个(或两个)节点上，且集群中的其他节点可能被闲置。
- 可移植性——应用程序可能不需要 RDBMS 提供的全部额外特性(例如，有类型的列、二级索引、事务、高级查询语言等)。通过简单地修改 JDBC 驱动，无法将基于 RDBMS 构建的应用程序“移植”到 HBase。从 RDBMS 迁移到 HBase 需要完全重新设计应用程序。

### 2.2.4 HBase 新特性

以下是最近加入到 HBase 中的两个显著特性：

- HFile v2 格式
- 协处理器

#### 1. HFile v2 格式

现有 HFile 格式的问题在于内存使用高且域服务器启动时间长，这是由于庞大的布隆过滤器和块索引造成的。

在当前的 HFile 格式中，内存中必须一直保存着一个单独的索引文件。这会导致每台服务器有数 GB 的内存被块索引消耗掉，这对域服务器的可扩展性和性能有显著的负面影响。此外，由于直到加载完所有块索引数据之后，才能认为域启动完成，因此这样的块索引大小会明显地拖慢域的启动速度。

为解决此问题，HFile v2 格式将块索引拆分为一个根索引块和多个叶块。只有根索引(索引数据块)必须一直保存在内存中。叶索引以块的级别存储，这意味着它是否在内存中出现取决于块是否在内存中出现。叶索引仅在加载块时被加载到内存中，并且在块移出时被移出内存。此外，叶级别索引的结构化方式允许在不进行反序列化的情况下，对键进行二叉搜索。

类似的方案被 HFile v2 实现者用于布隆过滤器。每个数据块有效地利用自身的布隆过滤器，块填满后立即将其写入磁盘。在读取时，利用对键的二叉搜索确定合适的布隆过滤器块，将其加载、缓存并查询。复合布隆过滤器不依赖于对将要向布隆过滤器添加的键的数量的估计，因此它们可以达到更加精确的误报率。

以下是 HFile v2 的一些额外的增强功能：

? 统一的 HFile 块格式支持在不使用块索引的情况下高效地查找之前的块。

? HFile 重构为读取器和写入器层次结构，显著提升了代码的可维护性。

? 稀疏的锁实现简化了用于层次化块索引实现的块操作同步。

当前 HFile v2 读取器实现方案的一个重要特性是能够同时读取 HFile v1 和 v2。而另一方面，写入器实现仅能写入 HFile v2。这允许将已存在的 HBase 安装从 HFile v1 无缝地过渡到 HFile v2。HFile v2 的使用给 HBase 的可扩展性和性能方面带来了明显改善。

目前还有一个针对 HFile v3 的提议，即改进压缩。

2. 协处理器

HBase 协处理器受到 Google BigTable 协处理器的启发，并在设计上支持高效的并行计算——超越 Hadoop MapReduce 可以提供的性能。此外，可以将协处理器用于实现新特性，例如二级索引、复杂过滤(下推谓词)和访问控制。

尽管是受 BigTable 的启发，但 HBase 协处理器在实现细节上与之存在差别。开发者们实现了一个框架，提供库和运行时环境，用于在 HBase 域服务器(即相同的 Java Virtual Machine(JVM)和主控服务器进程中执行用户代码。相反，Google 协处理器不在 tablet 服务器(等效于 HBase 域服务器)上运行，而是运行在其地址空间之外(HBase 开发者也正考虑在未来的实现中提供一个选项，支持在服务器进程之外部署协处理器代码)。

HBase 定义了两种类型的协处理器：

? 系统协处理器全局加载，适用于域服务器上的所有表和域。

? 表协处理器在所有域上加载，适用于某个表。

协处理器框架非常灵活，且允许实现两种基本的协处理器类型：

? 观察者(类似于常规数据库中的触发器)

? 终端(类似于常规数据库的存储过程)

观察者允许在 HBase 调用的执行中插入用户代码。此代码由 HBase 核心代码调用。协处理器框架处理调用用户代码的所有细节。协处理器实现只需要包含所需的功能。HBase 0.92 提供了三种观察者接口：

- RegionObserver——提供用于数据访问操作的钩子(Get、Put、Delete、Scan 等)，并支持使用用户代码补充这些操作的方法。RegionObserver 协处理器的实例会被加载到每个表域。其作用范围被限定为它所在的域。RegionObserver 需要被加载到每个 HBase 域服务器。
- WALObserver——提供用于预写日志(WAL)操作的钩子。这是一种使用自定义用户代码加强 WAL 写入和重建事件的方法。WALObserver 在域服务器上 WAL 处理的上下文中运行。WALObserver 需要被加载到每个 HBase 域服务器。
- MasterObserver——提供用于表管理操作(即创建、删除、修改表等)的钩子，并支持使用用户代码补充这些操作的方法。MasterObserver 在 HBase 主控节点的上下文中运行。

可以将给定类型的观察者链接在一起，从而以指定的优先级顺序执行。某个特定链上的协处理器可以通过在执行过程中传递信息，进而与彼此通信。

终端是一个用于动态远程过程调用(RPC)扩展的接口。终端的实现在服务器端安装，并可以由 HBase RPC 调用。客户端库提供了便捷的方法用于调用此类动态接口。

构建一个自定义终端的步骤序列如下：

(1) 创建一个继承自 CoprocessorProtocol 的新接口，支持 RPC 实现所需要的数据交换。数据转发必须以字节数组的形式实现。

(2) 使用(继承)抽象类 BaseEndpointCoprocessor 实现终端接口，该抽象类隐藏了一些内部实现细节(例如协处理器框架类的加载)。实现方案必须包含所有必需的协处理器功能，它被加载到域上下文并从中执行。没有什么可以阻止该实现进行 HBase 操作，而这可能会涉及额外的域服务器。

在客户端，新的 HBase 客户端 API 可以调用终端，该 API 允许在单个域服务器上或者在某个范围内的域服务器上执行它。

当前的实现支持两个用于配置自定义协处理器的选项：

- 从配置加载(发生在主控服务器或域服务器启动时)
- 从表属性加载(即当表打开或重新打开时动态加载)

当考虑在自己的开发中使用协处理器时，注意以下几点：

- 由于在当前的实现中，协处理器在域服务器执行上下文中执行，因此行为不当的协处理器可能会拖垮域服务器。
- 协处理器执行是非事务性的，这意味着即使某个 Put 协处理器在 Put 之外有其他的写操作失败，该 Put 操作也仍然是有效的。

尽管 HBase 提供了更加丰富的数据访问模型和通常更优秀的数据访问性能，但它在每行的数据大小方面存在局限。下一节讨论如何同时使用 HBase 和 HDFS 以更好地组织应用程序的数据。

## 2.3　将 HDFS 和 HBase 的组合用于高效数据存储

到本章目前为止，我们已经学习了两种基本的存储机制(HDFS 和 HBase)、它们的操作方式，以及用于数据存储的方式。HDFS 可以用于保存基本上是顺序访问的海量数据，而 HBase 的主要优势是快速随机访问数据。两者都有自己的优点，但二者没有一个能够独立解决一个常见的业务问题——快速访问大的(MB 或 GB 大小)数据条目。

此类问题在Hadoop用于存储和检索较大的条目时经常发生，例如PDF文件、大数据样本、图像、影片或其他多媒体数据。在这种情况下，直接使用HBase实现可能不是最佳方案，因为HBase并不适用于非常大的数据条目(由于分裂、域服务器内存饥饿等)。从技术上讲，HDFS提供了一种快速访问特定数据条目的机制——map文件，但它们不能随着键数目的增长得到很好扩展。

这类问题的解决方案在于结合 HDFS 和 HBase 的最优功能。

方法是创建一个包含大数据条目的 SequenceFile。在将数据写入该文件时，把特定数据条目的指针(从文件起始处的偏移)，以及所有必需的元数据保存到 HBase 中。此时，读取数据需要从 HBase 检索元数据(包括指向数据位置的指针和文件名)，这些信息可以用于访问实际的数据。第 9 章中将更详细地研究该方案的具体实现。

HDFS 和 HBase 均将数据(大多数部分)视为二进制流。这意味着借助于这些存储机制的大多数应用程序必须使用某种形式的二进制封装。一个这样的封装机制是 Apache Avro。

## 2.4　使用 Apache Avro

Hadoop 生态系统包含了一个新的二进制数据序列化系统——Avro。Avro 定义了一种数据格式，从设计上支持数据密集型应用程序，并在多种编程语言中提供对该格式的支持。其功能类似于其他封装系统，例如 Thrift、Protocol Buffers 等。Avro 主要的不同之处包括以下几点：

- 动态类型——Avro的实现总保持数据和其对应的结构在一起。其结果是，组装/反组装操作不需要代码生成或者静态数据类型。这也允许通用的数据处理。
- 无标签数据——由于保持数据和结构在一起，因此Avro组装/反组装不需要类型/大小信息，也不需要将人工指定的ID编码到数据中。其结果是，Avro序列化产生更小的输出。
- 增强的版本更新支持——在结构定义发生改变的情况下，Avro 包含改变前后的两个结构定义，这允许基于字段名进行不同的符号解析。

鉴于其高性能、代码量小以及结果数据紧凑的特点，Avro 不仅在 Hadoop 社区中得到广泛采用，而且还用于许多其他 NoSQL 实现(包括 Cassandra)。

Avro 的核心是数据序列化系统。Avro 可以使用反射来动态生成已有 Java 对象的结构，

或使用显式的 Avro 结构——JavaScript Object Notation(JSON)文档来描述数据格式。Avro 结构可以包含简单类型和复杂类型。

Avro 支持的简单数据类型包括 null、boolean、int、long、float、double、bytes 和 string。在这里，null 是一种特殊类型，对应于没有数据，并可以用于代替任何数据类型。

Avro 支持的复杂类型包括以下几个：

- Record——大致相当于 C 语言的结构体。Record 有名称以及可选的名称空间、文档和别名。它包含一个命名属性的列表，这些属性可以是任何 Avro 类型。
- Enum——值的枚举。Enum 有名字、可选的名称空间、文档和别名，并包含一个符号列表(合法的 JSON 字符串)。
- Array——相同类型条目的集合。
- Map——字符串类型键和特定类型值的映射表。
- Union——表示值的 or 选项。Union 的常见用法是指定可以为空的值。

**对结构演变的支持**

组装/反组装框架要考虑的一个重要问题是支持结构演变。为简化该问题的处理，Avro 支持以某种结构读取数据的能力，该结构可以不同于写入数据所使用的结构。在 Avro 中，用于写入数据的结构称为写入器的结构，而应用程序期望的结构称为读取器的结构。如果两个结构不匹配，就会出现错误。

若要匹配，必须满足以下条件之一：

- 两个结构中必须都有条目类型相匹配的 Array。
- 两个结构中必须都有条目类型相匹配的 Map。
- 两个结构中必须都有名称相匹配的 Enum。
- 两个结构中必须都有名称相匹配的 Record。
- 两个结构中必须都包含相同的基本类型。
- 任意一个结构是 Union。

类似于 Java 数据类型提升规则，写入器结构中的数据类型可能会被提升为读取器结构中的数据类型。关于 Avro 结构演变的另外一个忠告基于这样一个事实，即 Avro 不支持可选字段。其结果是，如下命题是正确的：

- 如果写入器的记录中包含的某个名称没有出现在读取器的记录中，则该字段对应的写入器的值会被忽略(写入器中的可选字段)。
- 如果读取器的记录结构中有一个包含默认值的字段，而写入器的结构中没有同名的字段，则读取器应该使用该字段的默认值(读取器中的可选字段)。
- 如果读取器的记录结构中有一个没有默认值的字段，而写入器的结构中没有同名的字段，则程序会抛出错误。

除纯粹的序列化之外，Avro 还支持 Avro RPC，允许定义 Avro 交互式数据语言(Interactive Data Language，IDL)，它基于 Avro 结构定义。据 Hadoop 开发者介绍，Avro RPC

正在逐步取代现有的 Hadoop RPC 系统，而该系统是目前 HDFS 和 MapReduce 使用的主要 Hadoop 通信机制。

最后，Avro项目已经为集成Avro和MapReduce做了大量的工作。新的org.apache.avro.mapred包提供了对在Avro数据上运行MapReduce作业的全部支持，以及提供了使用Java编写的map和reduce函数。Avro数据文件不包含Hadoop MapReduce API所期望的键/值对，而仅是一个值序列，它提供了Hadoop MapReduce API之上的一个层。

代码清单 2-11 展示了 Avro 结构的一个示例，该示例取自 Lucene HBase 实现(在第 9 章中将详细讨论该实现)。

**代码清单 2-11：Avro 结构示例**

```
{
  "type" : "record",
  "name" : "TermDocument",
  "namespace" : "com.navteq.lucene.hbase.document",
  "fields" : [ {
    "name" : "docFrequency",
    "type" : "int"
  }, {
    "name" : "docPositions",
    "type" : ["null", {
      "type" : "array",
      "items" : "int"
    }]
  } ]
}
```

一旦确定了结构，就可以使用简单的代码片段(如代码清单 2-12 中所示)来生成专门的 Avro 类。

**代码清单 2-12：编译 Avro 结构**

```
inputFile = File.createTempFile("input", "avsc");
fw = new FileWriter(inputFile);
fw.write(getSpatialtermDocumentSchema());
fw.close();
outputFile = new File(javaLocation);
System.out.println( outputFile.getAbsolutePath());
SpecificCompiler.compileSchema(inputFile, outputFile);
```

在生成 Avro 类之后，代码清单 2-13 所示的简单类就可以用于在 Java 和二进制 Avro 格式之间对数据进行组装/反组装了。

**代码清单 2-13：Avro 组装器/反组装器**

```
private static EncoderFactory eFactory = new EncoderFactory();
private static DecoderFactory dFactory = new DecoderFactory();
```

```
private static SpecificDatumWriter<singleField> fwriter = new
   SpecificDatumWriter<singleField>(singleField.class);
private static SpecificDatumReader<singleField> freader = new
   SpecificDatumReader<singleField>(singleField.class);
private static SpecificDatumWriter<FieldsData> fdwriter = new
   SpecificDatumWriter<FieldsData>(FieldsData.class);
private static SpecificDatumReader<FieldsData> fdreader = new
   SpecificDatumReader<FieldsData>(FieldsData.class);
private static SpecificDatumWriter<TermDocument> twriter = new
   SpecificDatumWriter<TermDocument>(TermDocument.class);
private static SpecificDatumReader<TermDocument> treader = new
   SpecificDatumReader<TermDocument>(TermDocument.class);
private static SpecificDatumWriter<TermDocumentFrequency> dwriter = new
   SpecificDatumWriter<TermDocumentFrequency>
   (TermDocumentFrequency.class);
private static SpecificDatumReader<TermDocumentFrequency> dreader = new
   SpecificDatumReader<TermDocumentFrequency>
   (TermDocumentFrequency.class);
private AVRODataConverter(){}

public static byte[] toBytes(singleField fData)throws Exception{

   ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
   Encoder encoder = eFactory.binaryEncoder(outputStream, null);
   fwriter.write(fData, encoder);
   encoder.flush();
   return outputStream.toByteArray();
}

public static byte[] toBytes(FieldsData fData)throws Exception{

   ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
   Encoder encoder = eFactory.binaryEncoder(outputStream, null);
   fdwriter.write(fData, encoder);
   encoder.flush();
   return outputStream.toByteArray();
}

public static byte[] toBytes(TermDocument td)throws Exception{

   ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
   Encoder encoder = eFactory.binaryEncoder(outputStream, null);
   twriter.write(td, encoder);
   encoder.flush();
   return outputStream.toByteArray();
}

public static byte[] toBytes(TermDocumentFrequency tdf)throws Exception{
```

```
    ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
    Encoder encoder = eFactory.binaryEncoder(outputStream, null);
    dwriter.write(tdf, encoder);
    encoder.flush();
    return outputStream.toByteArray();
}

public static singleField unmarshallSingleData(byte[] inputStream)throws
          Exception{

    Decoder decoder = dFactory.binaryDecoder(inputStream, null);
    return freader.read(null, decoder);
}

public static FieldsData unmarshallFieldData(byte[] inputStream)throws
          Exception{

    Decoder decoder = dFactory.binaryDecoder(inputStream, null);
    return fdreader.read(null, decoder);
}

public static TermDocument unmarshallTermDocument(byte[]
          inputStream)throws Exception{

    Decoder decoder = dFactory.binaryDecoder(inputStream, null);
    return treader.read(null, decoder);
}

public static TermDocumentFrequency unma rshallTermDocumentFrequency(byte[]
          inputStream)throws Exception{

    Decoder decoder = dFactory.binaryDecoder(inputStream, null);
    return dreader.read(null, decoder);
}
```

**当前** Avro **在使用上的一个复杂之处在于没有用于设计** Avro **结构的现成工具，对于庞大、复杂的数据结构来说，这项工作可能会相当繁重。解决该问题的一个有趣的方案是利用现有的** XML **结构**(XSD)**设计工具，并接着通过编程将这些** XSD **转换为** Avro **结构。考虑到** XSD **是目前数据定义/设计的事实标准，且有大量工具可以用于** XSD **设计，因此该方案允许数据设计者使用熟悉的工具完成** Avro **结构设计。**

**许多运行在** Hadoop **生态系统中的应用程序可能需要访问相同的数据。**HCatalog **提供了一个中心化的数据定义注册表，很多应用程序都可以使用它。**

## 2.5　利用 HCatalog 管理元数据

**在** HDFS **和** HBase **之间，**Hadoop **提供了许多保存数据的方法，使得数据可以被多个**

应用程序访问。但将数据集中存储并提供给多个应用程序访问，这样做产生了一系列新的挑战，包括以下几点：

- 如何分享数据，才能够使数据以用户想要的任何形式来保存和处理？
- 如何能够将不同的 Hadoop 应用程序和其他系统集成？

访问数据的常见方法之一是通过表抽象，该方法通常用于访问关系型数据库，并且为许多开发者所熟知(和广泛采用)。一些流行的 Hadoop 系统，例如 Hive 和 Pig，也采用了这种方法。这种抽象解除了数据如何存储(HDFS 文件、HBase 表)与应用程序如何处理数据(表格式)之间的耦合。此外，它允许从较大的数据语料库中“过滤”感兴趣的数据。

为支持这种抽象，Hive 以关系型数据库的形式提供了元存储，这允许我们捕获实际物理文件(和 HBase 表)与用于访问该数据的表(虚拟的)之间的依赖关系。

### Hive 和 Pig

传统上，数据保存在数据库中，SQL 是提供给数据工作者的主要接口。

Hadoop 的数据仓库系统——Hive，旨在为这些数据工作者简化 Hadoop 的使用，它提供了 HiveQL——一种类似 SQL 的语言，用于访问和操作基于 Hadoop 且保存在 HDFS 和 HBase 中的数据。通过将请求透明地转换为 MapReduce 执行，HiveQL 支持专有的查询、连接、摘要等。其结果是，Hive 查询不是实时执行，而是作为批量任务执行。

Pig 是另外一个 Hadoop 数据仓库系统，与类 SQL 的语言不同，它使用 Pig 的专用脚本语言——Pig Latin。Pig Latin 将数据视为一个元组集合(字段的有序集合)，允许将输入元组转换为输出。类似于 Hive，Pig 支持专有的查询、连接和其他操作，且将 Pig Latin 代码转换为 MapReduce 执行。Pig 还支持大量的并行机制和诸多优化技巧，使其能够处理非常庞大的数据集。

第 13 章有关于 Hive 和 Pig 更详细的信息。

一个新的 Apache 项目(HCatalog)扩展了 Hive 的元存储，同时保留了 Hive DDL 中用于表定义的组件。其结果是，Hive 的表抽象(当使用了 HCatalog 时)可以用于 Pig 和 MapReduce 应用程序，这带来了以下一些主要优势：

- 它使得数据消费者不必知道其数据存储的位置和方式。
- 它允许数据生产者修改物理数据存储和数据模型，同时仍然支持以旧格式存储的现有数据，从而数据消费者不需要修改他们的处理流程。
- 它为 Pig、Hive 和 MapReduce 提供了共享的结构和数据模型。

HCatalog 应用程序的数据模型以表的形式组织，表可以放入数据库中。可以基于一个或多个键对表进行散列分区，这允许我们将包含一个(或一组)给定键值的所有行组织在一起。例如，如果使用日期对一个包含三天数据的表进行分区，那么表中将会有三个分区。可以从表中动态地创建和删除新分区。分区是多维度的，而非层次化的。

分区包含多条记录。一旦创建了分区，相应的记录集就确定了，并且不能修改。记录被划分为多列，每列均有名称和数据类型。HCatalog 支持与 Hive 相同的数据类型。

HCatalog 还为“存储格式开发者”提供了一个 API，用于定义如何读取和写入保存在实际物理文件或 HBase 表中的数据(与 Hive 序列化/反序列化——SerDe 相比)。HCatalog 的默认数据格式是 RCFile。但如果数据以不同格式存储，那么用户可以实现 HCatInputStorageDriver 和 HCatOutputStorageDriver 来定义底层数据存储和应用程序记录格式之间的转换。StorageDriver 的作用域是一个分区，允许底层存储灵活地支持分区修改，或者将不同布局的多个文件合并为一个单独的表。

以下是 HCatalog 的三个基本用途：

? 工具间通信——大多数复杂的 Hadoop 应用程序都会使用多种工具来处理相同的数据。它们可能将 Pig 和 MapReduce 的组合用于抽取、转换、加载(ETL)的实现，MapReduce 用于实际的数据处理，而 Hive 用于分析查询。中心化元数据存储库的使用简化了数据共享，并确保了某个工具的执行结果总是对其他工具可见。

? 数据发现——对于大型 Hadoop 集群来说，常见的情形是应用程序和数据具有多样性。通常，一个应用程序的数据可以被其他应用程序使用，但试图发现这些情况需要大量跨应用程序的信息。在这种情况下，可以将 HCatalog 用作对任何应用程序可见的注册表。将数据在 HCatalog 中发布就可以让其他应用程序发现它们。

? 系统集成——HCatalog 所提供的 REST 服务，打开了 Hadoop 数据和处理的大门，使其可以应用在整体的企业级数据和处理基础设施中。Hadoop 以简易 API 和类似 SQL 语言的形式提供了简单的接口。

本节概述了存储数据的一些方法，以及如何对其进行组装/反组装。下一节介绍一些关于如何为特定应用程序设计数据布局的指南。

## 2.6　为应用程序选择合适的 Hadoop 数据组织形式

选择合适的数据存储是 Hadoop 整体应用程序设计中最重要的部分之一。要正确地做到这一点，必须了解哪些应用将会访问数据，以及它们的访问模式是什么。

例如，如果数据由某个 MapReduce 实现独占访问，那么 HDFS 可能是最好的选择——我们需要顺序地访问数据，而且数据本地性在整体性能中起着重要的作用。HDFS 可以很好地支持这些特性。

一旦确定了数据存储机制，接下来的任务就是选取实际的文件格式。通常，SequenceFile 是最好的选项——它的语义非常适合 MapReduce 处理，允许灵活地扩展数据模型，且支持值的独立压缩(在值数据类型较大的情况下尤其重要)。当然，我们可以使用其他文件类型，尤其是如果需要与有特定数据格式要求的其他应用程序集成的话。然而要知道，使用自定义格式(尤其是二进制)可能会导致读取、分裂和写入数据时额外的复杂性。

当然，决策过程并没有到此结束。我们还必须考虑要进行计算的类型。如果所有计算总要使用全部数据，那么就没有要额外考虑的因素了。但这种情况非常少见。通常，特定的计算仅使用数据的一个子集，这一般需要利用数据分区来避免不必要的数据读取。实际

的分区结构依赖于应用程序的数据使用模式。例如，对于空间应用的情况，常见的分区结构是基于块的分区。对于日志处理，常见的方案是两级分区——根据时间(天)和服务器。这两个级别可能顺序不同，具体取决于计算需求。要创建合适的分区结构，一个通用方法是评估数据用于计算的需求。

这种选择数据存储的方法很有效，除了有新数据产生的情况。当数据需要根据计算结果进行更新时，我们就需要考虑不同的设计方案。Hadoop 提供的唯一可更新的存储机制是 HBase。因此，如果 MapReduce 计算要更新(而非创建)数据，那么 HBase 通常是用于数据存储的最好选择。做决定时务必慎重考虑数据的大小。

如前所述，在数据(列值)过大的情况下，HBase 不是最好的选择。在这些情况下，通常的解决方案是使用 HBase/HDFS 组合——HDFS 用于存储实际数据，HBase 用于存储其索引。在这种情况下，应用程序在一个新的 HDFS 文件中写入结果，同时更新基于 HBase 的元数据(索引)。这种实现通常需要自定义数据合并(类似于之前描述的 HBase 合并)。

将 HBase 用作数据存储机制时，通常不需要应用层的数据分区——HBase 会对数据进行分区。另一方面，在使用 HBase/HDFS 组合的情况下，通常需要对 HDFS 数据进行分区，且可以将前面所述的普通 HDFS 数据分区中的相同原则作为指导。

如果数据要用于实时访问，那么根据不同的数据大小，Hadoop提供了一些可用的解决方案。如果数据的键空间相对较小，而且数据不经常改动，SequenceFile可能是一个相当不错的解决方案。对于键空间较大和有数据更新需求的情况，HBase或HBase/HDFS组合通常是最适合的解决方案。

一旦选定了数据存储，我们就必须选出一种将数据转换为字节流的方法——即 Hadoop/HBase 用于内部保存数据的格式。尽管有不同的潜在选项可以将应用特定的数据组装/反组装为字节流(从标准的 Java 序列化到自定义的封装方案)，但 Avro 提供了一种合理的通用方法，支持显著地简化封装，同时保持了性能和紧凑的数据大小。它还允许将数据定义与数据本身一起保存，从而提供了对数据版本化的强大支持。

选择合适数据存储的最后(但当然不是最不重要的)考虑是安全性(第10章提供关于 Hadoop 安全性的深入讨论)。HDFS 和 HBase 都有相当一部分安全风险，尽管目前已经修复了其中一些，但整体安全性的实现仍需要应用/企业特定的解决方案来确保数据安全性。例如，这些方案可能包括以下几点：

- 数据加密，在数据落到错误的人手中的情况下，限制信息暴露。
- 自定义防火墙，限制其他企业对 Hadoop 数据和执行的访问。
- 自定义服务层，中心化对 Hadoop 数据和执行的访问，并在服务层面实现所需要的安全性。

对于每个软件实现来说，在使用 Hadoop 时都有必要保证数据的安全。然而，我们应该仅实现真正必需的安全性。引入的安全性越多，系统将会变得越复杂(和昂贵)。

## 2.7　小结

本章讨论了 Hadoop 为数据存储提供的选项。我们学习了主要数据存储方案(HDFS 和 HBase)的架构和 Java API。我们还学习了如何使用 Avro 实现任意数据结构与用于实际物理存储的二进制流之间的转换。最后，我们学习了在为特定应用程序选择数据存储时要考虑的主要因素。

既然掌握了如何在 Hadoop 中存储数据，第 3 章就讨论如何将数据用于计算——具体来说，即用于 MapReduce 应用程序。